9th International Conference on Fun with Algorithms

FUN 2018, June 13–15, 2018, La Maddalena Island, Italy

Edited by

Hiro Ito
Stefano Leonardi
Linda Pagli
Giuseppe Prencipe

LIPICS

LIPIcs – Vol. 100 – FUN 2018 www.dagstuhl.de/lipics

Editors

Hiro Ito
School of Informatics and Engineering
The University of Electro-Communications
itohiro@uec.ac.jp

Stefano Leonardi
Dipartimento di Ing. Informatica Automatica e Gestionale
Sapienza Università di Roma
leonardi@diag.uniroma1.it

Linda Pagli
Dipartimento di Informatica
Università di Pisa
linda.pagli@unipi.it

Giuseppe Prencipe
Dipartimento di Informatica
Università di Pisa
giuseppe.prencipe@unipi.it

ACM Classification 2012
Theory of computation → Complexity classes, Theory of computation → Algorithm design techniques, Theory of computation → Computability, Theory of computation → Approximation algorithms analysis, Mathematics of computing → Combinatorics, Mathematics of computing → Combinatorial algorithms, Computing methodologies

ISBN 978-3-95977-067-5

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/978-3-95977-067-5.

Publication date
June, 2018

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

Digital Object Identifier: 10.4230/LIPIcs.FUN.2018.0

ISBN 978-3-95977-067-5 **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.

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Invited Papers

Regular Papers

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

FUN with Algorithms is dedicated to the use, design, and analysis of algorithms and data structures, focusing on results that provide amusing, witty but nonetheless original and scientifically profound contributions to the area. Donald Knuth's famous quote captures this spirit nicely:*.... pleasure has probably been the main goal all along. But I hesitate to admit it, because computer scientists want to maintain their image as hard-working individuals who deserve high salaries. Sooner or later society will realise that certain kinds of hard work are in fact admirable even though they are more fun than just about anything else.*

The previous FUNs were held in Elba Island, Italy; in Castiglioncello, Tuscany, Italy; in Ischia Island, Italy; in San Servolo Island, Venice, Italy; in Lipari Island, Sicily, Italy; and in La Maddalena Island, Sardinia, Italy. Special issues of Theoretical Computer Science, Discrete Applied Mathematics, and Theory of Computing Systems were dedicated to them.

This volume contains the papers presented at the 9th International Conference on Fun with Algorithms 2018, held on June 13-5, 2018, on La Maddalena Island, Italy. The call for papers attracted 55 submissions from all over the world, addressing a wide variety of topics, reviewed by three Program Committee members. After a careful reviewing process and a thorough discussion, the committee decided to accept 30 papers. In addition, the program featured two invited talks by Martin Farach-Colton and Kokichi Sugihara. Extended versions of selected papers will appear in a special issue of the journal Theoretical Computer Science.

We thank all authors who submitted their work to FUN 2018, all Program Committee members for their expert assessments and the ensuing discussions, all external reviewers for their kind help, and Atsuki Nagao for taking care of the web management of the conference. We used EasyChair (http://www.easychair.org/), that greatly facilitated the entire preparation of the conference, for handling submissions, reviews, the selection of papers, and the production of this volume. Warm thanks also go to Michael Wagner for following carefully the process of proceedings' publication in LIPIcs series.

May, 2018

Hiro Ito
Stefano Leonardi
Linda Pagli
Giuseppe Prencipe

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Conference Organization

Program Committee

Anna Bernasconi, U. Pisa, Italy
Allan Borodin, U. Toronto, Canada
Artur Czumaj, U. Warwick, UK
Erik Demaine, MIT, USA
David Eppstein, U. California Irvine, USA
Guy Even, Tel-Aviv University, Israel
Michele Flammini, GSSI & L'Aquila U., Italy
Rudolf Fleischer, GUtech, Oman
Paola Flocchini, Ottawa U., Canada
Fedor Fomin, U. Bergen, Norway
Naveen Garg, IIT Delhi, India
Fabrizio Grandoni, IDSIA Lugano, Switzerland
Takashi Horiyama, Saitama U., Japan
John Iacono, U. Libre Bruxelles, Belgium
Hiro Ito, UEC, Japan (co-chair)
Marc van Kreveld, Utrecht U., Netherlands
Stefan Langerman, U. Libre Bruxelles, Belgium
Stefano Leonardi, Sapienza U. Rome, Italy (co-chair)
Anna Lubiw, U. Waterloo, Canada
Flaminia Luccio, Ca' Foscari U. Venice, Italy
S. Muthu Muthukrishnan, Rutgers U., USA
Yoshio Okamoto, UEC, Japan
Mike Paterson, U. Warwick, UK
David Peleg, Weizmann Inst. Sci., Israel
Nadia Pisanti, ERABLE Team INRIA & U. Pisa, Italy
Geppino Pucci, U. Padova, Italy
Laura Sanità, U. Waterloo, Canada
Aravind Srinivasan, U. Maryland, College Park, USA
Hideki Tsuiki, Kyoto U., Japan
Ryuhei Uehara, JAIST, Japan
Yushi Uno, Osaka Prefecture U., Japan
Aaron Williams, Bard College at Simon's Rock, USA

Steering Commitee

Erik Demaine, MIT, USA
Fabrizio Grandoni, IDSIA, Switzerland
Linda Pagli, U. Pisa, Italy
Giuseppe Prencipe, U. Pisa, Italy
Nicola Santoro, Carleton U., Canada
Ugo Vaccaro, U. Salerno, Italy

Organizers

Linda Pagli, U. Pisa, Italy
Giuseppe Prencipe, U. Pisa, Italy
Atsuki Nagao, Ochanomizu U., Japan (web manager)

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

External Reviewers

Afrouz Jabalameli
Akihiro Uejima
Akira Suzuki
Alessio Conte
Andrea Marino
Ben Sach
Claudio Gallicchio
Daniele Frigioni
Davide Bilò
Eyal Kushilevitz
Giovanni Viglietta
Jun Kawahara
Junichi Teruyama
Kazuhisa Seto
Kazuo Iwama
Konstantinos Georgiou
Luca Versari
Luciano Gualà
Paweł orSchmidt
Stefano Leucci
Suguru Tamaki
Tom van der Zanden
Waldo Gálvez
Ziv Scully

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Adam Hesterberg
Massachusetts Institute of Technology
United States
achester@mit.edu

Alessandro Panconesi
Sapienza University of Rome
Italy
ale@di.uniroma1.it

Andreas Tönnis
Univesidad de Chile
Chile
atoennis@uni-bonn.de

Andrés Cristi
Universidad de Chile
Chile
andres.cristi.e@gmail.com

Augustin Chaintreau
Columbia University
United States
augustin@cs.columbia.edu

Christine Markarian
University of Paderborn
Germany
chrissm@mail.uni-paderborn.de

Chuzo Iwamoto
Hiroshima University
Japan
chuzo@hiroshima-u.ac.jp

Clemens Thielen
University of Kaiserslautern
Germany
thielen@mathematik.uni-kl.de

Daniel Smolyak
University of Maryland
United States
dsmolyak@gmail.com

Danny Krizanc
Wesleyan University
United States
dkrizanc@wesleyan.edu

David Eppstein
University of California, Irvine
United States
david.eppstein@gmail.com

Davide Bilò
University of Sassari, Italy
Italy
davide.bilo@uniss.it

Dorian Mazauric
INRIA
France
dorian.mazauric@inria.fr

Erik D. Demaine
Massachusetts Institute of Technology
United States
edemaine@mit.edu

Erik Metz
University of Maryland
United States
emetz1618@gmail.com

Evangelos Kranakis
Carleton UniversityComputer Science
Canada
kranakis@scs.carleton.ca

Fábio Botler
Universidad de Valparaíso
Chile
fabio.botler@gmail.com

Florian David Schwahn
University of Kaiserslautern
Germany
fschwahn@mathematik.uni-kl.de

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Friedhelm Meyer Auf der Heide
Heinz Nixdorf Institute & Department of Computer Science, University of Paderborn
Germany
fmadh@upb.de

Guangqi Cui
Montgomery Blair High School
United States
bestwillcui@gmail.com

Guido Proietti
Università L'Aquila, Italy and Istituto di Analisi dei Sistemi ed Informatica, IASI-CNR, Roma
Italy
guido.proietti@univaq.it

Guillaume Ducoffe
ICI
Romania
guillaume.ducoffe@ici.ro

Hans L. Bodlaender
Utrecht University
Netherlands
H.L.Bodlaender@uu.nl

Heiko Hamann
University of Lübeck
Germany
hamann@iti.uni-luebeck.de

Hideaki Sone
Tohoku University
Japan
tm-paper+cardanysone@g-mail.tohoku-university.jp

Isaac Grosof
Carnegie Mellon University
United States
isaacbg227@gmail.com

Jacob Prinz
University of Maryland
United States
jacobeliasprinz@gmail.com

Jan-Tobias Maurer
"Institut f""ur Informatik, Universit""at Giessen"
Germany
jan.t.maurer@math.uni-giessen.de

Jannik Dreier
LORIA, Université de Lorraine, INRIA, CNRS
France
jannik.dreier@loria.fr

Jaroslav Opatrny
Concordia University
Canada
opatrny@cs.concordia.ca

Jayson Lynch
Massachusetts Institute of Technology
United States
jaysonl@mit.edu

Jean-Claude Bermond
CNRS
France
jean-claude.Bermond@inria.fr

Jean-Guillaume Dumas
Université Grenoble Alpes
France
jean-guillaume.dumas@univ-grenoble-alpes.fr

Jeffrey Bosboom
Massachusetts Institute of Technology
United States
jbosboom@csail.mit.edu

John Dickerson
University of Maryland
United States
john@cs.umd.edu

Joshua Lockhart
University College London
United Kingdom
jlockhart06@qub.ac.uk

Jurek Czyzowicz
Universite du Quebec en Outaouais
Canada
jurek.czyzowicz@uqo.ca

Justin Kopinsky
Massachusetts Institute of Technology
United States
jkopin@mit.edu

Kazumasa Shinagawa
Tokyo Institute of Technology, AIST
Japan
shinagawa.k.aa@m.titech.ac.jp

Kei Kimura
Toyohashi University of Technology
Japan
kimura@cs.tut.ac.jp

Kevin Schewior
Universidad de Chile
Chile
kschewior@gmail.com

Konstantinos Georgiou
Ryerson University
Canada
konstantinos@ryerson.ca

Lata Narayanan
Concordia University
Canada
lata@cs.concordia.ca

Leonid Sedov
Linkoping University
Sweden
leo@gmail.com

Linus Hamilton
Massachusetts Institute of Technology
United States
linyks@gmail.com

Luciano Gualà
Dipartimento di Matematica, Universitá di Tor Vergata, Roma
Italy
guala@mat.uniroma2.it

Manuel Lafond
University of Ottawa
Canada
mlafond2@uOttawa.ca

Markus Holzer
Institut für Informatik, Universität Giessen
Germany
holzer@informatik.uni-giessen.de

Masashi Kiyomi
Yokohama City University
Japan
masashi@yokohama-cu.ac.jp

Masato Haruishi
Hiroshima University
Japan

Matteo Almanza
Sapienza University of Rome
Italy
matteojug@gmail.com

Mehdi Khosravian Ghadikolaei
LAMSADE, Université Paris Dauphine
France
m.khosravian@gmail.com

Michael Coulombe
MIT
United States
mcoulomb@mit.edu

Michael Lampis
LAMSADE, Université Paris Dauphine
France
michail.lampis@dauphine.fr

Mikhail Rudoy
Massachusetts Institute of Technology
United States
mrudoy@gmail.com

Mirko Rossi
University of Rome Tor Vergata
Italy
r.mirko25@gmail.com

Mostafa Wahby
University of Lübeck
Germany
wahby@iti.uni-luebeck.de

Naoki Kitamura
Nagoya Institute of Technology
Japan
ktmr522@yahoo.co.jp

Naveen Durvasula
Montgomery Blair High School
United States
140.naveen.d@gmail.com

Naveen Raman
Richard Montgomery High School
United States
nav.j.raman@gmail.com

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

Paolo Boldi
University of Milan
Italy
paolo.boldi@unimi.it

Pascal Lafourcade
LIMOS, University Clermont Auvergne
France
pascal.lafourcade@udamail.fr

Petra Wolf
University of Tübingen
Germany
wolfp@informatik.uni-tuebingen.de

Rémy Belmonte
The University of Electro-Communications
Japan
remybelmonte@gmail.com

Ronald de Haan
University of Amsterdam
Netherlands
me@ronalddehaan.eu

Ruben Hoeksma
Universität Bremen
Germany
hoeksma@uni-bremen.de

Ryan Killick
Carleton University
Canada
RyanKillick@cmail.carleton.ca

Sebastiano Vigna
University of Milan
Italy
sebastiano.vigna@unimi.it

Stefano Leucci
ETH Zurich
Italy
stefano.leucci@inf.ethz.ch

Sung Hyun Yoo
Bergen County Academies
United States
sunnyyoo812@gmail.com

Sunil Shende
Rutgers University
United States
sunil.shende@rutgers.edu

Taisuke Izumi
Nagoya Institute of Technology
Japan
t-izumi@nitech.ac.jp

Takaaki Mizuki
Tohoku University
Japan
mizuki@cc.tohoku.ac.jp

Takuya Kamehashi
Toyohashi University of Technology
Japan
kamehashi@algo.cs.tut.ac.jp

Tami Tamir
The Interdisciplinary Center
Israel
tami@idc.ac.il

Tatsuaki Ibusuki
Hiroshima University
Japan

Tatsuya Sasaki
Tohoku University
Japan
tatsuya.sasaki.p2@dc.tohoku.ac.jp

Tom van der Zanden
Utrecht University
Netherlands
T.C.vanderZanden@uu.nl

Toshihiro Fujito
Toyohashi University of Technology
Japan
fujito@cs.tut.ac.jp

Valentin Polishchuk
Linkoping University
Sweden
valentin.polishchuk@liu.se

William Gasarch
University of Maryland
United States
gasarch@cs.umd.edu

Xavier Bultel
Université d'Auvergne
France
xavier.bultel@udamail.fr

Yota Otachi
Kumamoto University
Japan
otachi@cs.kumamoto-u.ac.jp

Yuya Kawabata
Nagoya Institute of Technology
Japan
29414043@stn.nitech.ac.jp

Zachary Abel
Massachusetts Institute of Technology
United States
zabel@math.mit.edu

Mind the Gap

Martín Farach-Colton[1]
Rutgers University, Department of Computer Science, Piscataway, NJ 08854, USA
martin@farach-colton.com
https://orcid.org/0000-0003-3616-7788

Abstract

As a New Yorker, I'm painfully aware of space. There is, after all, nothing more luxurious than empty space! So when it comes to algorithms, I'm all in favor of leaving holes in my data structures. In this talk, I'll explore the advantages of pampering algorithms with some much needed breathing room.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases library sort, Italian island, packed memory arrays, weight balanced trees, Italians know how to throw a conference

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.1

Category Invited Paper

[1] Supported by NSF CCF 1637458, NIH 1 U01 CA198952-01, a NetAPP Faculty Fellowship and a gift from Dell/EMC.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 1; pp. 1:1–1:1
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Evolution of Impossible Objects

Kokichi Sugihara
Meiji Institute for Advanced Study of Mathematical Sciences, Meiji University, 4-21-1 Nakano, Nakano-ku, Tokyo 164-8525, Japan
http://www.isc.meiji.ac.jp/~kokichis/
kokichis@isc.meiji.ac.jp

Abstract

Impossible objects – 3D objects that can create a visual effect that seems impossible – can be classified by generation based on the order in which they were discovered or produced. The first generation consists of objects whose appearance when observed from a certain viewpoint matches a picture of an impossible object. Many such objects can be created, as there are multiple 3D objects that will project the same two-dimensional picture, including shapes that the human vision system is unable to perceive. The gap between the mathematical and the psychological can also create other types of "impossible" visual effects. Impossible objects are here classified into seven groups.

2012 ACM Subject Classification Applied computing → Computer-aided design

Keywords and phrases Ambiguous cylinder, anomalous picture, impossible motion, impossible object, optical illusion

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.2

Category Invited Paper

Acknowledgements This work is supported by the Grant-in-Aid for Basic Scientific Research (A) No. 16H01728 of MEXT.

1 Introduction

"Anomalous pictures" or "pictures of impossible objects" are a class of pictures that give viewers the impression of a 3D structure that is perceived to be impossible [2, 16]. A typical example of such a picture is the endless loop of stairs proposed by Penrose and Penrose [6] and which appears in Escher's "Ascending and Descending" [4].

It was once thought that impossible objects exist only in the mind and that they could not be constructed as actual 3D structures. However, several tricks were soon found for creating 3D structures that could reproduce pictures of impossible objects. One such trick is to make a discontinuous structure appear to be continuous when seen from a particular viewpoint [5, 7]. A second trick is to use curved surfaces instead of planar surfaces [1, 3].

Sugihara describes a third trick [8, 9] in which the creator uses angles other than 90 degrees to produce a rectangular look. He called this the "non-rectangularity trick". He extended the trick in various directions and proposed new types of impossible objects, including "impossible motion objects" in which the inserted motions appear to be impossible; "ambiguous cylinders", whose mirror images appear to be impossible; and "partly invisible objects", parts of which disappear when reflected in a mirror.

In this presentation, we classify Sugihara's impossible objects according to their generation and present typical examples. We also touch on some of the underlying mathematics.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 2; pp. 2:1–2:8
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

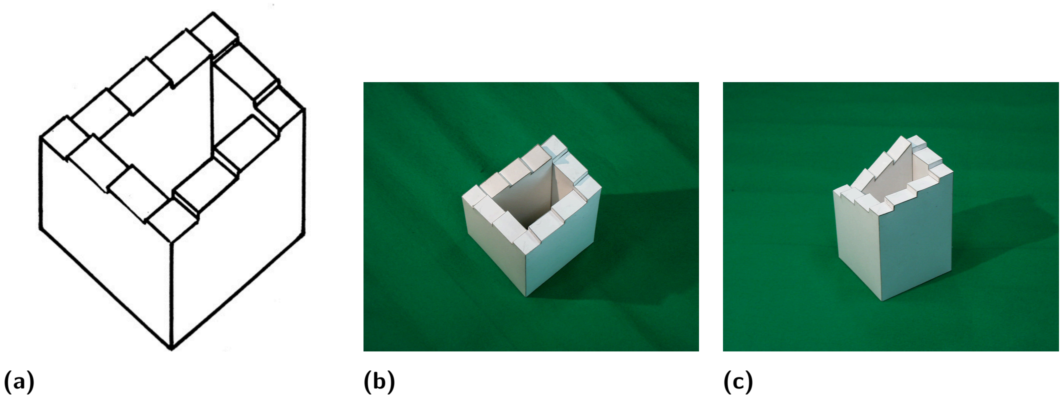


Figure 1 First-generation "anomalous object": (a) picture of an endless loop of stairs; (b) solid object producing the same picture; (c) another view of the object.

2 First-Generation "Anomalous Objects"

Not all of the 3D objects represented in anomalous pictures are impossible; in some cases, they can be constructed as 3D solid objects. An example is shown in Fig. 1, where panel (a) shows a picture of an endless loop of stairs and panel (b) shows an object made from paper whose appearance from above coincides with the picture. The stairs at the top of the four walls form a loop, suggesting that if we continue to ascend the stairs, we will eventually come back to the starting point, which is impossible since ascending the stairs should bring us always to a higher position. Fig. 1 (c) shows another view of the object, from which we can see that the stairs on the left rear wall are not normal. Note that in this realization, all faces are planar and the structures that look connected are actually connected.

This kind of realization can be found mathematically in the following way. As illustrated in Fig. 2, let us fix an (x, y, z) Cartesian coordinate system, place a picture on the $z = 1$ plane, and fix the viewpoint at the origin. We are interested in judging whether the object represented by the picture is realizable. Let $(x_i, y_i, 1)$ be the coordinates of the i-th vertex in the picture. The associated vertex in 3D space, if it exists, should have coordinates $(x_i/t_i, y_i/t_i, 1/t_i)$, where t_i is an unknown variable, because the vertex should be on the half-line emanating from the origin and passing through the vertex $(x_i, y_i, 1)$. For the j-th face, let

$$a_j x + b_j y + c_j z + 1 = 0 \tag{1}$$

be the equation of the plane containing the face. All the coefficients a_j, b_j, c_j are unknowns.

Suppose that the i-th vertex is on the j-th face. Then we can substitute the coordinates of the vertex into the face equation and obtain

$$a_j x_i + b_j y_i + c_j + t_i = 0,$$

which is linear in the unknowns t_i, a_j, b_j, c_j. We obtain a similar equation for each such vertex and face pair. Collecting them all, we get a system of linear equations

$$Aw = 0, \tag{2}$$

where A is a constant matrix and w is the vector of unknown variables.

Pictures also have relative depth information. As shown in Fig. 3, let l be an edge separating the j-th face and the k-th face. Suppose that l is a convex edge and the k-th face

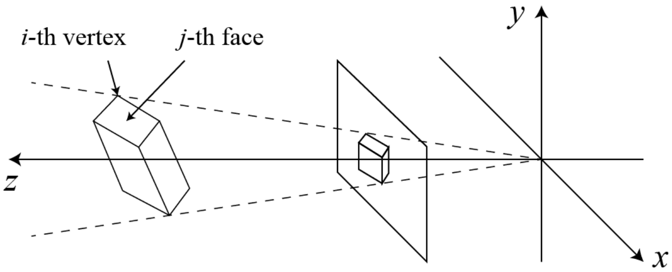


Figure 2 Object and its projection.

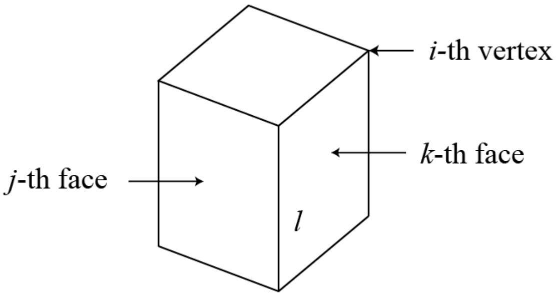


Figure 3 Relative depth.

contains the i-th vertex. Then the plane containing the j-th face passes between the i-th vertex and the viewpoint, which is represented by

$$a_j x_i + b_j y_i + c_j + t_i < 0. \tag{3}$$

Collecting all similar inequalities, we get the system of linear inequalities

$$Bw > 0. \tag{4}$$

We can then prove that the picture represents a polyhedral object if and only if the system of equations (2) and inequalities (4) has solutions [8]. Thus, we can determine the realizability of impossible objects. The object shown in Fig. 1 corresponds to a solution of the system of equations and inequalities. In this way, we can construct a potentially large number of 3D objects whose projections coincide with anomalous pictures. We classify this type of impossible object as first generation and call the objects “anomalous objects”.

3 Second-Generation “Impossible Motion Objects”

If a given picture is correct, then the associated system of equations (2) and inequalities (4) has solutions but they are not unique. We can utilize this property to construct another type of impossibility. Let D be a picture of an ordinary 3D object. The system of (2) and (4) contains a solution corresponding to the original object, but it contains many other solutions. Each solution corresponds to an object whose appearance is the same as the original object but whose actual shape is different. For example, a picture of a slope ascending rightward also contains a solution corresponding to a slope ascending leftward. By choosing an appropriate

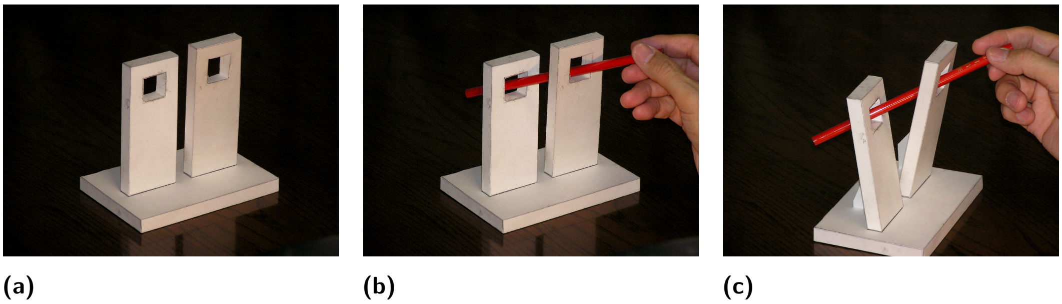


Figure 4 Second-generation “impossible motion object”: (a) two walls with holes; (b) a rod penetrating two windows; (c) another view.

solution, we can create the impression of impossible motion such as a ball climbing a slope against gravity [10].

An example is shown in Fig. 4, where panel (a) shows an object composed of two walls with rectangular holes, and panel (b) shows the motion of a rigid straight rod penetrating through two windows. Fig. 4(c) shows another view of the object, from which we can see that the actual shape of the object is different from what we perceive when we see the image in Fig. 4(a).

We classify this type of impossible object as second generation and name the objects “impossible motion objects”.

4 Cylinder-Type Impossible Objects

Anomalous objects and impossible motion objects create an “impossible” visual effect when they are seen from a single special viewpoint. Another way to generate a sense of impossibility is to observe an object from two or more viewpoints. If the appearance of the object is so different when seen from the different viewpoints that the viewer is unable to believe that it is the same object, then we may well have found a new impossible object. On the basis of this idea, we can construct several additional classes of impossible objects.

4.1 Third-Generation “Ambiguous Cylinders”

The system of linear equations and equalities described in (2) and (4) has infinitely many solutions if the associated picture is correct. This implies that the same 2D appearance can be realized by many different 3D shapes, and, consequently, we may construct a 3D object that projects two desired appearances when the object is seen from two special viewpoints. This kind of object can be found by solving two systems of (2) and (4) corresponding to two pictures with respect to two viewpoints.

An example is shown in Fig. 5. As shown in panel (a), when the cylinder is viewed directly, it has the shape of a full moon, but when it is reflected in the vertical mirror behind, it has the shape of a star.

We classify these types of impossible objects as third generation and call them “ambiguous cylinders” [11].

4.2 Fourth-Generation “Partly invisible Objects”

The design method used for ambiguous cylinders can also be used to create another visual illusion in which part of an object disappears when it is seen from a second viewpoint. To

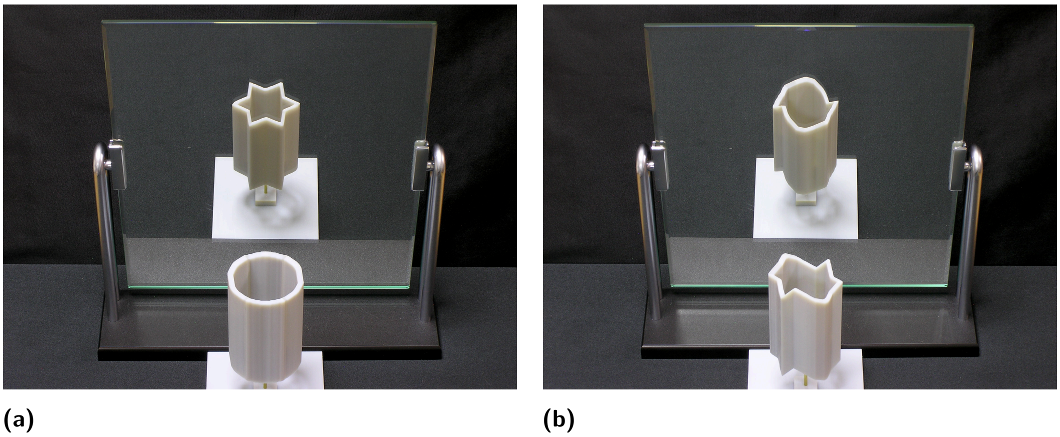


Figure 5 Third-generation "ambiguous cylinder": (a) full moon and star; (b) another view.

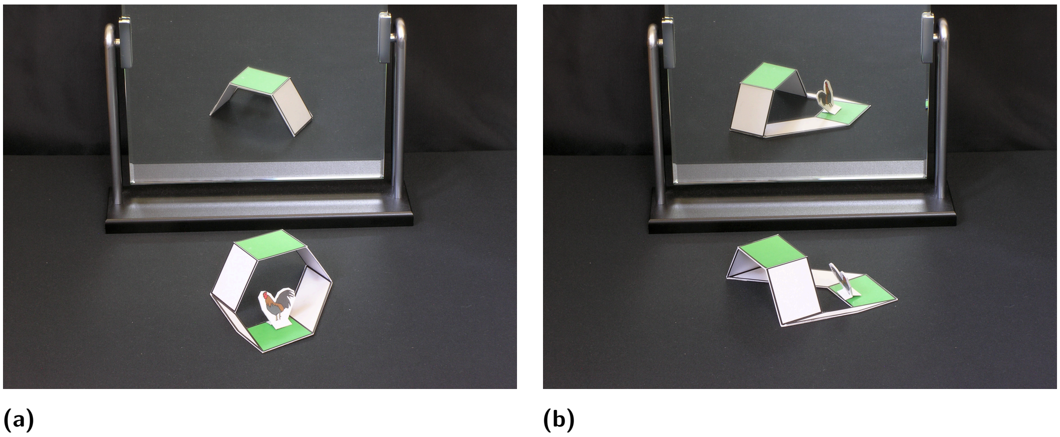


Figure 6 Fourth-generation "partly invisible object": (a) object VH; (b) another view.

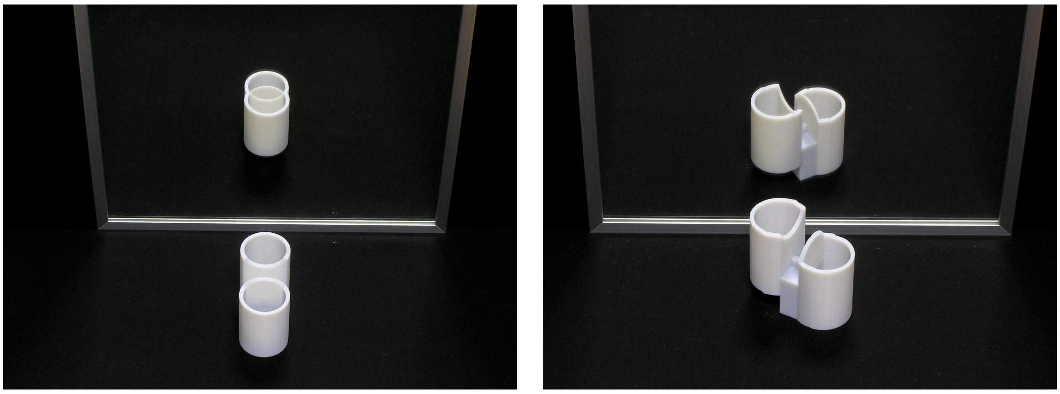
understand how this might work, consider an object composed of two parts, A and B. We can construct an ambiguous cylinder in such a way that part A appears as it is from both viewpoints, whereas part B appears as A when seen from the second viewpoint. The resulting object is such that, when we see the object from the second viewpoint, parts A and B overlap and, as a consequence, one is hidden by the other.

An example is given in Fig. 6. In panel (a), the object appears to be a regular hexagonal cylinder on its side, while the lower half disappears in the mirror. As shown in panel (b), the lower half is actually horizontal and gets hidden behind the upper half when seen from the second viewpoint.

We classify these types of impossible objects as fourth generation and name them "partly invisible objects" [14].

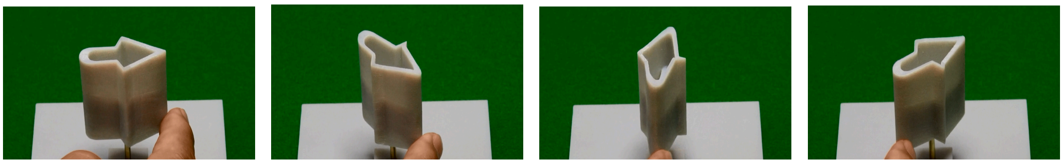
4.3 Fifth-Generation "Topology-Disturbing Objects"

We can apply the design method for ambiguous cylinders in still another way, one whereby we create objects whose topology changes in the mirror. An example is shown in Fig. 7. As shown in panel (a), the object appears in both views to consist of two circular cylinders; however, the cylinders are separated in the direct view, while they appear to be mutually



(a) (b)

Figure 7 Fifth-generation “topology-disturbing objects”: (a) two vertically aligned cylinders; (b) another view.



(a) (b) (c) (d)

Figure 8 Sixth-generation “deformable objects”: (a) arrow that likes to face rightward; (b), (c), (d) sequence of other views.

intersecting in the mirror. In other words, the shape of each part does not change, but their topology is disturbed in the mirror. The actual shape of the object can be understood when we see it from another direction, as in panel (b).

We classify these types of impossible objects as fifth generation and name them “topology-disturbing objects” [15].

4.4 Sixth-Generation “Deformable Objects”

Some ambiguous cylinders create another interesting visual effect in the sense that the rotation of the object around a vertical axis generates the impression of a dynamic change of shape. An example is shown in Fig. 8, where panels (a), (b), (c), and (d) show pictures of the object being rotated around a vertical axis by approximately 0, 40, 100, and 150 degrees. The object appears to be an arrow facing rightward; however, if we rotate it by 180 degrees around the vertical axis, it again faces rightward. Moreover, during the rotation, the object appears to be deforming continuously [13].

We call this type of impossible object sixth generation and name the objects “deformable objects”.

4.5 Eighth-Generation “Reflexively Fused Objects”

Another application of the ambiguous cylinder is to make part of the goal shape as a solid object and to provide the remaining part by its mirror image. An example is shown in Fig. 9, where panel (a) shows an object, panel (b) shows the same object placed on a horizontally

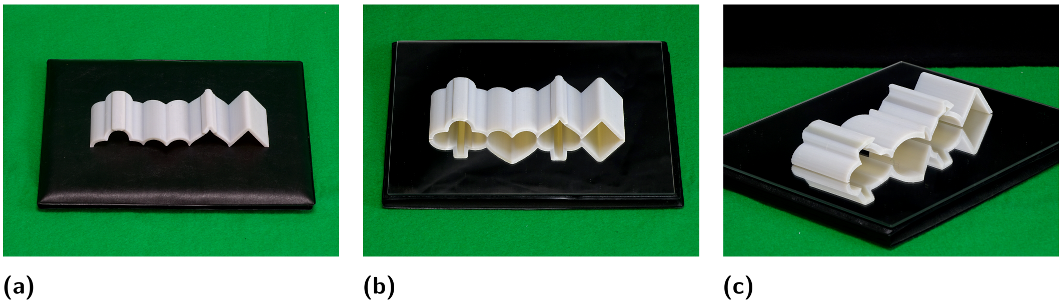


Figure 9 Eighth-generation “reflexively fused object”: (a) object alone; (b) object on a horizontally oriented mirror; (c) another view.

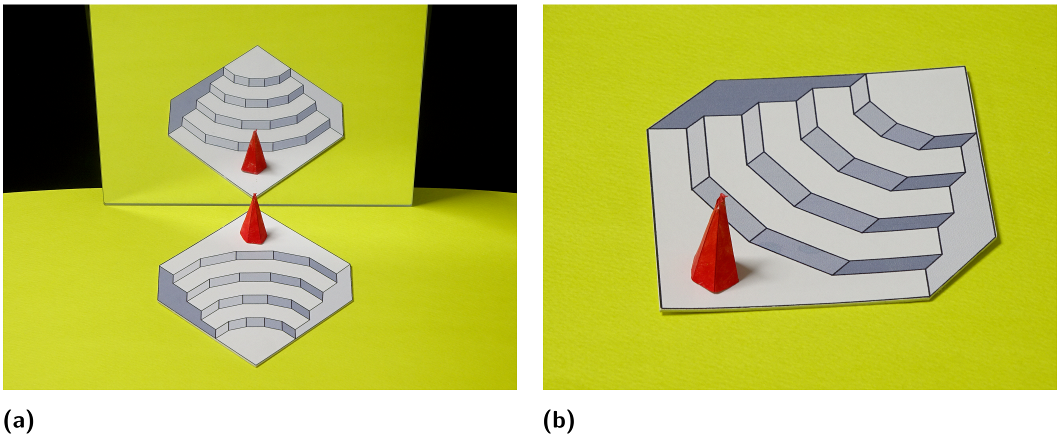


Figure 10 Seventh-generation “height-reversal object”: (a) amphitheater and hill; (b) another view.

oriented mirror so that the object and its mirror image are fused, and panel (c) shows another view of the object on the mirror. The object itself is meaningless, but the object together with its mirror image gives a meaningful shape. This type of an object can be constructed in the following way. We first decompose a goal shape into an upper part and a lower part, next transform the lower part into its height-reversed version, and finally apply the ambiguous cylinder method to this pair of the shapes.

We call this type of impossible object eighth generation and name the objects “reflexively fused objects”.

5 Seventh-Generation “Height Reversal Objects”

A picture placed on a horizontal plane can sometimes generate two interpretations of a 3D object whose height is reversed when the object is seen from mutually opposite sides with the same slant angle [12]. If we add to such a picture a 3D object showing the direction of gravity and place a vertical mirror behind it, then the direct view and the mirror image give quite different impressions of the 3D surfaces. An example is shown in Fig. 10. The direct view appears to be an amphitheater with the stage at the bottom, whereas the mirror image appears to be a hill.

We classify these types of impossible objects as seventh generation and name them “height-reversal objects” [12].

6 Concluding Remarks

We have classified objects that generate the impression of impossibility into several generations and shown an example object for each generation, where all of objects involve visual illusions.

The mathematics behind the illusions is based on the principle that a single image does not have depth information and hence there are many possible 3D shapes that give the same 2D appearance. By combining this mathematical property with the psychological preferences of human vision systems, we can effectively create many new visual effects, with potential applications to toys, tourism, magic, and so on. Our next goal is to realize these various applications.

References

1 G. Elber. Modeling (seemingly) impossible models. *Computer and Graphics*, 35(3):632–638, 2011.

2 B. Ernst. *The Eye Beguiled: Optical Illusion*. Benedikt Taschen Verlag GmbH, 1992.

3 B. Ernst. *Impossible World*. Taschen GmbH, 2006.

4 M. C. Escher. *M. C. Escher: The Graphic Work, 25th Edition*. Taschen America, 2008.

5 R. L. Gregory. *The Intelligent Eye*. Weidenfeld and Nicolson, 1970.

6 L. S. Penrose and R. Penrose. Impossible objects: a special type of visual illusion. *British Journal of Psychology*, 49(1):31–33, 1958.

7 A. I. Seckel. *Master of Deception*. Sterling Publishing Co., Inc., 2004.

8 Kokichi Sugihara. *Machine Interpretation of Line Drawings*. The MIT Press, 1996.

9 Kokichi Sugihara. Three-dimensional realization of anomalous pictures: An application of picture interpretation theory to toy design. *Pattern Recognition*, 30(7):1061–1067, 1997.

10 Kokichi Sugihara. Design of solids for antigravity motion illusion. *Computational Geometry: Theory and Applications*, 47(6):675–682, 2014.

11 Kokichi Sugihara. Ambiguous cylinders: A new class of impossible objects. *Computer Aided Drafting, Design and Manufacturing*, 25(3):19–25, 2015.

12 Kokichi Sugihara. Height reversal generated by rotation around a vertical axis. *Journal of Mathematical Psychology*, 68-69(October-December):7–12, 2015.

13 Kokichi Sugihara. Anomalous mirror symmetry generated by optical illusion. *Symmetry*, 8(4):2–22, 2016.

14 Kokichi Sugihara. A new type of impossible objects that become partly invisible in a mirror. *Japan Journal of Industrial and Applied Mathematics*, 33(3):525–535, 2016.

15 Kokichi Sugihara. Topology-disturbing objects: A new class of 3d optical illusion. *Journal of Mathematics and the Arts*, 12(1):2–18, 2018.

16 J. T. Unruh. *Impossible Objects*. Sterling Publishing Co., Inc., 2001.

Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible

Zachary Abel
MIT EECS Department, 50 Vassar St., Cambridge, MA 02139, USA
zabel@mit.edu

Jeffrey Bosboom
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
jbosboom@csail.mit.edu

Erik D. Demaine
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
edemaine@mit.edu

Linus Hamilton
MIT Mathematics Department, 77 Massachusetts Avenue, Cambridge, MA 02139, USA
luh@mit.edu

Adam Hesterberg
MIT Mathematics Department, 77 Massachusetts Avenue, Cambridge, MA 02139, USA
achester@mit.edu

Justin Kopinsky
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
jkopin@mit.edu

Jayson Lynch
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
jaysonl@mit.edu

Mikhail Rudoy[1]
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
mrudoy@gmail.com

Abstract

We analyze the computational complexity of the many types of pencil-and-paper-style puzzles featured in the 2016 puzzle video game *The Witness*. In all puzzles, the goal is to draw a path in a rectangular grid graph from a start vertex to a destination vertex. The different puzzle types place different constraints on the path: preventing some edges from being visited (broken edges); forcing some edges or vertices to be visited (hexagons); forcing some cells to have certain numbers of incident path edges (triangles); or forcing the regions formed by the path to be partially monochromatic (squares), have exactly two special cells (stars), or be singly covered by given shapes (polyominoes) and/or negatively counting shapes (antipolyominoes). We show that any *one* of these clue types (except the first) is enough to make path finding NP-complete ("witnesses exist but are hard to find"), even for rectangular boards. Furthermore, we show that a final clue type (antibody), which necessarily "cancels" the effect of another clue in the same region, makes path finding Σ_2-complete ("witnesses do not exist"), even with a single antibody (combined with many anti/polyominoes), and the problem gets no harder with many antibodies.

[1] Now at Google Inc.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 3; pp. 3:1–3:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases video games, puzzles, hardness

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.3

Related Version https://arXiv.org/abs/1804.10193

Acknowledgements We thank Jason Ku and Quanquan Liu for helpful discussions related to this paper. Most figures were produced using SVG Tiler (https://github.com/edemaine/svgtiler), based on SVG source code from The Windmill (https://github.com/thefifthmatt/windmill-client/), a clone of Witness-style puzzles distributed under Apache License 2.0.

1 Introduction

The Witness [9] is an acclaimed 2016 puzzle video game designed by Jonathan Blow (who originally became famous for designing the 2008 platform puzzle game Braid, which is undecidable [5]). The Witness is a first-person adventure game, but the main mechanic of the game is solving 2D puzzles presented on flat panels (sometimes CRT monitors) within the game. The 2D puzzles are in a style similar to pencil-and-paper puzzles, such as Nikoli puzzles. Indeed, one clue type in Witness (triangles) is very similar to the Nikoli puzzle *Slitherlink* (which is NP-complete [10]).

In this paper, we perform a systematic study of the computational complexity of all single-panel puzzle types in The Witness, as well as some of the 3D "metapuzzles" embedded in the environment itself. Table 1 summarizes our single-panel results, which range from polynomial-time algorithms (as well as membership in L) to completeness in two complexity classes, NP (i.e., Σ_1) and the next level of the polynomial hierarchy, Σ_2. Table 3 summarizes our metapuzzle results, where PSPACE-completeness typically follows immediately.

For omitted proofs, see [1].

Witness puzzles. Single-panel puzzles in The Witness (which we refer to henceforth as *Witness puzzles*) consist of an $m \times n$ full rectangular grid;[2] one or more *start circles* (drawn as a large dot, ●); one or more *end caps* (drawn as half-edges leaving the rectangle boundary); and zero or more *clues* (detailed below) each drawn on a vertex, edge, or cell[3] of the rectangular grid. Figure 1 shows a small example and its solution. The goal of the puzzle is to find a path that starts at one of the start circles, ends at one of the end caps, and satisfies all the constraints imposed by the clues (again, detailed below). We focus on the case of a single start circle and single end cap, which makes our hardness proofs the most challenging.

We now describe the clue types and their corresponding constraints. Table 2 lists the clues by what they are drawn on — grid edge, vertex, or cell — which we refer to as *this* edge, vertex, or cell. While the last five clue types are drawn on a cell, their constraint

[2] While most Witness puzzles have a rectangular boundary, some lie on a general grid graph. This generalization is mostly equivalent to having broken-edge clues (defined below) on all the non-edges of the grid graph, but the change in boundary can affect the decomposition into regions. We focus here on the rectangular case because it is most common and makes our hardness proofs most challenging.

[3] We refer to the unit-square faces of the rectangular grid as *cells*, given that "squares" are a type of clue and "regions" are the connected components outlined by the solution path and rectangle boundary.

Table 1 Our results for one-panel puzzles in The Witness: computational complexity with various sets of allowed clue types (marked by ✓). Allowed polyomino clues are either arbitrary (✓), or restricted to be monominoes (✓■), vertical dominoes (✓▮), or rotatable dominoes (✓▰).

broken edge	hexagon	square	star	triangle	polyomino	antipolyomino	antibody	complexity
▪▪	⬢	●	✸	▲▲	▞	▱	⅄	
✓								∈ L
✓	✓vertices							NP-complete
	✓vertices							*OPEN*
	✓edges							NP-complete
		✓1 color						∈ P
		✓2 colors						NP-complete
			✓1 color					*OPEN*
			✓n colors					NP-complete
				✓▲				NP-complete
✓				✓▲▲				NP-complete
				✓▲▲				*OPEN*
				✓▲▲▲				NP-complete
✓					✓■			*OPEN*
					✓■	✓□		NP-complete
					✓▰			NP-complete
					✓▮			NP-complete
✓	✓	✓	✓	✓	✓	✓		∈ NP
✓	✓	✓	✓	✓			✓n	∈ NP
					✓		✓2	Σ_2-complete
					✓	✓	✓1	Σ_2-complete
✓	✓	✓	✓	✓	✓	✓	✓n	$\in \Sigma_2$

applies to the *region* that contains that cell (referred to as *this* region), where we consider the regions of cells in the rectangle as decomposed by the (hypothetical) solution path and the rectangle boundary.

The solution path must satisfy *all* the constraints given by all the clues. (The meaning of this statement in the presence of antibodies is complicated; see Section 8.) Note, however, that if a region has no clue constraining it in a particular way, then it is free of any such constraints. For example, a region without polyomino or antipolyomino clues has no packing constraint.

As summarized in Table 1, we prove that most clue types *by themselves* are enough to obtain NP-hardness. The exceptions are broken edges, which alone just define a graph search problem; and vertex hexagons, which are related to Hamiltonian path in rectangular grid graphs as solved in [6] but remain open. But vertex hexagons are NP-hard when we also add broken edges. For squares, we determine that exactly two colors are needed for hardness. For stars, we do not know whether one or any constant number of colors are hard. For triangles, we know that 1-triangles or 3-triangles alone suffice for hardness, but for 2-triangles the only hardness proof we know needs broken edges. For polyominoes, monominoes alone are easy to solve [8], but monominoes plus antimonominoes are hard, as are rotatable dominoes by themselves and vertical nonrotatable dominoes by themselves. All problems without antibodies or without (anti)polyominoes are in NP. Antibodies combined with (anti)polyominoes push the complexity up to Σ_2-completeness, but no further.

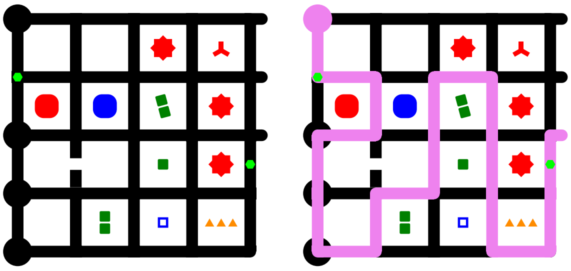


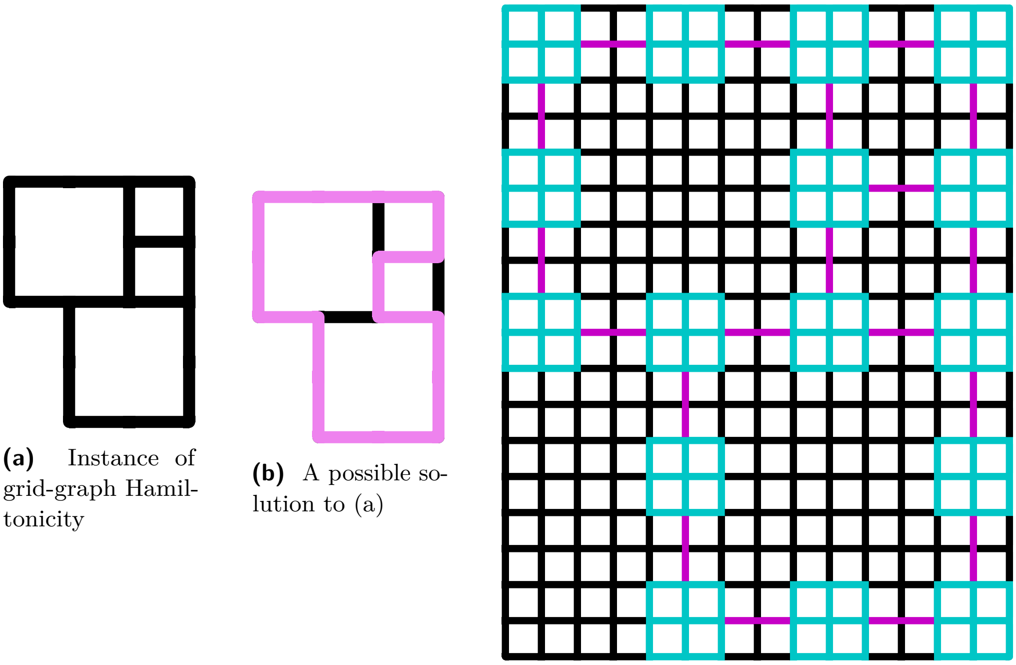
Figure 1 A small Witness puzzle featuring all clue types (left) and its solution (right). (Not from the actual video game.)

Table 2 Witness puzzle clue types and the definitions of their constraints.

clue	drawn on	symbol	constraint
broken edge	edge	▪ ▪	The solution path cannot include this edge.
hexagon	edge	⬢	The solution path must include this edge.
hexagon	vertex	⬢	The solution path must visit this vertex.
triangle	cell	▲▲	There are three kinds of triangle clues (▲, ▲▲, ▲▲▲). For a clue with i triangles, the path must include exactly i of the four edges surrounding this cell.
square	cell	■	A square clue has a color. This region must not have any squares of a color different from this clue.
star	cell	✸	A star clue has a color. This region must have exactly one other star, exactly one square, or exactly one antibody of the same color as this clue.
polyomino	cell	▞	A polyomino clue has a specified polyomino shape, and is either nonrotatable (if drawn orthogonally, like ▞) or rotatable by any multiple of 90° (if drawn at 15°, like ▞). Assuming no antipolyominoes, this region must be perfectly packable by the polyomino clues within this region.
antipolyomino	cell	□	Like polyomino clues, an antipolyomino clue has a specified polyomino shape and is either rotatable or not. For some $i \in \{0, 1\}$, each cell in this region must be coverable by exactly i layers, where polyominoes count as $+1$ layer and antipolyominoes count as -1 layer (and thus must overlap), with no positive or negative layers of coverage spilling outside this region.
antibody	cell	⅄	Effectively "erases" itself and another clue in this region. This clue also must be necessary, meaning that the solution path should not otherwise satisfy all the other clues. See Section 8 for details.

Table 3 Our results for metapuzzles in The Witness: computational complexity with various sets of environmental features.

features	complexity
sliding bridges	PSPACE-complete
elevators and ramps	PSPACE-complete
power cables and doors	PSPACE-complete



(a) Instance of grid-graph Hamiltonicity

(b) A possible solution to (a)

(c) Corresponding chambers and hallways

Figure 2 An example of the Hamiltonicity framework with $r = 1$ and $s = 4$.

Witness metapuzzles. We also consider some of the *metapuzzles* formed by the 3D environment in The Witness, which interact with the 2D single-panel puzzles. See Section 9 for details of these interaction models. Table 3 lists our metapuzzle results, which are all PSPACE-completeness proofs following the infrastructure of [2] (from FUN 2014).

2 Hamiltonicity Reduction Framework

We introduce a framework for proving NP-hardness of Witness puzzles by reduction from Hamiltonian cycle in a grid graph G of maximum degree 3. Roughly speaking, we scale G by a constant scale factor s, and replace each vertex by a block called a chamber; refer to Figure 2. Precisely, for each vertex v of G at coordinates (x, y), we construct a $2r + 1 \times 2r + 1$ subgrid of vertices $\{sx - r, \dots, sx + r\} \times \{sy - r, \dots, sy + r\}$, and all induced edges between them, called a *chamber* C_v. This construction requires $2r < s$ for chambers not to overlap. For each edge $e = \{v, w\}$ of G, we construct a straight path in the grid from sv to sw, and define the *hallway* $H_{v,w}$ to be the subpath connecting the boundaries of v's and w's chambers, which consists of $s - 2r$ edges. Figure 2 illustrates this construction on a sample graph G.

In each reduction, we define constraints to force the solution path to visit (some part of) each chamber at least once, to alternate between visiting chambers and traversing hallways that connect those chambers, and to traverse each hallway at most once. Because G has maximum degree 3, these constraints imply that each chamber is entered exactly once and exited exactly once. Next to one chamber on the boundary of G, called the *start/end chamber*, we place the start circle and end cap of the Witness puzzle. Thus any solution to the Witness puzzle induces a Hamiltonian cycle in G. To show that any Hamiltonian cycle in G induces a solution to the Witness puzzle, we simply need to show that a chamber can be traversed in each of the $\binom{3}{2}$ ways.

3 Hexagons and Broken Edges

Hexagons are placed on vertices or edges of the graph and require the path to pass through all of the hexagons. Broken edges are edges which cannot be included in the path. We show the positive result that puzzles with just broken edges are solvable in L, and the negative results that puzzles with just hexagons on edges are NP-complete and puzzles with just hexagons on vertices and broken edges are NP-complete. We leave open the question of puzzles with just hexagons on vertices (and no broken edges).

▶ **Lemma 1.** *Witness puzzles containing only broken edges, multiple start circles and multiple end caps are in L.*

Proof. We keep two pointers and a counter to track which pairs of starts and ends we have tried. For each start and end pair we run an (s, t) path existence algorithm, which is in L. If any of these return yes, the answer is yes. Thus we've solved the problem with a quadratic number of calls to a log-space algorithm, a constant number of pointers, and a counter, all of which only require logarithmic space. ◀

▶ **Lemma 2.** *It is NP-complete to solve Witness puzzles containing only broken edges and hexagons on vertices.*

Proof. Hamiltonian path in grid graphs is a strict subproblem. ◀

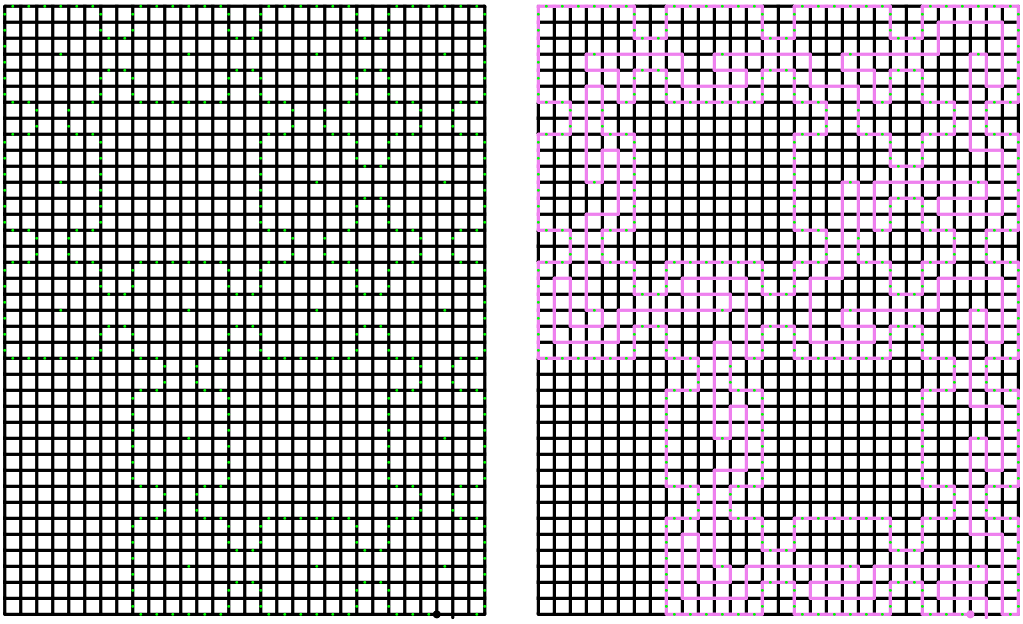
▶ **Theorem 3.** *It is NP-complete to solve Witness puzzles containing only hexagons on edges (and no broken edges).*

Proof sketch. We use the Hamiltonicity framework; refer to Figure 3. Noting that two edge hexagons incident to the same vertex must be consecutively traversed by the solution path, we carefully force the solution path to traverse the boundary of every chamber separate from the decision of which hallways to use. As with other Hamiltonicity framework reductions, we force each chamber to be visited with an edge hexagon in its center and can deduce the corresponding Hamiltonian cycle in the original grid graph from the set of used hallways. ◀

▶ **Open Problem 1.** *Is there a polynomial-time algorithm to solve Witness puzzles containing only hexagons on vertices?*

4 Squares

Each square clue has a color and is placed on a cell of the puzzle. Each region formed by the solution path and puzzle boundary must have at most one color of squares. If a puzzle has only a single color of squares, no non-trivial constraint is applied.



(a) Instance corresponding to Figure 2a

(b) Solution corresponding to Figure 2b

Figure 3 Example of the Hamiltonicity framework applied to Witness with edge hexagons.

4.1 Tree-Residue Vertex Breaking

Our reduction is from *tree-residue vertex breaking* [4]. Define *breaking* a vertex of degree d to be the operation of replacing that vertex with d vertices, each of degree 1, with the neighbors of the vertex becoming neighbors of these replacement vertices in a one-to-one way. The input to the tree-residue vertex breaking problem is a planar multigraph in which each vertex is labeled as "breakable" or "unbreakable". The goal is to determine whether there exists a subset of the breakable vertices such that breaking those vertices (and no others) results in the graph becoming a tree (i.e., destroying all cycles without losing connectivity). This problem is NP-hard even if all vertices are degree-4 breakable vertices or degree-3 unbreakable vertices[4].

4.2 Squares with Squares of Two Colors

▶ **Theorem 4.** *It is NP-complete to solve Witness puzzles containing only squares of two colors.*

Concurrent work [8] also proves this theorem. However, we prove this by showing that the stronger *Restricted Squares Problem* is also hard, which will be useful to reduce from in Section 5.

▶ **Problem 1** (Restricted Squares Problem). *An instance of the* Restricted Squares Problem *is a Witness puzzle containing only squares of two colors (red and blue), where each cell in the leftmost and rightmost columns, and each cell in the topmost or bottommost rows, contains a square clue; and of these square clues, exactly one is blue, and that square clue is not in a corner cell; and the start vertex and end cap are the two boundary vertices incident to that blue square; see Figure 4.*

▶ **Theorem 5.** *The Restricted Squares Problem is NP-complete.*

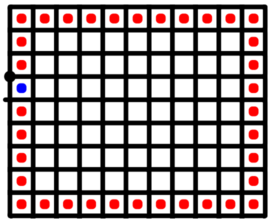
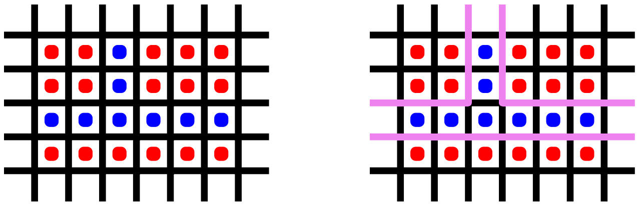
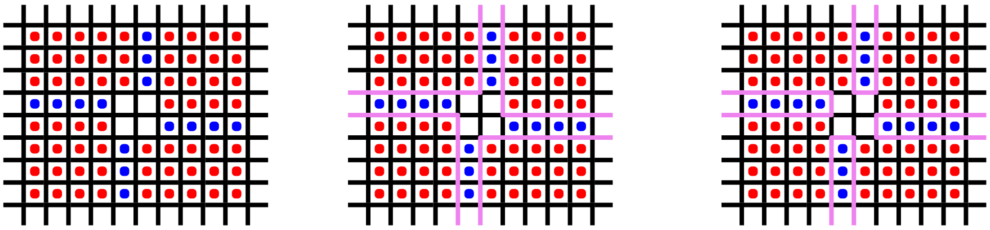


Figure 4 Boundary of the Restricted Squares Problem.



(a) Unsolved gadget. **(b)** The unique solution path.

Figure 5 Unbreakable degree-3 vertex gadget



(a) Unsolved gadget **(b)** Unbroken solution path. **(c)** Broken solution path.

Figure 6 Breakable degree-4 vertex gadget

Proof sketch. We reduce from tree-residue vertex breaking and construct gadgets for an unbreakeable degree 3 vertex (Figure 5) and a breakable degree 4 vertex (Figure 6) out of squares. We force the solution path to take an Euler tour of these gadgets, which can only be done if the underlying tree-residue vertex breaking graph is a tree. ◀

5 Stars

Star clues are in cells of a puzzle. If a region formed by the solution path and boundary of a puzzle has a star of a given color, then the number of clues (stars, squares, or antibodies) of that color in that region must be exactly two. A star imposes no constraint on clues with colors different from that of the star.

▶ **Theorem 6.** *It is NP-complete to solve Witness puzzles containing only stars (of arbitrarily many colors).*

Proof sketch. We reduce from the Restricted Squares Problem. For every square in the source instance, I, we use exactly one pair of stars of a distinct color corresponding to that square, as well as ten auxiliary colors. Figure 7 shows the high level structure of the reduction. A subrectangle, S, of the puzzle is designated for recreating I. For each pair of stars corresponding to a square, we place one of the two stars on the boundary of the puzzle, and the other in S in the same position as the corresponding square in I. The solution path will be forced to divide the overall puzzle into exactly two regions—an "inside" and an "outside"—such that all of the boundary stars corresponding to red squares are on the outside and all of the boundary stars corresponding to blue squares are on the inside. Then, inside of S, the solution path must ensure that all stars corresponding to red squares are in the outside region and all stars corresponding to blue squares are in the inside region, or else the star constraint will be violated. Then the solution path inside of S must correspond exactly to a solution path in I. ◀

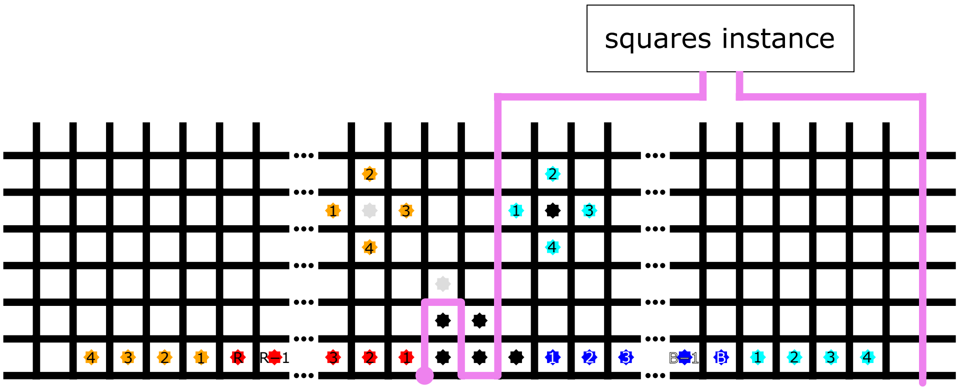


Figure 7 The boundary of the reduction. Each visual (color, number) pair represents a distinct color in the constructed instance. All stars depicted as blue correspond to blue squares in the source instance and must be in the inside region. Stars depicted as red correspond to red squares and must be in the outside region. The other stars enforce this.

Table 4 Summary of Slitherlink / Witness triangle constraints. New results are bold.

Clue types	*Complexity*
0	P [10]
1	**NP-complete [Theorem 7]**
2	*Open*
3	**NP-complete [Theorem 8]**
4	P [trivial]
0 and **2**	**NP-complete**

▶ **Open Problem 2.** *Is it NP-complete to solve Witness puzzles containing only a constant number of colors of stars?*

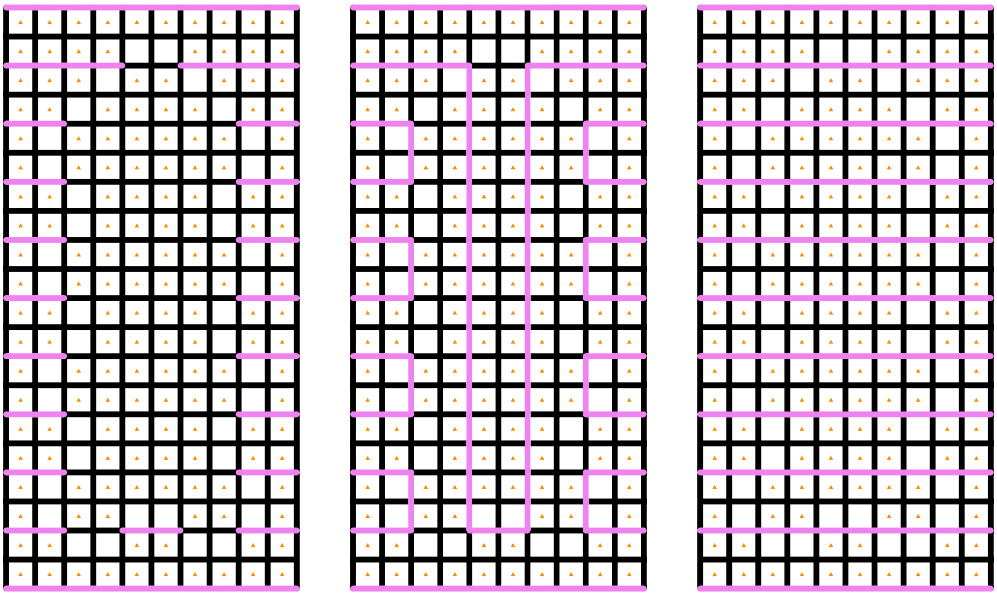
6 Triangles

Triangles are placed in cells. The number of solution path edges adjacent to that cell must match the number of triangles. This is similar to Slitherlink, which is known to be NP-complete [10]; however the proof in [10] relies critically on being able to force zero edges around a cell using 0-clues, which are not available in The Witness. We characterize all possibile combinations of constraints of these types for grid graphs. Table 4 summarizes what is known.

▶ **Open Problem 3.** *Is it NP-complete to solve Witness puzzles containing only 2-triangle clues (and no broken edges)?*

6.1 One Triangle Clues

Proving hardness of Witness puzzles containing only 1-triangle clues is made challenging by the fact that it is impossible to (locally) force turns on the interior of the puzzle. In particular, any rectangular interior region can be locally satisfied by a solution path which either traverses every second row of horizontal edges in the region or every second column of vertical edges in the region *regardless of the configuration of 1-triangle clues in the region.*



(a) Unsolved. **(b)** "All" or "active" solution. **(c)** "Nothing" or "inactive" solution.

Figure 8 All-or-nothing gadget.

Therefore, any local arguments we want to make about gadgets on the interior of the puzzle will need to admit the possibility of local solutions which are comprised of just horizontal or vertical paths straight through.

▶ **Theorem 7.** *It is NP-complete to solve Witness puzzles containing only 1-triangle clues.*

Proof sketch. We reduce from positive 1-in-3SAT, making use of the fact that the solution path must be a single closed path. We force the solution path to traverse all horizontal edges except for on the interior of gadgets, in which the solution path is allowed to connect adjacent horizontal path segments in a controlled manner (see Figure 8 for one key gadget), such that doing so corresponds to a solution to the source 1-in-3SAT instance. ◀

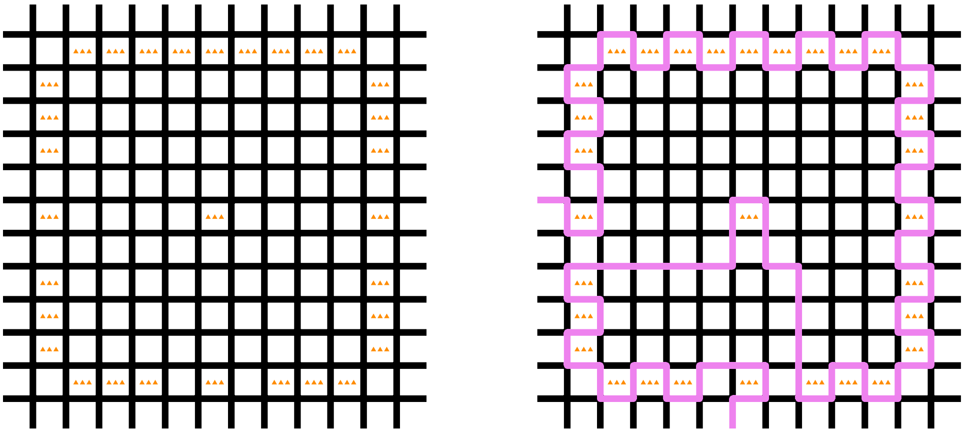
6.2 Three Triangle Clues

▶ **Theorem 8.** *It is NP-complete to solve Witness puzzles containing only 3-triangle clues.*

Proof sketch. We use the Hamiltonicity framework. Adjacent 3-triangle clues must be traversed consecutively by the solution path, so we can use them to for the solution path to trace the boundary of each chamber. Figure 9 shows the construction of a chamber. ◀

7 Polyominoes

This section covers various types of *polyomino* and *antipolyomino* clues. Polyomino clues can generally be characterized by the size and shape of the polyomino and whether or not they can be rotated (▮ vs. ▮). For each region, it must be possible to place all polyominoes and antipolyominoes depicted in that region's clues (not necessarily within the region) so that



(a) Unsolved.

(b) One possible solution, using the left and bottom edges.

Figure 9 A chamber with edges to the left, right, and below.

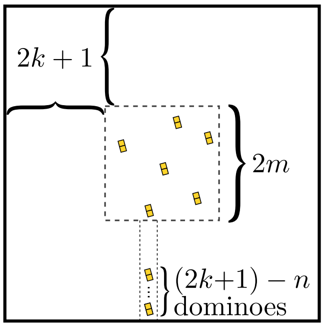


Figure 10 Overview of the rotatable dominoes NP-completeness proof.

for some $i \in \{0, 1\}$, each cell inside the region is covered by exactly i more polyomino than antipolyomino and each cell outside the region is covered by the same number of polyominoes and antipolyominoes. We give several negative results showing that some of the simplest (anti)polyomino clues suffice for NP-completeness.

Concurrent work [8] shows that Witness puzzles with squares of two colors for which every cell contains a square clue can be solved in polynomial time. Interestingly, such puzzles are equivalent to puzzles with only monominoes, by replacing one color of square with monominoes and the other color with blank cells. The only constraint on the two puzzle types is that there can be no region with a mix of square colors or, equivalently, monomino clues and blank cells. However, the question of whether puzzles with only monominoes and broken edges can be solved in polynomial time is still open.

7.1 Rotatable Dominoes

▶ **Theorem 9.** *It is NP-complete to solve Witness puzzles containing only rotatable dominoes.*

Proof sketch. We reduce from Rectilinear Steiner Tree: given n points with integer coordinates (x'_i, y'_i) in the plane, $i \in \{1, 2, \ldots, n\}$, and given an integer k, decide whether there exists a rectilinear tree connecting the n points having total length at most k. As illustrated in Figure 10, we embed the tree in the cells of a Witness puzzle, putting a domino clue at each vertex of the tree, which the solution path must therefore visit. The total number of dominoes is proportional to k, such that with careful counting, the area enclosed by the solution path must "look like" a tree of length exactly k in the original Steiner tree instance. ◀

7.2 Monominoes + Antimonominoes

▶ **Theorem 10.** *It is NP-complete to solve Witness puzzles containing only monominoes and antimonominoes.*

Proof sketch. The reduction is very similar to that of Theorem 9, except that the vertices of the Steiner tree contain antimonomino clues, and most of the other cells contain monomino clues. We force the solution path to partition the puzzle into two regions, an "outside" region which is entirely covered by monominoes, and an "inside" region which contains exactly as many antimonominoes as monominoes, thereby satisfying both. We show that doing this corresponds to a solution to the Steiner tree source instance. ◀

7.3 Nonrotatable Dominoes

▶ **Theorem 11.** *It is NP-complete to solve Witness puzzles containing only nonrotatable vertical dominoes.*

Proof sketch. We reduce from planar rectilinear monotone 3SAT [7]. Refer to Figure 11. We construct variable "wires" which are comprised of dominoes arranged on a diagonal which the solution path must enclose in one of two settings. Each clause needs to "connect" to at least one of its literals, but can only get close enough to do so if the corresponding variable is set appropriately. ◀

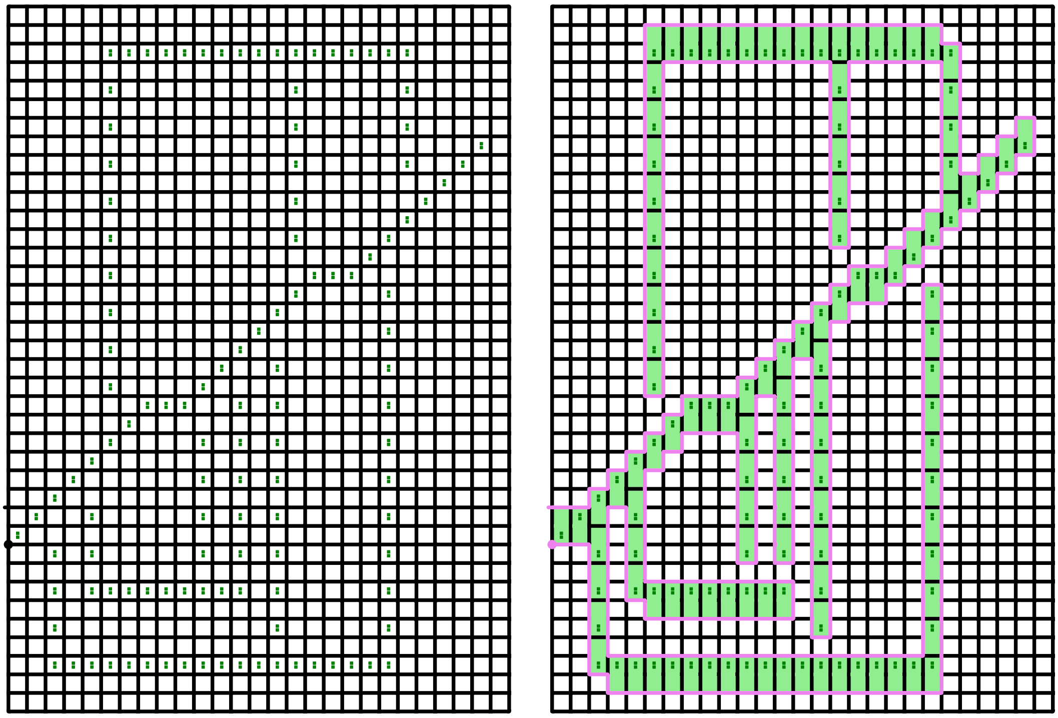
▶ **Open Problem 4.** *Is there a polynomial-time algorithm to solve Witness puzzles containing only monominoes and broken edges?*

8 Antibodies

An antibody (⅄) eliminates itself and one other clue in its region. For the antibody to be satisfied, this region must *not* be satisfied without eliminating a clue; that is, the antibody must be necessary. An antibody may be colored, but its color does not restrict which clues it can eliminate.[4] Very few Witness puzzles contain multiple antibodies, making the formal rules for the interactions between antibodies not fully determined by the in-game puzzles. We believe the following interpretation is a natural one: each antibody increments a count of clues that must necessarily be unsatisfied for their containing region to be satisfied. If there are k antibodies in a region, then there must be k clues which can be eliminated such that those k clues were unsatisfied and all other clues were satisfied; furthermore, there must not have been a set of fewer than k unsatisfied clues such that all other clues are satisfied[5]. Antibodies cannot eliminate other antibodies. The choice of clue to eliminate need not be

[4] Antibody color matters when checking if the antibody is necessary; a region containing only a star and an antibody of the same color is unsatisfied because the antibody is not necessary.

[5] Whether or not a clue is satisfied is usually determined only by the solution path; however, in the case of polyominoes and antipolyominoes, there might be several choices of packings which satisfy different sets of clues.



(a) The puzzle.

(b) The solution.

Figure 11 A Witness puzzle produced from $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg y)$ and its solution (x and y are FALSE, z is TRUE). Shaded cells show the domino tiling on the path's interior.

unique; for instance, a region with three white stars and one antibody is satisfied, even though the stars are not distinguished. Formally:

▶ **Definition 12** (Simultaneous Antibodies). A region with k antibody clues is satisfied if and only if there exists a set S of k non-antibody clues such that eliminating all clues in S and all k antibodies leaves the region satisfied, and there does *not* exist a set S' of non-antibody clues with $|S'| < k$ such that eliminating all clues in S' and only $|S'|$ of the antibodies leaves the region satisfied.

▶ **Theorem 13.** *Witness puzzles containing all clue types except polyominoes and antipolyominoes are in NP.*

Proof sketch. Other than antibodies, polyominoes, and antipolyominoes, whether or not a clue is satisfied can be easily determined from the solution path. Thus, checking whether an antibody which eliminates such a clue is necessary is easy. ◀

▶ **Theorem 14.** *Witness puzzles containing all clue types except antipolyominoes and for which at least one solution eliminates at most one polyomino in each region are in NP.*

Proof sketch. If at least one polyomino is eliminated in a region containing at least two polyominoes and the region is satisfied as a result, then the region can't be satisfied without deleting at least one polyomino because the total area of the polyominoes is greater than that of the region, and therefore there is no packing. ◀

▶ **Theorem 15.** *Witness puzzles containing any set of clue types (including polyominoes, antipolyominoes, and antibodies) are in Σ_2.*

Proof. Solving this Witness puzzle requires picking clues for antibodies to eliminate and finding a path which respects the remaining clues, such that the regions cannot be satisfied if only a subset of antibodies are used to eliminate clues. Membership in Σ_2 requires an algorithm which accepts only when there exists a certificate of validity for which there is no certificate of invalidity (i.e., one alternation of $\exists x \forall y$). A certificate of invalidity allows a polynomial-time algorithm to check whether an instance of a given problem is false. Our certificate of validity is a solution path, a mapping from antibodies to eliminated clues, and a packing witness for any region with at least one uneliminated polyomino. Our certificate of *invalidity* is the solution path (from the certificate of validity), a mapping of a *subset* of the antibodies to eliminated clues, and a packing witness for any region with at least one uneliminated polyomino.

Our verification algorithm begins checking the certificate of validity by verifying the packing witnesses and checking that the antibody mapping specifies distinct eliminated clues in the same region as each antibody. Then we remove all antibodies, polyomino and antipolyomino clues, and eliminated clues from the Witness puzzle and run the algorithm given in the proof of Theorem 13 to verify that the remaining clues in each region are satisfied under the solution path.

To verify the certificate of invalidity, we again check its packing witnesses and its (partial) antibody mapping. Then we remove the used antibodies, polyomino and antipolyomino clues, and eliminated clues from the Witness puzzle. We replace any unused colored antibodies with stars of their color if they are in the same region as an (uneliminated) star of that color, then remove any remaining antibodies. We run the polynomial-time algorithm given in the proof of Theorem 13 on the resulting Witness puzzle. Our algorithm accepts if and only if the certificate of validity is valid and all certificates of invalidity are invalid. ◀

Finally, we will show that Witness puzzles in general are Σ_2-complete. We will proceed in two steps, first considering puzzles which have two (or more) antibodies which might be eliminating polyominoes in the same region, and then considering puzzles which have only one antibody but both polyominoes and antipolyominoes. In both cases, we will reduce from *Adversarial-Boundary Edge-Matching*, a one-round two-player game defined as follows:

▶ **Problem 2** (Adversarial-Boundary Edge-Matching). *A* signed color *is a sign (+ or −) together with an element of a set C of colors. Two signed colors* match *if they have the same element of C and the opposite sign. A* tile *is a unit square with a signed color on each of its edges.*

An $n \times (2m)$ boundary-colored board *is an $n \times (2m)$ rectangle together with a signed color on each of the unit edges along its boundary. Given such a board and a multiset T of $2nm$ tiles, a* tiling *is a placement of the tiles at integer locations within the rectangle such that two adjacent tiles have matching colors along their shared edge, and a tile adjacent to the boundary has a matching color along the shared edge. There are two types of tiling according to whether tiles can only be translated or can also be rotated.*

The adversarial-boundary edge-matching game *is a one-round two-player game played on a $2n \times m$ boundary-colored board B and a multiset T of $2nm$ tiles. Name the unit edges along B's top boundary $e_0, e_1, \ldots, e_{2n}$ from left to right. During the first player's turn, for each even $i = 0, 2, 4, \ldots, 2n-2$, the first player chooses to leave alone or swap the signed colors on e_i and e_{i+1}. During the second player's turn, the second player attempts to tile the resulting boundary-colored board B' such that signed colors on coincident edges (whether on tiles or on the boundary of B') match. If the second player succeeds in tiling, the second player wins; otherwise, the first player wins.*

The adversarial-boundary edge-matching problem *is to decide whether the first player has a winning strategy for a given adversarial-boundary edge-matching game; that is, whether there exists a choice of top-boundary swaps such that there does* not *exist an edge-matching tiling of the resulting boundary-colored board.*

▶ **Lemma 16.** *Adversarial-boundary edge-matching is Σ_2-hard, with or without tile rotation, even when the first player has a losing strategy.*

Proof sketch. We reduce from from QSAT_2, which is the Σ_2-complete problem of deciding a Boolean statement of the form $\exists x_1 : \exists x_2 : \cdots : \exists x_n : \forall y_1 : \forall y_2 : \cdots : \forall y_n : f(x_1, x_2, \ldots, x_n; y_1, y_2, \ldots, y_n)$ where f is a Boolean formula using AND ($\wedge$), OR ($\vee$), and/or NOT ($\neg$). We convert this formula into a circuit, lay out the circuit on a square grid, and implement each circuit element as a set of tiles, one tile for each valid state (truth table row) of that element. The first player's boundary-edge swaps encode a setting of true or false for the first player's variables. Then, as part of solving the edge-matching problem, the second player must exhibit a setting of their variables that makes the formula false; otherwise the first player wins. ◀

▶ **Theorem 17.** *It is Σ_2-complete to solve Witness puzzles containing two antibodies and polyominoes.*

Proof. We reduce from adversarial-boundary edge-matching with the guarantee that the first player has a losing strategy. We create a Witness puzzle containing two antibodies. We will force the solution path to split the puzzle into two regions, with both antibodies in the same region and with part of the solution path encoding top-boundary swaps. In the construction, it will be easy to find a solution path satisfying all non-antibody clues when both antibodies are used to eliminate clues, but the antibodies themselves are only satisfied if they are necessary. When only one antibody is used, the remaining polyominoes in one of the regions, together with the solution path, simulate the adversarial-boundary edge-matching instance. The remaining polyominoes cannot pack the region (necessitating the second antibody and making the Witness solution valid) exactly when the adversarial-boundary edge-matching instance is a YES instance. (In the context of The Witness, the human player is the first player in an adversarial-boundary edge-matching game, and The Witness is the second player.)

Encoding signed colors. We encode signed colors on the edges of polyominoes in binary as unit-square tabs (for positive colors) or pockets (for negative colors) [3, Figure 7]. If the input adversarial-boundary edge-matching instance has c colors, we need $\lceil \log_2(c+1) \rceil$ bits to encode the color[6]. To prevent pockets at the corners of a tile from overlapping, we do not use the 2×2 squares at each corner to encode colors, so tiles are built out of squares with side length $w = \lceil \log_2(c+1) \rceil + 4$[7].

Clue sets. We consider the clues in the Witness puzzle to be grouped into two clue sets, A and B, which we place far apart on the board. We will argue that any valid solution path must partition the puzzle into two regions, such that each set is fully contained in one of the regions. Figure 12 shows (the intended packing of) most of the polyomino clues.

[6] We cannot use 0 as a color because we need at least one tab or pocket to determine the sign.

[7] At the cost of introducing disconnected polyomino clues, we could leave only one pixel at each corner out of the color encoding; that pixel is disconnected when the colors on its edges both have pockets next to it.

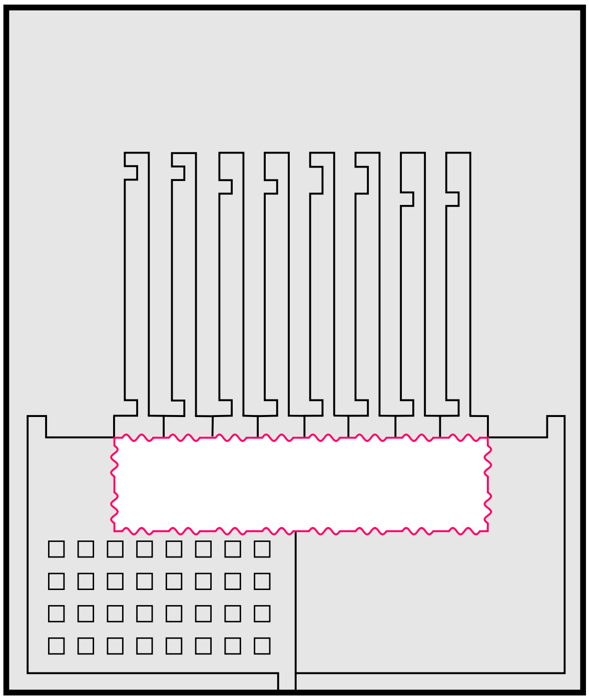


Figure 12 The intended packing of the puzzle after eliminating the medium polyomino (not to scale). The left and right board-frame polyominoes slot inside the large polyomino, and the monominoes fill the holes in the left board-frame polyomino. The stamps fill in their matching handle slots in the large polyomino, leaving only the boundary-colored board for the simulated adversarial-boundary edge-matching instance.

Clue set A contains:

- Two antibodies.
- $2nw - q$ monominoes, where q is the total number of pockets minus the total number of tabs across the "dies" of the "stamps" in clue set B (see below). There are $2n$ stamps each having up to $\lceil \log_2(c+1) \rceil$ tabs or pockets, so the total number of monominoes is between $2nw - 2n\lceil \log_2(c+1) \rceil = 8n$ and $2nw + 2n\lceil \log_2(c+1) \rceil = 4nw - 8n$ inclusive.
- A $w \times w$ square polyomino for each of the $2nm$ tiles in the adversarial-boundary edge-matching instance. The edges of each polyomino are modified with tabs and pockets encoding the signed colors on the corresponding edges of the corresponding tile. Call the upper-left corner of the $w \times w$ square the *key pixel* of that polyomino (even if tabs caused other pixels to be further up or to the left).
- A "medium" sized polyomino formed from a $2n(w+3) - 1 \times m(w+3) + 3$ rectangle polyomino; see Figure 13. Cut a hole out of this rectangle in the image of each tile polyomino, aligning the key pixel of each tile polyomino to a $2n \times m$ grid with upper-left point at the fourth row, second column of the rectangle and $w + 3$ intervals between rows and columns. Regardless of the pattern of tabs and pockets on each tile, this spacing ensures at least two rows of pixels above the top row of tile-shaped holes, at least one row on each other side, and at least one row between adjacent holes. Then add pixels above the upper-leftmost and upper-rightmost pixel of the rectangle (the *horns*) and below the middle-bottommost pixel of the rectangle (the *tail*). Finally, cut $2nw$ pixels out of the top row of the rectangle starting from the third pixel; this cutout is the *stamp accommodation zone.*

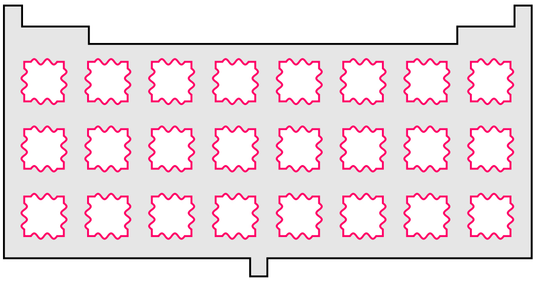


Figure 13 The medium polyomino, with boundary-colored holes matching each tile polyomino.

- Two *board-frame* polyominoes. Again, starting from a $2n(w+3)-1 \times m(w+3)+3$ rectangle polyomino, add horns and tail pixels in the same locations. Then cut out a $2nw \times mw$ rectangle whose upper-left pixel is the third pixel in the top row of the rectangle. The left, right and bottom edges of this cutout are modified with tabs and pockets encoding the signed colors on the corresponding sides of the boundary-colored board in the adversarial-boundary edge-matching instance. Split the polyomino vertically along the column of edges immediately to the right of the tail pixel.

  Finally, for each monomino in this clue set, cut a pixel out of the left board-frame polyomino, starting from the second-bottommost pixel in the second column, continuing across every other column, then continuing with the fourth-bottomost pixel in the second column, and so on. The left board-frame polyomino has width $nw+3n$, we cut pixels out of every other column, and we do not cut holes in its left or right columns, so we cut pixels out of $\frac{nw+3n-2}{2}$ columns. Below the mw-tall cutout and allowing two rows to ensure cut pixels do not join with pockets encoding signed colors along the edges of the cutout, we can cut pixels out of $\frac{3w-1}{2}$ rows (or $\frac{3w}{2}$, depending on parity). This allows up to $(\frac{nw+3n-2}{2})(\frac{3w-1}{2}) = \frac{n(w-4)^2+2w(nw-3)+13n+2}{4} + 4nw - 8n$ pixels to be cut out, but there are at most $4nw-8n$ monominoes, so we can always cut enough pixels without interfering with any other cuts.

Clue set B contains:

- A *stamp* polyomino for each of the $2n$ edge segments of the top edge of the boundary-colored board. Each stamp is composed of a $w \times 2$ rectangle modified to encode the signed color on the corresponding edge segment (called the *die*), a pixel centered above that rectangle, and a $2 \times h$ rectangular *handle* whose bottom-right pixel is immediately above that pixel, where $h = \max(m(w+3)+7, n)$. Stamps corresponding to 1-indexed edge segments $2i$ and $2i+1$ have pockets encoding i in binary cut into the left edge of their handle, starting from the second-to-top row of the handle.
- A "large" sized polyomino built from a $2n(w+3)+1 \times t$ rectangular polyomino, where t is the total area of all other polyominoes so far defined. Modify this polyomino by cutting out the middle pixel of the bottom row, the $2n(w+3)-1 \times m(w+3)+3$ horizontally-centered rectangle immediately above that removed pixel, and the pixels above the upper-left and upper-right removed pixels. (That is, cut out space for the medium polyomino, including the horns and tail but not including the stamp accommodation zone.) Then cut out the image of each stamp in the order of their corresponding edge segments in the adversarial-boundary edge-matching instance, aligning the leftmost-bottom pixel of the first stamp's die two pixels to the right of the upper-left removed pixel and aligning successive dies immediately adjacent to one another.

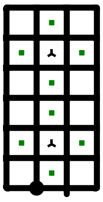


Figure 14 Because both antibodies are surrounded by monominoes, any region containing an antibody also contains at least one monomino.

Puzzle. The Witness puzzle is a $2n(w+3)+1 \times t$ rectangle. The start circle and end cap are at the middle two vertices of the bottom row of vertices.

Placement of A clues. We place a monomino from clue set A in the cell having the start circle and end cap as vertices, then place an antibody above that monomino, surrounded by a monomino in each of its other three neighbors. We then place the other antibody, surrounded by monominoes in its neighboring cells, three cells above the first antibody. (See Figure 14.) It is always possible to surround the antibodies in this way because there are at least $8n$ monominoes. We place the remaining clues from clue set A inside the $2n(w+3)-1 \times m(w+3)+3$ rectangle one row above the bottom of the puzzle; this is always possible because $|A| \leq 4nw - 8n + 2nm + 5$.

Placement of B clues. We place the large polyomino clue in the upper-left cell of the board and the stamp clues in the $2n$ cells to its right.

Argument. In any valid solution to the resulting puzzle, the large polyomino is not eliminated. If it were, it must be in the same region as an antibody. Because each antibody is surrounded by monomino clues, the number of polyomino clues in this region is strictly greater than the number of antibodies, so the region must be packed by the non-eliminated polyomino clues. The nearest (upper) antibody is $t-4$ columns and $nw+3n$ rows away from the large polyomino clue, so this region has area at least t. Recall that t is the total area of all polyomino clues except the large polyomino. If the large polyomino is eliminated, there is no way to pack this region, even if all other polyomino clues are used.

The large polyomino is as wide and as tall as the entire puzzle, so it has a unique placement. The large polyomino intersects its bounding box everywhere except one unit-length edge aligned with the start vertex and end cap, so any valid solution path can only touch the boundary at the start and end. Thus the solution path divides the puzzle into at most two regions (an inside and an outside).

Suppose the solution path places the entire puzzle into a single region; that is, suppose the solution path proceeds (in either direction) from the start vertex to the end cap without leaving the boundary. Then by the assumption that the first player has a losing strategy in the input adversarial-boundary edge-matching instance, we can pack the region while eliminating only one clue. The large polyomino's placement is fixed. We eliminate the medium polyomino, place the two board-frame polyominoes inside the large polyomino, and place the monominoes in the pixels cut out of the left board-frame polyomino. It remains to place the stamps and tiles. By the assumption, there is a losing set of top-boundary swaps; we swap the corresponding pairs of stamps when placing them into the cutouts in the large polyomino, and then place the tiles in the remaining uncovered area bordered by the board-frame polyominoes and stamp dies. Because we satisfied all non-antibody constraints after eliminating only one clue, the unused antibody is unsatisfied, so any solution path resulting in a single region is not a valid solution to the puzzle. Thus there are exactly two regions.

The cells containing the stamp clues are covered by the large polyomino, so any valid solution places the stamps in the same region as the large polyomino. The handles of the stamps are taller than the cutout in the bottom-middle of the large polyomino, so they must instead be placed in the stamp-shaped cutouts in the large polyomino. The pockets cut into the left edges of the handles ensure that stamps can only swap places corresponding to top-boundary swaps in the adversarial-boundary edge-matching instance.

All clues in set A are in the other region. The monomino clue in the cell having both the start circle and end cap as vertices cannot be in the same region as the large polyomino (else the path could not divide the puzzle into two regions). Because each antibody is surrounded by monomino clues, the number of polyomino clues in this region is strictly greater than the number of antibodies, so the region must be packed by the non-eliminated polyomino clues. When both antibodies are used to eliminate clues, they must eliminate both board-frame polyominoes, and when only one is used, it must eliminate the medium polyomino; any other elimination leaves polyomino clues with too much or too little area to pack the area of the puzzle not yet covered by the large polyomino or the stamps. Thus either the medium polyomino or both board-frame polyominoes will not be eliminated. The medium polyomino and board-frame polyominoes have unique placements within the large polyomino determined by the horns and tail. The intersection of the outlines of these placements covers all the A clues, so they are all in the same other region.

By this division of the clues into regions, any valid solution path traces the inner boundary of the large polyomino and the dies of the stamps (possibly after swapping some pairs). It remains to show that the solution path is valid exactly when the implied set of top-boundary swaps is a winning strategy in the adversarial-boundary edge-matching instance.

When using both antibodies to eliminate the board-frame polyominoes, the remaining polyominoes always pack their region. The medium polyomino's placement is fixed by the horns and tail; the stamp accommodation zone ensures this placement is legal regardless of the pattern of tabs on the dies of the stamps. The tile polyominoes fit directly into the cutouts in the medium polyomino and there are exactly enough monominoes to fill in the uncovered area in the stamp accommodation zone and the pockets of the dies.

The solution path is only valid if both antibodies are necessary. When using one antibody to eliminate the medium polyomino, the board-frame polyominoes' position is forced by the horns and tail. The monominoes are the only way to fill the single-pixel holes in the left board-frame polyomino and there are exactly enough monominoes to do so. Then the dies of the stamps and the edges of the rectangular cutout in the board-frame polyominoes models the boundary-colored board of the input adversarial-boundary edge-matching instance (see Figure 12). The tile polyominoes cannot pack this area, necessitating the second antibody and making the solution path valid, exactly when the set of top-boundary swaps is a winning strategy in the adversarial-boundary edge-matching instance. ◀

▶ **Theorem 18.** *It is Σ_2-complete to solve Witness puzzles containing one antibody, polyominoes and antipolyominoes.*

Proof sketch. As in the proof of Theorem 17, we reduce from adversarial-boundary edge-matching, and the reduction is similar. The primary difference is that the medium polyomino is also the singular board-frame polyomino. Besides the antibody and the tile polyominoes (same as before), clue set A contains an antipolyomino called the *antikit* shaped like a 1-

pixel-wide tree with the tile polyominoes (as antipolyominoes) at the leaves and a polyomino shaped like the 1-pixel-wide tree (the *sprue*). The medium polyomino has the kit polyomino attached to its right side and a cutout for the sprue and for the boundary-colored board.

The stamps must be placed in the large polyomino as in the previous proof. When the antibody eliminates the medium polyomino, the antikit annihilates the sprue and tile polyominoes, leaving no (anti)polyominoes in the inner region (so it is trivially satisfied). When the antibody is not used, the antikit annihilates the kit-shaped part of the medium polyomino and the sprue fits in the cutout in the medium polyomino, leaving only a boundary-colored board for the tile polyominoes to be placed. Placing the tile polyominoes is impossible, necessitating the antibody and making the solution path valid, exactly when the top-boundary swaps are a winning strategy in the adversarial-boundary edge-matching instance. ◀

By Theorem 14, Theorem 17 and Theorem 18 are tight.

9 Metapuzzles

In this section, we analyze several of the *metapuzzles* that appear in The Witness. Metapuzzles are puzzles which have one or more puzzle panels as a sub-component of the puzzle, and in which solving the puzzle panel affects the surrounding world in a way that depends on the choice of solution that was used to solve the panel.

9.1 Sliding Bridges

The marsh area contains sliding bridges. In this metapuzzle, each bridge has a corresponding puzzle panel, and solving the puzzle causes the bridge to move into the position depicted by the outline of the solution path. The following theorem demonstrates that, regardless of the difficulty of the puzzle panels (i.e., even if it is easy to find all solutions of each individual panel), it is PSPACE-complete to solve sliding bridge metapuzzles.

▶ **Theorem 19.** *It is PSPACE-complete to solve Witness metapuzzles containing sliding bridges.*

Proof sketch. We straightforwardly construct the *one-way* and *door* gadgets of [2], which are known to be sufficient for PSPACE-completeness. ◀

9.2 Elevators and Ramps

Another metapuzzle which appears in The Witness consists of groups of platforms that move vertically at one or both ends to form an elevator or ramp, controlled by the path drawn on puzzle panels. Because the player cannot jump or fall in The Witness, the player can walk onto an elevator platform only if it is at the same height as the player. The player can adjust the height of the platforms from anywhere with line-of-sight to the controlling panel, including while on the platforms themselves. Besides the sawmill, the other building in the quarry contains a ramp and an elevator. The marsh contains a single puzzle with a 3×3 grid of elevators controlled by two identical panels; as a metapuzzle, our puzzle could be built out of multiple marsh puzzles with two platforms and one panel each.

▶ **Theorem 20.** *It is PSPACE-complete to solve Witness metapuzzles containing elevator reconfiguration, even when each panel controls at most one elevator.*

Proof sketch. We construct *one-way* and *door* gadgets similar to Theorem 19. ◀

9.3 Power Cables and Doors

In the introductory area of The Witness, there are panels with two solutions, each of which activates a power cable. Activated cables can power one other panel (allowing it to the solved) or one door (opening it). If a cable connected to a door is depowered, the door closes. Cables cannot be split and panels can power at most one cable at a time.

▶ **Theorem 21.** *It is PSPACE-complete to solve Witness metapuzzles containing power cables and doors.*

Proof sketch. Again we construct *one-way* and *door* gadgets, with the slight complication that all powered doors in The Witness are initially closed, so we need to give the player a way to open exactly the set of doors which are initially open in the source instance. ◀

References

1 Zachary Abel, Jeffrey Bosboom, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, and Mikhail Rudoy. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. arXiv:1804.10193, 2018.

2 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.

3 Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics*, 23:195–208, June 2007.

4 Erik D. Demaine and Mikhail Rudoy. Tree-residue vertex-breaking: a new tool for proving hardness. In *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm Theory*, 2018. arXiv:1706.07900.

5 Linus Hamilton. Braid is undecidable. arXiv:1412.0784, 2014.

6 Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, November 1982.

7 Donald E. Knuth and Arvind Raghunathan. The problem of compatible representatives. *SIAM Journal on Discrete Mathematics*, 5(3):422–427, 1992.

8 Irina Kostitsyna, Maarten Löffler, Max Sondag, Willem Sonke, and Jules Wulms. The hardness of Witness puzzles. In *Abstracts from the 34th European Workshop on Computational Geometry*, 2018.

9 Wikipedia. The Witness (2016 video game). https://en.wikipedia.org/wiki/The_Witness_(2016_video_game), 2018.

10 Takayuki Yato. On the NP-completeness of the Slither Link puzzle (in Japanese). *IPSJ SIG Notes*, AL-74:25–32, 2000.

Tracks from hell – when finding a proof may be easier than checking it

Matteo Almanza
Dipartimento di Informatica, Sapienza Università di Roma, Italy.
almanza.1597415@studenti.uniroma1.it

Stefano Leucci
Institute of Theoretical Computer Science, ETH Zürich, Switzerland.
stefano.leucci@inf.ethz.ch
https://orcid.org/0000-0002-8848-7006

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

Abstract

We consider the popular smartphone game Trainyard: a puzzle game that requires the player to lay down tracks in order to route colored trains from departure stations to suitable arrival stations. While it is already known [Almanza et al., FUN 2016] that the problem of finding a solution to a given Trainyard instance (i.e., game level) is NP-hard, determining the computational complexity of *checking* whether a candidate solution (i.e., a track layout) solves the level was left as an open problem. In this paper we prove that this verification problem is PSPACE-complete, thus implying that Trainyard players might not only have a hard time finding solutions to a given level, but they might even be unable to efficiently recognize them.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases puzzle games, solitaire games, Trainyard, verification

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.4

1 Introduction

The relationship between the Catholic Church and science and technology in the course of history has been a long and complex one. On the one hand, there are dark and immensely sad episodes like the condemnation of Galileo Galilei. On the other, one must acknowledge the fantastic contribution to science and mathematics throughout the centuries at the hands of Catholic clergymen, an all star team that includes the likes of Gregor Mendel, Francis Bacon, Nicolaus Copernicus, and Bernard Bolzano, to name just a few.

Perhaps nothing better than the history of the railways illustrates the ambivalent relationship of the Catholic Church toward science and technology. Early adoption within the Papal States of this revolutionary means of transportation was stymied by Pope Gregory XVI (1765-1846) who famously warned *"Chemins de fer, chemin d'enfer"* ("road of iron, road of hell") [4, p. 164]. His successor Pious IX however, realized the potential of railway transportation for the purposes of the Holy See. The roads of iron could not only lead to hell, but also to holy places like the sanctuary of Lourdes. The steam engine became a facilitator of mass pilgrimages. The all powerful Roman Curia finally gave in on October 2, 1934, when the stately Vatican City Central Station opened [14, p. 653]. This imposing building has been to this day the headquarters of the smallest railway system in the world—300 meters of

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 4; pp. 4:1–4:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

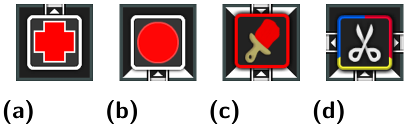


Figure 1 Different types of tiles: (a) a red departure station, (b) a red arrival station, (c) a red painter, (d) a splitter.

tracks in total!—an apparent contradiction that epitomizes an ambivalent attitude toward a profound dilemma.

One must wonder why such a potentially useful and, from the point of view of Catholic orthodoxy, apparently innocuous technology was met with such a high degree of suspicion. The answer, it turns out, is hidden in the odd meanderings of the computational complexity of games. The puzzle game Trainyard beautifully captures the inherent tension between two moral imperatives, finding the right path and making sure that the path taken is indeed the virtuous one.

In a landmark paper, Almanza et al. proved that finding the layout of a railway network in Trainyard—a computational task easily seen to be in PSPACE—is NP-hard [2]. They left open the question of verification, namely the computational complexity of checking whether a given track layout delivers all trains safely to destination. In this paper we show that, surprisingly, this task is PSPACE-complete. Interestingly, it is still possible that Trainyard lies in NP. If this were the case then checking certificates for Trainyard would be more difficult than finding them, unless NP = PSPACE. In moral terms, walking along the path of virtue could prove impossibly arduous, even when you have found it. This is a bizarre state of affairs that vindicates the cautious approach of the Roman Curia toward the railway question.

2 Problem Definition

In a Trainyard level we are given a rectangular board in which each of the cells is either empty or occupied by a *tile*. The two main tile types are departure and arrival stations: a departure station (shown in Figure 1 (a)) hosts a single train, initially colored either blue or red, while an arrival station (shown in Figure 1 (b)) accepts one train of a given color. The player's task is to route trains from departure stations to arrival stations by placing (possibly rotated versions of) the *rail pieces* of Figure 2 into empty cells. The rail pieces of Figure 2 (a)–(d) are traversed by trains in the straightforward way, while the pieces of Figure 2 (e) and (f) are called *switches* and route trains going in the bottom-to-top direction towards two neighboring tiles in an alternating fashion, i.e., each transiting train flips the *state* of the switch. Trains traveling in the opposite directions are routed to the tile immediately below the *switch* but they still affect the switch's state.

To further complicate[1] things, the grid also contains other special tiles that interact with incoming trains in various ways (see Figure 1 (c) and (d)), namely:

Painters: trains traversing this tile will acquire the color of the painter tile itself. A painter can be either red or blue. Note that a painter gadget has only two entry points located on opposite sides of the cell.

[1] Actually, the original game is even more complex that the one described in the present work. The subset of rules and tiles described here, however, suffices for the purposes of our reduction.

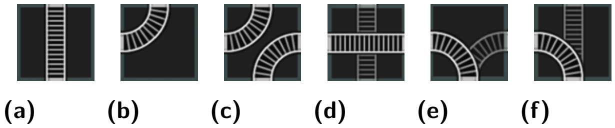


Figure 2 Different kinds of rail pieces. Each of the pieces can be rotated by 90, 180 or 270 degrees.

Splitters: 1 splitters act both on the number of trains and on their color. A splitter has only a single one-way entry gate, and two one-way exit gates located at the left and right side of the cell. When a train enters a splitter, it vanishes but two new trains are created and exit from the sides. If the incoming train was red or blue, then the outgoing trains will also have that same color. If the incoming train was purple,[2] then the new train exiting from the left side will be blue, while the other outgoing trail will be red.

After all the rail pieces have been placed by the player, the design is put to the test: trains exit from departure stations simultaneously and they travel on rails at a speed of one tile per second. When a train moves from a tile to the next, the directions of the involved rails must match. If this does not happen then the train will *crash* and the player loses. Moreover, whenever two trains simultaneously occupy the same rail piece while traveling in the same direction, they merge into a single train. The color of this resulting train depends on the color of the two merged trains. We are only interested in the following cases: if the merged trains had the same color, then the new train will also retain that color; if one merged train was blue and the other was red, then the resulting train will be purple. Two trains going in different directions can also occupy the same tile at the same time: in this case no merge occurs but the color of both trains is still changed according to the previous rules.

The player wins iff this process eventually reaches a state in which there are no traveling trains, each arrival station has received exactly one train of the associated color, and no crash has occurred.

We study the computational complexity of the Trainyard-Verification problem, i.e., the problem of deciding whether a solution (i.e., a placement of the rail pieces) to a Trainyard level results in a win for the player (see Figure 3). In some sense we investigate how hard it is for the player to recognize a correct solution to a Trainyard level. Unfortunately for the player, this tasks can not be performed efficiently, unless PSPACE = P. Indeed, we are able to show the following:

▶ **Theorem 1.** Trainyard-Verification *is* PSPACE-*complete.*

3 Other related works

Trainyard belongs to the broad class of casual games: these games are characterized by an intuitive gameplay which is usually organized into a series of small puzzles of increasing difficulty. Interestingly enough, many casual games are *hard* to solve, not only for human players, but also for machines: it is often the case that the computational problem of finding a solution to a given level (instance) of a casual game turns out to be at least NP-hard. Notable examples include e.g., Candy Crush [8], 2048 [12], Flow-Free [1], Sokoban [3], Rush

[2] Purple trains can appear when a red and a blue train meet, as we explain in the following.

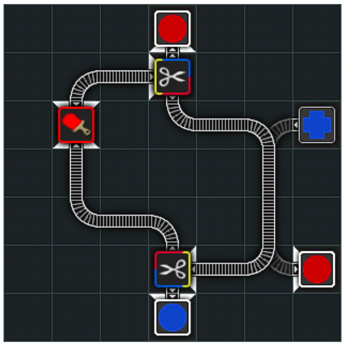


Figure 3 A correct solution to a Trainyard level, i.e., a yes-instance of TRAINYARD-VERIFICATION.

Hour [6], Two-Dots [13], or Peg-Solitaire [15, 9] and it might even be the case that the success of these games is due, in part, to their challenging levels, as suggested in [5]. For a discussion of other NP-hard and PSPACE-hard puzzles and of general tecniques for showing their hardness, we refer the interested reader to [11] and [10].

4 Our Reduction

4.1 Preliminaries

Our reduction is from the *Iterated Monotone Boolean Circuit* problem (IT-MON-BC for short) [7]. A monotone boolean circuit C is a directed acyclic graph whose non-source vertices are labeled with either $\wedge$ ("*and*") or $\vee$ ("*or*"). The source vertices are called *input-vertices*, while all the other vertices are called *gates*. Gates that are also sinks in G are called *output-vertices*. Let $v^I_1, \dots, v^I_n$ (resp. $v^O_1, \dots, v^O_m$) be the input-vertices (resp. output-vertices) of C, in some order. Once a boolean value is associated to each input vertex, it is possible to *evaluate* the circuit by computing a boolean value for each output-vertex. This is done by propagating the input values from the sources towards the sinks, i.e., by considering the gates of C in preorder and assigning a truth value to them depending on their label and on the truth value of their in-neighbors. If $\overline{x} = \langle x_1, \dots, x_n \rangle$ is the boolean vector containing the input values (where x_i is the initial truth value for vertex v^I_i), then the evaluation of C computes a vector $\overline{y} = f_C(\overline{x})$, where f_C is the function from $\{\texttt{True}, \texttt{False}\}^n$ to $\{\texttt{True}, \texttt{False}\}^m$ *implemented* by the circuit and the i-th entry y_i of $\overline{y}$ is the truth value corresponding to vertex v^O_i. Without loss of generality we assume that all the gates of C have in-degree 2. If $n = m$ then we can also compute

$$f^t_C(\overline{x}) = (\underbrace{f_C \circ \cdots \circ f_C}_{t \text{ times}})(\overline{x})$$

by evaluating C t times, where the input of i-th evaluation, for $i > 1$, consists of output of the $i-1$-th evaluation.

The IT-MON-BC problem asks, given a monotone boolean circuit with $n = m$, an input vector x, and an index h, to determine whether there exists a positive integer t such that $f^t_C(\overline{x})_h = \texttt{True}$, i.e., if iterating C with a $\overline{x}$ as the initial input eventually causes the h-th output to become $\texttt{True}$.

▶ **Theorem 2** (Lemma 3 in [7]). IT-MON-BC *is* PSPACE*-complete even when restricted to circuits of in-degree* 2 *and out-degree* 2*.*

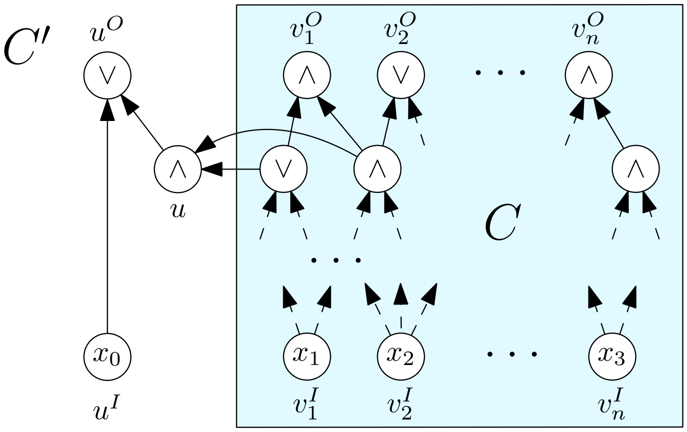


Figure 4 Transformation of the circuit of an It-Mon-BC instance $\langle C, \overline{x} \rangle$ into an equivalent circuit C' having one additional input/output vertex pair (namely, u^I and u^O). Initially, the truth value of the new variable x_0 associated with u^I is `False`. The output value associated to u^O becomes `True` as soon as the value of v_1^O is `True` and remains `True` in all subsequent iterations.

Notice that, since there can be at most 2^n distinct input vectors, if $(f_C^t(\overline{x}))_h = \texttt{False}$ for each $t = 1, \dots, 2^n$ then, by the pigeonhole principle, we can conclude that the answer to an instance of It-Mon-BC is false. For technical convenience we assume w.l.o.g. that $h = 1$ and that, once the first output becomes `True`, it will remain `True` in all the subsequent iterations of the circuit. Notice that this latter assumption is not restrictive and can be removed by transforming the circuit C into an equivalent circuit C' having the desired property as follows (see also Figure 4): initially C' is a copy of C, then we add to C' (i) a new input-vertex u^I, (ii) a new gate u having the same label as v_1^O in C, and (iii) a new output-vertex u^O having label $\vee$; then, for each each edge (v, v_1^O) in C, we add a corresponding edge (v, u) to C', and finally we add the two edges (u^I, u_O) and (u, u_O) to C'. It is easy to check that, for every input vector $\overline{x}$ and any $x_0 \in \{\texttt{True}, \texttt{False}\}$, the truth value of the vertices $v_1^O, \dots, v_n^O$ of C with input $\overline{x}$ coincides with the truth value of the corresponding vertices of C' with input $\langle x_0, x_1, \dots, x_n \rangle$ where x_0 is the value initially assigned to u^I. Moreover, as soon as v_1^O become `True`, u_O will also become `True` and will retain the `True` value in all subsequent iterations. Hence, we have obtained a circuit C' having the desired property, modulo a renaming of its input/output vertices.

4.2 Overview

In the following we show how to convert, in polynomial time, an instance $\langle C, x, h \rangle$ of It-Mon-BC into a instance of Trainyard-Verification that is a valid solution iff $(f_C^{2^n}(\overline{x}))_h = \texttt{True}$, thus establishing the PSPACE-hardness of Trainyard-Verification. Notice that Trainyard-Verification clearly belongs to PSPACE as simulating a solution only requires polynomial space. Moreover, since the number of possible states that can occur during the simulation is at most exponential in the instance's size, there exists an upper limit T_0 to the number of simulation steps needed: if the Trainyard level is still not solved after T_0 steps, then the simulation must be stuck in some cyclic sequence of states, hence the Trainyard-Verification instance has a "no" answer.

A high-level picture of our reduction is shown in Figure 5. The initial input $\overline{x}$ of the circuit C is encoded in the color of n departing trains in the "input area", if $x_i = \texttt{True}$ (e.g., x_2 in Figure 5) then the i-th departure station is red, while if $x_i = \texttt{False}$ (e.g., x_1 and x_3 in Figure 5) the corresponding departure station is blue. The departing trains (moving in the

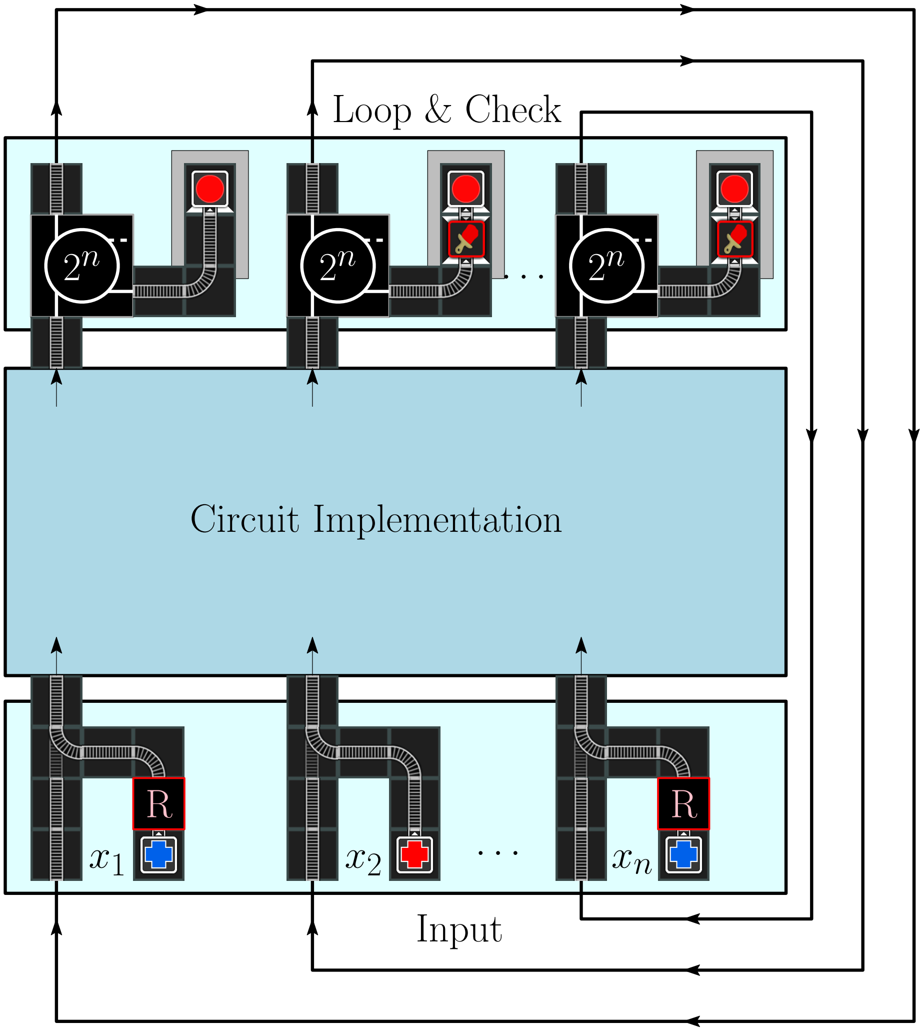
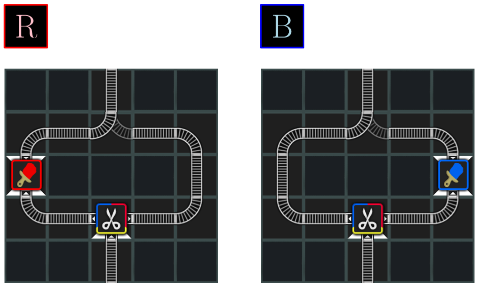


Figure 5 Overview of our reduction from It-Mon-BC to Trainyard-Verification. The input trains departing from the stations in the "Input" region traverse the "Circuit Implementation" area exactly 2^n times (using the outer loops to return to the bottom) and are then routed to the gray *check* gadgets in the "Loop and Check" area.

bottom-to-top direction) then enter the "Circuit Implementation" area where they traverse a series of gadgets that simulate the gates of the circuit C. Eventually, exactly n trains exit from the "Circuit Implementation" area, the i-th of which encodes (in its color) the truth value of the i-th output of the circuit. These trains are now routed to the one final region of our Trainyard level, namely the "Loop and Check" area. This region contains n *loop* gadgets that act *transparently* for the first $2^n - 1$ times they are traversed, i.e., they let any train entering from the bottom exit unaltered from the top. These outgoing trains are then re-routed back into the inputs of the circuit and the whole process repeats. At the 2^n-th iteration, the loop gadgets stop acting transparently and instead divert the incoming trains to the rails exiting from their right. Here each train enters a *check* gadget (shown in gray in Figure 5) containing one arrival station. Collectively, these check gadgets allow the level to be completed if and only if the train corresponding to the first output of the circuit (which represents the first entry of $f_C^{2^n}$) encodes the value `True`.



(a) (b)

Figure 6 (a) Implementation of Red-Shift gadget. (b) Implementation of Blue-Shift gadget. These gadgets convert the color scheme of a train from CS2 to CS1 and vice-versa, respectively.

An interactive demonstration of our reduction is available at: `https://trainyard.isnphard.com/verification/`

4.3 Gadgets and Color Schemes

We encode the gates of C using a combination of *gadgets*, i.e., combinations of rails and special tiles, and the links of C using rails between the two corresponding gadgets. Gadgets will receive some trains as inputs (usually from the bottom), will perform some operations, and will let the result trains exit (usually from the top). We assume that all the input trains enter a gadget simultaneously: since a train always takes the same number of steps to traverse a gadget[3], this property can be guaranteed by choosing suitable lengths for the rails connecting two consecutive gadgets.

The evaluation of C will be simulated by trains that will carry truth values from one gadget to the other. Those truth values will be encoded using train colors according to two different color schemes.

CS1: A red train represents the value `True`, a purple train represents the value `False`. Blue trains are not allowed.

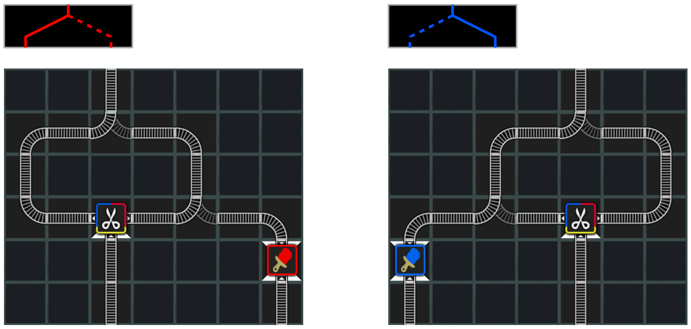
CS2: A purple train represents the value `True`, a blue train represents the value `False`. Red trains are not allowed.

4.3.1 Converting between color schemes

Here we show how to change the color of a train that is carrying a truth value encoded using CS1 to the correct encoding according to CS2, and vice-versa.

The conversion from CS2 to CS1 is performed by the *Red-Shift* gadget shown in Figure 6 (a). The input train enters the gadget from the rail on the bottom and is split into two trains exiting from the sides of the splitter tile. If the incoming train is blue then these two trains will be blue as well, the one on the left side will pass through a red painter tile and will merge back with the blue train coming from the right side into a purple train. If incoming train is purple then the train exiting the spliter tile from the left side will be blue while the one exiting from the right will be red. Due to the red painter gadget, the left train will also become red, and the two trains will merge thus causing a single red train to exit from the top of the gadget. Notice also that if the input of a Red-Shift gadget is a red train, then the train will retain its red color. This property will be useful in the sequel.

[3] with the exception of loop gadgets, as will be discussed in the following.



(a) (b)

Figure 7 Implementation of red (a) and blue (b) absorption gadgets.

The conversion from CS2 to CS1 is done by the *Blue-Shift* gadget shown in Figure 6 (b), whose operation is symmetric to the Red-Shift gadget. Notice again that blue trains retain their colors when traversing a *Red-Shift* gadget.

These two gadgets allow us to change the color of each train entering a gadget in order to meet the expected color scheme. We can hence assume that all the appropriate color conversions happen on the railways connecting the output of a gadget to the input of next one. This will be particularly useful in implementing gadgets for “and” and “or” gates which require the two input trains to be colored according to different colors schemes. As an example notice how the blue trains that will exit the departure stations of Figure 5 (i.e., the stations corresponding to `False` inputs) immediately traverse a Red-Shift gadget. This ensures that all the trains leaving the “Input” area will be colored according to CS1.

4.3.2 Train Absorption Gadget

Some of the gadgets will generate additional trains as a side effect of their operation. In order to get rid of these spurious trains we would like them to merge with a train that we call a *main* train. This operation should be performed with care to ensure that the color of the main train is preserved. This can be done by using the *absorption gadgets* shown in Figure 7.

Let us focus on the *red absorption gadget* of Figure 7 (a) which shows how a main train colored according to CS1 and entering the gadget from the leftmost rail on the bottom can be merged with any other train entering from the rightmost rail. Once the main train reaches the splitter, the train exiting from the right side will always be red (recall that, according to CS1, the main train must be either red or purple). We can therefore safely merge it with the spurious train, which has been colored red by the red painter tile, to obtain a single red train. This red train is then merged back with the train exiting the left side of the splitter, resulting in a single train of the same color of the incoming main train.

The *blue absorption gadget* gadget of Figure 7 (b) merges a main train colored according to CS2 with any other train (entering from the leftmost rail on the bottom) and its operation is analogous the the red absorption gadget just described.

4.3.3 Crossover Gadget

The *crossover* gadget allows two trains entering (at the same time) from the bottom rails to exit (at the same time) from the top two rails in the opposite order. Its implementation is straightforward and it is shown in Figure 8 (a). The turns in the gadget ensure that the two input trains do not meet when the rails cross (as this might alter their original colors).

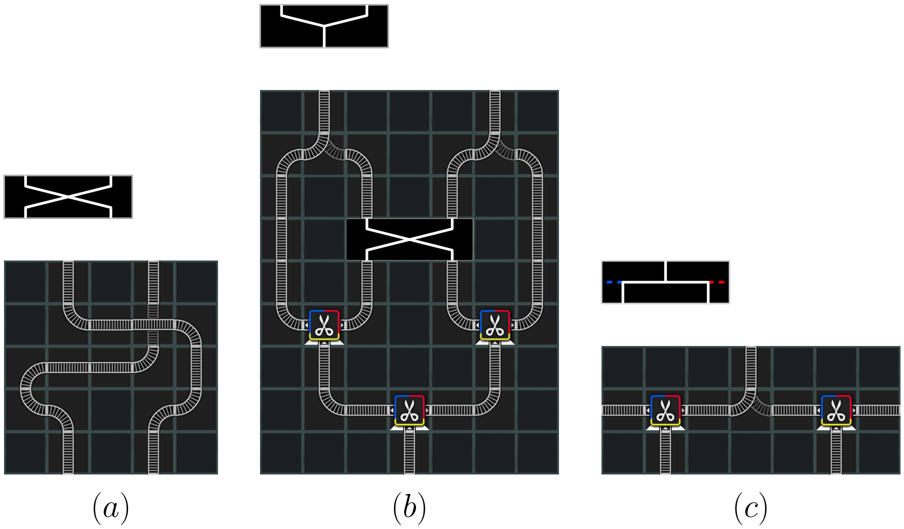


Figure 8 Implementation of the Crossover (a), Fanout (b), and Blend (c) gadgets.

4.3.4 Fanout Gadget

The *fanout* gadget duplicates an input train entering from the bottom rail into two identical trains exiting from the top two rails. This gadget is necessary as, in general, input-vertices and gates might have two or more out-neighbors. By chaining together several fanout gadgets it is possible to create exactly one train for each out-neighbor of a vertex, each encoding the same truth value. The implementation of this gadget is shown in Figure 8 (b).

4.3.5 Blend Gadget

We still have to show how “and” and “or” gates of the circuit C are implemented. To this aim it is useful to first describe an intermediate gadget that we call a *blend gadget* (see Figure 8 (c)). This gadget takes two input trains entering from the bottom side: the leftmost train is colored according to CS2 while the rightmost train is colored according to CS1. There are four possible combinations of (valid) inputs, corresponding to the four values in $\{\texttt{True}, \texttt{False}\}^2$. In all of these cases the gadget will output three trains. Two of them, namely the ones exiting from the left and right side of the gadget, are spurious trains and will be dealt with by the “and” and “or” gadgets. The other train is the actual output of the gadget and its color depends on the truth values of the input trains, as detailed in the following table:

Left Value	**Right Value**	**Left Color (CS2)**	**Right Color (CS1)**	**Output Color**
`True`	`True`	purple	red	red
`True`	`False`	purple	purple	purple
`False`	`True`	blue	red	purple
`False`	`False`	blue	purple	blue

In other words, the output train is blue if both the inputs are false, red if they are both true, and purple if exactly one of the inputs is true. Notice that, so far, this coloring scheme does not adhere neither to CS1 nor to CS2.

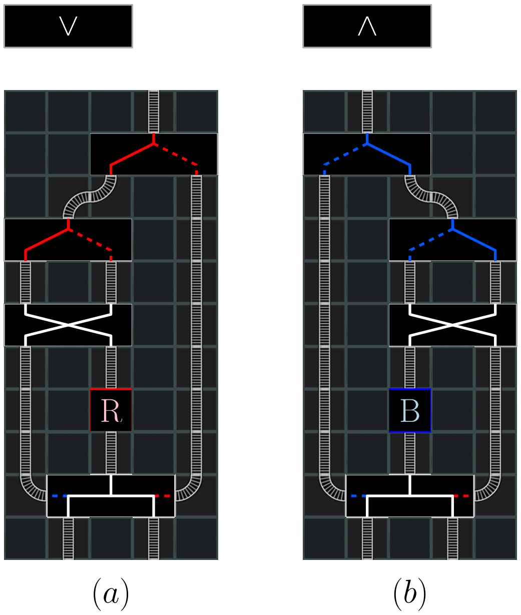


Figure 9 Implementation of the “or” (a) and “and” (b) gadgets.

4.3.6 Or Gadget

The *or* gadget is shown in Figure 9 (a) and implements an “or” gate of the circuit C. It is obtained by appending a Red-Shift gadget to the output of a Blend gadget. It is easy to check that the color of the output train is red if at least one of the two inputs of the blend gadget encodes the values `True`, and purple otherwise. In other words, the gadget computes the logical or between the two signals carried by the input trains and encodes the result according to CS1. To take care of the two extra trains exiting the Blend gadget we use two Absorption gadgets to merge them into the output train.

4.3.7 And Gadget

The implementation of the *and* gadget is similar to the one of the *or* gadget: by appending a Blue-Shift gadget to the output of a Blend gadget we have that the color output train is blue if at least one of the two input trains carries the logical value `False`, and purple otherwise. Thus, the and gadget computer the logical and of the two input signals and encodes the output according to CS2. As before, we dispose of the the two spurious trains exiting from the Blend gadget by using two Absorpion gadgets, as shown in Figure 9 (b).

4.3.8 Loop Gadget

The purpose of this gadget is to ensure that the circuit is evaluated (i.e., traversed by the trains) exactly 2^n times, where the trains exiting from the output-vertices of one iteration constitute the input of the next evaluation. After 2^n evaluations this feedback loop will break and the output trains will be allowed to reach the check area.

The gadget implementation is recursive and it is shown in Figure 10 (a)–(d). Figure 10 (a) shows the details of a 2-loop gadget: the first train to enter the gadget from the single rail on the bottom will exit from the top and will cause the two switches to flip, we call the first encountered switch (i.e., the one closer to the bottom) a *counting switch*. When a second

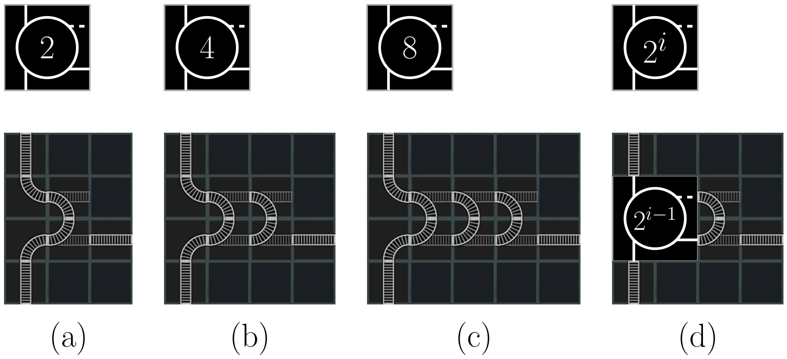


Figure 10 Implementation of a 2^i loop-gadget for $i = 1$ (a), $i = 2$ (b), $i = 3$ (c), and in the general case (d).

train enters the gadget, it will now exit from the rail on the right side due to the position of the counting switch (that will now revert to the initial state). Notice that the position of the other (non-counting) switch plays no role in the gadget operation. Indeed, its sole purpose is to provide an extra input that allows any train entering *from* the right side (i.e., moving in the right-to-left direction) to exit from the top side of the gadget without affecting the state of the counting switch. This allows a 2-loop to be used as a sub-gadget in the 4-loop construction of Figure 10 (b): here the first train to exit from the right side of a 2-loop traverses once again a counting and a non-counting switch (in order) and is routed back to the top exit of the 2-loop; the second train to exit the 2-loop (which corresponds to the 4-th train entering from the bottom rail) will revert the counting-switch to its original state and it will proceed to the right. By repeating this construction one can easily obtain the 8-loop of figure Figure 10 (c) and, more generally, the 2^i-loop of Figure 10 (d). The first $2^i - 1$ trains entering a 2^i-loop will exit from the top, while the 2^i-th train will exit from the right. Notice how after the j trains have traversed a 2^i-loop, the state of the counting switches encodes the number $i \bmod 2^i$ in binary, where the least-significant-bit corresponds to the innermost counting switch.

We lay the tracks so that all the n trains entering the “Loop and Check” area (see Figure 5) reach the 2^n-loop gadgets at the same time (this can be guaranteed by tuning the length of the railways connecting to the inputs of the loop gadgets). This is because, in contrast to the other gadgets, the time required for a train to traverse a 2^n-loop gadgets depends on the specific iteration (i.e., on the state of the counting switches). By requiring this additional property, we can ensure that, on every iteration, all the 2^n-loop gadgets will always change their state simultaneously and hence the n incoming trains will also exit the gadgets simultaneously. This allows us to also synchronize the trains once they are routed back to the “Input” area.

4.3.9 Check Gadgets

The purpose of the check gadgets (highlighted in gray in the “loop and check” area of Figure 5) is to ensure that the solution to the Trainyard level will be valid if and only if the circuit C of the It-Mon-BC instance is such that the first entry of $\overline{y} = f_C^{2^n}(\overline{x})$ is `True`. When the loop gadgets in the “loop and check” area are traversed 2^n times, the trains encoding the n output values of $f_C^{2^n}(\overline{x})$ will exit from the right side of their respective loop gadget. Since we are not interested in the values $y_2, \dots, y_n$, each of the corresponding trains is colored

red (thus discarding its currently encoded truth value) and fed into an arrival station that expects exactly one red train. As for the output train corresponding to y_1, we assume w.l.o.g. that it is encoded according to CS1 (if this is not the case, then it suffices to use a Red-Shift gadget just after the output of the corresponding 2^n-loop gadget). Then, the train will be red if y_1 is `True` and purple otherwise. Hence, it suffices to route this train into an arrival station expecting one red train. If $y_1 =$ `True`, all the arrival stations will be satisfied and the Trainyard level is won, otherwise the purple train training to enter a red station will crash and the level will be lost.

5 Conclusions

We have proved that the problem of checking whether a candidate solution to a Trainyard instance actually solves the level is PSPACE-complete. What is the exact complexity of *finding* such a solution, however, is still an open question: we know from [2] that this problem is NP-hard and it is easy to check that it also belongs to PSPACE (since it is possible to enumerate all the possible track layouts). If Trainyard belongs to NP, this would mean that computing a solution could be easier than checking it, unless NP = PSPACE. It might be the case that a more involved certificate than the natural one (i.e., the placement of the tracks in a solution) is needed, or that every Trainyard level that admits a solution also allows for a *simple* solution, i.e., a solution that can be recognized in polynomial time.

Other examples of games for which checking the natural solution is at least NP-hard[4] include *Settlers of Catan* and *Carcassonne*[5], as the scoring rules involve longest-path computations.

References

1 Aaron B. Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-zag numberlink is np-complete. *JIP*, 23(3):239–245, 2015. `doi:10.2197/ipsjjip.23.239`.

2 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-hard. *Theoretical Computer Science*, 2017. `doi:10.1016/j.tcs.2017.09.039`.

3 Joseph Culberson. Sokoban is PSPACE-complete. In *Proceedings of the 1st International Conference on Fun with Algorithms (FUN'98)*, volume 4, pages 65–76, 1998.

4 Giuseppe Pasolini dall'Onda and Pier Desiderio Pasolini. *Memoir of Count Giuseppe Pasolini*. Longmans, Green, and Company, 1885.

5 David Eppstein. Computational complexity of games and puzzles. `https://www.ics.uci.edu/~eppstein/cgt/hard.html`.

6 Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theor. Comput. Sci.*, 270(1-2):895–911, 2002. `doi:10.1016/S0304-3975(01)00173-6`.

7 Eric Goles, Pedro Montealegre, Ville Salo, and Ilkka Törmä. Pspace-completeness of majority automata networks. *Theor. Comput. Sci.*, 609:118–128, 2016. `doi:10.1016/j.tcs.2015.09.014`.

[4] More precisely, we are given the final state of the game and we want to check whether a player claiming to be the winner is actually the winner.

[5] With a suitable set of expansions: *Carcassonne: Abbey & Mayor* allows to have biforcations in roads (i.e., vertices of degree 3 in the road graph) and *Carcassonne: King and Scout* awards bonus points to the player that completes the longest road.

8 Luciano Gualà, 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, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932866.

9 Luciano Gualà, Stefano Leucci, Emanuele Natale, and Roberto Tauraso. Large peg-army maneuvers. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.18.

10 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* CRC Press, 2009.

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

12 Stefan Langerman and Yushi Uno. Threes!, fives, 1024!, and 2048 are hard. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.22.

13 Neeldhara Misra. Two dots is np-complete. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 24:1–24:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.24.

14 Italia : Consiglio superiore dei lavori pubblici. *Annali dei lavori pubblici.* Eredi di A. De Gaetani. In italian.

15 Ryuhei Uehara and Shigeki Iwata. Generalized Hi-Q is NP-complete. *IEICE Transactions (1976-1990)*, 73(2):270–273, 1990.

How Bad is the Freedom to Flood-It?

Rémy Belmonte
The University of Electro-Communications, Chofu, Tokyo, Japan
remy.belmonte@uec.ac.jp

Mehdi Khosravian Ghadikolaei
Université Paris-Dauphine, PSL Research University, CNRS, UMR,
LAMSADE, 75016 Paris, France
mehdi.khosravian-ghadikolaei@dauphine.fr

Masashi Kiyomi
Yokohama City University, Yokohama, Japan
masashi@yokohama-cu.ac.jp

Michael Lampis
Université Paris-Dauphine, PSL Research University, CNRS, UMR,
LAMSADE, 75016 Paris, France
michail.lampis@dauphine.fr

Yota Otachi
Kumamoto University, Kumamoto, Japan
otachi@cs.kumamoto-u.ac.jp
https://orcid.org/0000-0002-0087-853X

Abstract

Fixed-Flood-It and Free-Flood-It are combinatorial problems on graphs that generalize a very popular puzzle called *Flood-It*. Both problems consist of recoloring moves whose goal is to produce a monochromatic ("flooded") graph as quickly as possible. Their difference is that in Free-Flood-It the player has the additional freedom of choosing the vertex to play in each move. In this paper, we investigate how this freedom affects the complexity of the problem. It turns out that the freedom is *bad* in some sense. We show that some cases trivially solvable for Fixed-Flood-It become intractable for Free-Flood-It. We also show that some tractable cases for Fixed-Flood-It are still tractable for Free-Flood-It but need considerably more involved arguments. We finally present some combinatorial properties connecting or separating the two problems. In particular, we show that the length of an optimal solution for Fixed-Flood-It is always at most twice that of Free-Flood-It, and this is tight.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms, Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases flood-filling game, parameterized complexity

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.5

Related Version A full version of the paper is available at `http://arxiv.org/abs/1804.08236`.

Funding This work is partially supported by JSPS and MAEDI under the Japan-France Integrated Action Program (SAKURA) Project GRAPA 38593YJ.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 5; pp. 5:1–5:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

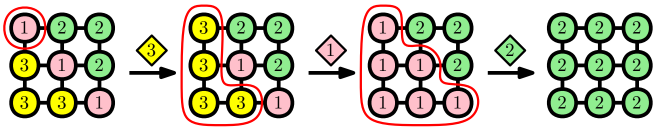


Figure 1 A flooding sequence on a 3 × 3 grid.

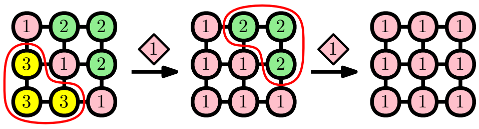


Figure 2 A flooding sequence with no restriction on selected monochromatic components.

1 Introduction

Flood-It is a popular puzzle, originally released as a computer game in 2006 by LabPixies (see [1]). In this game, the player is presented with (what can be thought of as) a vertex-colored grid graph, with a designated special *pivot* vertex, usually the top-left corner of the grid. In each move, the player has the right to change the color of all vertices contained in the same monochromatic component as the pivot to a different color of her choosing. Doing this judiciously gradually increases the size of the pivot's monochromatic component, until the whole graph is *flooded* with one color. The goal is to achieve this flooding with the minimum number of moves. See Figure 1 for an example.

Following the description above, *Flood-It* immediately gives rise to a natural optimization problem: given a vertex-colored graph, determine the shortest sequence of flooding moves that wins the game. This problem has been extensively studied in the last few years (e.g. [10, 12, 14, 13, 7, 3, 17, 4, 15, 9]; a more detailed summary of known results is given below), both because of the game's popularity (and addictiveness!), but also because the computational complexity questions associated with this problem have turned out to be surprisingly deep, and the problem has turned out to be surprisingly intractable.

The goal of this paper is to add to our understanding of this interesting, puzzle-inspired, optimization problem, by taking a closer look at the importance of the *pivot vertex*. As explained above, the classical version of the game only allows the player to change the color of a special vertex and its component and has been studied under the name Fixed-Flood-It [12, 14, 13] (or Flood-It in some papers [1, 17, 3, 4, 9]). However, it is extremely natural to also consider a version where the player is also allowed to play a different vertex of her choosing in each turn. This has also been well-studied under the name Free-Flood-It [1, 10, 12, 14, 13, 3, 17]. See Figure 2.

Since both versions of this problem have been studied before, the question of the impact of the pivot vertex on the problem's structure has (at least implicitly) been considered. Intuitively, one would expect Free-Flood-It to be a harder problem; after all, the player has to choose a color to play *and* a vertex to play it on, and is hence presented with a larger set of possible moves. The state of the art seems to confirm this intuition, as only some of the positive algorithmic results known for Fixed-Flood-It are known also for Free-Flood-It, while there do exist some isolated cases where Fixed-Flood-It is tractable and Free-Flood-It is hard, for example co-comparability graphs [5, 7] and grids of height 2 [1, 13]. Nevertheless, these results do not completely pinpoint the added complexity brought by the task of selecting a vertex to play, as the mentioned algorithms for Fixed-Flood-It are already non-trivial, and hence the jump in complexity is likely to be the result of *the combination* of the tasks of picking a color and a vertex. More broadly, [3] presented a generic reduction from Fixed-Flood-It to Free-Flood-It that preserves a number of crucial parameters (number of colors, optimal value, etc.) and gives convincing evidence that Free-Flood-It is always at least as hard as Fixed-Flood-It, but not necessarily harder.

Our Results: We investigate the complexity of Free-Flood-It, mostly from the point of view of parameterized complexity,[1] as well as the impact on the combinatorics of the game of allowing moves outside the pivot. Due to space constraints, some proofs are ommited and marked with ★.

Our first result is to show that Free-Flood-It is W[2]-hard parameterized by the number of moves in an optimal solution. We recall that for Fixed-Flood-It this parameterization is trivially fixed-parameter tractable: when a player has only k moves available, then we can safely assume that the graph uses at most (roughly) k colors, hence one can easily consider all possible solutions in FPT time. The interest of our result is, therefore, to demonstrate that the task of deciding which vertex to play next is sufficient to make Free-Flood-It significantly harder than Fixed-Flood-It. Indeed, the W[2]-hardness reduction we give, implies also that Free-Flood-It is not solvable in $n^{o(k)}$ time under the ETH. This tightly matches the complexity of a trivial algorithm which considers all possible vertices and colors to be played. This is the first concrete example showing a case where Fixed-Flood-It is essentially trivial, but Free-Flood-It is intractable.

Motivated by this negative result we consider several other parameterizations of the problem. We show that Free-Flood-It is fixed-parameter tractable when parameterized by the number of possible moves and the clique-width. This result is tight in the sense that the problem is hard when parameterized by only one of these parameters. It also implies the fixed-parameter tractability of the problem parameterized by the number of colors and the modular-width. In a similar vein, we present a polynomial kernel when Free-Flood-It is parameterized by the input graph's neighborhood diversity and number of colors. An analogous result was shown for Fixed-Flood-It in [4], but because of the freedom to select vertices, several of the tricks used there do not apply to Free-Flood-It, and our proofs are slightly more involved. Our previously mentioned reduction also implies that Free-Flood-It does not admit a polynomial kernel parameterized by vertex cover, under standard assumptions. This result was also shown for Fixed-Flood-It in [4], but it does not follow immediately for Free-Flood-It, as the reduction of [3] does not preserve the graph's vertex cover.

Motivated by the above results, which indicate that the complexity of the problem can be seriously affected if one allows non-pivot moves, we also study some more purely combinatorial questions with algorithmic applications. The main question we pose here is the following. It is obvious that for all instances the optimal number of moves for Free-Flood-It is upper-bounded by the optimal number of moves for Fixed-Flood-It (since the player has strictly more choices), and it is not hard to construct instances where Fixed-Flood-It needs strictly more moves. Can we bound the optimal number of Fixed-Flood-It moves needed as a function of the optimal number of Fixed-Flood-It moves? Somewhat surprisingly, this extremely natural question does not seem to have been explicitly considered in the literature before. Here, we completely resolve it by showing that the two optimal values cannot be more than a factor of 2 apart, and constructing a family of simple instances where they are exactly a factor of 2 apart. As an immediate application, this gives a 2-approximation for Free-Flood-It for every case where Fixed-Flood-It is known to be tractable.

[1] For readers unfamiliar with the basic notions of this field, we refer to standard textbooks [2, 6].

We also consider the problem's monotonicity: Fixed-Flood-It has the nice property that even an adversary that selects a single bad move cannot increase the optimal (that is, in the worst case a bad move is a wasted move). We construct minimal examples which show that Free-Flood-It does not have this nice monotonicity property, even for extremely simple graphs, that is, making a bad move may not only waste a move but also make the instance strictly worse. Such a difference was not explicitly stated in the literature, while the monotonicity of Fixed-Flood-It was seem to be known or at least assumed. The only result we are aware of is the monotonicity of Free-Flood-It on paths shown by Meeks and Scott [12].

Known results: In 2009, the NP-hardness of Fixed-Flood-It with six colors was sketched by Elad Verbin as a comment to a blog post by Sariel Har-Peled [16]. Independently to the blog comment, Clifford et al. [1] and Fleischer and Woeginger [5] started investigations of the complexity of the problem, and published the conference versions of their papers at FUN 2010. Below we mostly summarize some of the known results on Free-Flood-It. For more complete lists of previous result, see e.g. [7, 10, 4].

Free-Flood-It is NP-hard if the number of colors is at least 3 [1] even for trees with only one vertex of degree more than 2 [10, 3], while it is polynomial-time solvable for general graphs if the number of colors is at most 2 [1, 12, 10]. Moreover, it is NP-hard even for height-3 grids with four colors [12]. Note that this result implies that Free-Flood-It with a constant number colors is NP-hard even for graphs of bounded bandwidth. If the number of colors is unbounded, then it is NP-hard for height-2 grids [13], trees of radius 2 [3], and, proper interval graphs and caterpillars [7]. Also, it is known that there is no constant-factor approximation with a factor independent of the number of colors unless P = NP [1].

There are a few positive results on Free-Flood-It. Meeks and Scott [14] showed that every colored graph has a spanning tree with the same coloring such that the minimum number of moves coincides in the graph and the spanning tree. Using this property, they showed that if a graph has only a polynomial number of vertex subsets that induce connected subgraphs, then Free-Flood-It (and Fixed-Flood-It) on the graph can be solved in polynomial time. This in particular implies the polynomial-time solvability on subdivisions of a fixed graph. It is also known that Free-Flood-It for interval graphs and split graphs is fixed-parameter tractable when parameterized by the number of colors [7].

2 Preliminaries

For a positive integer k, we use $[k]$ to denote the set $\{1, \dots, k\}$. Given a graph $G = (V, E)$, a coloring function $\mathbf{col}\colon V \to [c_{\max}]$, where $c_{\max}$ is a positive integer, and $u \in V$, we denote by $\mathbf{Comp}(\mathbf{col}, u)$ the maximal set of vertices S such that for all $v \in S$, $\mathbf{col}(u) = \mathbf{col}(v)$ and there exists a path from u to v such that for all its internal vertices w we have $\mathbf{col}(w) = \mathbf{col}(u)$. In other words, $\mathbf{Comp}(\mathbf{col}, u)$ is the monochromatic connected component that contains u under the coloring function $\mathbf{col}$.

Given $G, \mathbf{col}$, a *move* is defined as a pair (u, i) where $u \in V$, $i \in [c_{\max}]$. The *result* of the move (u, c) is a new coloring function $\mathbf{col}'$ defined as follows: $\mathbf{col}'(v) = c$ for all $v \in \mathbf{Comp}(\mathbf{col}, u)$; $\mathbf{col}'(v) = \mathbf{col}(v)$ for all other vertices. In words, a move consists of changing the color of u, and of all vertices in the same monochromatic component as u, to c. Given the above definition we can also define the result of a sequence of moves $(u_1, c_1), (u_2, c_2), \dots, (u_k, c_k)$ on a colored graph with initial coloring function $\mathbf{col}_0$ in the natural way, that is, for each $i \in [k]$, $\mathbf{col}_i$ is the result of move (u_i, c_i) on $\mathbf{col}_{i-1}$.

The Free-Flood-It problem is defined as follows: given a graph $G = (V, E)$, an integer k, and an initial coloring function $\mathbf{col}_0$, decide if there exists a sequence of k moves $(u_1, c_1), (u_2, c_2), \dots, (u_k, c_k)$ such that the result $\mathbf{col}_k$ obtained by applying this sequence of moves on $\mathbf{col}_0$ is a constant function (that is, $\forall u, v \in V$ we have $\mathbf{col}_k(u) = \mathbf{col}_k(v)$).

In the Fixed-Flood-It problem we are given the same input as in the Free-Flood-It problem, as well as a designated vertex $p \in V$ (the pivot). The question is again if there exists a sequence of moves such that $\mathbf{col}_k$ is monochromatic, with the added constraint that we must have $u_i = p$ for all $i \in [k]$.

We denote by $\mathrm{OPT}_{\mathrm{Free}}(G, \mathbf{col}), \mathrm{OPT}_{\mathrm{Fixed}}(G, \mathbf{col}, p)$ the minimum k such that for the input $(G, \mathbf{col})$ (or $(G, \mathbf{col}, p)$ respectively) the Free-Flood-It problem (respectively the Fixed-Flood-It problem) admits a solution.

Graph parameters: The graph parameters considered in this paper are the *vertex cover number* $\mathsf{vc}(G)$, the *neighborhood diversity* $\mathsf{nd}(G)$, the *modular-width* $\mathsf{mw}(G)$, and the *clique-width* $\mathsf{cw}(G)$. It is known that $\mathsf{cw}(G) \le \mathsf{mw}(G) \le \mathsf{nd}(G) \le 2^{\mathsf{vc}(G)} + \mathsf{vc}(G)$ for every graph G [8, 11]. (See [11, 8, 2] for definitions.)

3 W[2]-hardness of Free-Flood-It

The main result of this section is that Free-Flood-It is W[2]-hard when parameterized by the minimum length of any valid solution (the natural parameter). The proof consists of a reduction from Set Cover, a canonical W[2]-complete problem.

Before presenting the construction, we recall two basic observations by Meeks and Vu [15], both of which rest on the fact that any single move can (at most) eliminate a single color from the graph, and this can only happen if a color induces a single component.

▶ **Lemma 3.1** ([15]). *For any graph $G = (V, E)$, and coloring function $\mathbf{col}$ that uses $c_{\max}$ distinct colors, we have $OPT_{Free}(G, \mathbf{col}) \ge c_{\max} - 1$.*

▶ **Lemma 3.2** ([15]). *For any graph $G = (V, E)$, and coloring function $\mathbf{col}$ that uses $c_{\max}$ distinct colors, such that for all $c \in [c_{\max}]$, $G[\mathbf{col}^{-1}(c)]$ is a disconnected graph, we have $OPT_{Free}(G, \mathbf{col}) \ge c_{\max}$.*

The proof of Theorem 3.6 relies on a reduction from a special form of Set Cover, which we call Multi-Colored Set Cover (MCSC for short). MCSC is defined as follows:

▶ **Definition 3.3.** In Multi-Colored Set Cover (MCSC) we are given as input a set of elements R and k collections of subsets of R, $\mathcal{S}_1, \dots, \mathcal{S}_k$. We are asked if there exist k sets $S_1, \dots, S_k$ such that for all $i \in [k]$, $S_i \in \mathcal{S}_i$, and $\cup_{i \in [k]} S_i = R$.

Observe that MCSC is just a version of Set Cover where the collection of sets is given to us pre-partitioned into k parts and we are asked to select one set from each part to form a set cover of the universe. It is not hard to see that any Set Cover instance $(\mathcal{S}, R)$ where we are asked if there exists a set cover of size k can easily be transformed to an equivalent MCSC instance simply by setting $\mathcal{S}_i = \mathcal{S}$ for all $i \in [k]$, since the definition of MCSC does not require that the sub-collections $\mathcal{S}_i$ be disjoint. We conclude that known hardness results for Set Cover immediately transfer to MCSC, and in particular MCSC is W[2]-hard when parameterized by k.

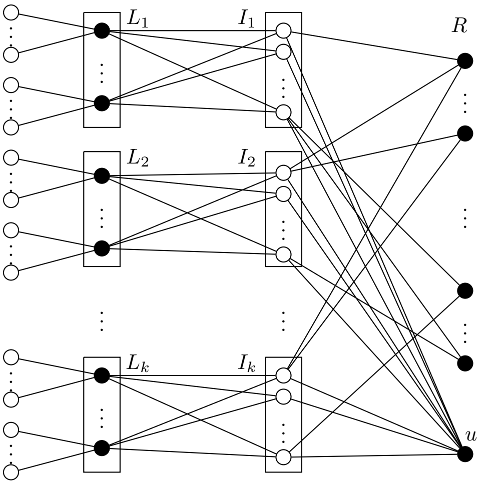


Figure 3 The graph $G = (V, E)$ of Free-Flood-It constructed from the given MCSC instance. All the vertices in each I_i have color i and all black vertices have color $k + 1$. Boxes containing black vertices have size $3k$. Also each vertex in L_i has k neighbors with degree 1 colored $1, ..., k$.

Construction

We are now ready to describe our reduction which, given a MCSC instance with universe R and k collections of sets $\mathcal{S}_i, i \in [k]$, produces an equivalent instance of Free-Flood-It, that is, a graph $G = (V, E)$ and a coloring function **col** on V. We construct this graph as follows:

- for every set $S \in \mathcal{S}_i$, construct a vertex in V. The set of vertices in V corresponding to sets of $\mathcal{S}_i$ is denoted by I_i and $\mathbf{col}(v) = i$ for each $v \in I_i$. $I_1 \cup ... \cup I_k$ induces an independent set colored $\{1, ..., k\}$.
- for each $i \in [k]$, construct $3k$ new vertices, denoted by L_i and connect all of them to all vertices of I_i such that $L_i \cup I_i$ induces a complete bipartite graph of size $3k \times |I_i|$. Then set $\mathbf{col}(v) = k + 1$ for each $v \in L_i$, for all $i \in [k]$.
- for each vertex $v \in L_i$ for $1 \leq i \leq k$, construct k new leaf vertices connected to v with distinct colors $1, ..., k$.
- for each element $e \in R$, construct a vertex e. For each $S \in \mathcal{S}_i$ such that $e \in S$ we connect e to the vertex of I_i that represents S.
- add a special vertex u with $\mathbf{col}(u) = k + 1$ which is connected it to all vertices in I_i for $i \in [k]$.

An illustration of G is shown in Fig.3. In the following we will show that $(G, \mathbf{col})$ as an instance of Free-Flood-It is solvable with at most $2k$ moves if and only if the given MCSC instance has a set cover of size k which contains one set of each $\mathcal{S}_i$.

▶ **Lemma 3.4.** *If* $(\mathcal{S}_1, \ldots, \mathcal{S}_k, R)$ *is a YES instance of* MCSC *then* $OPT_{Free}(G, \mathbf{col}) \leq 2k$.

Proof. Suppose that there is a solution $S_1, \ldots, S_k$ of the given MCSC instance, with $S_i \in \mathcal{S}_i$, for $i \in [k]$ and $\cup_{i \in [k]} S_i = R$. Recall that for each S_i there is a vertex in I_i in the constructed graph representing S_i. Our first k moves consist of changing the color of each of these k vertices to $k + 1$ in some arbitrary order.

Observe that in the graph resulting after these k moves the vertices with color $k + 1$ form a single connected component: because $\cup S_i$ is a set cover, all vertices of R have a neighbor with color $k + 1$; all vertices with color $k + 1$ in some I_i are in the same component as u; and all vertices of $\cup_{i \in [k]} L_i$ are connected to one of the vertices we played. Furthermore, observe that this component dominates the graph: all remaining vertices of $\cup I_i$, as well as all leaves attached to vertices of $\cup_{i\in[k]}L_i$ are dominated by the vertices of $\cup_{i\in[k]}L_i$. Hence, we can select an arbitrary vertex with color $k+1$, say u, and cycle through the colors $1,\dots,k$ on this vertex to make the graph monochromatic. ◀

Now we establish the converse of Lemma 3.4.

▶ **Lemma 3.5.** *If $OPT_{Free}(G,\mathbf{col}) \le 2k$, then $(\mathcal{S}_1,\dots,\mathcal{S}_k,R)$ is a YES instance of* MCSC.

Proof. Suppose that there exists a sequence of at most $2k$ moves solving $(G,\mathbf{col})$. We can assume without loss of generality that the sequence has length exactly $2k$, since performing a move on a monochromatic graph keeps the graph monochromatic. Let $(u_1,c_1),\dots,(u_{2k},c_{2k})$ be a solution, let $\mathbf{col}_0 = \mathbf{col}$, and let $\mathbf{col}_i$ denote the coloring of G obtained after the first i moves. The key observation that we will rely on is the following:

(i) For all $i\in[k]$, there exist $j\in[k], v\in I_i$ such that $\mathbf{col}_j(v)=k+1$.

In other words, we claim that for each group I_i there exists a vertex that received color $k+1$ at some point during the first k moves. Before proceeding, let us prove this claim. Suppose for contradiction that the claim is false. Then, there exists a group I_i such that no vertex in that group has color $k+1$ in any of the colorings $\mathbf{col}_0,\dots,\mathbf{col}_k$. We now consider the vertices of L_i and their attached leaves. Since L_i contains $3k > k+2$ vertices, there exist two vertices v_1,v_2 of L_i such that $\{u_1,\dots,u_k\}$ contains neither v_1,v_2, nor any of their attached leaves. In other words, there exist two vertices of L_i on which the winning sequence does not change colors by playing them or their private neighborhood directly. However, since v_1,v_2 only have neighbors in I_1 (except for their attached leaves), and no vertex of I_1 received color $k+1$, we conclude that $\mathbf{col}_k(v_1)=\mathbf{col}_k(v_2)=k+1$, that is, the colors of these two vertices have remained unchanged, and the same is true for their attached leaves. Consider now the graph G with coloring $\mathbf{col}_k$: we observe that this coloring uses $k+1$ distinct colors, and that each color induces a disconnected graph. This is true for colors $1,\dots,k$ because of the leaves attached to v_1,v_2, and true of color $k+1$ because of v_1,v_2 and the fact that no vertex of I_i has color $k+1$. We conclude that $\mathrm{OPT}_{\mathrm{Free}}(G,\mathbf{col}_k)\ge k+1$ by Lemma 3.2, which is a contradiction, because the whole sequence has length $2k$.

Because of claim (i) we can now conclude that for all $i\in[k]$ there exists a $j\in[k]$ such that $\mathbf{col}_{j-1}(u_j)=i$. In other words, for each color i there exists a move among the first k moves of the solution that played a vertex which at that point had color i. To see that this is true consider again for contradiction the case that for some $i\in[k]$ this statement does not hold: this implies that vertices with color i in $\mathbf{col}_0$ still have color i in $\mathbf{col}_1,\dots,\mathbf{col}_k$, which means that no vertex of I_i has received color $k+1$ in the first k moves, contradicting (i).

As a result of the above, we therefore claim that for all $j\in[k]$, we have $\mathbf{col}_{j-1}(u_j)\ne k+1$. In other words, we claim that none of the first k moves changes the color of a vertex that at that point had color $k+1$. This is because, as argued, for each of the other k colors, there is a move among the first k moves that changes a vertex of that color. We therefore conclude that for all vertices v for which $\mathbf{col}_0(v)=k+1$ we have $\mathbf{col}_j(v)=k+1$ for all $j\in[k]$. In addition, because in $\mathbf{col}_0$ all colors induce independent sets, each of the first k moves changes the color of a single vertex. Because of claim (i), this means that for each $i\in[k]$ one of the first k moves changes the color of a single vertex from I_i to $k+1$. We select the corresponding set of $\mathcal{S}_i$ in our MCSC solution.

We now observe that, since all vertices of $\cup_{i\in[k]}L_i$ retain color $k+1$ throughout the first k moves, $\mathbf{col}_k$ is a coloring function that uses $k+1$ distinct colors, and colors $1,\dots,k$ induce disconnected graphs (because of the leaves attached to the vertices of each L_i). Thanks to

Lemma 3.2, this means that $\mathbf{col}_k^{-1}(k+1)$ must induce a connected graph. Hence, all vertices of R have a neighbor with color $k+1$ in $\mathbf{col}_k$, which must be one of the k vertices played in the first k moves; hence the corresponding element is dominated by our solution and we have a valid set cover selecting one set from each $\mathcal{S}_i$. ◀

We are now ready to combine Lemmas 3.4 and 3.5 to obtain the main result of this section.

▶ **Theorem 3.6.** FREE-FLOOD-IT *is W[2]-hard parameterized by OPT_{Free}, that is, parameterized by the length of the optimal solution. Furthermore, if there is an algorithm that decides if a* FREE-FLOOD-IT *instance has a solution of length k in time $n^{o(k)}$, then the ETH is false.*

Proof. The described construction, as well as Lemmas 3.4 and 3.5 give a reduction from MCSC, which is W[2]-hard parameterized by k, to an instance of FREE-FLOOD-IT with $k+1$ colors, where the question is to decide if $\mathrm{OPT}_{\mathrm{Free}}(G,\mathbf{col}) \le 2k$. Furthermore, it is known that MCSC generalizes DOMINATING SET, which does not admit an algorithm running in time $n^{o(k)}$, under the ETH [2]. Since our reduction only modifies k by a constant, we odtain the same result for FREE-FLOOD-IT. ◀

We note that because of Lemma 3.1 we can always assume that the number of colors of a given instance is not much higher than the length of the optimal solution. As a result, FREE-FLOOD-IT parameterized by $\mathrm{OPT}_{\mathrm{Free}}$ is equivalent to the parameterization of FREE-FLOOD-IT by $\mathrm{OPT}_{\mathrm{Free}} + c_{\max}$ and the result of Theorem 3.6 also applies to this parameterization. Also, as a byproduct of the reduction above, we can show a kernel lower bound for FREE-FLOOD-IT parameterized by the vertex cover number.

▶ **Theorem 3.7 (★).** FREE-FLOOD-IT *parameterized by the vertex cover number admits no polynomial kernel unless* $\mathrm{PH} = \Sigma_3^{\mathrm{p}}$.

4 Clique-width and neighborhood diversity

In this section, we consider as a combined parameter for FREE-FLOOD-IT the length of an optimal solution and the clique-width. We show that this case is indeed fixed-parameter tractable by using the theory of the monadic second-order logic on graphs. As an application of this result, we also show that combined parameterization by the number of colors and the modular-width is fixed-parameter tractable.

▶ **Theorem 4.1 (★).** *Given an instance $(G,\mathbf{col})$ of* FREE-FLOOD-IT *such that G has n vertices and clique-width at most w, it can be decided in time $O(f(k,w)\cdot n^3)$ whether $OPT_{Free}(G,\mathbf{col}) \le k$, where f is some computable function.*

▶ **Corollary 4.2 (★).** *Given an integer k and an instance $(G,\mathbf{col})$ of* FREE-FLOOD-IT *such that G has n vertices and modular-width at most w, it can be decided in time $O(f(c_{\max},w)\cdot n^3)$ whether $OPT_{Free}(G,\mathbf{col}) \le k$, where f is some computable function.*

Since the modular-width of a graph is upper bounded by its neighborhood diversity, the corollary above implies that FREE-FLOOD-IT is fixed-parameter tractable when parameterized by both the neighborhood diversity and the number of colors. Here we show that FREE-FLOOD-IT admits a polynomial kernel with the same parameterization.

▶ **Theorem 4.3.** FREE-FLOOD-IT *admits a kernel of* $\mathsf{nd}(G)\cdot c_{\max}\cdot(\mathsf{nd}(G)+c_{\max}-1)$ *vertices.*

Our reduction rules are as follows:

- Rule TT: Let u and v be true twins of the same color in $(G, \mathbf{col})$. Remove v.
- Rule FT: Let F be a set of false-twin vertices of the same color in $(G, \mathbf{col})$ such that $|F| = \mathsf{nd}(G) + c_{\max}$. Remove arbitrary one vertex in F.

Observe that after applying TT and FT exhaustively in polynomial time, the obtained graph can have at most $\mathsf{nd}(G) \cdot c_{\max} \cdot (\mathsf{nd}(G) + c_{\max} - 1)$ vertices. This is because each set of twin vertices can contain at most $\mathsf{nd}(G) + c_{\max} - 1$ vertices. Hence, to prove Theorem 4.3, it suffices to show the safeness of the rules.

▶ **Lemma 4.4** (★). *The rules TT and FT are safe.*

5 Relation Between Fixed and Free Flood-It

The main theorem of this section is the following:

▶ **Theorem 5.1.** *For any graph $G = (V, E)$, coloring function* $\mathbf{col}$ *on G, and $p \in V$ we have*

$$OPT_{Free}(G, \mathbf{col}) \leq OPT_{Fixed}(G, \mathbf{col}, p) \leq 2OPT_{Free}(G, \mathbf{col}).$$

Theorem 5.1 states that the optimal solutions for Free-Flood-It and Fixed-Flood-It can never be more than a factor of 2 apart. It is worthy of note that we could not hope to obtain a constant smaller than 2 in such a theorem, and hence the theorem is tight.

▶ **Theorem 5.2.** *There exist instances of* Fixed-Flood-It *such that $OPT_{Fixed}(G, \mathbf{col}, p) = 2OPT_{Free}(G, \mathbf{col})$*

Proof. Consider a path on $2n + 1$ vertices properly colored with colors $1, 2$. If we set the pivot to be one of the endpoints then $\mathrm{OPT}_{\mathrm{Free}} = 2n$. However, it is not hard to obtain a Free-Flood-It solution with n moves by playing every vertex at odd distance from the pivot. ◀

Before we proceed to give the proof of Theorem 5.1, let us give a high-level description of our proof strategy and some general intuition. The first inequality is of course trivial, so we focus on the second part. We will establish it by induction on the number of non-pivot moves performed by an optimal Free-Flood-It solution. The main inductive argument is based on observing that a valid Free-Flood-It solution will either at some point play a neighbor u of the component of p to give it the same color as p, or if not, it will at some point play p to give it the same color as one of its neighbors. The latter case is intuitively easier to handle, since then we argue that the move that changed p's color can be performed first, and if the first move is a pivot move we can easily fall back on the inductive hypothesis. The former case, which is the more interesting one, can be handled by replacing the single move that gives u the same color as p, with two moves: one that gives p the same color as u, and one that flips p back to its previous color. Intuitively, this basic step is the reason we obtain a factor of 2 in the relationship between the two versions of the game.

The inductive strategy described above faces some complications due to the fact that rearranging moves in this way may unintentionally re-color some vertices, which makes it harder to continue the rest of the solution as before. To avoid this we define a somewhat generalized version of Free-Flood-It, called Subset-Free-Flood-It.

▶ **Definition 5.3.** Given $G = (V, E)$, a coloring function $\mathbf{col}$ on G, and a pivot $p \in V$, a *set-move* is a pair (S, c), with $S \subseteq V$ and $S = \mathbf{Comp}(\mathbf{col}, u)$ for some $u \in V$, or $\{p\} \subseteq S \subseteq \mathbf{Comp}(\mathbf{col}, p)$. The result of (S, c) is the coloring $\mathbf{col}'$ that sets $\mathbf{col}'(v) = c$ for $v \in S$; and $\mathbf{col}'(v) = \mathbf{col}(v)$ otherwise.

We define Subset-Free-Flood-It as the problem of determining the minimum number of set-moves required to make a graph monochromatic, and Subset-Fixed-Flood-It as the same problem when we impose the restriction that every move must change the color of p, and denote as $\mathrm{OPT}_{\text{S-Free}}, \mathrm{OPT}_{\text{S-Fixed}}$ the corresponding optimum values.

Informally, a set-move is the same as a normal move in Free-Flood-It, except that we are also allowed to select an arbitrary connected monochromatic set S that contains p (even if S is not maximal) and change its color. Intuitively, one would expect moves that set S to be a proper subset of $\mathbf{Comp}(\mathbf{col}, p)$ to be counter-productive, since such moves split a monochromatic component into two pieces. Indeed, we prove below in Lemma 5.4 that the optimal solutions to Fixed-Flood-It and Subset-Fixed-Flood-It coincide, and hence such moves do not help. The reason we define this version of the game is that it gives us more freedom to define a solution that avoids unintentionally recoloring vertices as we transform a given Free-Flood-It solution to a Fixed-Flood-It solution.

▶ **Lemma 5.4.** *For any graph $G = (V, E)$, coloring function $\mathbf{col}$ on G, and pivot $p \in V$ we have $OPT_{Fixed}(G, \mathbf{col}, p) = OPT_{S\text{-}Fixed}(G, \mathbf{col}, p)$.*

Proof. First, observe that $\mathrm{OPT}_{\text{S-Fixed}}(G, \mathbf{col}, p) \leq \mathrm{OPT}_{\text{Fixed}}(G, \mathbf{col}, p)$ is trivial, as any solution of Fixed-Flood-It is a solution to Subset-Fixed-Flood-It by playing the same sequence of colors and always selecting all of the connected monochromatic component of p.

Let us also establish the converse inequality. Consider a solution $(S_1, c_1), \dots, (S_k, c_k)$ of Subset-Fixed-Flood-It, where by definition we have $p \in S_i$ for all $i \in [k]$. We would like to prove that $(p, c_1), (p, c_2), \dots, (p, c_k)$ is a valid solution for Fixed-Flood-It. Let $\mathbf{col}_i$ be the result of the first i set-moves of the former solution, and $\mathbf{col}'_i$ be the result of the first i moves of the latter solution. We will establish by induction the following:

1. For all $i \in [k]$ we have $\mathbf{Comp}(\mathbf{col}_i, p) \subseteq \mathbf{Comp}(\mathbf{col}'_i, p)$.
2. For all $i \in [k], u \in V \setminus \mathbf{Comp}(\mathbf{col}'_i, p)$ we have $\mathbf{col}_i(u) = \mathbf{col}'_i(u)$.

The statements are true for $i = 0$. Suppose that the two statements are true after $i - 1$ moves. The first solution now performs the set-move (S_i, c_i) with $S_i \subseteq \mathbf{Comp}(\mathbf{col}_{i-1}, p) \subseteq \mathbf{Comp}(\mathbf{col}'_{i-1}, p)$. We now have that $\mathbf{Comp}(\mathbf{col}_i, p)$ contains S_i plus the neighbors of S_i which have color c_i in $\mathbf{col}_{i-1}$. Such vertices either also have color c_i in $\mathbf{col}'_{i-1}$, or are contained in $\mathbf{Comp}(\mathbf{col}'_{i-1}, p)$; in both cases they are included in $\mathbf{Comp}(\mathbf{col}'_i, p)$, which establishes the first condition. To see that the second condition continues to hold observe that every vertex for which $\mathbf{col}_{i-1}(u) \neq \mathbf{col}_i(u)$ or $\mathbf{col}'_{i-1}(u) \neq \mathbf{col}'_i(u)$ belongs in $\mathbf{Comp}(\mathbf{col}'_i, p)$; the colors of other vertices remain unchanged. Since in the end $\mathbf{Comp}(\mathbf{col}_k, p) = V$ the first condition ensures that $\mathbf{Comp}(\mathbf{col}'_k, p) = V$. ◀

We are now ready to state the proof of Theorem 5.1.

Proof of Theorem 5.1. As mentioned, we focus on proving the second inequality as the first inequality follows trivially from the definition of the problems. Given a graph $G = (V, E)$, an initial coloring function $\mathbf{col} = \mathbf{col}_0$, and a pivot $p \in V$, we suppose we have a solution to Free-Flood-It $(u_1, c_1), (u_2, c_2), \dots, (u_k, c_k)$. In the remainder, we denote by $\mathbf{col}_i$ the coloring that results after the moves $(u_1, c_1), \dots, (u_i, c_i)$. We can immediately construct an equivalent solution to Subset-Free-Flood-It from this, producing the same sequence of colorings: $(\mathbf{Comp}(\mathbf{col}_0, u_1), c_1), (\mathbf{Comp}(\mathbf{col}_1, u_2), c_2), \dots, (\mathbf{Comp}(\mathbf{col}_{k-1}, u_k), c_k)$. We will transform this solution to a solution of Subset-Fixed-Flood-It of length at most $2k$, and then invoke Lemma 5.4 to obtain a solution for Fixed-Flood-It of length at most $2k$. More precisely, we will show that for any $G, \mathbf{col}, p$ we have $\mathrm{OPT}_{\text{S-Fixed}}(G, \mathbf{col}, p) \leq 2\mathrm{OPT}_{\text{S-Free}}(G, \mathbf{col}, p)$.

For a solution $\mathcal{S} = (S_1, c_1), (S_2, c_2), \ldots, (S_k, c_k)$ to Subset-Free-Flood-It we define the number of bad moves of $\mathcal{S}$ as $b(\mathcal{S}) = |\{(S_i, c_i) \mid p \notin S_i\}|$. We will somewhat more strongly prove the following statement for all $G, \mathbf{col}, p$: for any valid Subset-Free-Flood-It solution $\mathcal{S}$, we have

$$\mathrm{OPT}_{\text{S-Fixed}}(G, \mathbf{col}, p) \le |\mathcal{S}| + b(\mathcal{S}).$$

Since $|\mathcal{S}| + b(\mathcal{S}) \le 2|\mathcal{S}|$, the above statement will imply the promised inequality and the theorem.

We prove the statement by induction on $|\mathcal{S}| + 2b(\mathcal{S})$. If $|\mathcal{S}| + 2b(\mathcal{S}) \le 2$ then $\mathcal{S}$ is already a Subset-Fixed-Flood-It solution, so the statement is trivial. Suppose then that the statement holds when $|\mathcal{S}| + 2b(\mathcal{S}) \le n$ and we have a solution $\mathcal{S}$ with $|\mathcal{S}| + 2b(\mathcal{S}) = n + 1$. We consider the following cases:

- The first move (S_1, c_1) has $p \in S_1$. By the inductive hypothesis there is a Subset-Fixed-Flood-It solution of length at most $|\mathcal{S}| + b(\mathcal{S}) - 1$ for $(G, \mathbf{col}_1, p)$. We build a solution for Subset-Fixed-Flood-It by appending this solution to the move (S_1, c_1), since this is a valid move for Subset-Fixed-Flood-It.
- There exists a move (S_i, c_i) with $S_i = \mathbf{Comp}(\mathbf{col}_{i-1}, u)$, for some vertex u in $N(\mathbf{Comp}(\mathbf{col}_{i-1}, p)) \setminus \mathbf{Comp}(\mathbf{col}_{i-1}, p)$ such that $c_i = \mathbf{col}_{i-1}(p)$. That is, there exists a move that plays a vertex u that currently has a different color than p, and as a result of this move the component of u and p merge, because u receives the same color as p and u has a neighbor in the component of p.

  Consider the first such move. We build a solution $\mathcal{S}'$ as follows: we keep moves $(S_1, c_1) \ldots (S_{i-1}, c_{i-1})$; we add the moves $(\mathbf{Comp}(\mathbf{col}_{i-1}, p), \mathbf{col}_{i-1}(u))$, $(\mathbf{Comp}(\mathbf{col}_{i-1}, p) \cup \mathbf{Comp}(\mathbf{col}_{i-1}, u), \mathbf{col}_{i-1}(p))$; we append the rest of the previous solution $(S_{i+1}, c_{i+1}), \ldots$. To see that $\mathcal{S}'$ is still a valid solution we observe that $\mathbf{Comp}(\mathbf{col}_{i-1}, p) \cup \mathbf{Comp}(\mathbf{col}_{i-1}, u)$ is monochromatic and connected when we play it, and that the result of the first $i-1$ moves, plus the two new moves is exactly $\mathbf{col}_i$. We also note that $\mathcal{S}' + b(\mathcal{S}') = \mathcal{S} + b(\mathcal{S})$ because we replaced one bad move with two good moves. However, $\mathcal{S}' + 2b(\mathcal{S}') < \mathcal{S} + 2b(\mathcal{S})$, hence by the inductive hypothesis there exists a Subset-Fixed-Flood-It solution of the desired length.
- There does not exist a move as specified in the previous case. We then show that this reduces to the first case. If no move as described in the previous case exists and the initial coloring is not already constant, $\mathcal{S}$ must have a move (S_i, c_i) where $\{p\} \subseteq S_i \subseteq \mathbf{Comp}(\mathbf{col}_0, p)$ and $c_i = \mathbf{col}_{i-1}(u)$ for $u \in N(\mathbf{Comp}(\mathbf{col}_0, p)) \setminus \mathbf{Comp}(\mathbf{col}_0, p)$. In other words, this is a good move (it changes the color of p), that adds a new vertex u to the connected monochromatic component of p. Such a move must exist, since if the initial coloring is not constant, the initial component of p must be extended, and we assumed that no move that extends it by recoloring one of its neighbors exists.

Consider the first such good move (S_i, c_i) as described above. We build a solution $\mathcal{S}'$ as follows: the first move is $(\mathbf{Comp}(\mathbf{col}_0, p), \mathbf{col}_0(u))$, where u is, as described above, the neighbor of $\mathbf{Comp}(\mathbf{col}_0, p)$ with $\mathbf{col}_{i-1}(u) = c_i$. For $j \in [i-1]$ we add the move (S_j, c_j) if $u \notin S_j$, or the move $(\mathbf{Comp}(\mathbf{col}_{j-1}, u) \cup \mathbf{Comp}(\mathbf{col}_0, p), c_j)$ if $u \in S_j$. In other words, we keep other moves unchanged if they do not affect u, otherwise we add to them $\mathbf{Comp}(\mathbf{col}_0, p)$. We observe that these moves are valid since we maintain the invariant that $\mathbf{Comp}(\mathbf{col}_0, p)$ and u have the same color and since none of the first $i-1$ moves of $\mathcal{S}$ changes the color of p (since we selected the first such move). The result of these i moves is exactly $\mathbf{col}_i$. We now append the remaining move $(S_{i+1}, c_{i+1}), \ldots$, and we have a solution that starts with a good move, has the same length and the same (or smaller) number of bad moves as $\mathcal{S}$ and is still valid. We have therefore reduced this to the first case. ◀

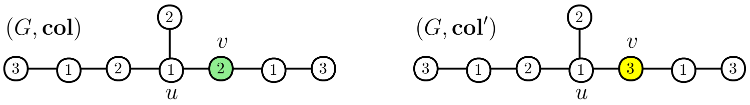


Figure 4 Non-monotonicity of Free-Flood-It.

6 Non-monotonicity of Free-Flood-It

As a final remark, we consider the (non-)monotonicity of the problem. A game has the *monotonicity property* if no legal move makes the situation worse. That is, if Fixed-Flood-It (or Free-Flood-It) has the monotonicity property, then no single move increases the minimum number of steps to make the input graph monotone. We believe that the monotonicity of Fixed-Flood-It was known as folklore and used implicitly in the literature. On the other hand, we are not sure that the non-monotonicity of Free-Flood-It was widely known. The only result we are aware of is by Meeks and Scott [12] who showed that on paths Free-Flood-It has the monotonicity property. Figure 4 shows that Free-Flood-It loses its monotonicity property as soon as the underlying graph becomes a path with one attached vertex. The instance $(G, \mathbf{col}')$ is obtained from $(G, \mathbf{col})$ by playing the move $(v, 3)$. We can show that $\mathrm{OPT}_{\mathrm{Free}}(G, \mathbf{col}) < \mathrm{OPT}_{\mathrm{Free}}(G, \mathbf{col}')$.

References

1 Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach. The complexity of flood filling games. *Theory Comput. Syst.*, 50(1):72–92, 2012. doi:10.1007/s00224-011-9339-2.

2 Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015. doi:10.1007/978-3-319-21275-3.

3 Michael R. Fellows, Uéverton dos Santos Souza, Fábio Protti, and Maise Dantas da Silva. Tractability and hardness of flood-filling games on trees. *Theor. Comput. Sci.*, 576:102–116, 2015. doi:10.1016/j.tcs.2015.02.008.

4 Michael R. Fellows, Fábio Protti, Frances A. Rosamond, Maise Dantas da Silva, and Uéverton dos Santos Souza. Algorithms, kernels and lower bounds for the Flood-It game parameterized by the vertex cover number. *Discrete Applied Mathematics*, 2017. in press. doi:10.1016/j.dam.2017.07.004.

5 Rudolf Fleischer and Gerhard J. Woeginger. An algorithmic analysis of the honey-bee game. *Theor. Comput. Sci.*, 452:75–87, 2012. doi:10.1016/j.tcs.2012.05.032.

6 Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

7 Hiroyuki Fukui, Yota Otachi, Ryuhei Uehara, Takeaki Uno, and Yushi Uno. On complexity of flooding games on graphs with interval representations. In Jin Akiyama, Mikio Kano, and Toshinori Sakai, editors, *Computational Geometry and Graphs - Thailand-Japan Joint Conference, TJJCCGG 2012, Bangkok, Thailand, December 6-8, 2012, Revised Selected Papers*, volume 8296 of *Lecture Notes in Computer Science*, pages 73–84. Springer, 2012. doi:10.1007/978-3-642-45281-9_7.

8 Jakub Gajarský, Michael Lampis, and Sebastian Ordyniak. Parameterized algorithms for modular-width. In Gregory Z. Gutin and Stefan Szeider, editors, *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4-6, 2013, Revised Selected Papers*, volume 8246 of *Lecture Notes in Computer Science*, pages 163–176. Springer, 2013. doi:10.1007/978-3-319-03898-8_15.

9 Wing-Kai Hon, Ton Kloks, Fu-Hong Liu, Hsiang Hsuan Liu, and Hung-Lung Wang. Flood-it on AT-free graphs. *CoRR*, abs/1511.01806, 2015. arXiv:1511.01806.

10 Aurélie Lagoutte, Mathilde Noual, and Eric Thierry. Flooding games on graphs. *Discrete Applied Mathematics*, 164:532–538, 2014. doi:10.1016/j.dam.2013.09.024.

11 Michael Lampis. Algorithmic meta-theorems for restrictions of treewidth. *Algorithmica*, 64(1):19–37, 2012.

12 Kitty Meeks and Alexander Scott. The complexity of flood-filling games on graphs. *Discrete Applied Mathematics*, 160(7-8):959–969, 2012. doi:10.1016/j.dam.2011.09.001.

13 Kitty Meeks and Alexander Scott. The complexity of free-flood-it on 2×n boards. *Theor. Comput. Sci.*, 500:25–43, 2013. doi:10.1016/j.tcs.2013.06.010.

14 Kitty Meeks and Alexander Scott. Spanning trees and the complexity of flood-filling games. *Theory Comput. Syst.*, 54(4):731–753, 2014. doi:10.1007/s00224-013-9482-z.

15 Kitty Meeks and Dominik K. Vu. Extremal properties of flood-filling games. *CoRR*, abs/1504.00596, 2015. arXiv:1504.00596.

16 Elad Verbin. Comment to “Is this game NP-Hard? by Sariel Har-Peled. http://sarielhp.org/blog/?p=2005#comment-993, 2009. Accessed: 2018-01-18.

17 Uéverton ßdos Santos Souza, Fábio Protti, and Maise Dantas da Silva. An algorithmic analysis of Flood-it and Free-Flood-it on graph powers. *Discrete Mathematics & Theoretical Computer Science*, 16(3):279–290, 2014. URL: http://dmtcs.episciences.org/2086.

How long does it take for all users in a social network to choose their communities?

Jean-Claude Bermond
Université Côte d'Azur, CNRS, Inria, I3S, France

Augustin Chaintreau
Columbia University in the City of New York, USA

Guillaume Ducoffe[1]
National Institute for Research and Development in Informatics and Research Institute of the University of Bucharest, Bucureşti, România

Dorian Mazauric[2]
Université Côte d'Azur, Inria, France

Abstract

We consider a community formation problem in social networks, where the users are either friends or enemies. The users are partitioned into conflict-free groups (*i.e.*, independent sets in the *conflict graph* $G^- = (V, E)$ that represents the enmities between users). The dynamics goes on as long as there exists any set of at most k users, k being any fixed parameter, that can change their current groups in the partition *simultaneously*, in such a way that they all strictly increase their utilities (number of friends *i.e.*, the cardinality of their respective groups minus one). Previously, the best-known upper-bounds on the maximum time of convergence were $\mathcal{O}(|V|\alpha(G^-))$ for $k \leq 2$ and $\mathcal{O}(|V|^3)$ for $k = 3$, with $\alpha(G^-)$ being the independence number of G^-. Our first contribution in this paper consists in reinterpreting the initial problem as the study of a dominance ordering over the vectors of integer partitions. With this approach, we obtain for $k \leq 2$ the tight upper-bound $\mathcal{O}(|V| \min\{\alpha(G^-), \sqrt{|V|}\})$ and, when G^- is the empty graph, the exact value of order $\frac{(2|V|)^{3/2}}{3}$. The time of convergence, for any fixed $k \geq 4$, was conjectured to be polynomial [7, 14]. In this paper we disprove this. Specifically, we prove that for any $k \geq 4$, the maximum time of convergence is an $\Omega(|V|^{\Theta(\log |V|)})$.

2012 ACM Subject Classification Networks, Theory of computation

Keywords and phrases communities, social networks, integer partitions, coloring games, graphs

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.6

Related Version The full version of the paper can be found at `https://hal.inria.fr/hal-01780627`.

[1] Part of this work has been done as PhD student in the project Coati at Université Côte d'Azur and during visits at Columbia University in the City of New York. This work was also supported by the Institutional research programme PN 1819 "Advanced IT resources to support digital transformation processes in the economy and society - RESINFO-TD" (2018), project PN 1819-01-01"Modeling, simulation, optimization of complex systems and decision support in new areas of IT&C research", funded by the Ministry of Research and Innovation, Romania.

[2] Part of this work has been done during his post-doc at Columbia University in the City of New York.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 6; pp. 6:1–6:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Foreword: The organizers of a wedding (party) have difficulties in arranging place settings for the guests as there are many incompatibilities among those who do not want to be at the same table as an "enemy" (ex girl (boy) friend, boss or employee, student or supervisor, etc...). The organizers realize that they have no set of 5 pairwise friends and so allow people place themselves. Successively each person joins a table where she has no enemies or starts a new table. At any time a person can move from one table to another table (of course where she has no enemy) if in doing so she increases strictly the number of friends she has at the new table. The process converges relatively fast (linear time). Some time later the organizers of FUN having heard about this scenario decide to use the same process to place the participants in different groups for the social activities of the afternoon. Each participant registers first in her own group. The organizers decide to accelerate the process by authorizing not just one person but any subset of 4 persons to change their mind and leave the group in which they are registered to join another group or create a new group; these persons move only if they desire to do so, that is, they increase strictly the number of friends. Surprisingly the process takes a very long (exponential) time and night arrives before groups are formed. As we will discover, the exponential time derives from the fact that at FUN all the persons are friends and there are no enemies due to the use of moves implying 4 persons. At this point the reader (and the organizers) might ask why we see such a difference in behaviors and how long does it takes for users of a social network to form groups. The answers to these questions and "all you wanted to know but were afraid to ask" will be revealed in this paper.

1 Introduction

Community formation is a fundamental problem in social network analysis. It has already been modeled in several ways, each trying to capture key aspects of the problem. The model studied in this paper has been proposed in [14] in order to reflect the impact of information sharing on the community formation process. Although it is a simplified model, we show that its understanding requires us to solve combinatorial problems that are surprisingly intricate. More precisely, we consider the following dynamics of formation of groups (communities) in social networks. Each group represents a set of users sharing about some information topic. We assume for simplicity that each user shares about a given topic in only one group. Therefore the groups will partition the set of users. We follow the approach of [14]. An important feature is the emphasis on incompatibility between some pairs of users that we will call enemies. Two enemies do not want to share information and so will necessarily belong to different groups. In the general model one consider different degrees of friendship or incompatibilities. Here we will restrict to the case where two users are either friends or enemies – as noted in [14], even a little beyond this case, the problem quickly becomes intractable. As an example, if we add a neutral (indifference) relation, there are instances for which there is no stability.

The social network is often modeled by the friendship graph G^+ where the vertices are the users and an edge represents a friendship relation. We will use this graph to present the first notions and examples. However, for the rest of the article and the proofs we will use the complementary graph, that we call the *conflict graph* and denote by G^-; here the vertices represent users and the edges represent the incompatibility relation. We assign each user a *utility* which is the number of friends in the group to which she belongs. Equivalently, the utility is the size of the group minus one, as in a group there is no pair of enemies; in [14] this is modeled by putting the utility as $-\infty$ when there is an enemy in the group.

In the example of Figure 1, the graph depicted is the friendship graph: the edges represent the friendship relation, and if there is no edge it corresponds to a pair of enemies. Figure 1(a)

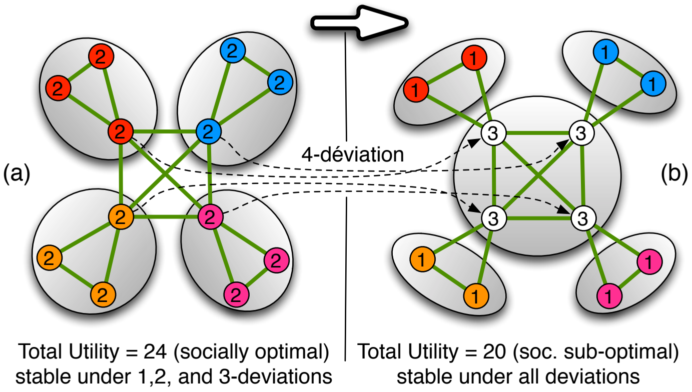

(a)
(b)
4-déviation
2
3
1
Total Utility = 24 (socially optimal)
stable under 1,2, and 3-deviations
Total Utility = 20 (soc. sub-optimal)
stable under all deviations

Figure 1 A friendship graph with 12 vertices (users). **(a)** 3-stable partition that is not 4-stable but it is optimal in terms of total utility. **(b)** k-stable partition for any $k \geq 1$ that is not optimal in terms of total utility.

depicts a partition of 12 users composed of 4 non-empty groups each of size 3. The integers on the vertices represent the utilities of the users which are all equal to 2. Figure 1(b) depicts another partition consisting of 5 groups with one group of size 4 (where users have utility 3) and 4 groups of size 2 (where users have utility 1).

In this study we are interested in the dynamics of formation of groups. Another important feature of [14], taken into account in the dynamics, is the notion of bounded cooperation between users. More precisely, the dynamics is as follows: initially each user is alone in her own group. In the simplest case, a move consists for a specific user to leave the group to which she belongs to join another group but only if this action increases strictly her utility (acting in a selfish manner); in particular, it implies that a user does not join a group where she has an enemy. In the k-bounded mode of cooperation, a set of at most k-users can leave their respective groups to join another group, again, only if each user increases strictly their utility. If the group they join is empty it corresponds to creating a new group. We call such a move a *k-deviation*. Note that this notion is slightly different from that of $(k+1)$-defection of [14]. We will say that *a partition is k-stable* if there does not exist a k-deviation for this partition.

The partition of Figure 1(a) is k-stable when $k \in \{1, 2, 3\}$. Indeed each user has at least one enemy in each non empty other group and so cannot join another group. Furthermore, when $k \leq 3$, if k users join an empty group their utility will be at most 2 and so will not strictly increase. However, this partition is not 4-stable because there is a 4-deviation: the four central users can join an empty group and so they increase their utilities from 2 to 3. The partition obtained after such a 4-deviation is depicted in Figure 1(b). This partition is k-stable for any $k \geq 1$. Note that the utility of the other users is now 1 (instead of 2). Thus, we deduce that this partition is not optimal in terms of total utility (the total utility has decreased from 24 to 20); but it is now stable under all deviations. This illustrates the fact that users act in a selfish manner as some increase their utility, but on the contrary the global utility decreases. For more information on the suboptimality of k-stable partitions, *i.e.*, bounds on the price of anarchy and the price of stability, the reader is referred to [14].

1.1 Related work

This above dynamics has been also modeled in the literature with *coloring games.* A coloring game is played on the conflict graph. Players must choose a color in order to construct a proper coloring of the graph, and the individual goal of each agent is to maximize the number of agents with the same color as she has. On a more theoretical side, coloring games have been introduced in [18] as a game-theoretic setting for studying the chromatic number in graphs. Specifically, the authors in [18] have shown that for every coloring game, there exists a Nash equilibrium where the number of colors is exactly the chromatic number of the graph. Since then, these games have been used many times, attracting attention in the study of information sharing and propagation in graphs [4, 7, 14]. Coloring games are an important subclass of the more general Hedonic games, of which several variations have been studied in the literature in order to model coalition formation under selfish preferences of the agents [10, 12, 15, 5, 8, 16]. We stress that while every coloring game has a Nash equilibrium that can be computed in polynomial-time [18], deciding whether a given Hedonic game admits a Nash equilibrium is NP-complete [1].

If the set of edges of the conflict graph is empty (edgeless conflict graph), there exists a unique k-stable partition, namely, that consists of the group of all the users. In [14], it is proved that there always exists a k-stable partition for any conflict graph, but that it is NP-hard to compute one if k is part of the input (this result was also proved independently in [7]). Indeed, if k is equal to the number of users, a largest group in such a partition must be a maximum independent set of the conflict graph. In contrast, it can be computed a k-stable partition in polynomial time for every fixed $k \leq 3$, by using simple *better-response dynamics* [18, 7, 14]. In such an algorithm one does a k-deviation until there does not exist any one. That corresponds to the dynamics of formation of groups that we study in this work for larger values of k.

1.2 Additional related work and our results

In this paper we are interested in analyzing in this simple model the convergence of the dynamics with k-deviations, in particular in the worst case. It has been proved implicitly in [14] that the dynamics always converges within at most $\mathcal{O}(2^n)$ steps. Let $L(k, G^-)$ be the size of a longest sequence of k-deviations on a conflict graph G^-. We first observe that the maximum value, denoted $L(k, n)$, of $L(k, G^-)$ over all the graphs with n vertices is attained on the edgeless conflict graph $G^\emptyset$ of order n. Prior to this work, no lower bound on $L(k, n)$ was known, and the analysis was limited to potential function that only applies when $k \leq 3$ [7, 14] giving upper bounds of $O(n^2)$ in the case $k = 1, 2$ and $O(n^3)$ in the case $k = 3$. In order to go further in our analysis, the key observation is that when the conflict graph is edgeless, the dynamics depends only of the size of the groups of the partitions generated. Following [3], let an integer partition of $n \geq 1$, be a non-increasing sequence of integers $Q = (q_1, q_2, \ldots, q_n)$ such that $q_1 \geq q_2 \geq \ldots \geq q_n \geq 0$ and $\sum_{i=1}^{n} q_i = n$. If we rank the groups by non increasing order of their size, there is a natural relation between partition in groups and integer partitions (the size q_i of the group X_i corresponding to the integers q_i of the partition of n). Using this relation, we prove in Section 3 that the better response dynamics algorithm reaches a stable partition in p_n steps, where $p_n = \Theta((e^{\pi\sqrt{2n/3}})/n)$ denotes the number of integer partitions. This is already far less than 2^n, which was shown to be the best upper bound that one can obtain for $k \geq 4$ when using an additive potential function [14].

Table 1 Previous bounds and results we obtained on $L(k,n)$ and $L(k,G^-)$.

k	Prior to our work	**Our results**	
1	$\mathcal{O}(n^2)$ [14]	exact analysis, which implies $L(1,n) \sim \frac{(2n)^{3/2}}{3}$	Theorem 6
2	$\mathcal{O}(n^2)$ [14]	exact analysis, which implies $L(2,n) \sim \frac{(2n)^{3/2}}{3}$	Theorem 9
1-2	$\mathcal{O}(n\alpha(G^-))$ [18]	$L(k,G^-) = \Omega(n\alpha(G^-))$ for some G^- and $\alpha(G^-) = \mathcal{O}(\sqrt{n})$	Theorem 12
3	$\mathcal{O}(n^3)$ [7, 14]	$L(3,n) = \Omega(n^2)$	Theorem 13
≥ 4	$\mathcal{O}(2^n)$ [14]	$L(k,n) = \Omega(n^{\Theta(\ln(n))})$, $L(k,n) = \mathcal{O}(\exp(\pi\sqrt{2n/3})/n)$	Theorem 14

Table 1 summarizes our contributions described below.

- For $k = 1, 2$, we refine the relation between partitions into groups and integer partitions as follows.
  - In the case $k = 1$ (Section 4.1), we prove that there is a one to one mapping between sequences of 1-deviations in the edgeless conflict graph and chains in the dominance lattice of integer partitions. Then, we use the value of the longest chain in this dominance lattice obtained in [9] to determine exactly $L(1,n)$. More precisely, if $n = \frac{m(m+1)}{2} + r$, with $0 \leq r \leq m$, $L(1,n) = 2\binom{m+1}{3} + mr$. The latter implies in particular $L(1,n)$ is of order $\mathcal{O}(n^{\frac{3}{2}})$, thereby improving the previous bound $\mathcal{O}(n^2)$.
  - In Section 4.2, we prove that any 2-deviation can be "replaced" (in some precise way) either by one or two 1-deviations, and so, $L(2,n) = L(1,n)$.
  - For $k = 1, 2$ and a general conflict graph G^-, the value of $L(k,G^-)$ depends on the independence number $\alpha(G^-)$ (cardinality of a largest independent set) of the conflict graph. In [18] it was proved that the convergence of the dynamics is in $\mathcal{O}(n\alpha(G^-))$. In the case of edgeless conflict graph, we have seen that $L(1,n) = \mathcal{O}(n^{3/2})$ and so the preceding upper-bound was not tight. So we inferred that the convergence of the dynamics was in $\mathcal{O}(n\sqrt{\alpha(G^-)})$. Yet in fact we prove in Section 4.3 that, for any $\alpha(G^-) = \mathcal{O}(\sqrt{n})$, there exists a conflict graph G^- with n vertices and independence number $\alpha(G^-)$ for which we need a sequence of at least $\Omega(n\alpha(G^-))$ 1-deviations to reach a stable partition. *For the wedding's example of the foreword, $\alpha(G^-) = 4$ and so the sequence is linearly bounded.*
- Finally, our main contribution is obtained for $k \geq 3$. Prior to our work, it was known that $L(3,n) = \mathcal{O}(n^3)$, which follows from another application of the potential function method [14]. But nothing proved that $L(3,n) > L(2,n)$, and in fact it was conjectured in [7] that both values are equal. In Section 5, we prove (Theorem 13) that $L(3,n) = \Omega(n^2)$ and thus we show for the first time that deviations can delay convergence and that the gap between $k = 2$ and $k = 3$ obtained from potential function is indeed justified. It was also conjectured in [14] that $L(k,n)$ was polynomial in n for k fixed. In Section 5.1 we disprove this conjecture and prove in Theorem 14 that $L(4,n) = \Omega(n^{\Theta(\ln(n))})$. This shows that 4-deviations are responsible for a sudden complexity increase, as no polynomial bounds exist for $L(4,n)$. *This explains why in the foreword it takes an exponential time for the organizers of FUN to schedule the groups.*

2 Notations

Conflict graph. We refer to [2] for standard graph terminology. For the remaining of the paper, we suppose that we are given a *conflict graph* $G^- = (V,E)$ where V is the set of vertices (called users or players in the introduction) and edges represent the incompatibility relation (*i.e.*, an edge means that the two users are enemies). The number of vertices is denoted by $n = |V|$. The independence number of G^-, denoted $\alpha(G^-)$, is the maximum cardinality of an independent set in G^-. In particular, if $\alpha(G^-) = n$ then the conflict graph is edgeless and we denote it by $G^\emptyset = (V, E = \emptyset)$ and call it the empty graph.

Partitions and utilities. We consider any partition $P = X_1, \ldots, X_i, \ldots, X_n$ of the vertices into n independent sets X_i called groups (colors in coloring games), with some of them being possibly empty. In particular, two enemies are not in the same group. We rank the groups by non increasing size, that is $|X_i| \geq |X_{i+1}|$. For any $1 \leq i \leq n$ and for any $v \in X_i$, the *utility* of vertex v is the number of other vertices in the same group as it, that is $|X_i| - 1$.

We use in our proofs two alternative representations of the partition P. The *partition vector* associated to P is defined as $\overrightarrow{\Lambda}(P) = (\lambda_n(P), \ldots, \lambda_1(P))$, where $\lambda_i(P)$ is the number of groups of size i. The *integer partition* associated to P is defined as $Q = (q_1, q_2, \ldots, q_n)$ such that $q_1 \geq q_2 \geq \ldots \geq q_n \geq 0$ and $\sum_{i=1}^{n} q_i = n$, where $q_i = |Xi|$.

In the example of Figure 1(a) we have a partition P of the 12 vertices into 4 groups each of size 3 and so $\lambda_3(P) = 4$ and $\lambda_i(P) = 0$ for $i \neq 3$; in other words $\overrightarrow{\Lambda}(P) = (0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 0, 0)$. The corresponding integer partition is $Q(P) = (3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0)$. In the example of Figure 1(b) we have a partition P' of the 12 vertices into one group of size 4 and 4 groups each of size 2 and so $\lambda_4(P') = 1$, $\lambda_2(P') = 4$ and $\lambda_i(P') = 0$ for $i \neq 2, 4$; in other words $\overrightarrow{\Lambda}(P) = (0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 4, 0)$. The corresponding integer partition is $Q(P') = (4, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0)$.

k-deviations and k-stability. We can think of a k-deviation as a move of at most k vertices which leave the groups to which they belong in P, to join another group (or create a new group) with the necessary condition that each vertex strictly increases its utility, thereby leading to a new partition P'. A *k-stable partition* is simply a partition for which there exists no k-deviation. We write $L(k, G^-)$, resp. $L(k, n)$, for the length of a longest sequence of k-deviations to reach a stable partition in G^-, resp. in any conflict graph with n vertices. Recall that we start with the partition consisting of n groups of size 1, that is, $\overrightarrow{\Lambda}(P) = (\ldots, 0, 0, 0, n)$.

We next define a natural vector representation for k-deviations. The *difference vector* $\overrightarrow{\varphi}$ associated to a k-deviation φ from P to P' is equal to $\overrightarrow{\varphi} = \overrightarrow{\Lambda}(P') - \overrightarrow{\Lambda}(P)$. In concluding this section, we define the difference vectors for some of the k-deviations used in our proofs:

- $\overrightarrow{\alpha}[p, q]$, the 1-deviation where a vertex leaves a group of size $q + 1$ for a group of size $p - 1$ (valid when $p \geq q + 2$). In that case $\alpha_p = 1, \alpha_{p-1} = -1, \alpha_{q+1} = -1, \alpha_q = -1$, and $\alpha_i = 0$ for any $i \notin \{q, q + 1, p - 1, p\}$ (we omit for ease of reading the brackets $[p, q]$).
- $\overrightarrow{\gamma}[p]$, the 3-deviation where one vertex in each of 3 groups of size $p - 1$ moves to a group of size $p - 3$ to form a new group of size p (valid if there are at least 3 groups of size $p - 1$ and one of size $p - 3$). In that case $\gamma_p = 1, \gamma_{p-1} = -3, \gamma_{p-2} = 3, \gamma_{p-3} = -1$, and $\gamma_i = 0$ for any $i \notin \{p - 3, p - 2, p - 1, p\}$.
- $\overrightarrow{\delta}[p]$, the 4-deviation where one vertex in each of 4 groups of size $p - 1$ moves to a group of size $p - 4$ to form a new group of size p (valid if there are at least 4 groups of size $p - 1$ and one of size $p - 4$). In that case $\delta_p = 1, \delta_{p-1} = -4, \delta_{p-2} = 4, \delta_{p-4} = -1$, and $\delta_i = 0$ for any $i \notin \{p - 4, p - 2, p - 1, p\}$. As an example, the move from the partition of Figure 1(a) to the partition of Figure 1(b), is a 4-deviation with difference vector $\overrightarrow{\delta[4]}$.

Algorithm 1 Dynamics of the system

Input: a positive integer $k \geq 1$, and a conflict graph G^-.
Output: a k-stable partition for G^-.

```
Let $P_1$ be the partition composed of $n$ singletons groups.
Set $i = 1$.
while there exists a $k$-deviation for $P_i$ do
    Set $i = i + 1$.
    Choose one $k$-deviation and compute the partition $P_i$ after this $k$-deviation.
Return the partition $P_i$.
```

3 Preliminary results

In [14], the authors prove that there always exists a k-stable partition, but that it is NP-hard to compute one if k is part of the input (this result was also proved independently in [7]). In contrast, it can be computed a k-stable partition in polynomial time for every fixed $k \leq 3$, by using simple *better-response dynamics* [18, 7, 14]. The latter results question the role of the value of k in the complexity of computing stable partitions.

Formally, a better-response dynamics proceeds as follows. We start from the trivial partition P_1 consisting of n groups with one vertex in each of them. In particular, the partition vector $\overrightarrow{\Lambda}(P_1)$ is such that $\lambda_1(P_1) = n$ and, for all other $j \neq 1$, $\lambda_j(P_1) = 0$. Provided there exists a k-deviation with respect to the current partition P_i, we pick any one of these k-deviations φ and in so doing we obtain a new partition P_{i+1}. If there is no k-deviation, the partition P_i is k-stable. An algorithmic presentation is given in Algorithm 1.

We now prove in Proposition 1 that better-response dynamics can be used for computing a k-stable partition *for every fixed* $k \geq 1$ (but not necessarily in polynomial time). It shows that for every fixed $k \geq 1$, the problem of computing a k-stable partition is in the complexity class PLS (Polynomial Local Search), that is conjectured to lie strictly between P and NP [13]. Recall that the problem becomes NP-hard when k is part of the input.

▶ **Proposition 1.** *For any $k \geq 1$, for any conflict graph G^-, Algorithm 1 converges to a k-stable partition.*

Proof. Let P_i, P_{i+1} be two partitions for G^- such that P_{i+1} is obtained from P_i after some k-deviation φ. Let S be the set of vertices which move ($|S| \leq k$) and let j be the size of the group they join ($j = 0$ if they create a new group). Then, the new group obtained has size $p = j + |S|$. Note that all the vertices of S have increased their utilities and so, they belonged in P_i to groups of size $< p$. Therefore, the coordinates of the difference vector $\overrightarrow{\varphi}$ satisfy $\varphi_p = 1$ and $\varphi_j = 0$ for $j > p$, and so $\overrightarrow{\Lambda}(P_i) <_L \overrightarrow{\Lambda}(P_{i+1})$ where $<_L$ is the lexicographical ordering. Finally, as the number of possible partition vectors is finite, we obtain the convergence of Algorithm 1. ◀

An instrumental observation for our next proofs is the following:

▶ **Observation 1.** *$L(k, n)$ is always attained on the empty conflict graph $G^\emptyset$ of order n.*

Indeed, any sequence of k-deviations on a conflict graph G^- is also a sequence in the empty conflict graph with the same vertices. Note that the converse is not true as it can happen that some moves allowed in the empty conflict graph are not allowed in G^- as they bring two enemies in the same group.

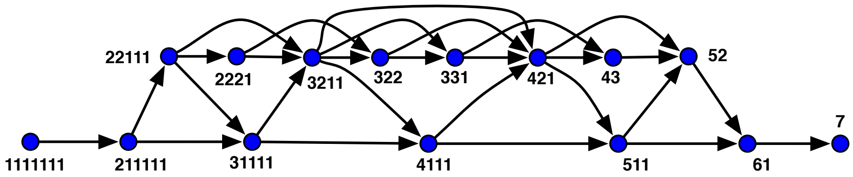


Figure 2 The lattice of integer partitions for $n = 7$.

Recall that we can associate to any partition $P = X_1, \ldots, X_i, \ldots, X_n$ of the vertices the integer partition $Q = (q_1, q_2, \ldots, q_n)$ such that $q_1 \geq q_2 \geq \ldots \geq q_n \geq 0$ and $\sum_{i=1}^{n} q_i = n$ by letting $q_i = |X_i|$. The converse is not true in general; as an example it suffices to consider a partition with $q_1 > \alpha(G^-)$. However the converse is true when the conflict graph is empty; indeed it suffices to associate to an integer partition any partition of the vertices obtained by putting in the group X_i a set of q_i vertices .

We can now use the value p_n of the number of integer partitions (see [11]) to obtain the following proposition which follows from Proposition 1.

▶ **Proposition 2.** *Algorithm 1 reaches a stable partition in at most* $p_n = \Theta((e^{\pi\sqrt{\frac{2n}{3}}})/n)$ *steps.*

Note that this is already far less than 2^n, which was shown to be the best upper bound that one can obtain for $k \geq 4$ when using an additive potential function [14].

4 Analysis for k $\leq$ 2

In [14], the authors proved that for $k \leq 2$, Algorithm 1 converges to a stable partition in at most a quadratic time (by using a potential function). Indeed when performing a 1-deviation $\overrightarrow{\alpha}[p, q]$, a vertex moves from a group of size $q + 1$ to a group of size $p - 1$ (with $p \geq q + 2$); the utility of this vertex increases by $p - q - 1$, the utility of the q other vertices of the group of size $q + 1$ decreases by 1, while the utility of the vertices of the group of size $p - 1$ increases by 1. So the global utility increases by $2p - 2q - 2 \geq 2$ as $p \geq q + 2$. Furthermore, in a k-stable partition, the utility of a vertex is at most $n - 1$ and the global utility is at most $n(n - 1)/2$. As a result, $L(k, n) = O(n^2)$.

In the next subsections we improve this result as we completely solve this case and give the exact (non-asymptotic) value of $L(k, n)$ when $k \leq 2$. The gist of the proof is to use a partial ordering that was introduced in [3], and is sometimes called the dominance ordering.

4.1 Exact analysis for k = 1 and empty conflict graph

In [3] the author has defined an ordering over the integer partitions, sometimes called the dominance ordering which creates a lattice of integer partitions. This ordering is a direct application of the theory of majorization to integer partitions [17].

▶ **Definition 3.** (dominance ordering) Given two integer partitions of $n \geq 1$, $Q = (q_1, q_2, \ldots, q_n)$ and $Q' = (q'_1, q'_2, \ldots, q'_n)$, we say that Q' dominates Q if $\sum_{j=1}^{i} q'_j \geq \sum_{j=1}^{i} q_j$, for all $1 \leq i \leq n$.

The example of Figure 2 shows the dominance lattice for $n = 7$. We did not write in the figure the integers equal to 0.

The two next lemmas show that there is a one to one mapping between chains in the dominance lattice and sequences of 1-deviations in the empty conflict graph.

▶ **Lemma 4.** *Let P be a partition of the vertices and P' be the partition obtained after a 1-deviation φ. Then, the integer partition $Q' = Q(P')$ dominates $Q = Q(P)$.*

Proof. In the 1-deviation φ a vertex v moves from a group X_k to a group X_j with $q_j = |X_j| \geq q_k = |X_k|$. W.l.o.g. we can suppose that the groups (ranked in non increasing order of size) are ranked in a such a way that X_j is the first group with size $|X_j|$ and X_k the last group with size $|X_k|$. Thus, the integer partition $Q(P)$ associated to P satisfies $q_1 \geq q_2 \ldots \geq q_{j-1} > q_j \geq q_{j+1} \ldots \geq q_k > q_{k+1} \ldots \geq q_n$. After the move the groups of P' are the same as those of P except we have replaced X_j with the group $X_j \cup v$ and X_k with $X_k - v$. Therefore the integer partition Q' associated to P' has the same elements as Q except $q'_j = q_j + 1$ and $q'_k = q_k - 1$ and so, Q' dominates Q. Note that this lemma holds for any conflict graph. ◀

In the case $n = 7$, consider the partition P with one group of size 3, one of size 2 and two of size 1. The integer partition associated to P is $Q = (3, 2, 1, 1, 0, 0, 0)$. Let φ be the 1-deviation where a vertex in the group of size 1 moves to the group of size 3. We obtain the partition P' with one group of size 4, one of size 2 and one of size 1. The integer partition associated to P' is $Q' = (4, 2, 1, 0, 0, 0, 0)$ which dominates Q.

▶ **Lemma 5.** *Let $G^{\emptyset}$ be the empty conflict graph and let Q, Q' be two integer partitions of $n = |V|$ such that Q' dominates Q. For any partition P associated to Q, there exists another partition P' associated to Q' such that P' is obtained from P by doing a sequence of 1-deviations.*

Proof. As proved in [3], we have that if Q' dominates Q then there is a finite sequence of integer partitions $Q^0, \ldots, Q^r, \ldots, Q^s$,with $Q = Q^0$ and $Q' = Q^s$ such that for each $0 \leq r < s$, Q^{r+1} dominates Q^r and differs from it only in two elements j_r and k_r with $q^{r+1}_{j_r} = q^r_{j_r} + 1$ and $q^{r+1}_{k_r} = q^r_{k_r} - 1$.

The proof is now by induction on r, starting from any partition $P^0 = P$ associated to Q. For $r > 0$, we consider the partition P^r associated to Q^r. Recall that Q^r and Q^{r+1} differ only in the two groups X_{j_r} and X_{k_r}. As $q^{r+1}_{j_r} = q^r_{j_r} + 1$ and $q^{r+1}_{k_r} = q^r_{k_r} - 1$, P^{r+1} can be obtained from P^r by moving a vertex from X_{k_r} to X_{j_r}. This move is valid as the conflict graph is empty. (Note that the lemma is not valid for a general conflict graph.) ◀

As an example, consider the two integer partitions $Q = (2, 2, 2, 1, 0, 0, 0)$ and $Q' = (5, 1, 1, 0, 0, 0, 0)$ where Q' dominates Q. The sequence of integer partitions is $Q_0 = Q$, $Q_1 = (3, 2, 1, 1, 0, 0, 0)$, $Q_2 = (4, 1, 1, 1, 0, 0, 0)$, $Q_3 = (5, 1, 1, 0, 0, 0, 0)$. Partition P^1 is obtained from P^0 by moving a vertex of a group of size 2 to another group of size 2. Then, P^2 is obtained by moving a vertex of the group of size 2 to the group of size 3 and P' is obtained from P^2 by moving a vertex of one group of size 1 to that of size 4.

In summary we conclude that a sequence of 1-deviations with an empty conflict graph corresponds to a chain of integer partitions, and vice versa. Therefore, by Observation 1, the length of the longest sequence of 1-deviations with an empty conflict graph is the same as the length of the longest chain in the dominance lattice of integer partitions. Since it has been proven in [9] that for $n = \frac{m(m+1)}{2} + r$, the longest chain in the Dominance Lattice has length $2\binom{m+1}{3} + mr$, we obtain the *exact* value for $L(1, n)$.

▶ **Theorem 6.** *Let m and r be the unique non negative integers such that $n = \frac{m(m+1)}{2} + r$, and $0 \leq r \leq m$. Then, $L(1, n) = 2\binom{m+1}{3} + mr$.*

We note that the proof in [9] is not straightforward. One can think that the longest chain is obtained by taking among the possible 1-deviations the one which leads to the smallest partition in the lexicographic order. Unfortunately this is not true. Indeed let $n = 9$. After 6 steps we get the integer partition $(3, 3, 2, 1, 0, 0, 0, 0, 0)$. Then, by choosing the 1-deviation that gives the smallest partition (in the lexicographic order), we get the partition $(3, 3, 3, 0, 0, 0, 0, 0, 0)$ and then $(4, 3, 2, 0, 0, 0, 0, 0, 0)$. But there is a longer chain of length 3 from $(3, 3, 2, 1, 0, 0, 0, 0, 0)$ to $(4, 3, 2, 0, 0, 0, 0, 0, 0)$, namely, $(4, 2, 2, 1, 0, 0, 0, 0, 0)$, $(4, 3, 1, 1, 0, 0, 0, 0, 0)$, $(4, 3, 2, 0, 0, 0, 0, 0, 0)$. However the proof in [9] implies that the following simple construction works for any n. (see the full version).

▶ **Proposition 7.** *A longest sequence of* 1*-deviations in the empty conflict graph is obtained by choosing, at a given step, among all the possible* 1*-deviations, any one of which leads to the smallest increase of the global utility.*

4.2 Analysis for k = 2

Interestingly we will prove that any 2-deviation can be replaced either by one or two 1-deviations and so, we will prove in Theorem 9 that $L(2, n) = L(1, n)$.

▶ **Claim 8.** *If the conflict graph G^- is empty, then any* 2*-deviation can be replaced either by one or two* 1*-deviations*

Proof. Consider a 2-deviation which is not a 1-deviation. In that case case two vertices u_i and u_j leave their respective group X_i and X_j (which can be the same) to join a group X_k. Let $|X_i| \geq |X_j|$; in order for the utility of the vertices to increase, we should have $|X_k| \geq |X_i| - 1$ $(\geq |X_j| - 1)$.

- Case 1: $|X_k| \geq |X_j|$. In that case the 2-deviation can be replaced by a sequence of two 1-deviations where firstly a vertex u_j leaves X_j to join X_k and then a vertex u_i leaves X_i to join the group $X_k \cup u_j$ whose size is now at least that of X_i.
- Case 2: $|X_k| = |X_i| - 1 = |X_j| - 1 = p - 2$ and $X_i = X_j$. In that case the effect of the 2-deviation is to replace the group X_i of size $p - 1$ with a group of size $p - 3$ and to replace the group X_k of size $p - 2$ with a group of size p. Said otherwise, the difference vector $\overrightarrow{\varphi}$ associated to the 2-deviation has as non null coordinates $\varphi_p = 1, \varphi_{p-1} = -1, \varphi_{p-2} = -1, \varphi_{p-3} = 1$. We obtain the same effect by doing the 1-deviation $\overrightarrow{\alpha}[p - 1, p - 2]$ where a vertex leaves X_k to join X_i.
- Case 3: $|X_k| = |X_i| - 1 = |X_j| - 1 = p - 2$ and $X_i \neq X_j$. In that case the effect of the 2-deviation is to replace the 2 groups X_i and X_j of size $p - 1$ with two groups of size $p - 2$ and to replace the group X_k of size $p - 2$ with a group of size p. Said otherwise, the difference vector $\overrightarrow{\varphi}$ associated to the 2-deviation has as non null coordinates $\varphi_p = 1, \varphi_{p-1} = -2, \varphi_{p-2} = 1$. We obtain the same effect by doing the 1-deviation $\overrightarrow{\alpha}[p - 1, p - 1]$ where a vertex leaves X_j to join X_i.

Note that the fact that G^- is empty is needed for the proof. Indeed, in the case 2, it might happen that all the vertices of X_k have some enemy in X_i and so, the 1-deviation we describe is not valid. Similarly, in case 3, it might happen that all the vertices of X_i have some enemy in X_j and so, the 1-deviation we describe is not valid. ◀

▶ **Theorem 9.** $L(2, n) = L(1, n)$.

Proof. Clearly, $L(2, n) \geq L(1, n)$ as any 1-deviation is also a 2-deviation. By Observation 1, the value of $L(2, n)$ is obtained when the conflict graph G^- is empty. In that case, Claim 8 implies that $L(2, n) \leq L(1, n)$. ◀

4.3 Analysis for k $\leq$ 2 and a general conflict graph

Using the potential function introduced at the beginning of this section, Panagopoulou and Spirakis ([18]) proved that for every conflict graph G^- with independence number $\alpha(G^-)$, the convergence of the dynamics is in $\mathcal{O}(n\alpha(G^-))$. Indeed as we have seen each 1-deviation increases the global utility by at least 2. But the global utility of a stable partition is at most $n(\alpha(G^-)-1)$ as the groups have maximum size $\alpha(G^-)$. If the conflict graph is empty we have seen that $L(1,n) = \Theta(n^{3/2})$ that is in that case $\mathcal{O}(n\sqrt{\alpha(G^-)})$. This led one of us ([6], page 131) to conjecture that in the case of 1-deviations the worst time of convergence of the dynamics is $\mathcal{O}(n\sqrt{\alpha(G^-)})$. We disprove the conjecture by proving the following theorem:

▶ **Theorem 10.** *For $n = \binom{m+1}{2}$, there exists a conflict graph G^- with $\alpha(G^-) = m = \Theta(\sqrt{n})$ and a sequence of $\binom{m+1}{3}$ valid 1-deviations, that is a sequence of $\Omega(n^{\frac{3}{2}}) = \Omega(n\alpha(G^-))$ 1-deviations.*

Proof. We will use part of the construction of Greene and Kleitman ([9]). Namely, they prove that, if $n = \binom{m+1}{2}$, there is a sequence of $\binom{m+1}{3}$ 1-deviations transforming the partition P_1 consisting of n groups each of size 1 (the coordinates of $\overrightarrow{\Lambda}(P_1)$ satisfy $\lambda_1 = n$) into the partition P_m consisting of m groups, one of each possible size i for $1 \leq i \leq m$ (the coordinates of $\overrightarrow{\Lambda}(P_m)$ satisfy $\lambda_i = 1$ for $1 \leq i \leq m$). Furthermore they prove that the moves used are V-steps (see the proof of proposition 7 in the full version) which are nothing else than $\overrightarrow{\sigma}[p+1,p-1]$ for some p (one vertex leaves a group of size p to join a group of the same size p). One can note that in such a move the utility increases only by 2 and as the total utility of P_m is $\sum_{i=1}^{m} i(i-1) = (m+1)m(m-1)/3$ the number of moves is $(m+1)m(m-1)/6$.

The conflict graph of the counterexample will consist of m complete graphs $K^j, 1 \leq j \leq m$ where K^j has exactly j vertices. An independent set is therefore formed by taking at most one vertex in each K^j and $\alpha(G^-) = m$. We will denote the elements of K^j by $\{x_i^j\}$ with $1 \leq i \leq j \leq m$. The group of P_m of size i will be $X_i = \bigcup x_i^j$ with $m+1-i \leq j \leq m$. So these groups are independent sets.

Recall that $n = m(m+1)/2$. For each $p, 1 \leq p \leq m$ let us denote by P_p the partition consisting of 1 group of each size i for $1 \leq i \leq p$ and $n - p(p+1)/2$ groups of size 1 (said otherwise the coordinates of $\overrightarrow{\Lambda}(P_p)$ satisfy $\lambda_i = 1$ for $2 \leq i \leq p$ and $\lambda_1 = 1 + n - p(p+1)/2$). We will now describe the sequence $\overrightarrow{\sigma}[p-1]$ of $p(p-1)/2$ 1-deviations which transform the partition P_{p-1} into P_p. One way to do the Greene-Keitman sequence is obtained by doing successively the sequences $\sum_{p=2}^{m} \overrightarrow{\sigma}[p-1]$. More precisely we will prove by induction the following fact:

▶ **Claim 11.** *There exists a sequence $\overrightarrow{\sigma}[p-1]$ of $p(p-1)/2$ valid 1-deviations which transform the partition P_{p-1} into P_p such that after this sequence the group $X_i[p]$ of size i, $1 \leq i \leq p$ contains exactly the vertices $X_i[p] = \bigcup x_{i+m-p}^j$ with $m+1-i \leq j \leq m$.*

Proof. (see example given after the proof)

We suppose we have built the sequence till $p-1$ and that, for $1 \leq i \leq p-1$, $X_i[p-1] = \bigcup x_{i+m-p+1}^j$ with $m+1-i \leq j \leq m$. In a first phase we consider the subpartition of $n-p+1$ elements obtained by removing the group $X_{p-1}[p-1]$. Namely, this above subpartition consists of the groups $X_i[p-1]$ for $1 \leq i \leq p-2$ and groups of size 1. In particular, the subpartition is isomorphic to P_{p-2} with $p-1$ singleton groups removed. Our construction ensure that these $p-1$ singleton groups that are missing are not used for $\overrightarrow{\sigma}[p-2]$. So, we can do the transformation $\overrightarrow{\sigma}[p-2]$ consisting of $(p-1)(p-2)/2$ valid moves on the partition of $n-p+1$ elements not contained in $X_{p-1}[p-1]$. It gives rise to the groups

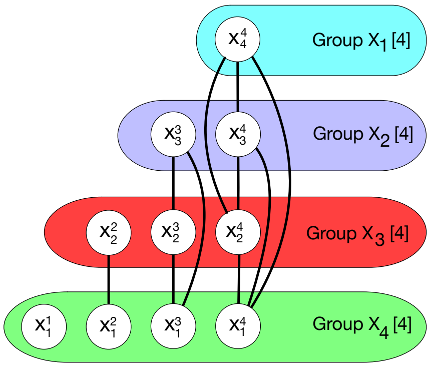


Figure 3 Illustration for Example 4.

$X_i[p] = X_{i-1}[p-1] + x^{m+1-i}_{i+m-p}$. Note that at this stage we have two groups of size $p-1$, namely, the original one $X_{p-1}[p-1]$ and the new one constructed $X_{p-1}[p]$. The second phase consists in doing $p-1$ successive 1-deviations with the vertex x^{m+1-p}_m. More precisely we move this vertex to the group $X_1[p]$ created in the first phase, then from this group to $X_2[p]$ and so on till $X_{p-2}[p]$ and finally from $X_{p-2}[p]$ to the original $X_{p-1}[p-1]$. The moves are valid as we move a vertex from K^{m+1-p} and the groups did not contain any vertex of this complete graph. Groups created in the first phase are eventually left unchanged as x^{m+1-p}_m joins such groups and then leaves them. Finally we have constructed a new group $X_p[p] = X_{p-1}[p-1] \cup x^{m+1-p}_m$. The groups are exactly those described in the claim. ◀

In order to end the proof of Theorem 10, it suffices to note that the groups X_i form an independent set and that after $\sum_{p=2}^{m} \overrightarrow{\sigma}[p-1]$ we have obtained the desired groups of P_m which gives the counterexample. ◀

Example for m = 4. (See Figure 3.)

- After $\overrightarrow{\sigma}[1]$, we have the 2 groups $X_2[2] = x^4_4 \cup x^3_4$ and $X_1[2] = x^4_3$.
- First phase of $\overrightarrow{\sigma}[2]$: we do the move of $\overrightarrow{\sigma}[1]$ on the vertices not in $X_2[2]$ and create the groups $X_2[3] = x^4_3 \cup x^3_3$ and $X_1[3] = x^4_2$.
- Second phase of $\overrightarrow{\sigma}[2]$: now we move x^2_4 to $X_1[3]$ and then from $X_1[3]$ to the original $X_2[2] = x^4_4 \cup x^3_4$, thereby creating the group $X_3[3] = x^4_4 \cup x^3_4 \cup x^2_4$.
- First phase of $\overrightarrow{\sigma}[3]$: we do the 3 moves of $\overrightarrow{\sigma}[2]$ on the vertices not in $X_3[3]$ and create the groups $X_3[4] = x^4_3 \cup x^3_3 \cup x^2_3$, $X_2[4] = x^4_2 \cup x^3_2$, $X_1[4] = x^4_1$.
- Second phase of $\overrightarrow{\sigma}[3]$: now we move x^1_4 to $X_1[4]$, then from $X_1[4]$ to $X_2[4]$ and finally from $X_2[4]$ to the original $X_3[3] = x^4_4 \cup x^3_4 \cup x^2_4$, thereby creating the group $X_4[4] = x^4_4 \cup x^3_4 \cup x^2_4 \cup x^1_4$.

We can prove a theorem analogous to Theorem 10 for any independence number $\alpha(G^-)$.

▶ **Theorem 12.** *For any $\alpha = \mathcal{O}(\sqrt{n})$, there exists a conflict graph G^- with n vertices and independence number $\alpha(G^-) = \alpha$, and a sequence of at least $\Omega(n\alpha)$ 1-deviations to reach a stable partition.*

Proof. Let G^-_0 be the graph of Theorem 10 for $m = \alpha$. G^-_0 has $n_0 = \mathcal{O}(\alpha^2)$ vertices, independence number α, and furthermore there exists a sequence of $\Theta(\alpha^3)$ valid 1-deviations for G^-_0. Let G^- be the graph obtained by taking the complete join of $k = n/n_0$ copies of G^-_0 (*i.e.*, we add all possible edges between every two copies of G^-_0). By construction, G^-

has order $n = kn_0 = \mathcal{O}(n\alpha^2)$ and the same independence number α as G_0^-. Furthermore, there exists a sequence of $k\Theta(\alpha^3) = \Omega(n\alpha)$ valid 1-deviations for G^-. ◀

Note that in any 2-deviation the global utility increases by at least 2 and so the number of 2 deviations when the conflict graph has independence number $\alpha(G^-)$ is also at most $\mathcal{O}(n\alpha(G^-))$. This bound is attained by using only 1-deviations as proved in Theorem 12, which is also valid for $k = 2$.

5 Lower bounds for k $>$ 2

The classical dominance ordering does not suffice to describe all k-deviations as soon as $k \geq 3$. As noted before, there is only one k-stable partition P_{max} in the empty conflict graph $G^{\emptyset}$, namely, the one consisting of one group of size n, with integer partition $Q_{max} = (n, 0, \ldots, 0)$ and partition vector $(1, 0, \ldots, 0)$. Let $d(Q)$ be the length of a longest sequence in the dominance lattice from the integer partition Q to the integer partition Q_{max}. For $k = 4$ let P be the partition consisting of 4 groups of size 4 and one group of size 1 with integer partition $Q = (4, 4, 4, 4, 1)$. Apply the 4-deviation where one vertex of each group of size 4 joins the group of size 1; it leads to the partition P' with integer partition $Q' = (5, 3, 3, 3, 3)$. Q is covered in the dominance lattice by the integer partition $(5, 4, 4, 3, 1)$ while Q' is at distance 3 from it via $(5, 4, 3, 3, 2)$ and $(5, 4, 4, 2, 2)$ and so, $d(Q') = d(Q) + 2$.

Prior to our work, it was known that $L(3, n) = O(n^3)$ ([14]). But nothing proved that $L(3, n) > L(2, n)$, and in fact it was conjectured in [7] that both values are equal. Theorem 13 proves for the first time that deviations can delay convergence and that the gap between $k = 2$ and $k = 3$ obtained from potential function is indeed justified. It was also conjectured in [14] that $L(k, n)$ was polynomial in n for k fixed. We disprove this conjecture and prove in Theorem 14 a much more significant result: 4-deviations are responsible for a sudden complexity increase, as we prove that no polynomial bounds exist for $L(4, n)$.

▶ **Theorem 13.** $L(3, n) = \Omega(n^2)$.

▶ **Theorem 14.** $L(4, n) = \Omega(n^{\Theta(\ln(n))})$.

The main idea of the proofs consists in doing repeated shifted sequences (called cascades) of deviations similar to the ones given in the example above. The proof of Theorem 13 can be found in the full version. In the next section, we give the proof of Theorem 14 for $k = 4$. We use sequences (cascades) of 4-deviations, called $\delta[p]$, and various additional tricks such that the repetition of the process by using cascades of cascades. Our motivation for using $\delta[p]$ as a basic building block for our construction is that it is the only type of 4-deviation which decreases the global utility.

5.1 Case k = 4. Proof of Theorem 14

Definition of $\delta[p]$: Consider a partition P containing at least 4 groups of size $p - 1$ and 1 group of size $p - 4$. In the 4-deviation $\delta[p]$ one vertex in each of the 4 groups of size $p - 1$ moves to the group of size $p - 4$ to form a new group of size p. The example given at the beginning of this section corresponds to the case $p = 5$. The coordinates of the associated difference vector (where we omit the bracket $[p]$ for ease of reading) are:

Figure 4 gives a visual description of these cascades. Here we start with a sequence of t 4-deviations $\delta[p]$ represented by black rectangles ($t = 16$ in the figure). The cascade so obtained, called $\overrightarrow{\delta}^1[p, t]$, is represented in red. Then we do $(t - 2)$ such cascades represented

Table 2 Difference vector of $\delta[p]$.

...	δ_p	δ_{p-1}	δ_{p-2}	δ_{p-3}	δ_{p-4}	...
...0	1	-4	4	0	-1	0...

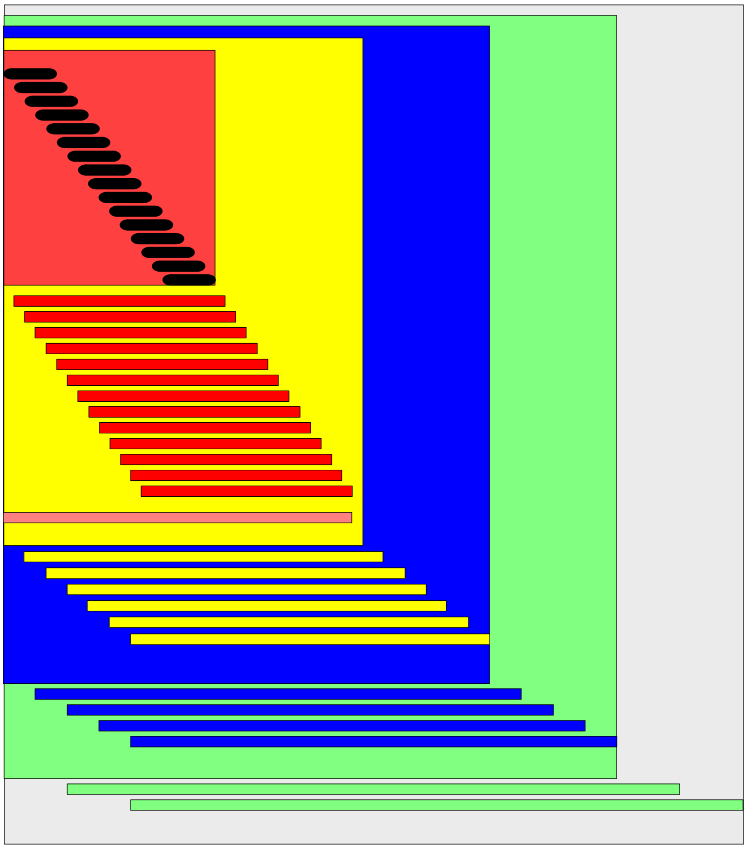


Figure 4 Cascades of cascades.

by red rectangles getting the cascade $\overrightarrow{\delta}^2[p, t-2]$ represented in yellow which contains $224(= 16 \cdot 14)$ 4-deviations. We apply some 1-deviations to get a deviation called $\overrightarrow{\zeta}^2[p]$ with the so-called Nice Property enabling us to do recursive constructions. We do a cascade of these $\overrightarrow{\zeta}^2[p]$ (shifted by 2) represented by yellow rectangles getting the blue cascade called $\overrightarrow{\zeta}^3[p]$. We do a cascade of these $\overrightarrow{\zeta}^3[p]$ (shifted by 3) represented by blue rectangles getting the green cascade called $\overrightarrow{\zeta}^4[p]$ and we finally do a cascade of these $\overrightarrow{\zeta}^4[p]$ (shifted by 5) represented by green rectangles getting the grey cascade called $\overrightarrow{\zeta}^5[p]$. The reader has to realize that, in this example, $\overrightarrow{\zeta}^5[p]$ contains 3 cascades $\overrightarrow{\zeta}^4[p]$ each containing 5 cascades $\overrightarrow{\zeta}^3[p]$ each consisting of 7 cascades $\overrightarrow{\zeta}^2[p]$. Altogether the cascade $\overrightarrow{\zeta}^5[p]$ of this example contains 23520 4-deviations $\delta[p]$.

The cascade $\overrightarrow{\delta}^1[p, t]$: we first do a cascade consisting of a sequence of t shifted 4-deviations $\delta[p], \delta[p-1], \ldots, \delta[p-t+1]$, for some parameter t which will be chosen later to give the maximum number of 4-deviations.

The reader can follow the construction in Table 3 with $t = 7$. The coordinates of $\overrightarrow{\delta}^1[p, t]$, are given in Claim 15 and Table 4. We note that there are lot of cancellations and only 8

Table 3 Computation of $\delta^1[p,7]$.

	...0	p	p-1	p-2	p-3	p-4	p-5	p-6	p-7	p-8	p-9	p-10	0...
$\delta[p]$	...0	1	-4	4	0	-1	0	0	0	0	0	0	0...
$+\delta[p-1]$	...0	0	1	-4	4	0	-1	0	0	0	0	0	0...
$+\delta[p-2]$	...0	0	0	1	-4	4	0	-1	0	0	0	0	0...
$+\delta[p-3]$	...0	0	0	0	1	-4	4	0	-1	0	0	0	0...
$+\delta[p-4]$	...0	0	0	0	0	1	-4	4	0	-1	0	0	0...
$+\delta[p-5]$	...0	0	0	0	0	0	1	-4	4	0	-1	0	0...
$+\delta[p-6]$	...0	0	0	0	0	0	0	1	-4	4	0	-1	0...
$= \overrightarrow{\delta}^1[p,7]$	...0	1	-3	1	1	0	0	0	-1	3	-1	-1	0...

Table 4 Difference vector $\delta^1[p,t]$.

...	δ^1_p	δ^1_{p-1}	δ^1_{p-2}	δ^1_{p-3}	...	δ^1_{p-t}	δ^1_{p-t-1}	δ^1_{p-t-2}	δ^1_{p-t-3}	...
...0	1	-3	1	1	0...0	-1	3	-1	-1	0...

non zero coordinates. Indeed consider the groups of size $p-i$ for $4 \le i \le t-1$; we have deleted such a group when doing the 4-deviation $\overrightarrow{\delta}[p+4-i]$, then created 4 such groups with $\overrightarrow{\delta}[p+2-i]$, then deleted 4 such groups with $\overrightarrow{\delta}[p+1-i]$, and finally created one with $\overrightarrow{\delta}[p-i]$. The reader can follow these cancellations in Table 3 for $i = 4,5,6$. The variation of the number of groups of a given size $p-i$ (which correspond to the coordinate δ^1_{p-i}) is obtained by summing the coefficients appearing in the corresponding column and so is 0 for $p-4, p-5, p-6$.

▶ **Claim 15.** *For $3 \le t \le p-3$, the coordinates of the cascade $\overrightarrow{\delta}^1[p,t] = \sum_{i=0}^{t-1} \overrightarrow{\delta}[p-i]$ satisfy: $\delta^1_p = 1$, $\delta^1_{p-1} = -3$, $\delta^1_{p-2} = 1$, $\delta^1_{p-3} = 1$, $\delta^1_{p-t} = -1$, $\delta^1_{p-t-1} = 3$, $\delta^1_{p-t-2} = -1$, $\delta^1_{p-t-3} = -1$, and $\delta^1_j = 0$ for all the others j (see Table 4).*

Proof. We have $\delta^1_j = \sum_{i=0}^{t-1} \delta_j[p-i]$. For a given j, $\delta_j[p-i] = 0$ except for the following values of i such that $0 \le i \le t-1$: $i = p-j$ where $\delta_j[j] = 1$; $i = p-j-1$ where $\delta_j[j+1] = -4$; $i = p-j-2$ where $\delta_j[j+2] = 4$; $i = p-j-4$ where $\delta_j[j+4] = -1$ (in the table it corresponds to the non zero values in a column, whose number is at most 4). Therefore, for $j > p$: $\delta^1_j = 0$; $\delta^1_p = 1$; $\delta^1_{p-1} = -4+1 = -3$; $\delta^1_{p-2} = 4-4+1 = 1$; $\delta^1_{p-3} = 0+4-4+1 = 1$; for $p-4 \ge j \ge p-t+1$, $\delta^1_{p-j} = -1+0+4-4+1 = 0$; $\delta^1_{p-t} = -1+0+4-4 = -1$; $\delta^1_{p-t-1} = -1+0+4 = 3$; $\delta^1_{p-t-2} = -1+0 = -1$; $\delta^1_{p-t-3} = -1$ and, for $j \le p-t-4$, $\delta^1_j = 0$. ◀

Validity of the cascades. We have to see when the cascades are valid, that is, to determine how many groups we need at the beginning. For the cascade $\overrightarrow{\delta}^1[p,t]$ we note that the coordinates of any subsequence of the cascade, *i.e.*, the coordinates of some $\overrightarrow{\delta}^1[p,r]$, are all at least -1 except δ^1_{p-1}: which is -4 when $r = 1$ and then -3. Therefore such a cascade is valid as soon as we have at least 4 groups of size $p-1$ and one group of each other size $p-i$ $(2 \le i \le t+3)$. To deal in general with the validity of cascades let us now introduce the notion of *h-balanced sequence.*

▶ **Definition 16.** Let h be a positive integer and let $\overrightarrow{\Phi} = \sum_{j=1}^{s} \overrightarrow{\varphi}^j$ be a cascade consisting of s k-deviations. We call this cascade h-balanced if, for any $1 \le i \le s$, the sum of the i first vectors, namely, $\sum_{j=1}^{i} \overrightarrow{\varphi}^j$, has all its coordinates greater than or equal to $-h$.

Table 5 Computation of $\delta^2[p,5]$.

	...	p	p-1				p-5				p-9				p-13	p-14	...
$\delta^1[p,7]$	...0	1	-3	1	1	0	0	0	-1	3	-1	-1	0	...			
$+\delta^1$[p-1,7]	...0	0	1	-3	1	1	0	0	0	-1	3	-1	-1	0	...		
$+\delta^1$[p-2,7]	...0	0	0	1	-3	1	1	0	0	0	-1	3	-1	-1	0	...	
$+\delta^1$[p-3,7]	...0	0	0	0	1	-3	1	1	0	0	0	-1	3	-1	-1	0	...
$+\delta^1$[p-4,7]	...0	0	0	0	0	1	-3	1	1	0	0	0	-1	3	-1	-1	0...
$= \overrightarrow{\delta}^2[p,5]$	...0	1	-2	-1	0	0	-1	2	0	2	1	0	0	1	-2	-1	0...

Table 6 Difference vector $\delta^2[p,t-2]$.

...	δ^2_p	δ^2_{p-1}	δ^2_{p-2}	...	δ^2_{p-t+2}	δ^2_{p-t+1}	δ^2_{p-t}	δ^2_{p-t-1}	δ^2_{p-t-2}	...	δ^2_{p-2t+2}	δ^2_{p-2t+1}	δ^2_{p-2t}	...
0	1	-2	-1	0	-1	2	0	2	1	0	1	-2	-1	0

For example, the cascade $\overrightarrow{\delta}^1[p,t]$ described before is 4-balanced. The interest of this notion lies in the following fact: Let $p_{\max}$ be the largest index j that satisfies $\overrightarrow{\Phi}_j \neq 0$. Then, if we start from a partition with at least h groups of each size j, for $1 \leq j \leq p_{\max}$, an h-balanced sequence is valid.

Note that a sequence is itself composed of sub-sequences and the following lemma will be useful to bound the value h of a sequence.

▶ **Lemma 17.** *Let $\overrightarrow{\Phi}^1$ be an h_1-balanced sequence and $\overrightarrow{\Phi}^2$ be an h_2-balanced sequence. Then, $\overrightarrow{\Phi}^1 + \overrightarrow{\Phi}^2$ is a (max $\{h_1, h_2 - min_i\Phi^1_i\}$)-balanced sequence.*

Proof. As $\overrightarrow{\Phi}^1$ is h_1-balanced, the coordinates of any subsequence of $\overrightarrow{\Phi}^1$ are greater than or equal to $-h_1$. Consider a subsequence $\overrightarrow{\Phi}^1 + \overrightarrow{\Phi}^3$ where $\overrightarrow{\Phi}^3$ is a subsequence of $\overrightarrow{\Phi}^2$. The j-th coordinate is $\Phi^1_j + \Phi^3_j$; by definition $\Phi^3_j \geq -h_2$ and so, $\Phi^1_j + \Phi^3_j \geq \Phi^1_j - h_2 \geq min_i\Phi^1_i - h_2$. ◀

The cascade $\overrightarrow{\delta}^2[p,t-2]$: We do now the following sequence of $t-2$ cascades $\overrightarrow{\delta}^2[p,t-2] = \sum_{i=0}^{t-3} \overrightarrow{\delta}^1[p-i,t]$. Altogether we have a sequence of $t(t-2)$ 4-deviations. There are a lot of cancellations and in fact, as shown in Claim 18, $\overrightarrow{\delta}^2[p,t-2]$ has only 10 non zero coordinates. Table 5 describes an example of computation of $\overrightarrow{\delta}^2[p,t-2]$ with $t = 7$.

▶ **Claim 18.** *For $3 \leq t \leq \frac{p}{2}$, the coordinates of the cascade $\overrightarrow{\delta}^2[p,t-2] = \sum_{i=0}^{t-3} \overrightarrow{\delta}^1[p-i,t]$ satisfy: $\delta^2_p = 1$, $\delta^2_{p-1} = -2$, $\delta^2_{p-2} = -1$, $\delta^2_{p-t+2} = -1$, $\delta^2_{p-t+1} = 2$, $\delta^2_{p-t-1} = 2$, $\delta^2_{p-t-2} = 1$, $\delta^2_{p-2t+2} = 1$, $\delta^2_{p-2t+1} = -2$, $\delta^2_{p-2t} = -1$, and $\delta^2_j = 0$ for all the others j (see Table 6). Furthermore this cascade is 4-balanced.*

Proof. We have $\delta^2_j = \sum_{i=0}^{t-3} \delta^1_j[p-i,t]$. Using the values of $\delta^1_j[p-i,t]$, we get that: for $j > p$, $\delta^2_j = 0$; $\delta^2_p = 1$; $\delta^2_{p-1} = -3+1 = -2$; $\delta^2_{p-2} = 1-3+1 = -1$; for $p-3 \geq j \geq p-t+3$, $\delta^2_j = 1+1-3+1 = 0$; $\delta^2_{p-t+2} = 1+1-3 = -1$; $\delta^2_{p-t+1} = 1+1 = 2$; $\delta^2_{p-t} = -1+1 = 0$; $\delta^2_{p-t-1} = 3-1 = 2$; $\delta^2_{p-t-2} = -1+3-1 = 1$; for $p-t-3 \geq j \geq p-2t+3$, $\delta^2_j = -1-1+3-1 = 0$; $\delta^2_{p-2t+2} = -1-1+3 = 1$; $\delta^2_{p-2t+1} = -1-1 = -2$, $\delta^2_{p-2t} = -1$, and for $j < p-2t$, $\delta^2_j = 0$.

Using Lemma 17 we get that $\overrightarrow{\delta}^2[p,t-2]$ is 7-balanced; but a careful analysis shows that this sequence is in fact 4-balanced. Indeed we will prove by induction that $\overrightarrow{\delta}^2[p,r] = \sum_{i=0}^{r-1} \overrightarrow{\delta}^1[p-i,t]$ is 4-balanced for any $r \leq t-3$. That is true for $r = 1$, as $\overrightarrow{\delta}^1[p,t]$ is 4-balanced. Suppose that it is true for r. We have $\overrightarrow{\delta}^2[p,r+1] = \overrightarrow{\delta}^2[p,r] + \overrightarrow{\delta}^1[p-r-1,t]$. All the coordinates of $\overrightarrow{\delta}^2[p,r]$ are by the computation above at least -3, and the coordinates

Table 7 Difference vector $\zeta^2[p]$.

...	ζ^2_p	ζ^2_{p-1}	ζ^2_{p-2}	ζ^2_{p-3}	...	ζ^2_{p-t}	ζ^2_{p-t-1}	...	ζ^2_{p-2t+2}	ζ^2_{p-2t+1}	ζ^2_{p-2t}	ζ^2_{p-2t-1}	...
...0	1	0	-1	-1	0...0	1	1	0...0	-1	-1	0	1	0...

of $\overrightarrow{\delta}^1[p-r-1,t]$ are greater than -1 except for $j = p-r-2$ where $\delta^1_{p-r-2}[p-r-1] = -4$; but $\delta^2_{p-r-2}[p,r] = 1$ (case $r = 1$) or 2 (case $r > 1$) and so, all the coordinates of $\overrightarrow{\delta}^2[p, r+1]$ are at least -4. ◀

At this stage we could continue and do a cascade of $\overrightarrow{\delta}^2[p, t-2]$ but there is no more the phenomenon of cancellation. In fact we will use the following "symmetrization" trick. We will transform the cascade $\overrightarrow{\delta}^2[p, t-2]$ into a sequence $\overrightarrow{\zeta}^2[p]$ by doing some sequence of 1-deviations whose coordinates are given in Claim 19 The sequence obtained has only 8 non zero coefficients (4 with values 1 and 4 with values -1) arranged in a very symmetric nice way (that we will call *Nice Property*). Furthermore we will be able to iterate a cascade process on it many times keeping the property.

For $p \geq q+2$, we will denote by $\overrightarrow{\alpha}[p, q]$ the 1-deviation, where a vertex leaves a group of size $q+1$ for a group of size $p-1$ (valid as $p \geq q+2$). Let $\overrightarrow{\alpha}^1[p, q, r] = \sum_{i=0}^{r-1} \overrightarrow{\alpha}[p-i, q+i]$ denote a cascade of r such 1-deviations (we need $p-r+1 \geq q+r+2$ in order it is valid). The coordinates of $\overrightarrow{\alpha}^1[p, q, r]$ are given in the following Claim 19.

▶ **Claim 19.** *For $p - r \geq q + r + 1$, $\overrightarrow{\alpha}^1[p, q, r] = \sum_{i=0}^{r-1} \overrightarrow{\alpha}[p-i, q+i]$ has only 4 non zero coordinates namely, $\alpha^1_p = 1$, $\alpha^1_{p-r} = -1$, $\alpha^1_{q+r} = -1$, and $\alpha^1_q = 1$.*

▶ **Claim 20.** *For $3 \leq t \leq \frac{p+1}{2}$, the coordinates of the sequence $\overrightarrow{\zeta}^2[p] = \overrightarrow{\delta}^2[p, t-2] + \overrightarrow{\alpha}^1[p-1, p-2t-1, t-2] + \overrightarrow{\alpha}^1[p-1, p-2t, 2] + \overrightarrow{\alpha}^1[p-t+2, p-2t+1, 1] + \overrightarrow{\alpha}^1[p-t, p-t-3, 1]$ satisfy: $\zeta^2_p = 1$, $\zeta^2_{p-2} = -1$, $\zeta^2_{p-3} = -1$, $\zeta^2_{p-t} = 1$, $\zeta^2_{p-t-1} = 1$, $\zeta^2_{p-2t+2} = -1$, $\zeta^2_{p-2t+1} = -1$, $\zeta^2_{p-2t-1} = 1$ (see Table 7). Furthermore this cascade is still 4-balanced.*

Proof. By Claim 19, we have the following coordinates:

- for $\overrightarrow{\alpha}^1[p-1, p-2t-1, t-2]$, $\alpha^1_{p-1} = 1$, $\alpha^1_{p-t+1} = -1$, $\alpha^1_{p-t-3} = -1$, $\alpha^1_{p-2t-1} = 1$;
- for $\overrightarrow{\alpha}^1[p-1, p-2t, 2]$, $\alpha^1_{p-1} = 1$, $\alpha^1_{p-3} = -1$, $\alpha^1_{p-2t+2} = -1$, $\alpha^1_{p-2t} = 1$;
- for $\overrightarrow{\alpha}^1[p-t+2, p-2t+1, 1]$, $\alpha^1_{p-t+2} = 1$, $\alpha^1_{p-t+1} = -1$, $\alpha^1_{p-2t+2} = -1$, $\alpha^1_{p-2t+1} = 1$;
- for $\overrightarrow{\alpha}^1[p-t, p-t-3, 1]$, $\alpha^1_{p-t} = 1$, $\alpha^1_{p-t-1} = -1$, $\alpha^1_{p-t-2} = -1$, $\alpha^1_{p-t-3} = 1$.

Therefore, using these values and the values of the coordinates of δ^2_j given in claim 18, we get $\zeta^2_p = 1$, $\zeta^2_{p-1} = -2+1+1 = 0$, $\zeta^2_{p-2} = -1$, $\zeta^2_{p-3} = 0-1 = -1$, $\zeta^2_{p-t+2} = -1+1 = 0$, $\zeta^2_{p-t+1} = 2-1-1 = 0$, $\zeta^2_{p-t} = 0+1 = 1$, $\zeta^2_{p-t-1} = 2-1 = 1$, $\zeta^2_{p-t-2} = 1-1 = 0$, $\zeta^2_{p-t-3} = 0-1+1 = 0$, $\zeta^2_{p-2t+2} = 1-1-1 = -1$, $\zeta^2_{p-2t+1} = -2+1 = -1$, $\zeta^2_{p-2t} = -1+1 = 0$ $\zeta^2_{p-2t-1} = 0+1$.

To prove that $\overrightarrow{\zeta}^2[p]$ is 4-balanced, apply Lemma 17 with $\overrightarrow{\Phi}^1 = \overrightarrow{\delta}^2[p, t-2]$ and $\overrightarrow{\Phi}^2 = \overrightarrow{\alpha}^1[p-1, p-2t-1, t-2] + \overrightarrow{\alpha}^1[p-1, p-2t, 1] + \overrightarrow{\alpha}^1[p-t+2, p-2t+1, 1] + \overrightarrow{\alpha}^1[p-t, p-t-3, 1]$. We have that $h_1 = 4$ and furthermore all the coefficients of $\overrightarrow{\Phi}^1$ are greater than -2 and $\overrightarrow{\Phi}^2$ is 2-balanced. Hence, $\overrightarrow{\zeta}^2[p]$ is $\max(4, 2+2) = 4$-balanced. ◀

Table 8 shows an example with $t = 7$.

▶ **Definition 21. Nice Property**: Let $k \geq 2$ be a positive integer. We will say the sequence $\overrightarrow{\zeta}^k[p]$ has the *Nice Property*, if there exist 3 integers $a(k)$, $b(k)$, and $s(k)$ satisfying $1 < a(k) < b(k) < 2a(k)$ and $b(k) < s(k) - 1 < p/2$ and such that all coordinates of $\overrightarrow{\zeta}^k$ are null except for:

Table 8 Computation of $\zeta^2[p]$ with $t = 7$.

	...	p	p-1			...			p-7	p-8			...				p-15	...
$\delta^2[p, 5]$	0	1	-2	-1	0	0	-1	2	0	2	1	0	0	1	-2	-1	0	
+α[p-1,p-15,5]	0	0	1	0	0	0	0	-1	0	0	0	-1	0	0	0	0	1	0
+α[p-1,p-14,2]	0	0	1	0	-1	0	0	0	0	0	0	-1	0	-1	0	1	0	0
+α[p-5,p-13,1]	0	0	0	0	0	0	1	-1	0	0	0	0	0	-1	1	0	0	0
+α[p-7,p-10,1]	0	0	0	0	0	0	0	0	1	-1	-1	1	0	0	0	0	0	0
$= \overrightarrow{\zeta}^2[p]$	0	1	0	-1	-1	0	0	0	1	1	0	0	0	-1	-1	0	1	0

- $\zeta^k_p = \zeta^k_{p+1-2s(k)} = 1$,
- $\zeta^k_{p-a(k)} = \zeta^k_{p-b(k)} = \zeta^k_{p+1-2s(k)+b(k)} = \zeta^k_{p+1-2s(k)+a(k)} = -1$, and
- $\zeta^k_{p+1-s(k)} = \zeta^k_{p-s(k)} = 1$.

We note the symmetry of the coordinates, as for any j, $\zeta^k_{p-j} = \zeta^k_{p+1-2s(k)+j}$. As an example, the sequence $\overrightarrow{\zeta}^2[p]$ satisfies the Nice Property with $a(2) = 2$, $b(2) = 3$ and $s(2) = t + 1$ and is 4-balanced. Now we will show how starting with a sequence $\overrightarrow{\zeta}^k[p]$ satisfying the Nice Property we can construct a sequence $\overrightarrow{\zeta}^{k+1}[p]$ having still the Nice Property.

▶ **Claim 22.** ***Main construction:*** *Let $\overrightarrow{\zeta}^k[p]$ be a sequence satisfying the Nice Property with parameters $a(k), b(k), s(k)$. Then, we can construct a sequence $\overrightarrow{\zeta}^{k+1}[p]$ satisfying the following properties:*

- *$\overrightarrow{\zeta}^{k+1}[p]$ satisfies the Nice Property with parameters*
  - *$a(k+1) = b(k)$,*
  - *$b(k+1) = b(k) + a(k)$,*
  - *$s(k+1) = s(k)+a(k)r(k)/2$, where $r(k)$ is the greatest even integer such that $r(k)a(k)+b(k) < s(k) - 1$;*
- *if $\overrightarrow{\zeta}^k[p]$ is $h(k)$-balanced, then $\overrightarrow{\zeta}^{k+1}[p]$ is $(h(k) + 1)$-balanced;*
- *$\overrightarrow{\zeta}^{k+1}[p]$ contains $r(k) + 1$ sequences $\overrightarrow{\zeta}^k[p]$.*

Proof. We will first do a cascade of $\overrightarrow{\zeta}^k[p]$, but we will take values of the parameters differing by a multiple of $a(k)$ in order for some of the coordinates to cancel. Specifically, let us define $\overrightarrow{\Psi}^r = \sum_{j=0}^{r} \overrightarrow{\zeta}^k[p - ja(k)]$. Using the values of Definition 21, we get the following values for the non zero coordinates:

(1) $\psi^r_p = 1$; $\psi^r_{p-ja(k)} = -1 + 1 = 0$ for $0 < j \le r$ (cancellation phenomenom); $\psi^r_{p-(r+1)a(k)} = -1$;

(2) $\psi^r_{p+1-2s(k)+a(k)} = -1$; $\psi^r_{p+1-2s(k)-(j-1)a(k)} = 1 - 1 = 0$ for $0 < j \le r$ (cancellation); $\psi^r_{p+1-2s(k)-ra(k)} = 1$;

(3) for $0 \le j \le r$, $\psi^r_{p-b(k)-ja(k)} = -1$;

(4) for $0 \le j \le r$, $\psi^r_{p+1-2s(k)+b(k)-ja(k)} = -1$;

(5) for $0 \le j \le r$, $\psi^r_{p+1-s(k)-ja(k)} = \psi^r_{p-s(k)-ja(k)} = 1$.

Since $a(k) < b(k) < 2a(k)$, all the indices of the coordinates are different provided we choose r even and nonzero such that $p - b(k) - ra(k) > p + 1 - s(k)$ (that is equivalent to $ra(k) + b(k) < s(k) - 1$). Let us denote $a(k + 1) = b(k)$, $b(k + 1) = b(k) + a(k)$ and $s(k + 1) = s(k) + a(k)r/2$. Then $\overrightarrow{\Psi}^r$ has already part of the Nice Property for $k + 1$. Indeed we have:

- $\psi^r_p = 1$ by (1) and $\psi^r_{p+1-2s(k+1)} = 1$ by (2) with $j = r$ (as $2s(k + 1) = 2s(k) + ra(k)$);
- $\psi^r_{p-a(k+1)} = \psi^r_{p-b(k)} = -1$, $\psi^r_{p-b(k+1)} = \psi^r_{p-b(k)-a(k)} = -1$ by (3) with $j = 0, 1$;

- $\psi^r_{p+1-2s(k+1)+b(k+1)} = -1$, $\psi^r_{p+1-2s(k+1)+a(k+1)} = -1$ by (4) with $j = r-1, r$;
- $\psi^r_{p+1-s(k+1)} = \psi^r_{p-s(k+1)} = 1$ by (5) with $j = r/2$.

The remaining non zero coordinates are in number $4r$: firstly there are r values -1, namely, $\psi^r_{p-b(k)-ja(k)} = -1$, for $2 \leq j \leq r$, and $\psi^r_{p-(r+1)a(k)} = -1$; then there are $2r$ values 1, namely, $\psi^r_{p+1-s(k+1)} = \psi^r_{p-s(k+1)} = 1$, for $j \neq r/2$; and finally there are r values -1, namely, $\psi^r_{p+1-2s(k)+a(k)} = -1$ and $\psi^r_{p+1-2s(k)+b(k)-ja(k)} = -1$, for $0 \leq j \leq r-2$. These values are disposed in a very symmetric way and can be written: for the values -1, in the form $\psi^r_{p-x_m}$ and $\psi^r_{p+1-2s(k+1)+x_m}$; and for the values 1, in the form $\psi^r_{p-y_m}$ and $\psi^r_{p+1-2s(k+1)+y_m}$ with $x_m < y_m$ $(0 \leq m \leq r-1)$. Furthermore, these r quadruples of values can be canceled by adding to $\overrightarrow{\Psi}^r$ the r sequences $\overrightarrow{\alpha}^1[p - x_m, p+1-2s(k+1)+x_m, y_m - x_m]$.

We claim that the sequence so obtained, with partition vector $\overrightarrow{\Psi}^r + \sum_{m=0}^{r-1} \overrightarrow{\alpha}^1[p-x_m, p+1-2s(k+1)+x_m, y_m-x_m]$, satisfies the Nice Property with parameters $a(k+1), b(k+1)$ and $s(k+1)$. Indeed $a(k+1) = b(k) < b(k)+a(k) = b(k+1)$, $b(k+1) = b(k)+a(k) < b(k)+b(k) = 2a(k+1)$ and $b(k+1) = b(k)+a(k) < s(k)-1+a(k) \leq s(k+1)-1$ as $r \geq 2$. We also have to ensure in the computations that p is chosen so that $p \geq 2s(k)-1$. In order to get the maximum number of deviations we will consider this sequence for the largest possible even integer r satisfying $ra(k)+b(k) < s(k)-1$, denoted $r(k)$ and we will denote the sequence for this $r(k)$ by $\overrightarrow{\zeta}^{k+1}[p]$.

We now prove that $\overrightarrow{\zeta}^{k+1}[p]$ is $(h(k)+1)$-balanced. We first prove by induction that $\overrightarrow{\Psi}^r$ is $(h(k)+1)$-balanced. That is true for $r=0$ as $\overrightarrow{\zeta}^k[p]$ is $h(k)$-balanced. Then suppose it is true for some r ; we apply Lemma 17 with $\overrightarrow{\Phi}^1 = \overrightarrow{\Psi}^r$ and $\overrightarrow{\Phi}^2 = \overrightarrow{\zeta}^k[p-(r+1)a(k)]$. We have that $h_1 = h(k)+1$ by induction hypothesis and furthermore all the coefficients of $\overrightarrow{\Phi}^1$ are greater than -1; furthermore $\overrightarrow{\Phi}^2$ is $h(k)$-balanced and so, $\overrightarrow{\Psi}^{r+1}$ is $(\max(h(k)+1, h(k)+1) = h(k)+1)$-balanced. Then when we add an $\overrightarrow{\alpha}^1[p-x_m, p+1-2s(k+1)+x_m, y_m-x_m]$ which is 1-balanced we still get an $(\max(h(k)+1, 1+1) = h(k)+1$-balanced sequence.

Finally, by construction, we get that $\overrightarrow{\zeta}^{k+1}[p]$ contains $r(k)+1$ sequences $\overrightarrow{\zeta}^k[p]$. ◀

End of the proof of Theorem 14. At this stage we have built a sequence $\overrightarrow{\zeta}^2[p]$ which satisfies the Nice Property with $a(2)=2$, $b(2)=3$ and $s(2)=t+1$ and is $h(2)$=4-balanced. Furthermore, it contains $t(t-2)$ 4-deviations. See Claim 20. Then, for some well-chosen K (to be defined later) we can apply $K-2$ times the main construction (Claim 22) to construct a sequence $\overrightarrow{\zeta}^K[p]$ which satisfies the Nice Property with parameters $a(K)$, $b(K)$ and $s(K)$ and is $h(K)$-balanced.

We have $a(k) = b(k-1)$, $b(k) = b(k-1)+a(k-1) = b(k-1)+b(k-2)$ and so, we recognize the Fibonacci recurrence relation. The k^{th} Fibonacci number $F(k)$ is denoted as follows:

$$F(k) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^k - \left(\frac{1-\sqrt{5}}{2}\right)^k\right).$$

Then, as $a(2) = 2 = F(3)$ and $b(2) = 3 = F(4)$, we get $a(K) = F(K+1)$ and $b(K) = F(K+2)$. In fact in what follows we will only use that $a(K) \leq 2^{K-1}$ and $b(K) \leq 2^K$. We have $s(k+1) = s(k)+a(k)r(k)/2$; but $a(k)r(k) < s(k-1)-b(k) < s(k-1)$ and so, $s(k+1) < (3/2) \times s(k)$ and $s(K) < s(2)(3/2)^{K-2} = (t+1)(3/2)^{K-2}$.

Recall that we should have $p \geq 2s(K)-1$ so we choose $p = 2s(K)$. Furthermore by induction we have that $h(K) = K+2$. So we need to start with a partition containing at least $K+2$ groups of each size i, $1 \leq i \leq p$. It is easy to obtain such a starting partition from the initial partition — which consists of n groups of size 1 — by doing a sequence of

1-deviation of size $(K-2)p(p+1)/2$; indeed we can create a group of any size i with $(i-1)$ 1-deviations. Therefore, we will take $n = (K-2)p(p+1)/2 \leq (K-2)s(K)(2s(K)+1)$. Using the inequality $s(K) < (t+1)(3/2)^{K-2}$ we get that

$$n = \mathcal{O}(t^2K(3/2)^{2K}). \tag{1}$$

On the other hand we have to lower bound the number of deviations. By construction $\overrightarrow{\zeta}^{k+1}[p]$ contains $r(k)+1$ sequences $\overrightarrow{\zeta}^{k}[p]$ and so, contains $t(t-2)\prod_{k=2}^{K-1}(r(k)+1)$ 4-deviations, as $\overrightarrow{\zeta}^{2}[p]$ contains $t(t-2)$ 4-deviations. Recall that $r(k)$ is the greatest even integer r such that $ra(k)+b(k) < s(k)-1$ and so, $r(k) \geq \lfloor \frac{s(k)-1-b(k)}{a(k)} \rfloor - 1$. Using the fact that $b(k)+1 \leq 2a(k)$ and $s(k) > s(2)-1 = t$, and $a(k) \leq a(K) < 2^{K-1}$ we get $r(k) \geq \frac{t}{2^{K-1}} - 3$. Then $\prod_{k=2}^{K-1}(r(k)+1) \geq (\frac{t}{2^{K-1}} - 2)^{K-2}$ and the number D of deviations satisfies:

$$D = \Omega(t^2(\frac{t}{2^{K-1}} - 2)^{K-2}). \tag{2}$$

We have now to choose K as a function of t. In order for the number of deviations as given by Equation 2 to increase we need that 2^{K-1} is small compared to t, that is, $K << \log_2(t)$. However in view of Equation 1 we want to choose the largest possible K. Therefore, a good choice is $K = 1/2(\log_2(t))$. In that case, we get by Equation 1 that $n = \mathcal{O}(t^2\log_2(t)(3/2)^{\log_2(t)})$, or equivalently $\log_2(n) = \mathcal{O}(2\log_2(t) + \log_2(\log_2(t) + \log_2(t)(\log_2(3) - \log_2(2)))$. Using $\log_2(3) - \log_2(2) > 0.585$ and the fact that for t large enough $\log_2(\log_2(t)) < 0.014\log_2(t)$ we get $\log_2(n) = \mathcal{O}(2.6\log_2(t))$, that is, $n = \mathcal{O}(t^{2.6})$. On the other hand we get by Equation 2: $D = \Omega((t^{1/2})^{1/2\log_2(t)}) = \Omega(t^{1/4\log_2(t)})$ and so, $D = \Omega(n^{clog_2(n)})$ with $c = \frac{1}{4\times(2.6)^2} \simeq 1/27$, thereby proving Theorem 14.

References

1 C. Ballester. NP-completeness in hedonic games. *Games and Economic Behavior*, 49(1):1–30, 2004.

2 J. A. Bondy and U. S. R. Murty. *Graph theory.* Grad. Texts in Math., 2008.

3 T. Brylawski. The lattice of integer partitions. *Discrete Mathematics*, 6(3):201–219, 1973.

4 I. Chatzigiannakis, C. Koninis, P. N. Panagopoulou, and P. G. Spirakis. Distributed game-theoretic vertex coloring. In *OPODIS'10*, pages 103–118, 2010.

5 J. Chen, R. Niedermeier, and P. Skowron. Stable marriage with multi-modal preferences. In *EC*, 2018. to appear.

6 G. Ducoffe. *Propriétés métriques des grands graphes.* PhD thesis, Université Côte d'Azur, December 2016.

7 B. Escoffier, L. Gourvès, and J. Monnot. Strategic coloring of a graph. *Internet Mathematics*, 8(4):424–455, 2012.

8 M. Flammini, G. Monaco, and Q. Zhang. Strategyproof mechanisms for additively separable hedonic games and fractional hedonic games. In *WAOA*, pages 301–316, 2017.

9 C. Greene and D. J. Kleitman. Longest chains in the lattice of integer partitions ordered by majorization. *European Journal of Combinatorics*, 7(1):1–10, jan 1986.

10 J. Hajduková. Coalition formation games: A survey. *International Game Theory Review*, 8(04):613–641, 2006.

11 G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers.* Oxford University Press, 1979.

12 M. Hoefer and W. Jiamjitrak. On proportional allocation in hedonic games. In *SAGT*, pages 307–319. Springer, 2017.

13 D. S. Johnson, C. H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of computer and system sciences*, 37(1):79–100, 1988.

14 J. Kleinberg and K. Ligett. Information-sharing in social networks. *Games and Economic Behavior*, 82:702–716, 2013.

15 M. Mnich and I. Schlotter. Stable marriage with covering constraints–a complete computational trichotomy. In *SAGT*, pages 320–332. Springer, 2017.

16 K. Ohta, N. Barrot, A. Ismaili, Y. Sakurai, and M. Yokoo. Core stability in hedonic games among friends and enemies: impact of neutrals. In *IJCAI*, 2017.

17 I. Olkin and A. W. Marshall. *Inequalities: theory of majorization and its applications*, volume 143. Academic press, 2016.

18 P. N. Panagopoulou and P. G. Spirakis. A game theoretic approach for efficient graph coloring. In *ISAAC'08*, pages 183–195, 2008.

On the Complexity of Two Dots for Narrow Boards and Few Colors

Davide Bilò
University of Sassari, Italy
davide.bilo@uniss.it
https://orcid.org/0000-0003-3169-4300

Luciano Gualà
University of Rome “Tor Vergata”, Italy
guala@mat.uniroma2.it
https://orcid.org/0000-0001-6976-5579

Stefano Leucci
ETH Zürich, Switzerland
stefano.leucci@inf.ethz.ch
https://orcid.org/0000-0002-8848-7006

Neeldhara Misra
Indian Institute of Technology, Gandhinagar
mail@neeldhara.com
https://orcid.org/0000-0003-1727-5388

Abstract

Two Dots® is a popular single-player puzzle video game for iOS and Android. A level of this game consists of a grid of colored dots. The player connects two or more adjacent dots, removing them from the grid and causing the remaining dots to fall, as if influenced by gravity. One special move, which is frequently a game-changer, consists of connecting a *cycle* of dots: this removes all the dots of the given color from the grid. The goal is to remove a certain number of dots of each color using a limited number of moves. The computational complexity of Two Dots has already been addressed in [Misra, FUN 2016], where it has been shown that the general version of the problem is NP-complete. Unfortunately, the known reductions produce Two Dots levels having both a large number of colors and many columns. This does not completely match the spirit of the game, where, on the one hand, only few colors are allowed, and on the other hand, the grid of the game has only a constant number of columns. In this paper, we partially fill this gap by assessing the computational complexity of Two Dots instances having a small number of colors or columns. More precisely, we show that Two Dots is hard even for instances involving only 3 colors or 2 columns. As a contrast, we also prove that the problem can be solved in polynomial-time on single-column instances with a constant number of goals.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases puzzle, NP-complete, perfect information, combinatorial game theory

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.7

1 Introduction

Two Dots® (`http://weplaydots.com/twodots.html`) is a popular single-player puzzle video game for iOS and Android. The game has been so much appreciated by the community that, not even after 3 years from its launch, a recently introduced follow-up game, called Dots&Co®

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 7; pp. 7:1–7:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

(`https://www.dots.co/dotsandco/`), has already passed the 5 millions of downloads. In its simplest form, the game is played on a vertical grid where each location initially contains 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). In particular, the player selects a path of dots of the same color which can be either simple or it can contain exactly one cycle. In the former case only the selected dots disappear, while, in the latter case, all dots of that color disappear. It turns out that the cyclic 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 tip: *"When in doubt, make squares"*. After a move, all the remaining dots in the area fall down as if influenced by gravity. The game provides a certain number of moves, and demands certain goals to 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 it is removed).

The computational complexity of the game has been analyzed in [16], where the author showed that the problem of deciding whether an instance can be won by the player is NP-complete even in very restricted settings. In particular, the problem remains hard when the board has only four rows, or when there is only one goal of collecting two dots of a particular color, even if there is no restriction on the number of moves. In [16] it is also shown that the problem is W[1]-hard when parameterized by the number of moves. It turns out that all these reductions use a large (i.e., typically linear in the size of the instance) number of both different colors and columns. However, this does not completely match the spirit of the game, where, on the one hand, only few colors are allowed, and on the other hand, the arena of the game has only a constant number of columns, while there can be many rows (even if the player can only see the few down-most ones). Understanding the complexity of the game under these more realistic conditions is explicitly mentioned as open problems in [16].

In this paper, we partially fill this gap by showing that:

- the game is NP-complete even when the instance has three colors, two moves, and two goals;[1]
- the game is NP-complete even when the board has two columns and there is no restriction on the number of moves;
- the game is polynomial-time solvable when the board has only one column, provided that the number of goals is constant;
- the game is NP-complete even when the board has two rows.

Observe that the first two results immediately imply that the problem is not fixed parameter tractable when parameterized w.r.t. the number of colors, or the number of columns, unless P=NP. We leave open the problem of setting the computational complexity of the game when the instance has *both* a constant number of colors and columns.

Other related results

Two Dots belongs to the class of *tile-matching* video games. Tile-matching games allow the player to select a subset of tiles on the board according to some matching rule. Once selected, the tiles are removed from the board and the board configuration is updated automatically following the game-specific rules (for instance, all the remaining tiles might move to fill the voids as if influenced by gravity). Other popular games of this class also exhibit a rich

[1] A playable version of this reduction is available at `https://twodots.isnphard.com`.

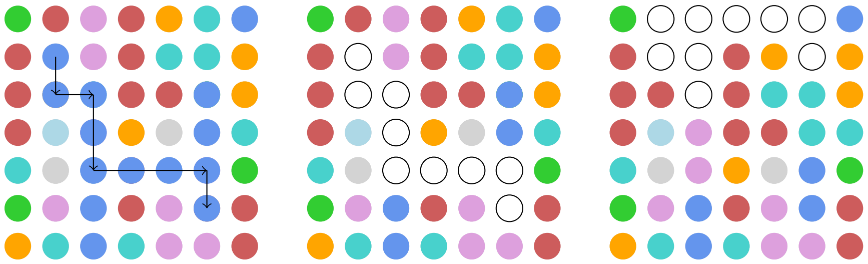


Figure 1 A depiction of a regular move. The first panel shows the set of locations of a move, the second panel shows the voids created, and the third panel shows how dots fall due to gravity.

combinatorial structure and have been studied from the computational complexity perspective. A noteworthy example is that of Candy Crush that has been shown to be NP-complete together with other *match-three* games in [12]. Another game having a somewhat similar mechanic to Two Dots is Flow Free: initially only a small number of dots of the board are colored, each color appearing exactly twice, while all the other positions are filled with *uncolored* dots. A move consists of connecting the two dots of a matching color by a path that traverses only uncolored-dots, which then inherit the color of the path's endpoints. The player is challenged to connect all the matching pairs while coloring all the dots on the board, which is equivalent to finding an embedding of monochromatic, non-intersecting, paths on the game board. This problem is also known as Zig-Zag Numberlink and has been shown to be NP-complete in [1]. The gameplay of Button and Scissors is also similar to the one of Two Dots: the player selects a monochromatic horizontal, vertical, or diagonal path traversing at least two dots (buttons), which are then removed (cut) from the board (the remaining dots are unaffected by gravity). It has been shown that clearing the board is NP-hard [11] even when only two colors are involved or when each color is used by at most 4 dots [6].

More broadly, all these games belong to the class of *casual* games. Casual games are often characterized by a puzzle-like gameplay and simple rules, which make these games easy to play, yet difficult to master. Indeed, the quality and enjoyability of puzzles has even been linked to their computational complexity [7]. It is then not surprising that many of most successful puzzles have been shown to be NP-complete, or even harder. This is the case, e.g., of Tetris [5], the (n^2-1)-puzzle (a generalized version of the famous 15-puzzle) [17], Rush Hour [9, 8], Peg-Solitaire [18, 13], Trainyard [3], Clickomania (also known as Same Game) [4, 2], 2048 [15], and many others [14].

Organization of the paper

The paper is organized as follows: Section 2 provides the problem definition and basic notation used through the paper, in Section 3 we show that the problem is NP-complete when the the number of colors is constant, and in Section 4 we address levels of the game with a constant number of columns. Finally, the results for levels with a constant number of rows are reported in Section 5.

2 Preliminaries

An instance of `Two Dots` consists of the following:

1. A $m \times n$ grid in which each position (i, j) is occupied by exactly one dot whose color belongs to a set $\mathcal{C}$.

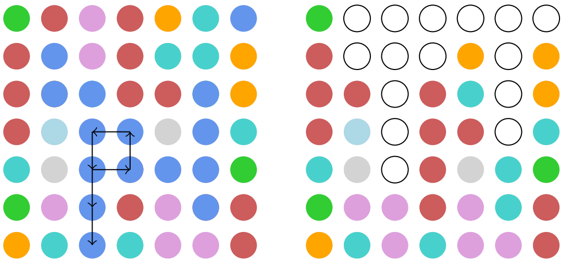


Figure 2 A depiction of a cyclic move, which has the effect of eliminating all the blue dots from the board. The example is rather similar to the above, but note the difference in the number of voids created.

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 $\ell \in \mathbb{N}$.

Intuitively, a player has a winning strategy in an instance of `Two Dots` if all the goals can be achieved within k moves. To formalize this, we need to first define moves, and the notion of dots being collected.

There are two types of moves in `Two Dots`: *regular moves* and *cyclic moves*. We first describe the regular moves, which essentially involve removing simple paths in the grid occupied by the same color. Recall that, two dots are adjacent if they occupy two neighboring position on the grid, either horizontally or vertically (dots aligned diagonally are *not* considered adjacent). Any move (either regular or cyclic) consists of a sequence of locations $\langle t_1, \ldots t_s \rangle$, with $s \geq 2$, such that all locations contain a dot of the same color and, for every $i = 2, \ldots, s$, t_i is adjacent to t_{i-1}.

- In a regular move all the locations of the sequence $\langle t_1, \ldots t_s \rangle$ are unique (see Figure 1).
- In a cyclic move all the locations of the subsequence $\langle t_1, \ldots t_{s-1} \rangle$ are unique and t_s coincides with t_j, for some $j \in \{1, \ldots, s-4\}$ (see Figure 2). Informally speaking, the locations induce a cycle with a (possibly empty) dangling path from t_j consisting of the locations in $\{t_1, \ldots, t_j\}$.

A regular move creates voids in all the locations corresponding to the sequence. A cyclic move creates voids in all the locations of the grid containing dots whose color match the color of the dots in the selected sequence.[2] All the removed dots are *collected* by the player. Then 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, pushing the voids to the top.[3] We refer the reader to Figure 1 for an illustration.

At the end of k moves, when the game is over, we say that the player has won if, for each goal (c, ℓ), the number of dots of color c collected by the player is at least ℓ.

We wish to determine whether a `Two Dots` level (i.e., an instance) can be won by the player, namely whether there exists a sequence of at most k moves that meets all the goals.

[2] Our reductions work for this simplified model which somehow contains "all the hardness" of the game. In the actual game, dots that are enclosed in the cycle of a cyclic move become bombs which, after falling down, explode and destroy their 8-neighborhood. Our reductions still work in this general model but we would then need to introduce additional gaps in our gadgets.

[3] This is a standard approach to generalize the game, but it differs slightly from the model used in [16], where new dots join the board to fill the voids. We point out that the results in [16] also work in our case.

3 Hardness of levels with three colors, two moves, and two goals

In this section, we show that `Two Dots` is NP-complete even when the number of colors is bounded by three. Since it is clear that `Two Dots` is in NP (a certificate being the sequence of moves of a solution), we now focus on showing that `Two Dots` is NP-hard. A playable version of the reduction is available at `https://twodots.isnphard.com`.

We reduce from the Exact Cover by 3-Sets problem (`X3C` for short). In an `X3C` instance we are given: (i) a set $\mathcal{I} = \{I_1, I_2, \dots, I_{3n}\}$ of $3n$ items; and (ii) a collection $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ of m subsets of $\mathcal{I}$, each subset having cardinality exactly 3. The problem is that of determining whether there exists a collection $\mathcal{S}' \subseteq \mathcal{S}$ of n sets such that $\bigcup_{S \in \mathcal{S}'} S = \mathcal{I}$, i.e., each item belongs to exactly one set in $\mathcal{S}'$. This problem is well-known to be NP-complete (see, for example, [10]). W.l.o.g. we will assume that n is an odd number.

The overview of our reduction is shown in Figure 3. We focus on describing the construct in the gray box, ignoring the first and last few columns. The middle columns correspond to items in groups of five — thus the first five columns encode the first item, and so forth. The dots are arranged in what we refer to as *wires*. There is one horizontal wire made of red dots and m horizontal wires made of blue dots. The red wire is a *check-wire* while the other m wires consisting of blue dots are *set-wires*. There are two empty rows between any pair of consecutive wires. To complete the board, we connect together the left (resp. right) endpoints of the set-wires using a single column of blue dots. Then, we introduce several more columns — the exact number of which we will specify later — on the extreme left and the extreme right of the board. These columns are populated with blue dots on rows corresponding to the set wires and are connected to the two blue columns joining the set-wires with a single dot each, on the top-right and on the bottom-left position, respectively (see Figure 3). All remaining dots in the grid are occupied by green dots.

The player is asked to eliminate a sufficiently large number of blue dots and all the red dots in two moves. The layout of the check-wire is such that all the red dots cannot all be eliminated in the first move. Further, we need to use at least one move to meet the goal for the blue dots. Therefore, in a winning strategy, the first move must involve the blue dots and achieve two things: (i) the move should clear the desired number of blue dots, and (ii) the move should result in the "alignment" of the red dots on the check-wire. The wires are set up in such a way that the dots on the check-wire align only when the dots removed from the set-wires correspond to the sets of a solution of the `X3C` instance.

We now describe the check-wire and set-wires in greater detail. First, let us establish some notation. Given a `X3C` instance with $3n$ items and m sets, the board in the reduced instance of `Two Dots` will have $5n + 2T$ columns and $4(m + 1)$ rows, where T is a parameter that we will fix later. In our discussion, we use $p_1, \dots, p_T$ and $q_1, \dots, q_T$ to label the first and last T columns, respectively. The remaining columns (indexed from $T + 1$ to $T + 5n$) are labeled by $c_1, \dots, c_{5n}$ and use C to refer to this subset of columns. On the other hand, we index the rows simply by their numbers, with the topmost row being the first.

The set-wires. For each set $S_j \in \mathcal{S}$ there is a two-cell tall set-wire traversing all columns in C. The j-th wire is on rows $r_j = 4(j + 1)$ and $r_j - 1$.

The wire is constructed as follows: For each item $I_i \in \mathcal{I}$ we consider the sub-grid consisting of 10 cells on rows r_j and $r_j - 1$, and on columns c_i to $c_i + 4$. If $I_i \notin S_j$ the lower row of this sub-grid is filled with blue dots while the top row remains empty (see the highlighted sub-grid corresponding to set S_2 and item I_4 in Figure 3). If $I_i \in S_j$ then the we place a blue dot on (i) all the cells of the bottom row of the sub-grid except for the one on column $c_i + 2$, and (ii) the cells on the top row of the sub-grid that are on columns $c_i + 1$, $c_i + 2$ and

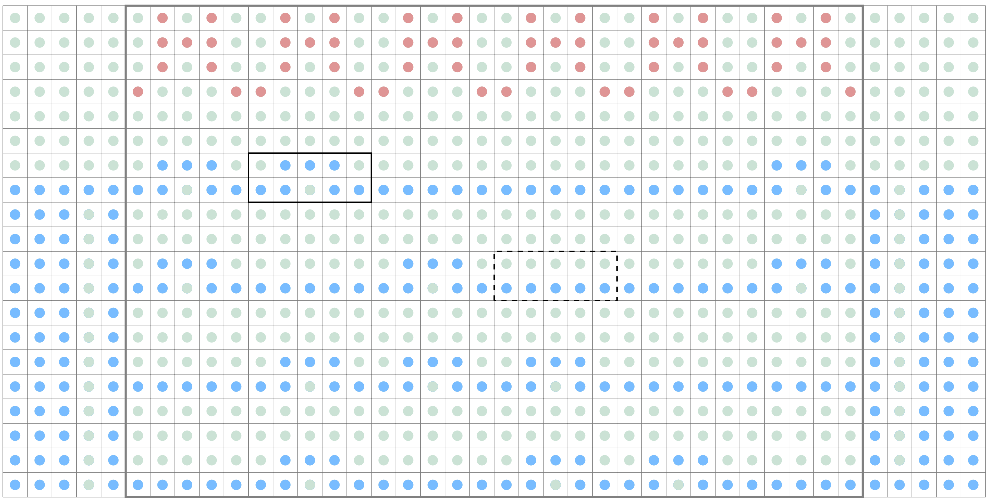


Figure 3 Overview of the reduction.

$c_i + 3$ (see the highlighted sub-grid corresponding to set S_1 and item I_2 in Figure 3).

Notice that if all the dots of the set-wire corresponding to, say, the set S_j are removed, then all the dots above row r_j will fall by exactly 1 cell, except for the ones on columns $c_i + 1$ and $c_i + 3$ where i is such that $I_i \in S_j$: in these columns, the dots above row r_j will fall by exactly 2 cells.

Finally, since the number of items contained in each set is exactly 3, notice that the number of blue dots in each set-wire is exactly $b := 5 \cdot (3n - 3) + 7 \cdot 3 = 15n + 6$.

The check-wire. The check-wire is a four-cell tall wire that is initially placed at top of the board. It is constructed simply by repeating the same pattern of 9 red dots every 5 columns, i.e., column c_{i+5} has the same layout of column c_i. The pattern is shown in Figure 4 (a).

Suppose that all the blue dots contained the set-wires corresponding to solution $\mathcal{S}'$ of the X3C instance are removed by the player, from top to bottom. This would cause all the dots of the check-wire to fall by exactly n rows, except for the dots on column $c_i + 1$ and $c_i + 3$ for $i = 1, \dots, 3n$ that will fall by exactly $(n + 1)$ cells. This will cause the dots on the items-wire to arrange in the configuration shown in Figure 4 (b). Notice that, in this configuration, all the $9 \cdot 3n = 27n$ dots of the items-wire can be removed by the player using a single move.

The reduction from X3C to Two Dots

We are now able to prove our result:

▶ **Theorem 1.** *Two Dots is* NP-complete *even when the numbers of colors, moves, and goals are bounded by 3, 2, and 2, respectively.*

Proof. Let $A = \langle \mathcal{I} = \{I_1, \dots, I_{3n}\}, \mathcal{S} = \{S_1, \dots, S_m\} \rangle$ be an instance of X3C and consider the corresponding instance B of Two Dots as described above, where T is such that the number η of blue dots in the first $T - 2$ columns (resp. the last $T - 2$ columns) is greater than the number of remaining blue dots.

Assume, without loss of generality, that $\mathcal{S}$ is not itself a set cover for $\mathcal{I}$ and remember that n is odd. We will show that at least 2η blue dots and all the red dots can be removed from B using at most two moves if and only if A admits an exact cover by 3-sets.

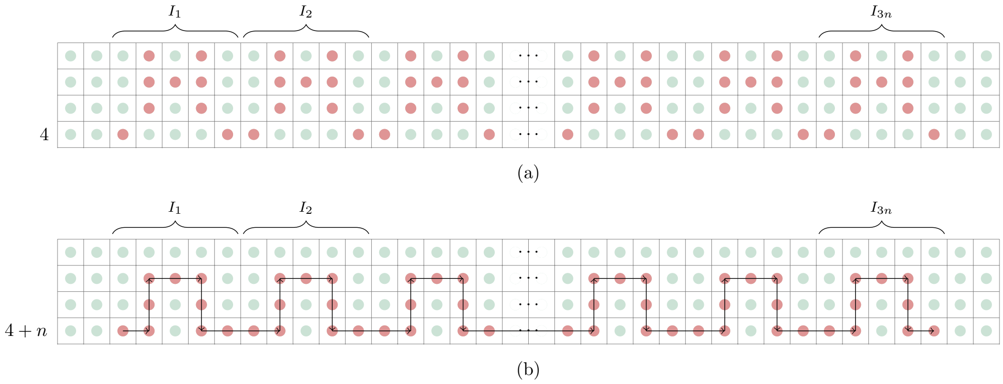


Figure 4 (a) Initial setup of the check-wire. (b) The check-wire once it gets aligned.

The Forward Direction. Let $\mathcal{S}' = \{S_{j_1}, S_{j_2}, \ldots, S_{j_n}\}$ exact cover by 3-sets for A where $j_k \in \{1, \ldots, m\}$ for $k = 1, \ldots, n$. We assume, w.l.o.g., that $j_1 < j_2 < \cdots < j_n$. A winning sequence of moves for the instance of B consists of:

1. connecting, in order and in a single move: all the blue dots in the fist $T - 2$ columns, the single dot in the $(T-1)^{th}$ column, all the dots of the set-wires on rows $r_{j_1}, \ldots r_{j_n}$ in a zig-zag fashion, the single dot in the $(T + 5n + 1)^{th}$ column, and finally all the dots in the last $T - 2$ columns. Notice that the above move is feasible since n is odd.
2. connecting, in a single move, all the red dots. This is possible since $\mathcal{S}'$ is an exact cover, and hence the previous move will cause the check-wire to align.

It is easily checked that both goals are satisfied (notice that the first and last $T - 2$ columns contain 2η blue dots).

The Reverse Direction. Suppose now that there is a solution to the instance B of `Two Dots`. From the fact that: (i) we are permitted only two moves; (ii) we have to clear all the red dots; and (iii) all the red dots are not aligned in the initial state, it follows that the first move has to meet the goal for the blue dots and also align the red dots on the check-wire so that the second move can be used to eliminate all of them in one move. This means that the first move cannot be a cycle-move, that it must involve dots that belong to both the first and the last $T - 2$ columns of the board, and that it must traverse set-wires entirely and in a zig-zag fashion. Let $\mathcal{S}'$ be the sets corresponding to the sets-wires whose dots have been removed in the first move. Suppose, towards a contradiction, that $\mathcal{S}'$ is not an exact cover for A. This means that at least one of the following conditions is true: (i) there exists an item I_j that is not covered by $\mathcal{S}'$; or (ii) there exists one item I_j that belongs $t \geq 2$ sets in $\mathcal{S}'$. In the former case the number of blue dots removed from the columns associated with item I_j is the same, and hence it will not be possible to connect all the dots in the check-wire in a single move. In the latter case, the number of blue dots removed from the 2^{nd} and 4^{th} column associated with item I_j exceeds the corresponding number of removed blue dots for the 1^{st}, 3^{rd}, and 5^{th} column by $2t$. Hence, after the first move, the red dots in the check-wire are not aligned and therefore it is not possible to meet the read goal using a single additional move. ◀

4 Boards with a constant number of columns

In this section, we address the complexity of `Two Dots` on boards that have a constant number of columns. More precisely, we show that the problem is NP-complete if the board has two or more columns, while it is polynomial-time solvable in the one-column case. Interestingly, our hardness result holds even when the player has an unlimited number of moves and only one goal to achieve.

4.1 Hardness of levels with two columns, one goal and unlimited moves

We proceed here by a reduction from `3-SAT`. Let $C_1, \ldots, C_m$ be a set of clauses over the variables $x_1, \ldots, x_n$. We assume, without loss of generality, that every clause consists of exactly three literals. The overall structure of the `Two Dots` instance that we construct is given in Figure 5. We describe the components starting from the bottom. First, we stack up a collection of *clause gadgets*, one corresponding to each variable of the `3-SAT` instance. Then, after a suitable gap, we introduce the *variable gadgets*, one corresponding to each variable of the instance. Finally, we have a *formula-check* gadget, which is the basis for the only goal that we have in this instance. We introduce one color for every literal and one for every clause of the `3-SAT` instance. Let the colors associated with the literals x_i and $\overline{x_i}$ be p_i and q_i, respectively; while we denote the color associated with the clause C_j by ℓ_j. We also have one special color that we denote by $\int$. We now describe each gadget separately and then explain the equivalence of the instances. In the following, when we speak of gaps in the board, we may assume these to be dots of "dummy" colors, which are newly introduced colors distinct from the colors mentioned already, and also distinct from each other.

The clause gadgets. Consider a clause C_j. The gadget corresponding to a clause is shown in part (a) of Figure 6. Let a, b, c be the colors corresponding to the literals of C_j. The first row is a gap row, and the next seven rows[4] consist of the following:

- Dots colored ℓ_j occupy the first column on all seven rows;
- Dots with colors a, b and c occupy the second column on the third, fifth and seventh rows, respectively; and
- Dots colored ℓ_j occupy the remaining rows on the second column.

After these rows, we introduce another gap row. Finally, the first column of the last three rows are occupied by dots colored a, b and c, respectively; while the second column on the last three rows are occupied by dummy dots. Note that because of the way the seven rows described above are "sandwiched" between gap rows, the only way to obtain a ℓ_j-colored square is to make a square move with at least one of a, b or c.

The variable gadgets. For a variable x_i, the variable gadget consists of four rows, alternatingly occupied by dots of colors p_i and q_i on both columns (see Figure 6(c)). To be specific, the first and third rows have dots colored p_i on both columns, while the second and fourth rows have dots colored q_i on both columns. Observe that within the scope of this gadget, any valid gameplay can involve a square move on either p_i or q_i, but not both.

[4] Recall that our convention is to count rows from the top.

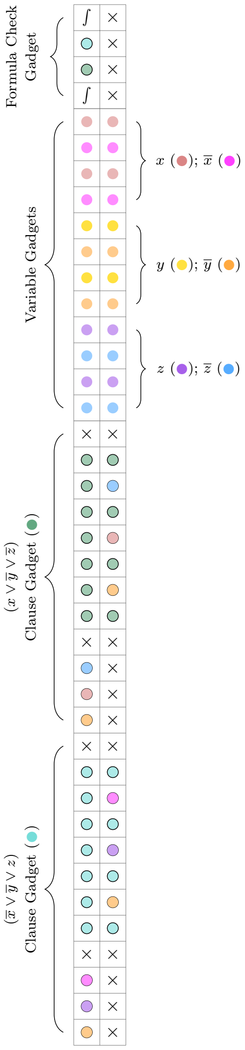

Formula Check Gadget
Variable Gadgets
x (●); $\overline{x}$ (●)
y (●); $\overline{y}$ (●)
z (●); $\overline{z}$ (●)
$(x \vee \overline{y} \vee \overline{z})$ Clause Gadget (●)
$(\overline{x} \vee \overline{y} \vee z)$ Clause Gadget (●)
$\int$
×

Figure 5 Overview of the reduction. The grid cells marked × are filled with distinct colors different from the ones used to represent variables and clauses. The goal of the game is to hit two dots of color $\int$ and the number of moves are unbounded.

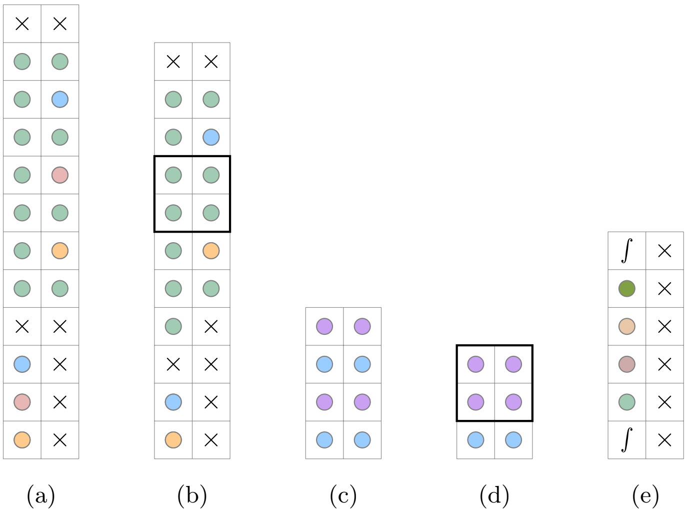


Figure 6 (a) Initial setup of the clause gadget. (b) The clause gadget in its aligned state. (c) Initial setup of the variable gadget. (d) The state of the variable gadget after one move on one of the literals. (e) The formula-check gadget.

The formula-check gadget. The formula-check gadget is depicted in Figure 6(e). It consists of $(m+2)$ rows, where the dots occupying the first and last row of the first column have color $\int$ and the interim rows comprise of one dot each of color ℓ_j, $1 \leq j \leq m$. For all rows, we have dummy dots occupying the second column. The only goal in the game will be to hit two dots colored $\int$.

We are now ready to prove our main theorem for this section:

▶ **Theorem 2.** *`Two Dots` is* NP-complete *on boards that have only two columns, even when the player has to achieve only one goal with an unlimited number of moves at his disposal.*

Proof. We proceed by a reduction from `3-SAT`. Let an instance $\mathcal{I}$ of `3-SAT` comprise of the clauses $C_1, \ldots, C_m$ over the variables $x_1, \ldots, x_n$, where every clause consists of exactly three literals. Let B denote the instance of `Two Dots` constructed as described above. Recall that the goal is to hit two dots colored $\int$ and there is no bound on the number of moves. We now establish the equivalence of the instances.

The Forward Direction. Let $\tau : V \to \{0,1\}$ be a satisfying assignment for the instance $\mathcal{I}$. For any variable x for which $\tau(x) = 1$, we eliminate the row containing dots colored q_i and perform a square move on the dots of color p_i, which becomes feasible once the q_i-colored dots are removed from either row of the variable gadget. On the other hand, for any variable x for which $\tau(x) = 0$, we eliminate the row containing dots colored p_i and perform a square move on the dots of color q_i. Observe that after each square move on a variable gadget, the design of the clause gadgets ensures that the number of dots hit in both columns is equal. Therefore, after these moves are complete, an ℓ_j-colored square is created for every $1 \leq j \leq m$. Making these moves in any order leaves us with a board where the $\int$-colored dots become adjacent, and the goal can be met with one final move.

The Reverse Direction. A winning gameplay involves a move that hits at least two $\int$-colored dots on board. Recall that the only $\int$-colored dots are available in the formula check gadget and that they are separated by m dots corresponding to the colors of the clauses. Since all other adjacent locations are occupied by dummy dots, it follows that the only way to arrive at a configuration where the two $\int$-colored dots are adjacent is to play a ℓ_j-cyclic move for all $1 \leq j \leq m$. For any such j, consider the clause gadget corresponding to C_j and let a, b, c denote the colors of the literals that appear in C_j. A cycle with dots colored ℓ_j can only manifest if there was a square move involving one of the colors a, b or c. We now propose an assignment based on these moves: set the variable x_i to 1 if the given gameplay involved a square move on the color p_i and set x_i to 0 if the gameplay involved a square move on the color q_i . If the gameplay did not have a square move on either p_i or q_i, then set the value of x_i arbitrarily. Note that this is a well-defined assignment since no valid gameplay can involve a square move on both p_i and q_i, by the design of the variable gadget and the fact that the variable gadget is the only part of the overall construction where dots of these colors are adjacent. To see that this is a satisfying assignment, proceed by contradiction: if a clause C_j is not satisfied, then it is easy to see that the choice of square moves amongst the variable gadget which led us to our assignment were such that no ℓ_j-squares were generated, which contradicts our assumption that we started with a winning gameplay. ◀

4.2 A polynomial-time algorithm for levels with one column

Let B be an instance of `Two Dots` where the board consists of one column with n dots. For $1 \leq i \leq n$, let $c(i)$ denote the color of the i^{th} dot in B. Moreover, for $i \leq j$, let $B_{i,j}$ denote the subsequence of dots starting at the i^{th} row and ending at the j^{th} row. By a slight abuse of notation, we also use $B_{i,j}$ to denote the natural instance of `Two Dots` associated with this (truncated) board. As a warm-up, and since this is instrumental to our general algorithm, we begin by considering the case in which the goal is to remove *all* the dots in the board (i.e., to *clear* the board) using the minimum number of moves.

4.2.1 Clearing the board

Let $C(i, j)$ be the minimum number of moves needed to clear $B_{i,j}$, or $+\infty$ if there exists no such sequence of moves. We now describe a dynamic programming algorithm to compute all the values $C(i, j)$ (and, in particular, $C(1, n)$).

If $j - i + 1 \leq 0$, or if $j - i + 1 = 1$, then clearly $C(i, j) = 0$ and $C(i, j) = +\infty$, respectively. We therefore consider the case in which $j \geq i + 1$. Notice that, in order to clear the board $B_{i,j}$, the dot on the first row of $B_{i,j}$ must be hit with a move connecting (at least) one other dot. Let h be the smallest index of a row in $B_{i,j}$ that contains one such dot. We distinguish two cases:

1. The move hitting the dot on the first row of $B_{i,j}$ connects exactly 2 dots.
2. The move hitting the dot on the first row of $B_{i,j}$ connects 3 or more dots.

If the former case we "guess" the location h of the dot that partners with the first dot of $B_{i,j}$. This decomposes our instance into two sub-instances corresponding to the boards $B_{i+1,h-1}$ and $B_{h+1,j}$. In formulas:

$$C_1(i, j) = \min_{i<h\leq j \text{ such that } c(h)=c(i)} \{C(i + 1, h - 1) + C(h + 1, j) + 1\},$$

where last term accounts for to the move that is used to hit the i^{th} dot and the h^{th} dot.

In the latter case we still guess the location h, but we now decompose the board into two different sub-instances, namely $B_{i+1,h-1}$ and $B_{h,j}$ (i.e., we still include the h^{th} dot in the second sub-instance). The h^{th} dot can then be combined with another dot in row $h' > h$

belonging to the same move that is used to hit both the i^{th} and the h^{th} dot. This can be repeated recursively until the last two dots hit by the move are considered, which falls into the former case, and thus accounts for the whole multi-dot move. In formulas:

$$C_2(i,j) = \min_{i<h\leq j \text{ such that } c(h)=c(i)} \{C(i+1,h-1) + C(h,j)\}.$$

Overall, our recurrence is given by $C(i,j) = \min\{C_1(i,j), C_2(i,j)\}$.

4.2.2 The general case

For the sake of simplicity we describe our algorithmic approach for the case when the game has only one goal, and our task is to determine the minimum number of moves that are needed to achieve said goal. In particular, suppose that the single goal demands the elimination of ℓ red dots. Nevertheless, our approach can be easily generalized for the case of multiple goals.

We now describe a dynamic programming routine to check if there is a sequence of at most k moves that hits at least ℓ red dots. To this aim, let $T(i,j,\delta)$ be the minimum number of moves needed to gain at least δ red dots in $B_{i,j}$, or $+\infty$ if there is no such sequence of moves. By our definition, we have that, if $\delta \leq 0$, then $T(i,j,\delta) = 0\ \forall i,j$. If $\delta > 0$ and $j <= i+1$, then $T(i,j,\delta) = +\infty$. Otherwise, we consider the following three cases:

1. There is a solution that does not hit the i^{th} dot.
2. There is a solution that hits the i^{th} dot along with one other dot.
3. There is a solution that hits the i^{th} dot along with two or more other dots.

To describe our recurrence corresponding to these cases, we denote by $\nabla(i',j')$ the number of red dots in $B_{i',j'}$. We start by addressing the first case, in which the i^{th} dot can be simply ignored from the current board:

$$T_1(i,j,\delta) = T(i+1,j,\delta).$$

In the second case, we "guess" the location h of the dot that partners with the i^{th} dot in an optimal move. To this end, we consider separately the subsequences of dots corresponding to $B_{i+1,h-1}$ and to $B_{h+1,n}$. This two sub-instances are handled differently as, in order to connect the i^{th} dot with the h^{th} dot, all the dots in $B_{i+1,h-1}$ need to be removed (while this is not true for $B_{h+1,n}$. We can write:

$$T_2(i,j,\delta) = \min_{i<h\leq j \text{ such that } c(h)=c(i)} \{C(i+1,h-1) + T(h+1,j,\delta - \nabla(i,h)) + 1\}.$$

In the last case, we have the following analogous recurrence:

$$T_3(i,j,\delta) = \min_{i<h\leq j \text{ such that } c(h)=c(i)} \{C(i+1,h-1) + T(h,j,\delta - \nabla(i,h-1))\}.$$

Overall, our recurrence is given by $T(i,j,\delta) = \min\{T_1(i,j,\delta), T_2(i,j,\delta), T_3(i,j,\delta)\}$, and we can state the following.

▶ **Theorem 3.** *`Two Dots` admits a polynomial time algorithm for a constant number of goals when the board consists of only one column.*

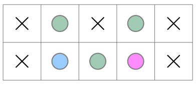


Figure 7 The edge gadget for the case of boards with two rows. Here we are representing an edge e (🟢) incident on vertices u (🔵) and v (🟣).

5 Boards with a constant number of rows

Here, we state and prove the following result, strengthening the NP-hardness result for four rows given in [16]. Unlike for the case of two columns, the reduction in this setting employs the use of many goals and many colors.

▶ **Theorem 4.** *`Two Dots` is* NP-complete *when the board has only two rows and can be solved in polynomial time for boards that have one row.*

Proof. The hardness for the case of two rows is by a reduction from `Vertex Cover`. Let the $(G = (V, E), k)$ be an instance of `Vertex Cover` where $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$. Our board consists of two rows and $2n + 5m$ columns. For every vertex v_i, introduce four dots of color c_i such that they form a square on the adjacent columns $2i - 1$ and $2i$. For every edge $e_j = (v_p, v_q)$, introduce the edge gadget shown in Figure 7, which involves three dots that have color d_j and one dot each of color c_p and c_q. The remaining five positions in the grid are filled with dummy dots that have colors distinct from the colors that correspond to edges and vertices. The goal is to hit at least two dots of color d_j for each $1 \leq j \leq m$ in at most $m + k$ moves.

We now argue the equivalence of these instances. In the forward direction, given a vertex cover $S \subseteq V$ of size k, perform the square moves on the colors corresponding to vertices in S. This uses up the first k moves. Since S is a vertex cover, this causes at least two dots of color d_j to become adjacent in the edge gadget corresponding to e_j. The remaining m moves can be used to now meet the demands of the game. In the other direction, assume we have a valid gameplay that meets all the goals. Note that at least m moves must be used to hit dots of color d_j. Let S denote the set of colors on which square moves were employed in the remaining moves. Note that this is a set of at most k colors, each of which corresponds to a vertex in the graph G. We claim that this subset is a vertex cover. Indeed, if not, observe that the edge gadget corresponding to any uncovered edge (say e_j) remains unchanged and therefore the goal for the color d_j cannot be met, contradicting our assumption that we started with a winning gameplay.

We now turn to the case of boards with one row, where we claim that a natural greedy algorithm solves `Two Dots` in polynomial time. First, note that the goal of hitting k dots of color c can be met if and only if the total number of dots colored c present in intervals of length at least two is at least k. Further, it is easily checked that in an optimal play, every move hits colors for which there is a non-trivial goal left (note that this is not true in the general game, where moves involving colors that have no goals associated with them can also help with meeting the goals of the game). Finally, observe that we may employ a greedy strategy here to meet any particular goal, where we proceed by hitting maximal intervals of the longest length of a particular color first. ◀

6 Conclusions

In this paper we have settled the computational complexity of several restrictions of Two Dots involving narrow boards and/or few colors. Some problems which are still open and that we regard as interesting are those of understanding the computational complexity of Two Dots for (i) boards with only two colors, and (ii) boards with a constant number of columns and colors, which nicely captures the spirit of the game. Finally, we remark that – by carefully positioning the wire gadgets, and employing some other small modifications – our reduction involving 3 colors, 2 moves, and 2 goals, can be adapted to require only 1 goal.

References

1 Aaron B. Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-zag numberlink is np-complete. *JIP*, 23(3):239–245, 2015. doi:10.2197/ipsjjip.23.239.

2 Aviv Adler, Erik D Demaine, Adam Hesterberg, Quanquan Liu, and Mikhail Rudoy. Clickomania is hard, even with two colors and columns. *The Mathematics of Various Entertaining Subjects: Research in Games, Graphs, Counting, and Complexity*, 2:325, 2017.

3 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is np-hard. *Theoretical Computer Science*, 2017. doi:10.1016/j.tcs.2017.09.039.

4 Therese C. Biedl, Erik D. Demaine, Martin L. Demaine, Rudolf Fleischer, Lars Jacobsen, and J. Ian Munro. The complexity of clickomania. *CoRR*, cs.CC/0107031, 2001. URL: http://arxiv.org/abs/cs.CC/0107031.

5 Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *Int. J. Comput. Geometry Appl.*, 14(1-2):41–68, 2004. doi:10.1142/S0218195904001354.

6 Kyle Burke, Erik D. Demaine, Harrison Gregg, Robert A. Hearn, Adam Hesterberg, Michael Hoffmann, Hiro Ito, Irina Kostitsyna, Jody Leonard, Maarten Löffler, Aaron Santiago, Christiane Schmidt, Ryuhei Uehara, Yushi Uno, and Aaron Williams. Single-player and two-player buttons & scissors games - (extended abstract). In Jin Akiyama, Hiro Ito, Toshinori Sakai, and Yushi Uno, editors, *Discrete and Computational Geometry and Graphs - 18th Japan Conference, JCDCGG 2015, Kyoto, Japan, September 14-16, 2015, Revised Selected Papers*, volume 9943 of *Lecture Notes in Computer Science*, pages 60–72. Springer, 2015. doi:10.1007/978-3-319-48532-4_6.

7 Davide Eppstein. Computational complexity of games and puzzles. https://www.ics.uci.edu/ eppstein/cgt/hard.html, accessed on the 23rd of February 2018.

8 Henning Fernau, Torben Hagerup, Naomi Nishimura, Prabhakar Ragde, and Klaus Reinhardt. On the parameterized complexity of the generalized rush hour puzzle. In *Proceedings of the 15th Canadian Conference on Computational Geometry, CCCG'03, Halifax, Canada, August 11-13, 2003*, pages 6–9, 2003. URL: http://www.cccg.ca/proceedings/2003/22.pdf.

9 Gary William Flake and Eric B. Baum. Rush hour is pspace-complete, or "why you should generously tip parking lot attendants". *Theor. Comput. Sci.*, 270(1-2):895–911, 2002. doi:10.1016/S0304-3975(01)00173-6.

10 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

11 Harrison Gregg, Jody Leonard, Aaron Santiago, and Aaron Williams. Buttons & scissors is np-complete. In *Proceedings of the 27th Canadian Conference on Computational Geometry, CCCG 2015, Kingston, Ontario, Canada, August 10-12, 2015.* Queen's University, Ontario,

Canada, 2015. URL: http://research.cs.queensu.ca/cccg2015/CCCG15-papers/48.pdf.

12 Luciano Gualà, 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, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932866.

13 Luciano Gualà, Stefano Leucci, Emanuele Natale, and Roberto Tauraso. Large peg-army maneuvers. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.18.

14 Robert A. Hearn and Erik D. Demaine. *Games, puzzles and computation.* A K Peters, 2009.

15 Stefan Langerman and Yushi Uno. Threes!, fives, 1024!, and 2048 are hard. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.22.

16 Neeldhara Misra. Two dots is np-complete. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 24:1–24:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.24.

17 Daniel Ratner and Manfred K. Warmuth. Nxn puzzle and related relocation problem. *J. Symb. Comput.*, 10(2):111–138, 1990. doi:10.1016/S0747-7171(08)80001-6.

18 Ryuhei Uehara and Shigeki Iwata. Generalized Hi-Q is NP-complete. *IEICE Transactions (1976-1990)*, 73(2):270–273, 1990.

On the PSPACE-completeness of Peg Duotaire and other Peg-Jumping Games

Davide Bilò
Dipartimento di Scienze Umanistiche e Sociali, University of Sassari, Italy.
davide.bilo@uniss.it
https://orcid.org/0000-0003-3169-4300

Luciano Gualà
Dipartimento di Ingegneria dell'Impresa, University of Rome "Tor Vergata", Italy.
guala@mat.uniroma2.it
https://orcid.org/0000-0001-6976-5579

Stefano Leucci
Institute of Theoretical Computer Science, ETH Zürich, Switzerland.
stefano.leucci@inf.ethz.ch
https://orcid.org/0000-0002-8848-7006

Guido Proietti
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, University of L'Aquila, Italy, and Istituto di Analisi dei Sistemi ed Informatica, CNR, Roma, Italy.
guido.proietti@univaq.it
https://orcid.org/0000-0003-1009-5552

Mirko Rossi
Dipartimento di Ingegneria dell'Impresa, University of Rome "Tor Vergata", Italy.
r.mirko25@gmail.com

Abstract

Peg Duotaire is a two-player version of the classical puzzle called Peg Solitaire. Players take turns making peg-jumping moves, and the first player which is left without available moves loses the game. Peg Duotaire has been studied from a combinatorial point of view and two versions of the game have been considered, namely the single- and the multi-hop variant. On the other hand, understanding the computational complexity of the game is explicitly mentioned as an open problem in the literature. We close this problem and prove that both versions of the game are PSPACE-complete. We also prove the PSPACE-completeness of other peg-jumping games where two players control pegs of different colors.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases peg duotaire, pspace-completeness, peg solitaire, two-player games

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.8

1 Introduction

Peg-Jumping games are games with one or more players that are played on boards of different shapes. Each position of the board can host at most one peg, and a move consists of *jumping* a peg over an (horizontally or vertically) adjacent peg into an empty position. The move causes the peg that is jumped over to be removed from the board (see Figure 1). Arguably, the most popular game in this class is the single-player puzzle called *Peg Solitaire* (also

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 8; pp. 8:1–8:15
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

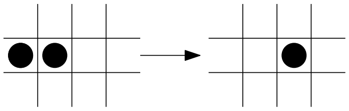


Figure 1 A move in peg-jumping games.

known as *Hi-Q*), in which the aim is to find a sequence of moves which reduces an initial placement of pegs into a single peg (and thus, the goal is that of clearing the board). A classical instance of the game has a cross-shaped board full of pegs except for the central position. The Peg Solitaire is an ancient game and its history dates back to at least the 17th century (see [2] for a comprehensive overview on the game).

Several other single-player peg-jumping games have been considered. For example, in the *Solitaire-Reachability* [15], the goal is, given an initial configuration of pegs, to find a sequence of moves that places any peg on a given target position (it is not required to remove all the other pegs). Another prominent game in this class, with a slightly different flavor, is the *Solitaire-Army* problem: given a *desert* region on a (usually infinite) board, and a target position in this region, one wishes to find an initial configuration of pegs outside of the desert that allows a peg to reach the target position through a valid sequence of moves. In its classical formulation, introduced by J.H. Conway in 1961, the desert is a half-plane, and the challenge is to understand what is the farthest distance in the desert that allows the target position to be reached. Conway devised an elegant potential argument to show that distance 5 cannot be reached on any finite board [18]. Other desert shapes have also been considered, such as square- and rhombus-shaped deserts [8].

In this paper, we focus on 2-player peg-jumping games, mainly on *Peg Duotaire*, a game introduced in [22], in which two players alternatively make a peg move and the winner is the last player to move. Two versions of Peg Duotaire have been considered: the *single-hop* Duotaire [22, 13], where each move consists of a single-hop jump, and the *multi-hop* Duotaire [21], where a series of (single-hop) jumps with the same peg can be made on a given turn. Both variants are impartial games, and they have been studied from a combinatorial point of view, while the problem of understanding the computational complexity of Peg Duotaire is mentioned as an open problem in the book by Hearn and Demaine (Section A.4 in [17]) and [21].

Our results

We study the problem of deciding whether the first player in a Peg Duotaire instance can force a win. As our main result, we show that this problem is PSPACE-complete for both versions of the game, namely the single- and the multi-hop variant. This closes the open problem given in [17] and [21].

We also consider another peg-jumping game, namely a 2-player version of Solitaire-Reachability. In this game, pegs are partitioned into white pegs (controlled by the first player) and black pegs (controlled by the second player). A move of the first (resp., the second) player consists of a jump involving only white (resp., black) pegs. However, pegs of a given color can prevent jumps of pegs of the other color since they occupy positions of the board. Moreover, each player has a target position that wants to reach with a peg (and the two target positions might coincide). As a natural extension of Solitaire-Reachability, we assume that the winner is the first player that reaches its target position.[1] However, different types of winning conditions can be considered here. For example, we can assume –as usual in the combinatorial game community– that the winner is the player that makes that last move, or a combination of the two mentioned rules, e.g., a player wins by either reaching his target position or by leaving his opponent with no available moves. We prove that all these variants are PSPACE-complete.

Related Results

Despite of the simplicity of their rules [10, 11, 7, 5], peg-jumping games exhibit a non-trivial combinatorial richness, and for this reason attracted the attention of many researchers over their long history [6, 3, 4]. From a computational point of view, it has been shown that single-player Peg Solitaire is NP-complete when the goal is to clear the entire board [23], or when the task is to decide whether a given target position can be reached [15]. On the other hand, deciding whether a given configuration can be transformed into a single peg is polynomial-time solvable for rectangular boards of fixed (constant) height, since solvable instances form a regular language [21, 22].

As far as 2-player peg-jumping games are concerned, single-hop Duotaire was introduced in [22], and then studied in [13, 21], while the multi-hop variant has been introduced in [21], where, besides other results, it is shown that even in the one-dimensional case, the set of instances for which the first player wins cannot be described by a context-free language.

Another work which is close in spirit to our is [16], where the PSPACE-completeness has been proved for another 2-player peg-jumping game called *Konane*, an ancient Hawaiian game in which pegs are of two different colors, and a player moves a peg of his color by jumping it over a peg of the opposite color in order to capture it. A peg may make multiple successive jumps in a single move, as long as they are in a straight line (while no turns are allowed within a single move). The first player that is unable to move wins. Due to the differences of the rules, the reduction in [16] cannot be easily adapted to prove PSPACE-completeness of the games we consider here. Finally, the present work contributes to the rich literature investigating the computational complexity of combinatorial games [17, 14, 19, 1, 20, 9].

2 Single-Hop Duotaire

2.1 Overview

In this section we focus on Single-Hop Duotaire and we prove that the problem of deciding whether the first player can force the win is PSPACE-complete.

Our reduction is from *directed vertex geography (DVG).* In this problem, we want to decide whether the first player can force a win in the following game. We are given a directed graph G and a distinguished vertex $s \in V(G)$ that initially contains a *token.* Two players take turns performing the following move: first the token is moved from its current vertex u to a neighboring vertex v, traversing the edge $(u, v) \in E(G)$, and then vertex u is deleted from the graph. The first player who has no legal move loses the game (and the other player wins).

The DVG problem is known to be PSPACE-complete even when G is planar, bipartite, all the vertices have maximum degree 3, maximum indegree 2, and maximum outdegree 2

[1] In this case, we can assume that a player with no available moves can skip his turn, and that the game can end with a draw, whenever no player can move and no target position has been reached yet.

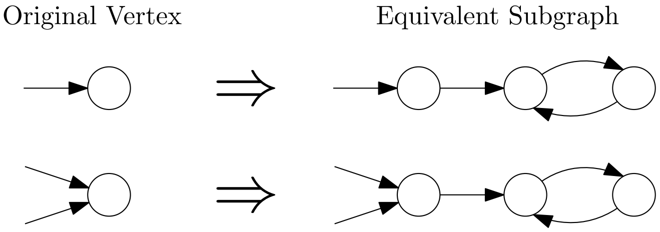


Figure 2 Vertex transformation. Original verteces with outdegree 0 (on the left), and equivalent subgraphs (on the right).

[12]. We will furthermore assume that vertex s has outdegree 2 and no incoming edges, while all the vertices in $V(G) \setminus \{s\}$ have either (i) indegree 1 and outdegree 2, (ii) indegree 2 and outdegree 1, or (iii) indegree 1 and outdegree 1. It turns out that these assumptions can be easily guaranteed by performing suitable transformations of the input graph, as we will discuss in the sequel.

The idea is to consider a planar embedding of G on a grid from which we build an equivalent instance of single-hop Duotaire where the token is simulated by a specific *token peg* (see Figure 7 for an example of the input graph G and of the associated single-hop Duotaire instance). We will simulate the behavior of the vertices of G using suitable *gadgets*: these gadgets take the token peg as an input, which encodes the act of placing the token on the associated DVG vertex (as a result of a previous move), and route it to a specific output, which corresponds to selecting the next position of the token in DVG (and hence to selecting an outgoing edge). The token peg will be transported from a vertex output (representing one endpoint of an edge) to a vertex input (representing the other endpoint) using a *wire* gadget.

Moreover, in an isolated area of the board, we place two adjacent pegs that allow the players to play a one-time extra move that we call *dummy* move. Except for this dummy move, all the other moves available to the players at any given point in time will involve the token peg.

2.2 Transforming the input DVG instance

Here we show how an instance of DVG on a planar, bipartite graph G in which all the vertices have maximum degree 3, maximum indegree 2, and maximum outdegree 2, can be transformed in order to further ensure that:

- s has outdegree 2 and no incoming edges.
- all the vertices in $V(G) \setminus \{s\}$ have either (i) indegree 1 and outdegree 2, (ii) indegree 2 and outdegre 1, or (iii) indegree 1 and outdegree 1.

First of all, we delete from G all the edges entering in s, and we iteratively remove all the other vertices of indegree 0. Then, while s has exactly one outgoing edge (s, s'), we perform the following operation: we delete s from G, we move the token to s', and we rename s' to s. It is easy to see that each such operation yields an equivalent instance in which the roles of the two players are exchanged: Player 1 can force a win in the instance preceding the operation iff Player 2 can win in the resulting instance (recall that Player 1 is always the first player to move, and that PSPACE is closed under complement).

After the transformation, G contains exactly 1 vertex with indegree 0 (i.e., s), which must also have outdegree 2. All the remaining vertices of G are either of one of the forms in (i), (ii), (iii), or they have outdegree 0 and indegree in $\{1, 2\}$. In the latter case, they can be replaced with the equivalent subgraphs shown in Figure 2.

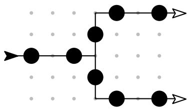


Figure 3 The gadget for a vertex with indegree 1 and outdegree 2.

2.3 Gadgets

Here we describe all the gadgets. Each gadget is meant to be played in one or more prescribed ways and is designed to ensure that any player deviating from the intended play will necessarily lose the game.

2.3.1 Vertices with outdegree two

In our reduction there are two kinds of vertices having outdegree 2, namely the starting vertex s (having indegree 0), and vertices having indegree 1.

Let us consider the case of vertices with indegree 1 first, which are implemented as shown in Figure 3. Players will be able to play the gadget whenever the token peg reaches the position marked with the black arrow. W.l.o.g., we assume that, once the token peg is in place, it is Player 1's turn. The intended play of the gadget follows the solid lines via alternating moves of the players. In particular, notice that Player 2 moves the token peg into the center of the gadget, where the solid lines meet. At this point, Player 1 can choose to jump over either the peg immediately above, or the peg immediately below. This corresponds to choosing which of the two outgoing edges of the associated vertex in the DVG instance the token traverses next. The following moves are straightforward and will bring the token peg to one of the two positions marked with the white arrows. Notice that this last move is performed by Player 1, and hence the turn will be up to Player 2.

We now argue that any deviation from the above strategy, will cause the deviating player to lose. In particular, all the deviations in this gadget consist of using one of the pegs of the gadget to jump over the token peg. However, if a player plays such a move he will bring the board in a configuration where the only available move is the dummy move. The opponent can then use this dummy move to win the game.

On the converse, if any player plays the dummy move instead of a move involving the token peg, the opponent can respond making a move that jumps over the token peg, thus reaching a configuration where no move is left available to the other player (thus winning the game).

As far as the starting vertex s is concerned, it suffices to implement it in the same way of vertices with indegree 1 we just described, with the only exception that a peg is initially placed in the position marked by the black arrow.

2.3.2 Vertices with outdegree 1

We first discuss vertices with outdegree 1 and indegree 2, which are implemented as shown in Figure 4 (a). The indented ways to play this gadget carries the token peg from any of the two input positions marked with the black arrows, to the output position marked with the white arrow. Notice that during the corresponding sequence of moves, the players will need to jump over the token peg along the solid black line. When this happens, the old token peg is removed from play, nevertheless, instead of thinking of the resulting state of the board as a configuration with no token peg, we promote the jumping peg to become the new token peg.

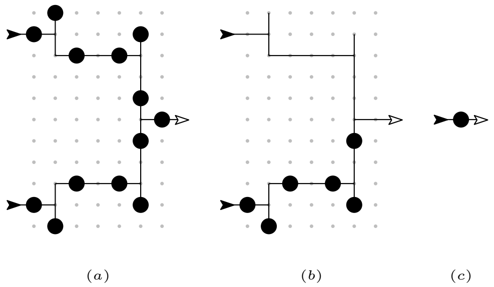


Figure 4 Gadgets for vertices with outdegree 1 and (c) indegree 1; (a) indegree 2. Picture (b) shows how the gadget in (a) looks like once players played it, which intuitively corresponds to a vertex already visited by the token in the associated DVG instance.

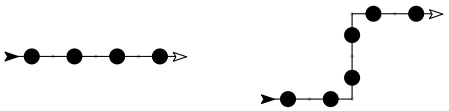


Figure 5 Wire gadgets.

As in the gadget encoding a vertex with outdegree 2, also in this case the first player making a move is also the player making the final move –placing the token peg in the output position (i.e., playing a gadget changes the turn of the next player to play).

We now argue that any player that deviates from the prescribed strategy is bound to lose. As in the previous case, notice that if a player makes a move that brings the token peg outside of the solid lines, or in the opposite direction w.r.t. the intended play, then the opponent can respond by playing the dummy move and winning the game. Notably, this also ensures that a token peg cannot be brought from its initial input position to the other input position (thus traversing the solid lines in the opposite direction).

We encode vertices having indegree and outdegree 1, with the gadget of Figure 4 (c), whose correctness is straightforward.

Wires Wire gadgets are used to encode directed edges in the DVG instance. Such a gadget receives the token peg as an input (which coincides with one of the outputs of the vertex gadget associated with the tail of the encoded edge), and carries it to its output (which coincides with the input of the vertex gadget associated with the head of the encoded edge), through an even number of alternating moves. This ensure that the player making the first move in the wire gadget will also be the next player to play after the wire gadget has been completely traversed. Some examples of wire gadgets are shown in Figure 5. Notice that by repeating the shown pattern, one can lengthen or shorten wires as needed, as well as perform 90, 180, and 270-degrees turns.

This is useful as the planar embedding of the graph G in the DVG instance will determine how to lay wires on the board. It might however happen that such an embedding results in wires with an odd number of moves. In this case, one can restore the desired parity by replacing any straight portion of a wire consisting of 4 moves (see Figure 6 (a)) with the gadget shown in Figure 6 (b), which uses the same input and output positions but requires 9 moves to be traversed.

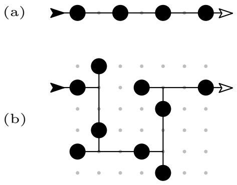


Figure 6 Picture (b): changing-parity gadget. Exactly 9 moves are needed to traverse it. The gadget can be used to change the parity of the length of a wire by replacing a portion of a 4-move straight wire (shown in (a)).

2.4 Putting all together

As we already described, we build the instance of single-hop Duotaire from a planar embedding of G, by replacing each vertex with its corresponding vertex gadget, and by connecting vertex gadgets through wires in the way prescribed by the planar embedding (see Figure 7 for an example).

When we embed all the gadgets on the board, we shall guarantee that every output of a gadget can be connected to the input of the consecutive gadget through a wire. We point out that all our gadgets are designed in such a way that this is always possible. Indeed, every gadget satisfies the following property: if an input is in the i-th row and the j-th column, where both i and j are even, then each output of the gadget is in i'-th row and j'-th column, with both i' and j' even. This is true for the wire gadget also, which allows us to connect an arbitrary pair of even-even positions.

We now show that if a player has a winning strategy in a DVG instance, then he can also force a win in the corresponding single-hop Duotaire instance. Let us consider the case in which Player 1 has a winning strategy first, and assume that all the gadgets are played in one of the intended ways (as otherwise the deviating player will lose if his opponent plays optimally). Remember that, initially, the token peg is placed on the input position of the gadget corresponding to vertex s (i.e., the black arrow of Figure 3). The two outputs of this vertex gadget correspond to the edges outgoing s in the DVG instance. Player 1 can then play the gadget in such a way that the token peg is carried to the output corresponding to the first edge traversed in his winning strategy, say $e = (s, u)$. This forces the players to traverse the wire corresponding to edge e until the input position of the gadget corresponding to vertex u is reached. Notice that, by our choice of the wire lengths, the turn is now up to Player 2. Since gadgets are always played in the intended way, Player 2 must also bring the token peg to one of the output positions of the gadget, which corresponds to an edge in DVG, say (u, v). Suppose that v is a vertex whose gadget has never been reached by the token peg so far; when the token peg reaches the corresponding input of the v's gadget (on Player 1's turn), Player 1 can respond by moving the peg towards another vertex gadget according to the DVG move prescribed by his winning strategy. Player 1 continues to play according to this scheme until he routes the peg toward a vertex w whose gadget was already traversed. Notice that vertex w must have indegree 2 (and outdegree 1), hence this is the situation depicted in Figure 4 (b) (up to symmetries). Since there are exactly 6 leftover moves along the solid lines (and any deviation causes the deviating player to lose), plus the dummy move, Player 1 is then able to win the game. In other words, any player attempting to move the token peg to a vertex that was already traversed will lose the game. A similar argument applies to the case in which Player 2 has a winning strategy in DVG.

The previous discussion, and the fact that a winning strategy for a single-hop Duotaire instance (if any) can be found by a DFS traversal of the (implicit) game tree (whose height is at most the number of pegs in the instance), allow us to state the following:

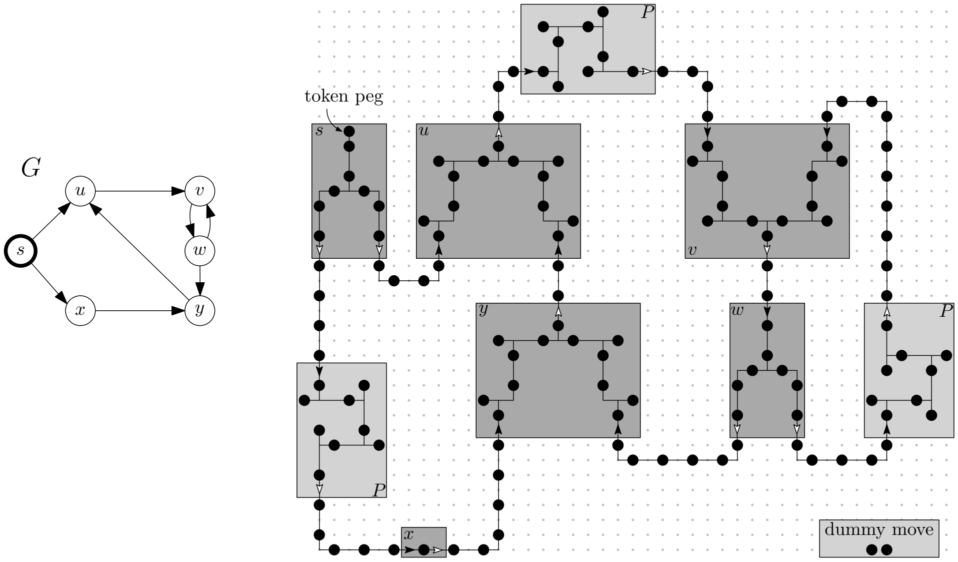


Figure 7 A DVG instance and its corresponding Single-Hop Duotaire one. Gadgets of Figure 6 are used to guarantee that every wire needs an even number of move to be traversed.

▶ **Theorem 1.** *Deciding whether the first player can force a win in single-hop Duotaire is* PSPACE-*complete.*

3 Multi-Hop Duotaire

In this section we prove that the problem of deciding whether the first player can force the win in the multi-hop Duotaire is PSPACE-complete.

Our reduction is from DVG on planar bipartite graphs of maximum degree 3. On the one hand, similarly to the reduction for single-hop Duotaire, both planarity and maximum-degree bounds are useful for embedding the instance on a board. On the other hand, bipartiteness is needed to uniquely associate each vertex with exactly one of the two players. Indeed, if the token is placed on a vertex of the left-hand side of the (vertex) bipartition and, w.l.o.g., it is Player 1's turn, then the player can only shift the token along an edge, if any, to reach a vertex of the right-hand side of the bipartition. Similarly, Player 2 can only shift the token along an edge, if any, to reach a vertex of the left-hand side of the bipartition.

The idea of the reduction is to have a token peg placed on each vertex gadget; however, each token peg needs to be *activated* by another token peg before it can be moved. At the beginning of the game, only the token peg contained in the vertex gadget representing s is active by default. Each token peg is *owned* by a specific player: in the intended play, the token pegs associated with the vertices of the left-hand side of the bipartition can be moved only by Player 1, while the remaining token pegs can be moved only by Player 2. When a token peg, say t, is active, the player owning t is first forced to jump over the token peg that activated t, and then, thanks to suitable *control* pegs, the player is forced to continue moving t along an edge until t reaches a new vertex gadget, if possible, and activates the token peg of that gadget. Therefore, once a player has moved his token peg away from a vertex gadget, no token peg is left in the gadget, and the player that tries to move his token peg inside a previously visited vertex gadget, will lose the game, due to the presence of one dummy move as in the single-hop Duotaire. Similarly to the reduction for single-hop Duotaire, every move other than the dummy move involves token pegs.

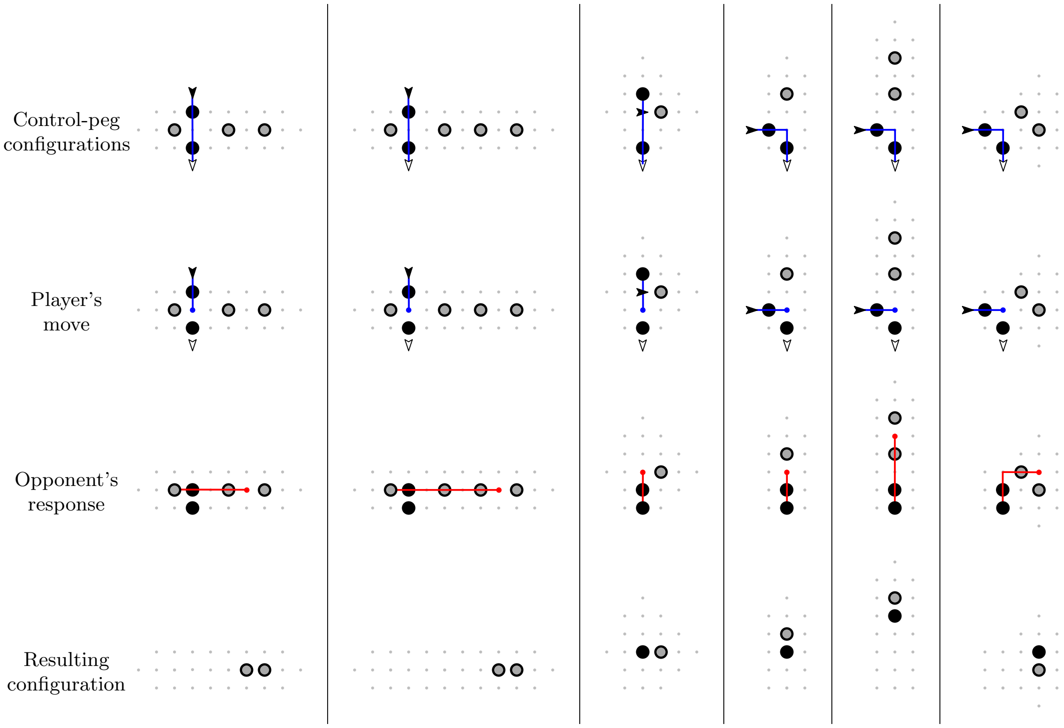


Figure 8 The 6 ways of embedding control pegs to monitor each pair of consecutive black pegs. The dots represent positions of the board nearby the control pegs that need to be empty. The picture shows the opponent's response when the moving player jumps only over the first of 2 consecutive black pegs. In all the 6 cases, the countermove of the opponent generates a dummy move. In the first 2 embeddings, the moving player can jump over the first black peg of the pair, and then also over a control peg. We observe that in this case the token peg has reached an empty area of the board, and the opponent wins the game by playing the dummy move.

3.1 Gadgets

Other than token pegs and (black) pegs that describe the main structure of the gadget, each gadget contains control (gray) pegs that are used to force players to behave in the desired way. Control pegs are embedded on the board to monitor each pair of consecutive black pegs whose corresponding Manhattan distance is equal to 2. We use 6 types of embeddings (see Figure 8). Basically, each configuration forces the player that is jumping over the first of 2 consecutive black pegs to jump also over the other black peg within the same move.

Wires. Wire gadgets are used to encode edges of the DVG instance as well as the vertex s. Each such gadget receives a token peg as an input, and carries it to its output through a single multi-hop move. This ensures that the player that is moving the token peg will also traverse the entire wire. Three examples of wires are shown in Figure 9. To avoid that the player moving the token peg would not traverse the entire wire, control pegs have been added all along the wire. Observe that the embedding of control pegs that monitor a pair of consecutive black pegs on straight wires is independent of the position of the other control gadgets placed along the wire. Finally, notice that a wire can also make 90-degree turns both clockwise and counterclockwise: indeed, the right-most drawing in Figure 9, being symmetric, can be traversed in both directions.

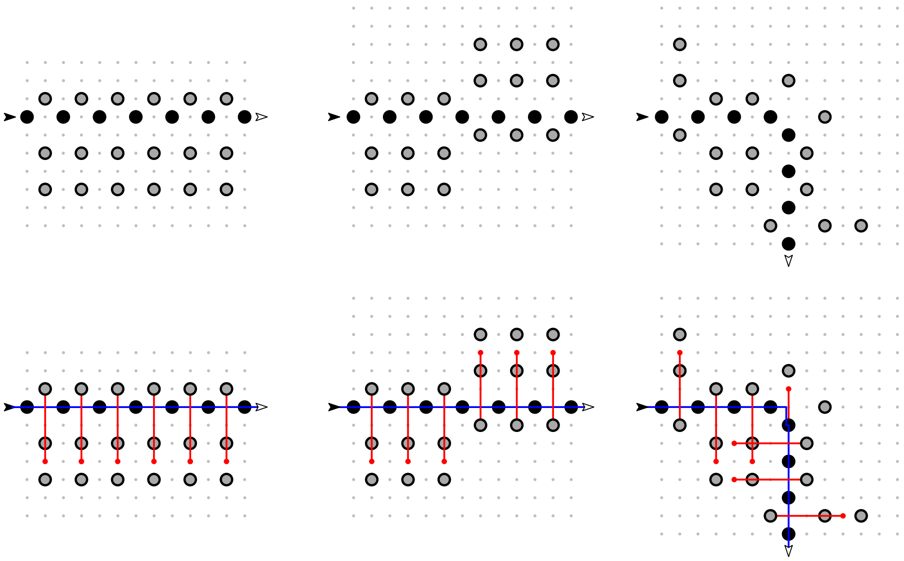


Figure 9 Straight wires and 90-degree turns.

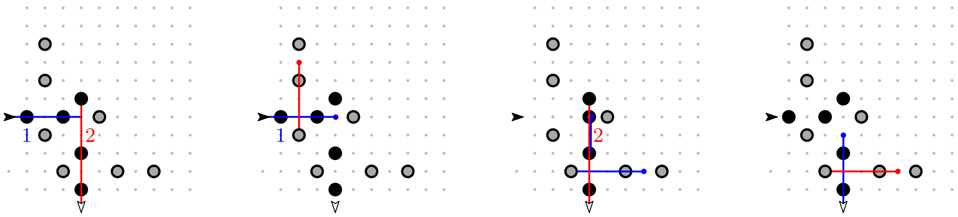


Figure 10 Vertex gadget. The rightmost drawing shows that the player that enters the gadget from its output loses.

Vertices. Vertex gadgets (see Figure 10) are used to encode vertices of the DVG instance that are different from s. Each such gadget receives the token peg of a player as an input, and outputs the token peg owned by the other player in exactly 2 moves. This is done thanks to the presence of control pegs which force the player that is entering the vertex gadget with a token peg he owns, to terminate his move exactly when his token peg is aligned with the token peg owned by the other player. Thus, the gadget forces the opponent to move his token peg to the output of the gadget. We observe that no token peg remains inside a vertex gadget once it has been visited. Furthermore, if we think of the game as if it were played on a chessboard, we also observe that if the input token peg comes from a black (resp., white) square of the chessboard, then the token peg contained in the vertex gadget is on a white (resp., black) square. Finally, we observe that a player that cheats and tries to enter the vertex gadget from its output position rather than from its input position, will lose the game.

Branches and one-way gadgets. A branch (see Figure 11) is used to model both the merge of two distinct wires into a single wire (i.e., vertices with indegree 2), as well as the split of one wire into two distinct wires (i.e., vertices with outdegree 2). The gadget takes one token peg in one of the two possible input positions, and forces the moving player to exit from the gadget either in the (unique) output position, or in the other input position. Branches are not sufficient by themselves to model wire splits and merges, and they need to be used together with *one-way* gadgets.

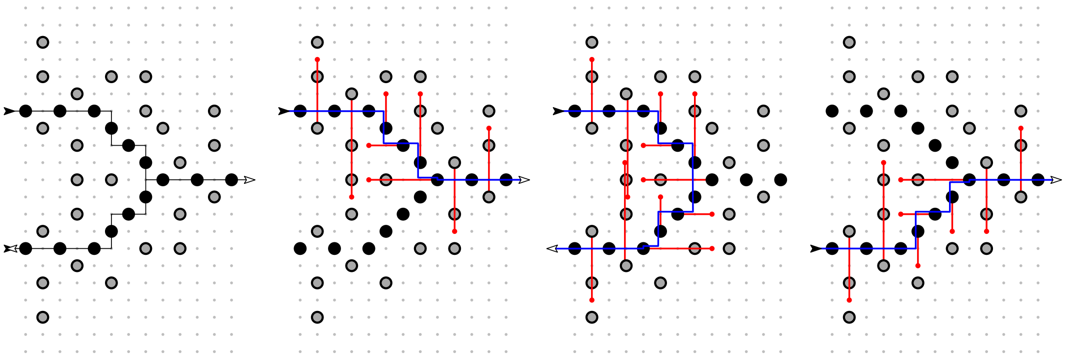


Figure 11 A branch.

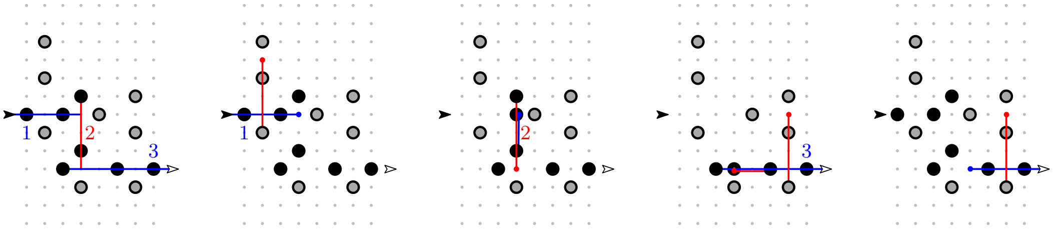


Figure 12 One-way gadget. The rightmost drawing shows that the player that enters the gadget from its output loses.

One-way gadgets (see Figure 12) are used to avoid that players can cheat by visiting branches starting from their corresponding output positions rather than from their corresponding input positions. The one-way gadget takes a token peg as an input, and outputs the token peg with 3 multi-hop moves. A one-way gadget contains two additional token pegs, one token peg for each player, each of which must be activated before it can be moved. Similarly to the vertex gadget, a one-way gadget outputs a token peg that is different from the one that entered the gadget. However, differently from the vertex gadget, the one-way gadget outputs a token peg owned by the same player that entered the gadget. A one-way gadget is designed in such a way that a player that cheats and tries to enter such a gadget from its output rather than its input, will lose the game.

The merge of two wires can now be modelled with a branch whose output is attached with the input of a one-way gadget. Similarly, the split of a wire into two wires can be modelled by using a branch and two one-way gadgets, whose inputs are attached to the output and to any of the 2 inputs of the branch, respectively.

3.2 Putting all together

The token pegs are placed over the (chess)board in such a way that, w.l.o.g., Player 1 owns the token pegs that are placed on black squares, while Player 2 owns the token pegs that are placed on white squares. We observe that this induces the position of each vertex gadget on the board, according to the vertex-player association (we recall that each player has been associated with a specific side of the bipartition). Therefore, when it is Player 1's turn, the (unique) active token peg is on a black square, and the player can only move such a token peg to activate a token peg placed on a white square, if possible. Similarly, when it is Player 2's turn, the (unique) active token peg is on a white square, and the player can only move such a token peg to activate a token peg placed on a black square, if possible.

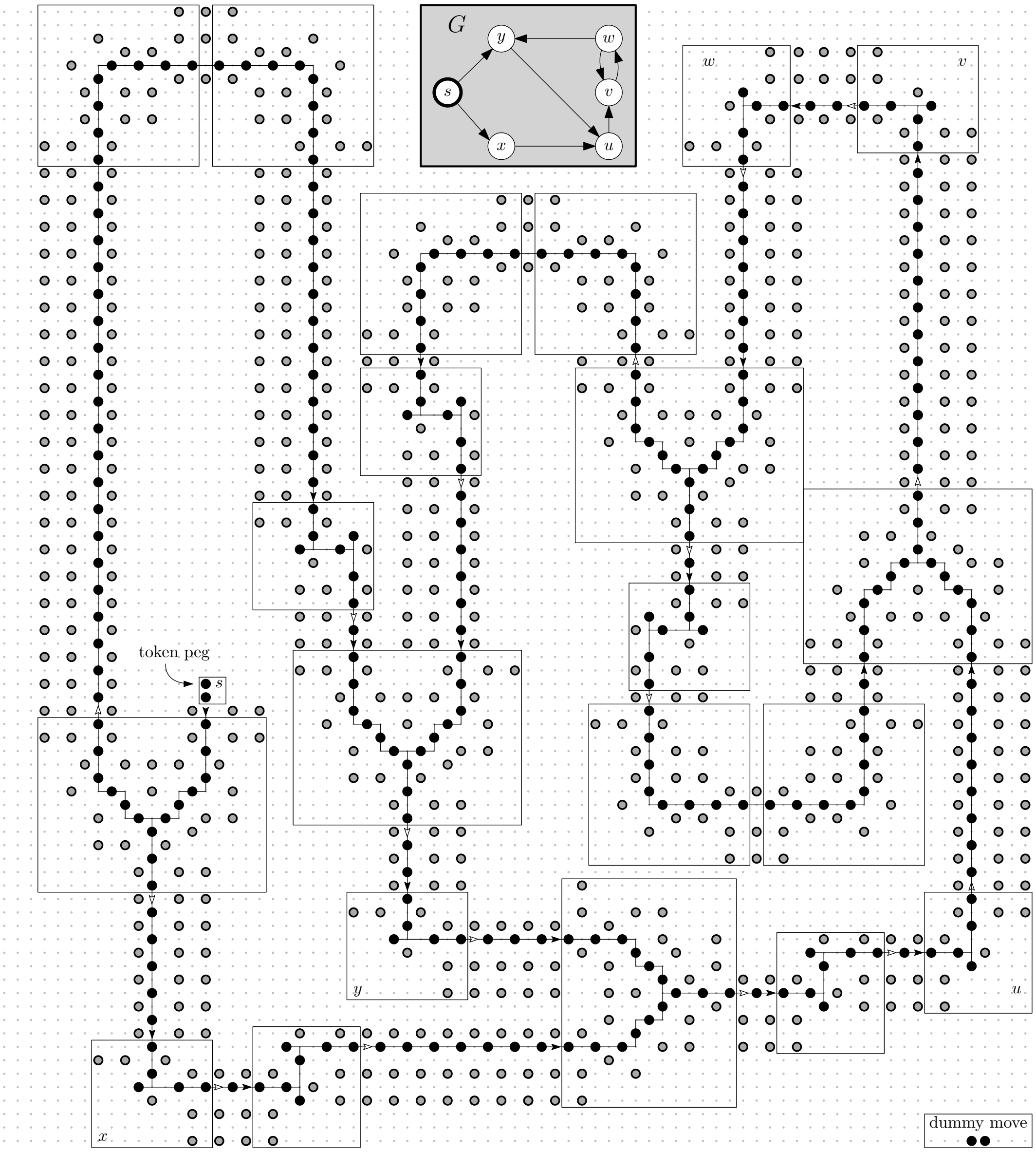


Figure 13 A DVG instance and its corresponding Multi-Hop Duotaire one. For the sake of readability, some (redundant) one-way gadgets have been removed.

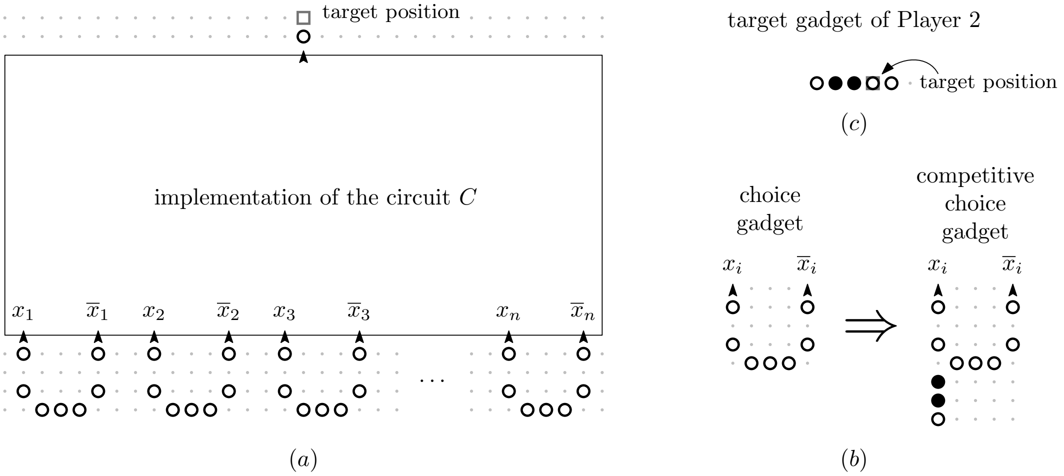


Figure 14 The three ingredients of our reduction for the 2-player version of Solitaire-Reachability. Picture (a) shows (the form of) the peg configuration resulting from the application of the transformation given in [15] to a circuit C (equivalent to the formula F). There is a choice gadget for each variable. Picture (b): the competitive choice gadget. Picture (c): the target gadget of Player 2. If Player 1 is forced to make the only available move in the gadget, then Player 2's target position becomes reachable.

Similarly to the single-hop Duotaire, the embedding shall guarantee that each output of any gadget can be connected to the corresponding input of the consecutive gadget. This could be a problem when the output is, for example, in an even-even position while the input is in a odd-odd position.[2] However, the one-way gadget can be used (also) to restore the desired parity. See Figure 13 for an example multi-hop Duotaire instance obtained from a corresponding DVG instance.

By using similar arguments to the ones we discussed for single-hop Duotaire, we can now state the following:

▶ **Theorem 2.** *Deciding whether the first player can force a win in multi-hop Duotaire is* PSPACE*-complete.*

4 2-player Solitaire Reachability

In this section we prove that the 2-player version of Solitaire Reachability discussed in the introduction is PSPACE-complete. We present the reduction when the winning rule is the following: a player wins when either he reaches his target position, or his opponent has no available moves. Next, we show how to adapt the reduction for other winning conditions.

The main idea of the reduction is borrowed from the PSPACE-completeness reduction of the *bounded 2-player constraint logic* presented in [17]. The reduction is from *Positive Conjunctive Boolean Formula Game* (POS CNF), where one wants to understand whether the first player can force a win in the following game. We are given a *monotone* boolean formula F in CNF, i.e., a formula containing positive literals only. The two players alternate choosing some variable of F that has not yet been chosen, and decide whether to assign either true or false to that variable. The game ends after all variables of F have been chosen.

[2] Notice that the cases even-odd and odd-even cannot occur because of the vertex-player association.

The first player wins if and only if F is true; therefore, the second player wins if and only if F is false. Observe that since F is monotone, the first player has convenience to set the chosen variables to true, while the second player will always set his variables to false.

It is well known that any CNF Boolean formula can be transformed, in polynomial time, into an equivalent planar Boolean circuit of NAND gates only, which in turn can be converted into an instance of Solitaire-Reachability [15]. More precisely, we use this fact to convert F into a configuration of white pegs and a target position having the form showed in Figure 14 (a), where there is a *choice gadget* for each input variable. Each choice gadget allows to set the corresponding variable either to true or false. The reduction given in [15] implies the following: there exists a sequence of moves placing a peg in the target position if and only if the chosen Boolean assignment satisfies F.

In our reduction we first replace each choice gadget with a *competitive choice gadget* (see Figure 14 (b)). The competitive choice gadget allows Player 1 to set the corresponding variable either to true or false, unless Player 2 forces Player 1 to set the variable to false. Therefore, the player that plays the gadget first essentially decides the assignment of the variable. To complete the description of the reduction, we add the target gadget of Player 2 (see Figure 14 (c)), in which the target position of the player is occupied by a peg of his opponent, and sufficiently many dummy moves that can be performed only by Player 2.

Clearly, since F is monotone, both players have convenience to first play all the competitive choice gadgets, and, once the truth assignment has been chosen, Player 1 has a sequence of moves that allows him to win the game only if the truth assignment satisfies the formula. Conversely, if the truth assignment does not satisfy the formula, then Player 1 runs out of moves before Player 2, frees the target position of his opponent, and Player 2 wins the game. Therefore, Player 1 can force a win in our instance if and only if he can force a win in the POS CNF instance.

Our reduction can be adapted to other winning conditions:

- The only way a player can win is reaching the target position. In this case, we can assume that a player with no available moves can skip his turn, and that the game can end with a draw, whenever no player can move and no target position has been reached yet. The reduction is exactly the same.
- There is no target position and the first player that has no available moves loses the game. The reduction is the same but we remove the target gadget of Player 2, and we add a number of sufficiently large moves for Player 1, that are triggered only if a (white) peg is placed in the old target position of Player 1.

References

1 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is np-hard. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 2:1–2:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.2.

2 John D. Beasley. *The Ins and Outs of Peg Solitaire*. Oxford University Press, Oxford Oxfordshire ; New York, 1985.

3 John D. Beasley. Solitaire: Recent Developments. *arXiv:0811.0851 [cs, math]*, nov 2008. arXiv: 0811.0851. URL: http://arxiv.org/abs/0811.0851.

4 John D. Beasley. John and Sue Beasley's Webpage on Peg Solitaire, oct 2015. URL: https://web.archive.org/web/20151010215541/http://jsbeasley.co.uk/pegsol.htm.

5 George I. Bell. A Fresh Look at Peg Solitaire. *Mathematics Magazine*, 80(1):16–28, feb 2007. URL: http://www.jstor.org/stable/27642987.

6 George I. Bell, Daniel S. Hirschberg, and Pablo Guerrero-Garcia. The minimum size required of a solitaire army. *arXiv:math/0612612*, 2006. arXiv: math/0612612. URL: http://arxiv.org/abs/math/0612612.

7 Arie Bialostocki. An application of elementary group theory to central solitaire - ProQuest. URL: http://search.proquest.com/openview/5b321e9dd162151b018ab7d63304daf2/1?pq-origsite=gscholar.

8 Bela Csakany and Rozalia Juhasz. The Solitaire Army Reinspected. *Mathematics Magazine*, 73(5):354–362, dec 2000.

9 Erik D. Demaine, Fermi Ma, Ariel Schvartzman, Erik Waingarten, and Scott Aaronson. The Fewest Clues Problem. In *8th International Conference on Fun with Algorithms (FUN 2016)*, volume 49, pages 12:1–12:12, 2016.

10 Edsger W. Dijkstra. The checkers problem told to me by M.O. Rabin, 1992. URL: http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1134.PDF.

11 Martin Gardner. *The Unexpected Hanging and Other Mathematical Diversions*. University of Chicago Press, Chicago, nov 1991.

12 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, 1990.

13 J. P. Grossman. Periodicity in one-dimensional peg duotaire. *Theor. Comput. Sci.*, 313(3):417–425, 2004. doi:10.1016/j.tcs.2002.11.003.

14 Luciano Gualà, 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, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932866.

15 Luciano Gualà, Stefano Leucci, Emanuele Natale, and Roberto Tauraso. Large peg-army maneuvers. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 18:1–18:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.18.

16 Robert A. Hearn. *Amazons, Konane, and Cross Purposes are PSPACE-complete*, page 287–306. Mathematical Sciences Research Institute Publications. Cambridge University Press, 2009. doi:10.1017/CBO9780511807251.015.

17 Robert A. Hearn and Erik D. Demaine. *Games, puzzles, and computation*. AK Peters Wellesley, 2009.

18 Ross Honsberger. A problem in checker jumping. *Mathematical Gems II*, pages 23–28, 1976.

19 David Lichtenstein and Michael Sipser. GO is polynomial-space hard. *J. ACM*, 27(2):393–401, 1980. doi:10.1145/322186.322201.

20 Neeldhara Misra. Two dots is np-complete. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 24:1–24:12. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.24.

21 Cristopher Moore and David Eppstein. One-Dimensional Peg Solitaire, and Duotaire. *arXiv:math/0008172*, 2000. arXiv: math/0008172. URL: http://arxiv.org/abs/math/0008172.

22 Bala Ravikumar. Peg-solitaire, string rewriting systems and finite automata. *Theor. Comput. Sci.*, 321(2-3):383–394, 2004. doi:10.1016/j.tcs.2004.05.005.

23 Ryuhei Uehara and Shigeki Iwata. Generalized Hi-Q is NP-Complete. *IEICE TRANSACTIONS (1976-1990)*, E73-E(2):270–273, feb 1990.

On the Exact Complexity of Polyomino Packing

Hans L. Bodlaender
Department of Computer Science, Utrecht University, Utrecht, The Netherlands and
Department of Mathematics and Computer Science, Eindhoven University of Technology,
Eindhoven, The Netherlands
H.L.Bodlaender@uu.nl

Tom C. van der Zanden
Department of Computer Science, Utrecht University, Utrecht, The Netherlands
T.C.vanderZanden@uu.nl

Abstract

We show that the problem of deciding whether a collection of polyominoes, each fitting in a $2 \times O(\log n)$ rectangle, can be packed into a $3 \times n$ box does not admit a $2^{o(n/\log n)}$-time algorithm, unless the Exponential Time Hypothesis fails. We also give an algorithm that attains this lower bound, solving any instance of polyomino packing with total area n in $2^{O(n/\log n)}$ time. This establishes a tight bound on the complexity of Polyomino Packing, even in a very restricted case. In contrast, for a $2 \times n$ box, we show that the problem can be solved in strongly subexponential time.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness, Mathematics of computing → Combinatorial algorithms

Keywords and phrases polyomino packing, exact complexity, exponential time hypothesis

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.9

1 Introduction

The complexity of games and puzzles is a widely studied topic, and the complexity of most games and puzzles in terms of completeness for a particular complexity class (NP, PSPACE, EXPTIME, ...) is generally well-understood (see e.g. [5] for an overview). Results in this area are not only mathematically interesting and fun, but are also a great educational tool for teaching hardness reductions. However, knowing that a game or puzzle is NP-complete does not provide a very detailed picture: it only tells us that there is unlikely to be a polynomial-time algorithm, but leaves open the possibility that there might be a very fast superpolynomial but subexponential-time algorithm. This issue was precisely the motivation for introducing the Exponential Time Hypothesis [6].

The Exponential Time Hypothesis (ETH) states that there exists no algorithm solving n-variable 3-SAT in $2^{o(n)}$ time. Assuming this hypothesis, and by designing efficient reductions (that do not blow up the instance size too much), it is possible to derive conditional lower bounds on the running time of an algorithm.

In this paper, we study the Polyomino Packing problem from the viewpoint of exact complexity. We give a reduction from 3-SAT, showing that Polyomino Packing can not be solved in $2^{o(n/\log n)}$ time, even if the target shape is a $3 \times n$ rectangle and each piece fits in a $2 \times O(\log n)$ rectangle. As the reduction is self-contained, direct from 3-SAT and rather elegant, it could be an excellent example to use for teaching. We also show that this is tight: Polyomino Packing can be solved in $2^{O(n/\log n)}$ time for any set of polyominoes of total area n that have to be packed into any shape.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 9; pp. 9:1–9:10
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Polyomino Packing appears to behave similarly to Subgraph Isomorphism on planar graphs, which has exact complexity $2^{\Theta(n/\log n)}$ [1] (i.e., there exist an algorithm solving the problem in $2^{O(n/\log n)}$ time on n-vertex graphs, and unless the ETH fails there is no $2^{o(n/\log n)}$-time algorithm).

Demaine and Demaine [4] showed that packing n polyominoes of size $\Theta(\log n) \times \Theta(\log n)$ into a square box is NP-complete. This result left open a gap, namely of whether the problem remained NP-complete for polyominoes of *area* $O(\log n)$. This gap was recently closed by Brand [3], who showed that Polyomino Packing is NP-complete even for polyominoes of size $3 \times O(\log n)$ that have to be packed into a square. However, Brand's construction effectively builds up larger (more-or-less square) polyominoes by forcing smaller (rectangular) polyominoes to be packed together in a particular way, by using jagged edges that correspond to binary encodings of integers to enforce that certain pieces are placed together.

Our reduction also uses binary encoding of integers to force that various pieces are placed together. However, in contrast, it gives hardness for a much more restricted case (packing polyomino pieces of size $2 \times O(\log n)$ into a rectangle of height 3) and also reduces directly from 3-SAT, avoiding the polynomial blowup incurred by Brand's reduction from 3-Partition, thus giving a tight (under the Exponential Time Hypothesis) lower bound. As 3-Partition is a frequently used tool for showing hardness of various types of packing puzzles and games, we believe that these techniques could be used to give (tight, or at least strong) lower bounds on the complexity of other games and puzzles.

This result is tight in another sense: we show that Polyomino Packing where the target shape is a $2 \times n$ rectangle admits a $2^{O(n^{3/4}\log n)}$-time algorithm, so $3 \times n$ is the smallest rectangle in which a $2^{\Omega(n/\log n)}$-time lower bound can be attained.

Note that our results are agnostic to the type (free, fixed or one-sided) of polyomino used. That is, it does not matter whether we are able to rotate (one-sided), rotate and flip (free) or not (fixed) our polyominoes. Our reduction creates instances whose solvability is preserved when changing the type of polyomino, while the algorithms can easily be adapted to work with any type of polyomino. In the following, we consider the Polyomino Packing problem, which asks whether a given set of polyominoes can be packed to fit inside a given target shape. If we include the additional restriction that the area of the target shape is equal to the total area of the pieces, we obtain the Exact Polyomino Packing problem.

2 Lower Bounds

▶ **Theorem 1.** *Unless the Exponential Time Hypothesis fails, there exists no $2^{o(n/\log n)}$-time algorithm for* Polyomino Packing, *even if the target shape is a $3 \times n$ box, and the bounding box of each polyomino is of size $2 \times \Theta(\log n)$.*

Proof. A weaker version of the statement follows by a simple reduction from the Orthogonal Vector Crafting problem [2]. However, because obtaining the bound on the piece size requires a deeper understanding of the proof, and to illustrate the technique, we give a self-contained proof that closely follows the presentation of [2].

We proceed by reduction from n-variable 3-SAT, which, unless the Exponential Time Hypothesis fails, does not admit a $2^{o(n)}$-time algorithm. By the Sparsification Lemma [7], we can assume that the number of clauses $m = O(n)$.

Using the following well-known construction, we can furthermore assume that each variable occurs as a literal at most 3 times: replace each variable x_i that occurs $k > 3$ times by k new variables $x_{i,1}, \dots, x_{i,k}$ and add the clauses $(\neg x_{i,1} \vee x_{i,2}) \wedge (\neg x_{i,2} \vee x_{i,3}) \wedge \dots \wedge (\neg x_{i,k-1} \vee x_{i,k}) \wedge (\neg x_{i,k} \vee x_{i,1})$. This only increases the total number of variables and clauses linearly (assuming we start with a linear number of clauses).

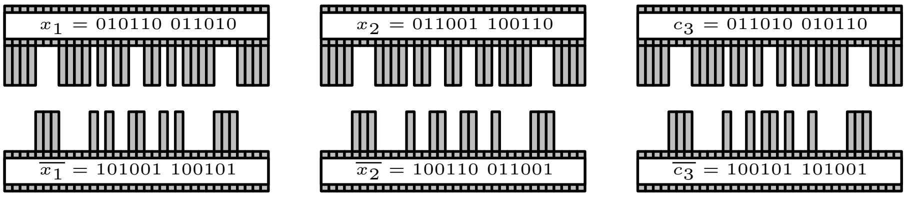

x_1 = 010110 011010
x_2 = 011001 100110
c_3 = 011010 010110
$\overline{x_1}$ = 101001 100101
$\overline{x_2}$ = 100110 011001
$\overline{c_3}$ = 100101 101001

Figure 1 Top: polyominoes corresponding to variables x_1, x_2 and clause c_3. Bottom: the complementary polyominoes, that mate with the polyominoes above them to form a $3 \times k$ square. Note that the polyominoes are depicted compressed horizontally.

We remark that our construction works for general SAT formulas. The Sparsification Lemma is only needed to achieve the stated $2^{\Omega(n/\log n)}$ lower bound, and the bound on the number of occurrences of a variable is only needed to obtain the bound on the piece size.

Our construction will feature three types of polyomino: n *formula-encoding polyominoes*, n *variable-setting* polyominoes and m *clause-checking* polyominoes. We number the variables of the input formula $1, \ldots, n$ and the clauses $n+1, \ldots, n+m$. With every clause or variable we associate a bitstring of length $22 + 4\lceil \log{(n+m)} \rceil$, which is obtained by taking the binary representation of that clause/variable's number, padding it with 0's to obtain a bitstring of length $\lceil \log{(n+m)} \rceil$, replacing every 0 by 01 and every 1 by 10 (thus ensuring the number of 1's in the bitstring is equal to the number of 0's, and that the bitstring contains at most 2 consecutive zeroes or ones) and then appending a reversed copy of the bitstring to itself (making it palindromic). Finally, we prepend 11110001111 and append 11110001111 (note that thus the start and end of the bitstring is the only place to feature 3 or more consequitive 0's).

For any bitstring, we can create a *corresponding polyomino*: given a bitstring of length k, its corresponding polyomino fits in a $2 \times k$ rectangle, whose top row consists of k squares, and whose bottom row has a square whenever the bitstring has a 1 in that position. For each such polyomino, we can also create a *complementary polyomino* that mates with it to form a $3 \times k$ rectangle (which can also be seen as a flipped version of the polyomino corresponding to the complement of the bitstring, i.e., the bitstring with all zeroes replaced by ones and vice-versa). Figure 1 shows several example corresponding polyominoes and their complements. Note that since the bitstrings are palindromic, the thus created polyominoes are achiral, i.e., invariant over being flipped.

We can *concatenate* two polyominoes corresponding to bitstrings b_1, b_2 by taking the polyomino corresponding to the concatenation of the two bitstrings $b_1 b_2$.

Note that the polyomino corresponding to a variable or clause can only mate with its complementary polyomino, it can not fit together with any polyomino corresponding to any other variable or clause or the complement thereof. Our construction uses as building blocks two more polyominoes: the *wildcard polyomino*, which is obtained as the polyomino corresponding to the bitstring 00001110000000...00000001110000 ($4\lceil \log{(n+m)} \rceil$ zeroes surrounded by 00001110000, and the *blocking polyomino*, which is the complementary polyomino for the wildcard. Note that the wildcard polyomino fits together with any clause or variable polyomino, while the blocking polyomino only fits together with the wildcard polyomino.

Since each variable occurs as a literal at most three times, we can assume that it appears at most twice in positive form, and at most twice negated (if the variable occurs exclusively positively or negated we can simply remove the clauses that contain it to obtain an equivalent instance).

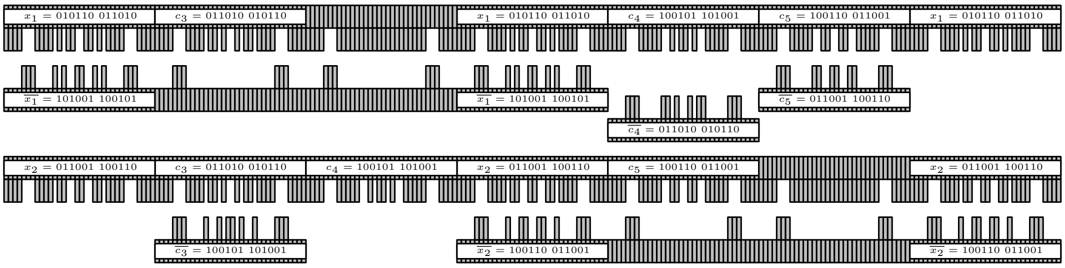


Figure 2 Example of our reduction for the formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$. Top-to-bottom, left-to-right: formula encoding polyomino for x_1, variable-setting polyomino for x_1, clause-checking polyomino for c_4, clause checking-polyomino for c_5, formula-encoding polyomino for x_2, clause-checking polyomino for c_3, variable-setting polyomino for x_2. The polyominoes are arranged in a way that suggests the solution $x_1 = false$, $x_2 = true$.

We are now ready to define the *formula-encoding polyominoes*. The construction will have n variable-encoding polyominoes, one for each variable x_i, and each consists of the concatenation of 7 polyominoes: we start with a polyomino corresponding to the bitstring of x_i. Next, for each time (at most two) x_i occurs positively in a clause, we take a polyomino corresponding to (the bitstring of) that clause. If x_i occurs only once in positive form, then we take (for padding) a copy of the blocking polyomino. Then, we take another copy of the polyomino for x_i. Next, we take the polyominoes corresponding to clauses in which x_i occurs negated. Again, we add the blocking polyomino if x_i only occurs negated once. Finally, we take another copy of the polyomino corresponding to x_i.

The *variable-setting polyomino* for x_i is the polyomino formed by concatenating, in the following order: (a) the complement polyomino for the variable, (b) 2 copies of the wildcard polyomino, (c) another copy of the complement polyomino.

The *clause-checking polyominoes* are simply the following: for each clause, we take a polyomino corresponding to the complement of its bitstring.

This completes the construction. An example of the construction is shown in Figure 2. Note that if fixed or one-sided polyominoes are used, the formula-encoding ones are provided with the solid row of squares on top, and the remaining polyominoes are provided with the solid row on the bottom. We claim this set of polyominoes can be packed into a $3 \times 7n(22 + 4\lceil\log{(n+m)}\rceil)$ box if and only if the formula is satisfiable.

($\Rightarrow$). Suppose the polyominoes can be packed in a $3 \times 7n(22 + 4\lceil\log{(n+m)}\rceil)$ box. We first examine the placement of the formula-encoding polyominoes. Because each formula-encoding polyomino starts with a row of four ones, and the largest "gap" of zeroes occurring in one is of length three, they cannot overlap vertically; each formula-encoding polyomino must be fully to the right of the previous. Moreover, since the width of the target rectangle matches exactly the total width of the formula-encoding polyominoes, they must be placed back-to-back in some arbitrary permutation.

Consider the placement of a single complementary polyomino for a clause or variable. Because wherever two formula-encoding polyominoes touch back-to-back there are 8 consecutive rows in which 2 squares are already occupied, and the longest "gap" in a complementary polyomino is of length at most 5 (and at the left and right edges, there is a gap of length exactly 4, we see that the rows in which this polyomino are placed can contain only a single formula-encoding polyomino. This rules out any undesirable shifts: no complementary polyomino can overlap (vertically) more than one formula-encoding polyomino. Moreover, note that this same phenomenon forces the vertical alignment of polyominoes corresponding to variables or clauses in the formula-encoding polyominoes with the complementary polyominoes in variable-setting and clause-checking polyominoes.

Now, consider the placement of a variable-setting polyomino (for variable x_i). Since it starts with a complementary polyomino for x_i, and also ends with one x_i, it must be placed such that it only overlaps at most (and exactly) one formula-encoding polyomino, namely the one for x_i. It thus suffices to consider each formula-encoding polyomino in isolation. Note that then, there are only two possible placements for the variable-setting polyomino for variable x_i: either overlapping the first half of the formula-encoding polyomino, with the wildcard polyominoes used as building blocks in the variable-setting polyomino overlapping (and thus blocking) the polyominoes corresponding to clauses that are satisfied by setting x_i to *true*, or, overlapping the second half of the formula-encoding polyomino, overlapping (and thus blocking) the polyominoes corresponding to clauses that are satisfied by setting x_i to $false$.

Thus, the placement of the variable-setting polyominoes (unsurprisingly) corresponds to an assignment for the variables of the formula. It is easy to see that the clause-checking polyominoes can then be packed into the space left only if the assignment is satisfying: if the assignment does not satisfy some clause, then all the places where the respective clause-checking polyomino could fit are blocked by variable-setting polyominoes.

($\Leftarrow$). We can consider each formula-encoding polyomino in isolation. An assignment for the formula immediately tells us how to pack the variable-setting polyomino for x_i into the formula-encoding polyomino for x_i (namely: if x_i is true we place the variable-setting polyomino in the second half, otherwise, we place it in the first half of the formula-encoding polyomino). It is easy to see that if the assignment is satisfying, then for each clause-checking polyomino there is at least one possible placement inside a formula-encoding polyomino. For an example of how the pieces fit together for a satisfying assignment, see Figure 2. ◀

Remark that our reduction leaves gaps inside the packing. If we consider the variant of the problem where total area of the pieces is equal to the area of the target shape, and thus the entire rectangle must be filled (Exact Polyomino Packing), the instance can be padded with several 1×1 polyominoes to make the total area of the pieces equal to the area of the target rectangle.

▶ **Corollary 2.** *Unless the Exponential Time Hypothesis fails, there exists no $2^{o(n/\log n)}$-time algorithm for* Exact Polyomino Packing*, even if the target shape is a $3 \times n$ box, and the bounding box of each polyomino is of size $2 \times O(\log n)$.*

This raises an interesting open problem: does Exact Polyomino Packing still admit a $2^{\Omega(n/\log n)}$-time lower bound when the pieces are similarly sized, that is, each piece must have area $\Theta(\log n)$ (or even just $\Omega(n)$). This seems to greatly limit the number of possible interactions between two polyomino pieces, since they cannot be combined in a way that creates small gaps.

Note that in the previous reduction we can fix the position of the formula-encoding polyominoes in advance. The problem then reduces to packing variable-setting and clause-checking polyominoes into the shape left when subtracting the formula-encoding polyominoes from the $3 \times n$ rectangle, which fits inside a $2 \times n$ rectangle. Doing so we obtain the following corollary:

▶ **Corollary 3.** *Unless the Exponential Time Hypothesis fails, there exists no $2^{o(n/\log n)}$-time algorithm for* Polyomino Packing *(resp.,* Exact Polyomino Packing*), even if the target shape fits inside a $2 \times n$ box, and the bounding box of each polyomino is of size $2 \times \Theta(\log n)$ (resp., $2 \times O(\log n)$).*

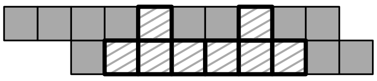


Figure 3 Packing an arbitrary $2 \times k$ polyomino into a Y-monotone polyomino results in several pieces that are again Y-monotone.

3 Algorithms

Our lower bound applies in a rather constrained case: even for packing polyominoes with a bounding box of size $2 \times O(\log n)$ into a rectangle of size $3 \times n$, there is no $2^{o(n/\log n)}$-time algorithm. As we will show later, a similar lower bound can not be established when the pieces are $1 \times k$ or $2 \times k$ rectangles (since the number of distinct such polyominoes is linear in their area rather than exponential). An interesting question, which we answer negatively, is whether a $2^{\Omega(n/\log n)}$-time lower bound can be obtained for packing polyominoes with a bounding box of size $2 \times O(\log n)$ into a rectangle of size $2 \times n$. Thus, the case for which we have derived our lower bound is essentially the most restrictive possible. Note that, while solvable in strongly subexponential time, this problem is NP-complete, as can be seen by a simple reduction from 3-Partition.

We say that a polyomino is *Y-monotone* if every row consists of a number of contiguous squares, that is, there are no gaps.

▶ **Theorem 4.** Polyomino Packing *for fixed, free or one-sided polyominoes can be solved in $2^{O(n^{3/4}\log n)}$ time if the target shape is a $2 \times n$ rectangle.*

Proof. First, consider a simple $O(2^n n^{O(1)})$-time dynamic programming algorithm that decides whether m polyominoes $p_1, \dots, p_m$ can be packed into a target polyomino of area n: for any subset S of (the squares of) the target polyomino (there are 2^n such subsets) and $1 \leq k \leq m$, let $B(S, k)$ be the proposition "the polyominoes $p_k, p_{k+1}, \dots, p_m$ can be packed into S". $B(S, m)$ is simply the proposition that S is the same polyomino as p_m; if $B(S, i-1)$ is known for all S then $B(S', i)$ can be computed by trying all (polyominally many) placements of p_i within S'.

If we are dealing with free or one-sided polyominoes we first guess how many (if any) of the 1×2 polyominoes should be used in the vertical orientation, and how many in the horizontal orientation. This thus converts them to fixed 1×2 or 2×1 polyominoes, and only increases the running time of the algorithm by a factor n.

We augment the previously presented algorithm with the following observation: when the target polyomino is a $2 \times n$ rectangle, and if we process the polyominoes in a fixed order, with the polyominoes that are $1 \times k$ rectangles being processed last (thus after the 2×1 polyominoes and any other polyominoes), then the target shapes considered by the dynamic programming algorithm are always the disjoint union of several Y-monotone polyominoes (c.f. Figure 3). Such polyominoes can be described by 3 integers: one giving the number of squares in the bottom row, one giving the number of squares in the top row, and one giving the shift of the top row relative to the bottom row. Note that this observation crucially depends on processing the $1 \times k$ polyominoes last, since removing them from a $2 \times k$ polyomino does not necessarily result in a shape that is Y-monotone, however, if only $1 \times k$ polyominoes remain, we can ensure this requirement remains satisfied because we can consider the top and bottom row of each polyomino in the target shape seperately.

If each of these integers is at most $n^{1/4} - 1$ we call the resulting polyomino *small*, otherwise, the polyomino is *large*. We can use the following more efficient description of the target shape: for each polyomino in the shape that is small, we give the number of such polyominoes in the

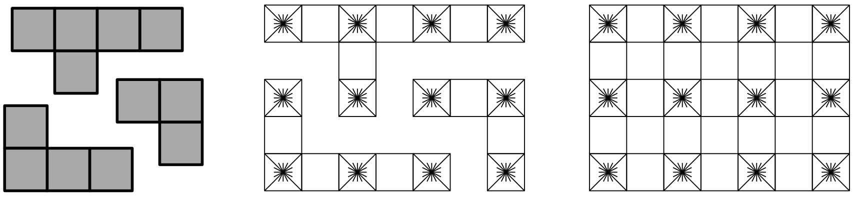


Figure 4 Polyomino Packing problem (left) modelled as Subgraph Isomorphism from pattern (middle) into host graph (right).

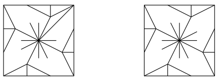


Figure 5 Alternative constructions to use with fixed (left) or one-sided (right) polyominoes.

target shape and we simply list each large polyomino. Since there are at most $n^{3/4}$ distinct small polyominoes, giving the quantity for each leads to at most $(2n)^{n^{3/4}} \leq 2^{n^{3/4}(\log n+1)}$ cases. There are at most n^3 distinct large polyominoes, but the target shape contains at most $2n^{3/4}$ of them (since each has area at least $n^{1/3}$), thus contributing $(n^3)^{2n^{3/4}} \leq 2^{6n^{3/4}(\log n+1)}$ cases. Thus, if we identify equivalent target shapes, the dynamic programming algorithm needs to consider at most $2^{6n^{3/4}(\log n+1)}n = 2^{O(n^{3/4}\log n)}$ subproblems, and each subproblem can be handled in polynomial time. ◀

Note that this algorithm only works when the target shape is a $2 \times n$ rectangle. Corollary 3 shows that we should not expect a similar algorithm for packing polyominoes into an arbitrary target shape, even if that target shape fits in a $2 \times n$ box.

Finally, we show that our $2^{\Omega(n/\log n)}$-time lower bound is tight:

▶ **Theorem 5.** Polyomino Packing *for free, fixed or one-sided polyominoes can be solved in $2^{O(n/\log n)}$ time if the target shape has area n.*

Proof. The problem can be modelled as Subgraph Isomorphism for an $O(n)$-vertex planar graph, for which a $2^{O(n/\log n)}$-time algorithm is known [1]. The construction is as follows: for every square in a polyomino, we take a cycle on four vertices, to which we add a fifth, universal vertex (which can be embedded in a planar embedding in the middle of this cycle). This fifth vertex is marked by adding a number of degree 1 vertices to it, to bring its degree up to (at least) 9. Each edge of this cycle is associated with an edge of the square in the polyomino. We make adjacent the endpoints of edges corresponding to adjacent edges in the polyomino. Both the host graph and the guest graph are constructed in this way, the host graph from the target shape (when viewed as a polyomino) and the guest graph from the set of input polyominoes (which will thus have one connected component corresponding to each separate polyomino that must be packed). An example for packing 3 polyominoes into a 3×4 rectangle is shown in Figure 4. The special (degree 9) vertices must be mapped to other vertices that are also degree 9, and this means that the cycles corresponding to squares can only be mapped to cycles corresponding to other squares (and not to cycles created by making cycles adjacent since those vertices have degree less than 9).

This construction works for free polyominoes. To restrict to fixed or one-sided polyominoes, we can modify the construction slightly to make the structure used to represent a square asymmetric. For one-sided polyominoes, we create a structure that is rotationally symmetric but achiral. To this end, we subdivide each edge of the cycle twice and identify one of the two vertices created by this subdivision, add another vertex, adjacent to this vertex, to its neighbours, and to the central vertex. For fixed polyominoes, we can add one additional edge (from the center to one of the vertices of the cycle to also remove the rotational symmetry. These constructions are depicted in Figure 5. ◀

To make the paper self-contained and more instructional, we give a direct proof of the following weaker version of Theorem 5 — which illustrates in a simpler way the principles from [1].

▶ **Theorem 6.** Polyomino Packing *for free, fixed or one-sided polyominoes can be solved in* $2^{O(n/\log n)}$ *time if the target shape is a rectangle of area* n*.*

Proof. If the rectangle is higher than it is wide, rotate it (and, if the polyominoes are fixed, the polyominoes as well) 90 degrees. Consider a scanline passing over the rectangle from left to right. At any given time, the scanline intersects at most $O(\sqrt{n})$ squares of the rectangle. We can specify how the intersection of the solution with the scanline looks by, for each square, specifying the polyomino (if any) that is placed there, along with its rotation and translation with respect to the square. This gives at most $O(n^3)$ cases for each square, and, since the scanline intersects at most $\sqrt{n}$ squares, $2^{O(\sqrt{n}\log n)}$ cases total.

We furthermore need to specify which polyominoes have already been used in the solution (to the left of the scanline) and which ones still need to be packed. Similar to [1], a polyomino is *large* if it has area greater than $c \log n$, and small otherwise. Since the number of polyominoes with area k is bounded by 4.65^k [8], the number of distinct small polyominoes it at most $4.65^{c \log n}$. For $c \leq 0.22$, this is at most $\sqrt{n}$. We can specify the *quantity* of each small polyomino left with a single number from 0 to n, giving $(n+1)^{\sqrt{n}} = 2^{O(\sqrt{n}\log n)}$ cases. Meanwhile, the number of large polyominoes is at most $n/(c \log n)$, and thus there are $2^{O(n/\log n)}$ possible subsets of them.

The problem can now be solved by dynamic programming. For each position of the scanline, we have $2^{O(n/\log n)}$ subproblems: can a given subset of pieces ($2^{O(n/\log n)}$ cases) be packed entirely to the left of the scanline (with only the pieces intersecting the scanline possibly sticking out to the right of it) such that the intersection with the scanline looks as specified ($2^{O(\sqrt{n}\log n)}$ cases) (and, in the case of Exact Polyomino Packing, leaving no gaps)? For each such subproblem, we can find its answer by deleting the pieces whose leftmost square(s) intersect the scanline, and checking whether the instance thus obtained is compatible with some subproblem with the scanline moved one position to the left. ◀

There is an interesting contrast between these two algorithms. Whereas the strongly subexponential algorithm for the case of the $2 \times n$ rectangle works by considering the input polyominoes in a fixed order (so that we always know which subset we have used) and uses a bound on the number of subsets of the target shape that have to be considered, the algorithm for the general case works the opposite way around: it considers subsets of the target shape in a (more-or-less) fixed order (by the scanline approach) and bounds the number of possible subsets of the input polyominoes.

Note that our $2^{\Omega(n/\log n)}$-time lower bound exploits the fact that we can construct exponentially many polyominoes that fit inside a $2 \times O(\log n)$ rectangle. If we consider polyominoes with simpler shapes, that is, polyominoes that are $a \times b$ rectangles, then the problem can be solved in strongly subexponential time:

▶ **Corollary 7.** Polyomino Packing *can be solved in* $2^{O(\sqrt{n}\log n)}$ *time if the polyominoes are rectangular and the target shape is a rectangle with area* n*.*

Proof. Consider the algorithm presented in the proof of Theorem 6. The running time is dominated by the number of cases for tracking a subset of the polyominoes. If the polyominoes are rectangles, then note that the number of distinct rectangles of area at most n is also at most n. Call a polyomino *large* if it has area $\geq \sqrt{n}$ and *small* otherwise: there are at most $\sqrt{n}$ large polyominoes in the input, and thus at most $2^{\sqrt{n}}$ subsets of them. The number of distinct small polyominoes is at most $\sqrt{n}$, and thus specifying the quantity for each leads to at most $n^{\sqrt{n}} = 2^{\sqrt{n}\log n}$ cases. ◀

4 Conclusions

In this paper, we have given a precise characterization of the complexity of (Exact) Polyomino Packing. For a set of polyominoes of total area n, the problem can be solved in $2^{O(n/\log n)}$ time. Even when restricted to the case where the pieces are of size $2 \times O(\log n)$ and they have to be packed into a $3 \times n$ rectangle or into a given shape which fits inside a $2 \times n$ rectangle, there is no faster (up to the base of the exponentiation) algorithm unless the Exponential Time Hypothesis fails. In contrast, in the case where the target shape is a $2 \times n$ rectangle, a strongly subexponential algorithm exists.

We conclude by listing several interesting open problems:

- Exact polyomino packing with excess pieces: we are given some target shape, and a set of polyominoes with total area possibly exceeding the target shape. Is it possible to use a subset of the polyominoes to build the target shape? Clearly this problem is at least as hard as (exact) polyomino packing; however, considering the set of pieces may be much larger than the target shape, it would be interested to study this problem from a parameterized perspective (where the parameter k is the area of the target shape). The problem can be solved in $2^k n^{O(1)}$-time (by the simple dynamic programming algorithm of Section 3; is there a $2^{o(k)} n^{O(1)}$-time (or even a $2^{o(k)} 2^{o(n/\log n)}$-time) algorithm?
- What is the (exact) complexity of Exact Polyomino Packing when every piece has area $\Omega(\log n)$ or $\Theta(\log n)$? Our lower bound construction uses 1×1 polyominoes to fill the gaps in the packing. Requiring that each piece has area $\Omega(\log n)$ seems to limit the number of possible interactions between two pieces significantly.
- We do not believe that our algorithm for packing polyominos into a $2 \times n$ rectangle is tight. What is the exact complexity of this problem? This is closely related to the exact complexity of 3-Partition with the input given in unary, which (to our knowledge) is also an open problem.

References

1 Hans L. Bodlaender, Jesper Nederlof, and Tom C. van der Zanden. Subexponential time algorithms for embedding h-minor free graphs. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 9:1–9:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.ICALP.2016.9.

2 Hans L. Bodlaender and Tom C. van der Zanden. Improved Lower Bounds for Graph Embedding Problems. In Dimitris Fotakis, Aris Pagourtzis, and Vangelis Th. Paschos, editors, *10th International Conference on Algorithms and Complexity (CIAC 2017)*, volume 10236 of *LNCS*, pages 92–103. Springer, 2017.

3 Michael Brand. Small polyomino packing. *Information Processing Letters*, 126:30–34, 2017.

4 Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics*, 23(1):195–208, 2007.

5 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* CRC Press, 2009.

6 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.

7 Russell Impagliazzo, Ramamohan Paturi, and Francis Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63:512–530, 2001.

8 David A. Klarner and Ronald L. Rivest. A procedure for improving the upper bound for the number of n-ominoes. *Canad. J. Math*, 25(3):585–602, 1973.

Kings, Name Days, Lazy Servants and Magic

Paolo Boldi
Università degli Studi di Milano, Italy
paolo.boldi@unimi.it
https://orcid.org/0000-0002-8297-6255

Sebastiano Vigna
Università degli Studi di Milano, Italy
sebastiano.vigna@unimi.it
https://orcid.org/0000-0002-3257-651X

Abstract

Once upon a time, a king had a very, very long list of names of his subjects. The king was also a bit obsessed with name days: every day he would ask his servants to look the list for all persons having their name day. Reading every day the whole list was taking an enormous amount of time to the king's servants. One day, the chancellor had a magnificent idea: he wrote a book with instructions. The number of pages in the book was equal to the number of names, but following the instructions one could find all people having their name day by looking at only a few pages – in fact, as many pages as the length of the name – and just glimpsing at the list. Everybody was happy, but in time the king's servants got lazy: when the name was very long they would find excuses to avoid looking at so many pages, and some name days were skipped. Desperate, the king made a call through its reign, and a fat sorceress answered. There was a way to look at much, much fewer pages using an additional magic book. But sometimes, very rarely, it would not work (magic does not always work). The king accepted the offer, and name days parties restarted. Only, once every a few thousand years, the magic book fails, and the assistants have to go by the chancellor book. So the parties start a bit later. But they start anyway.

2012 ACM Subject Classification Theory of computation → Sorting and searching

Keywords and phrases Suffix trees, suffix arrays, z-fast tries, prefix search

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.10

1 Introduction

From what we can ascertain reading the enthusiastic reports of the contemporary historians, the chancellor probably stumbled into an early version of *suffix arrays* [11]. His book might have contained a table of the initial points in the list for every name, sorted lexicographically by suffix. Indeed, the suffix array of a string s over an ordered alphabet Σ of σ elements, in modern terms, is simply the array of the starting points of the string $s\$$ (where \$ is a character larger than any character in Σ) sorted lexicographically by the corresponding suffix. Suffix arrays are an extremely effective way of looking for all occurrences of a pattern in a string, as once they are built (with some additional ancillary data), search requires only an amount of work linear in the length of the search pattern. In modern days, a large body of research has gone into building and representing suffix arrays efficiently (e.g., in compressed form) [6, 4, 13, 10, 7].

In fact, at the price of an additional (and very compressible) array of integers a suffix array can represent implicitly the suffix tree associated with the string s [1]. While asymptotically the two approaches give the same bounds, we know that managing a billion nodes or three

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 10; pp. 10:1–10:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

arrays of a billion integers is an entirely different business, especially if you are a poor servant that lives on bread and water.

So, why were the servants still unhappy? Well, since the pages to look at were as many as the letter in the names, with long names they had to jump through several pages of the book and of the list quite at random. The book was heavy, and the list too. Looking at consecutive pages was easy, jumping around much less.

Here is when the fat sorceress came in: she said that with an additional book and additional instructions, looking at a number of pages equal to the *logarithm* of the length of the name would have been sufficient to recover the same information, together with a few scans on the list. The only problem is that magic is tricky and once in a while (although very rarely) it would not work: if that happens, though, sevants would realize that something went wrong, and they could still go the old way.

Through a thorough research we have been able to reconstruct the spell. What the fat sorceress probably discovered is a way to apply *fat binary search*, the main ingredient of *z-fast tries* [2], to a suffix array.[1] To use fat binary search, one stores some additional linear-space information, the *z-map*. Then, given a pattern p of length m, one first preprocesses p in time $O(1 + (m \log \sigma)/w)$, and then performs $O(\log m)$ search steps, each accessing a constant amount of information. Then follows a verification phase, which accesses m characters, but the interesting fact is that the characters are accessed in sequential fashion at less than σ different positions of the original text s. Thus, in modern-day parlance, fat binary search makes it possible to find with high probability the occurrences of pattern p in s using time $O((m \log \sigma)/w + \log m + \sigma)$ and $O((m \log \sigma)/B + \log m + \sigma)$ I/Os in the cache-oblivious model.

2 Notation and Tools

Let Σ be a fixed alphabet (of cardinality σ) not including the special symbol \$, and define $\hat{\Sigma} = \Sigma \cup \{\$\}$. The alphabet Σ comes endowed with a specified (arbitrary) total order, that is inherited by $\hat{\Sigma}$ with the proviso that \$ is larger than any other character. We use $\leq$ to denote the induced lexicographic order on $\hat{\Sigma}^*$, whereas $\preceq$ is used to denote the prefix order.

If $x \in \hat{\Sigma}^*$ is a string, x juxtaposed with an interval is the substring of x with those indices (indices start from 0). Thus, for instance, $x[a\,..\,b]$ is the substring of x starting at position a (inclusive) and ending at position b (inclusive). We will write $x[a]$ for $x[a\,..\,a]$ and $x[a\,..\,]$ for $x[a\,..\,|x| - 1]$. By definition, $x[|x|\,..\,] = \varepsilon$.

We analyze our algorithms on a unit-cost word RAM with word size w in the cache-oblivious model [5]. In this model, the machine has a two-level memory hierarchy, where the fast level has an unknown size of M words and the slow level has an unbounded size and is where our data reside. We assume that the fast level plays the role of a cache for the slow level with an optimal replacement strategy where the transfers (a.k.a. I/Os) between the two levels are done in blocks of an unknown size of $B \leq M$ words; the I/O cost of an algorithm is the total number of such block transfers. *Scanning* is a fundamental building block in the design of cache-oblivious algorithms: given an array of N contiguous items the I/Os required for scanning is $O(1 + N/B)$.

We measure space in words. Thus, we will say that a suffix array takes linear space, even if it needs $O(n \log n)$ bits. A more detailed analysis can be performed when specific instances of the various support structure have been instantiated (e.g., a compressed vs. non-compressed lcp array).

[1] We should mention here that trying such a spell was suggested to the sorceress by the great sorcerer of sorcerers, Ricardo Baeza-Yates.

3 Static Z-Fast Tries on Arbitrary Languages

Fat binary search appeared for the first time in the context of *probabilistic static z-fast tries* [2], which were originally introduced for prefix-free binary languages. Their main purpose was to assign buckets to a larger set of strings. The basic idea is that of enriching a standard compacted trie with a kind of *acceleration map*, the *z-map*, which makes it possible navigate the trie quickly (i.e., using a number of accesses to the map logarithmic in the length of the search string).

Subsequently, fat binary search was applied to *dynamic z-fast tries* [3], again based on prefix-free binary strings. Dynamic z-fast tries exist in two versions: an *exact* version and a *signature-based* version. In the first case, we store a z-map from strings to nodes, in the second case a z-map from string *signatures* to nodes, and we have to handle collisions and false positives.

In this section we start introducing exact *static* z-fast tries on prefix-free languages L on a general alphabet Σ, and show how they can be used to solve the following problem: is p the prefix of some element of L? We describe the algorithms in the exact setting, adding assertions that may fail in the signature-based setting.

3.1 Compacted tries

A *compacted trie* [9] over Σ is a rooted tree such that

- every node α is endowed with a string $c_\alpha \in \Sigma^*$ (the *compacted path* of α)
- every arc connecting an internal node α with one of its children α' is labelled with a character $c_{\alpha,\alpha'} \in \Sigma$ and $c_{\alpha,\alpha'} \neq c_{\alpha,\alpha''}$ for any two distinct children α' and α'' of α
- every internal node has at least two children.

For every node α of a compacted trie, we define its *name* $n_\alpha \in \Sigma^*$ and its *extent* $e_\alpha \in \Sigma^*$ as follows:

- $n_{\text{root}} = \varepsilon$
- $e_\alpha = n_\alpha c_\alpha$
- if α' is a child of α, then $n_{\alpha'} = e_\alpha c_{\alpha,\alpha'}$.

For any given finite nonempty prefix-free language $L \subseteq \Sigma^*$, the *compacted trie of* L is the only compacted trie[2] $T(L)$ over Σ such that L is the set of all the extents of the leaves of $T(L)$.

In Figure 1, we show an example of a trie with the nomenclature just introduced (and some more that will be introduced in the following).

3.2 Exit nodes

Given a compacted trie over Σ, and given a string p, we let $\operatorname{exit}(p)$ be the exit node of p, that is, the only node α such that $n_\alpha \preceq p$ and for every other node α' if $n_{\alpha'} \preceq p$ then $n'_\alpha \preceq n_\alpha$. In other words, it is the node whose name is the longest possible prefix of p. (See Figure 1 for an example of an exit node). Moreover, we call *parex node of* p the *parent* of the exit node of p, or a special symbol $\perp$ if the exit node of p is the root (note that the parex of p is the node with the longest extent that is a *proper* prefix of p).

It is worth stating the following property of exit nodes:

[2] In [2], we studied compacted tries only for the binary case (i.e., $\Sigma = 2$).

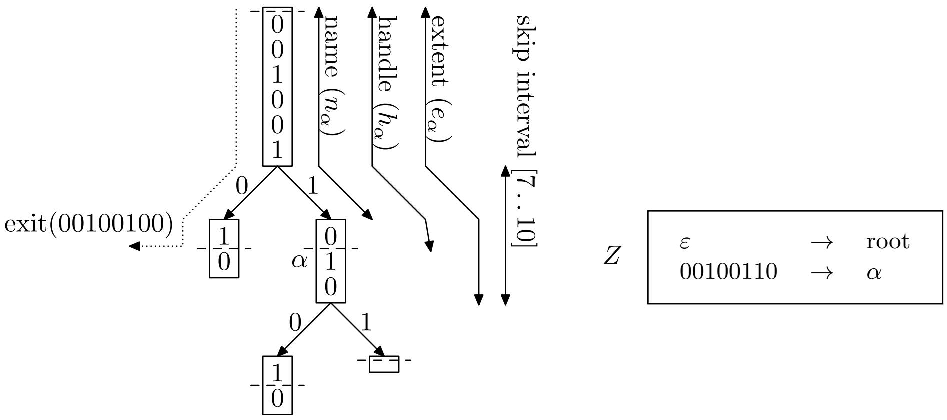


Figure 1 (above) The compacted trie $T(L)$ (with the corresponding nomenclature) and its z-map. Here $L = \{001001010, 0010011010010, 00100110101\}$.

▶ **Lemma 1.** *For a given string p, $\operatorname{exit}(p)$ is the only node γ such that $n_\gamma \preceq p$ and one of the following mutually exclusive properties holds:*

- *$p \preceq e_\gamma$;*
- *$e_\gamma \prec p$ and γ does not have a $p[|e_\gamma|]$ child;*
- *p and e_γ are $\preceq$-incomparable.*

Proof. The only remaining case is that $e_\gamma \preceq p$ with γ having a $p[|e_\gamma|]$-child: but in this case, that child would have a name that is still a prefix of p, longer than n_γ. ◀

Another easy consequence of the definition is the following:

▶ **Proposition 2.** *Let $L \subseteq \Sigma^*$ and consider the trie $T(L)$. A string $p \in \Sigma^*$ is a prefix of some element of L if and only if $p \preceq e_{\operatorname{exit}(p)}$. Moreover, if the latter happens then the set*

$$\{e_\alpha \mid \alpha \textit{ is a leaf descendant of } \operatorname{exit}(p)\}$$

is precisely the set of $x \in L$ such that $p \preceq x$.

In the example of Figure 1, $p = 00100100$ is not the prefix of any of the strings in the language. The name of its exit node $\operatorname{exit}(p)$ (the leftmost child of the root in the figure) is 0010010, which is in fact a prefix of p. Yet the extent of $\operatorname{exit}(p)$ is 001001010 of which p is not a prefix. If we had taken $p' = 00100110$ we would have exited at the rightmost child of the root, whose extent 0010011010 has p' as prefix: in fact, p' is the prefix of two of the elements of L, corresponding precisely to the rightmost two leaves in the trie.

A final important remark is the following: if p is *not* a prefix of an element of L, according to Proposition 2, $p \not\preceq e_{\operatorname{exit}(p)}$. But more than this is true: $p \not\preceq e_\alpha$ for all nodes α (for otherwise, a fortiori, p would be a prefix of the extent of a leaf).

3.3 Static z-fast tries

Let us assume that we have built the trie $T(L)$ for a given language L of size n. Proposition 2 gives an easy way to determine if a string p of length m is a prefix of some element of L: it is enough to locate $\operatorname{exit}(p)$ and then to check whether p is a prefix of its extent or not. Moreover, the second part of the statement suggests which elements of L have p as prefix.

Algorithm 1 Querying the z-fast trie using fat binary search. Given a string p, it will return either exit(p) or parex(p).

```
Input: a string $p$ of length $m$
Output: either parex($p$) or exit($p$)
$\ell, r \leftarrow 0, m$
while $\ell \leq r$ do
  $f \leftarrow$ the 2-fattest number in $[\ell \mathinner{..} r]$
  $\beta \leftarrow Z(p[0 \mathinner{..} f-1])$
  if $\beta \neq \perp$ then
    $\ell \leftarrow |e_\beta| + 1$
    $\gamma \leftarrow \beta$
  else
    $r \leftarrow f - 1$
  end if
end while
return $\gamma$
```

Locating exit(p) can be done trivially in $O(m\sigma)$ steps, going down in the trie starting from the root. In [2] we suggest an alternative, faster solution that needs an additional data structure, called *z-map*. We briefly recall the idea.

▶ **Definition 3** (2-fattest numbers and handles). The *2-fattest number* of an interval $[a \mathinner{..} b]$ of non-negative integers is the unique integer in $[a \mathinner{..} b]$ that is divisible by the largest power of two, or equivalently, that has the largest number of trailing zeroes in its binary representation. The *handle* h_α of a node α of a trie is the prefix of e_α whose length is 2-fattest in $[|n_\alpha| \mathinner{..} |e_\alpha|]$ (the *skip interval* of α).

In Figure 1 we show the (length of the) handles of each node, including the leaves: the handle is the string ending just above the dotted lines you can see in each node.

▶ **Definition 4** (z-map). The *z-map* $Z(-)$ for the trie $T(L)$ is a map from elements of Σ^* to nodes in the trie, which maps h_α to α for each *internal* node α.

In the example of Figure 1, there are only two internal nodes, so the map contains only two pairs. The z-map can be stored using any static dictionary with constant-time access; we assume that the dictionary returns the special value $\perp$ whenever the key is not in the dictionary.

The usefulness of the z-map is made evident by Algorithm 1, which takes as input a string $p \in \Sigma^*$ and outputs a node of the trie which is either exit(p) or parex(p): it is a general-alphabet version of the classic fat binary search [2], and in fact, it is exactly identical to the binary version, since the alphabet has no role in such searches. The proof from [2] goes along in the same way.

▶ **Theorem 5.** *Algorithm 1 is correct and its loop is executed at most* $\log m$ *times; in particular, the z-map is accessed at most* $\log m$ *times.*

Note that fat binary search does not use the trie structure: it only queries the z-map, each time using a prefix of p, and computes possibly the length of the extent of a node returned by the z-map.

If our purpose is determining whether p is the prefix of some element of L, we can start with Algorithm 1, but then we need two things: first we must understand whether the

Algorithm 2 Given a string p, it will return either exit(p) or $\bot$, depending on whether p is the prefix of a string in L or not.

```
Input: a string $p$
Output: if $p$ is the prefix of an element of $L$ then exit($p$), otherwise $\bot$
$\gamma \leftarrow$Algorithm 1 on $p$
assert $n_\gamma \preceq p$
if $p \preceq e_\gamma$ then
  return $\gamma$ /* $\gamma$ is the exit node */
else
  if $p$ and $e_\gamma$ are $\preceq$-incomparable then
    return $\bot$ /* $\gamma$ is the exit node, but $p \not\preceq e_{\text{exit}(p)}$ */
  else
    /* necessarily $e_\gamma \prec p$ */
    if $\gamma$ has a child labelled $p[|e_\gamma|]$ then
      let $\gamma'$ be the child /* $\gamma'$ is the exit node */
      if $p \preceq e_{\gamma'}$ then
        return $\gamma'$
      else
        assert $p$ and $e_{\gamma'}$ are $\preceq$- incomparable or $\gamma'$ does not have a $p[|e_{\gamma'}|]$-child
        return $\bot$
      end if
    else
      return $\bot$ /* $\gamma$ is the exit node, but $p \not\preceq e_{\text{exit}(p)}$ */
    end if
  end if
end if
```

algorithm returned the exit node or its parent (and in the latter case we must go down to the actual exit node), and second we need to use Proposition 2 to determine if $p \preceq e_{\text{exit}(p)}$ or not. Algorithm 2 does both things at the same time.

▶ **Theorem 6.** *Algorithm 2 is correct; moreover, if the length of the strings in L is bounded by $O(w/\log\sigma)$, the algorithm uses $O(\log m + \sigma)$ time and I/Os.*

Proof. First of all, note that since $n_\gamma \preceq p$ (as in the first **assert**), one can determine if $p \preceq e_\gamma$ just by comparing $p[|n_\gamma| .. |e_\gamma|]$ and c_γ. A similar consideration is true (later on in the algorithm) for deciding if $p \preceq e_{\gamma'}$.

Let now γ be the output of Algorithm 1; γ is either the exit node or the parex: in either case $n_\gamma \preceq p$ (which justifies the first assertion). If $p \preceq e_\gamma$, necessarily γ is the exit node and moreover p is a prefix of some element of L by Proposition 2. If p and e_γ are incomparable, once more γ is the exit node (because none of the children of γ can have a name that is a prefix of p), but (again using Proposition 2) we must return $\bot$. The last case is that $e_\gamma \prec p$. If γ has no child with label $p\big[|e_\gamma|\big]$, for the same reasons as in the last case, γ is the exit node but we must return $\bot$. Otherwise, the child with label $p\big[|e_\gamma|\big]$, say γ', is the real exit node. We must continue to check if $p \preceq e_{\gamma'}$ or not, but we can limit the check to the part of the string p that was not checked so far. The last assertion states that γ' is indeed the exit node.

For the complexity statement, the call to Algorithm 1 requires time $O(\log m)$ and the same amount of I/Os to query $Z(-)$ (every access to $Z(-)$ is done in constant time because of the assumption about the length of the strings in L). The comparison between p and

e_γ (and, later, $e_{\gamma'}$) requires time $O(1 + (m \log\sigma)/w)$ and $O(1 + (m\log\sigma)/B)$ I/Os (as we observed above, in both cases we are comparing a substring of p with the compacted path of a node), but $m = O(w/\log\sigma)$ so these quantities are both $O(1)$. The only case in which Algorithm 1 needs to access the trie structure is when it needs to find the child of γ with label $p\big[|e_\gamma|\big]$, which can be done in time $O(\sigma)$. ◀

It is worth noting that Algorithm 2 performs at most one access to the trie structure in case it needs to enumerate the children of γ. Moreover, the comparisons between p and other strings requires at most σ scans of overall m characters.

We also remark that if the z-map is modified to include also the handles of the leaves, Algorithm 1 on a string in L never outputs the parex, which means that Algorithm 2 needs just $O(\log m)$ time and I/Os to solve a membership query.

3.4 Signature-based static z-fast tries

If the length of the strings in L is not bounded by $O(w/\log\sigma)$, the map $Z(-)$ described in the previous section uses superlinear space and superconstant time at every access. This is why the "long string" version of dynamic z-fast tries [3] replaces handles with *signatures*: instead of storing pairs (h_α, α) we store pairs $(H(h_\alpha), \alpha)$ where $H(-)$ is a suitably chosen signature hash function. The signature-based version is designed to work with sets of at most $2^{O(w)}$ strings of length up to $2^{O(w)}$, but fat binary searches return the correct result only with high probability.

Note that by using hashes of size $(c + \varepsilon)\log n$, $c \geq 2$, we will find distinct hash values for all handles after a constant expected number of attempts. Indeed, under a full randomness assumption the probability of a collision when t elements are extracted from a set of u with replacement is well approximated by $1 - e^{-t^2/2u}$, which means that with the choice above the probability of having a hash collision between distinct handles is at most

$$1 - e^{-n^2/2^{1+(c+\varepsilon)\log n}} = 1 - e^{-n^2/2n^{c+\varepsilon}} \leq \frac{1}{2n^{c-2+\varepsilon}} \to 0 \qquad \text{as } n \to \infty.$$

Once the signatures are all distinct, in estimating the probability of error of a fat binary search we have to care just about at most $\log m$ false positives, which by the union bound happen with probability at most

$$2^{-(c+\varepsilon)\log n}\log m = \frac{1}{n^c n^\varepsilon}O(w) = o\left(\frac{1}{n^c}\right).$$

Note that each time we have to query the z-map, we have to compute the hash of a potentially long prefix: to this purpose, the dynamic z-fast trie uses hash functions that can hash any prefix of the pattern p in constant time after preprocessing p in time $O(1+(m\log\sigma)/w)$ and storing a linear amount of information. We will see that even stronger properties will be needed in Section 5.

Finally, the signature-based version needs that besides c_α, also n_α can be accessed in constant time from α. This can be obtained in different ways, but usually the simplest one (and the one used by the dynamic z-fast trie) is to store in α a pointer to a suitable element of L. Another possibility is to store explicitly e_α.

Due to signature collisions and false positives, Algorithm 1 may output a node γ that is neither the exit node nor the parex. Let us see how this fact impacts on Algorithm 2; looking back at Lemma 1, we have the following cases:

- the node γ returned by Algorithm 1 is in fact either the exit node or the parex (because no false positives were found during the execution, or because by chance we anyway landed in the correct place): in this case everything goes smoothly as before;
- the returned node γ is not even an ancestor of $\operatorname{exit}(p)$: in this case, the first assertion (that $n_\gamma \preceq p$) fails;
- finally, if γ is a proper ancestor of $\operatorname{parex}(p)$, then Algorithm 2 proceeds as if γ was the parex but then the second assertion of the algorithm fails (γ' would in that case be an ancestor of the parex, and not the exit node as expected).

Thus, when executing Algorithm 2 in the signature-based case, we have to actually verify the assertions, which requires time $O(1 + (m \log \sigma)/w)$ and $O(1 + (m \log \sigma)/B)$ I/Os, as we have to compare the *whole name* n_γ with p (the cost of verifying the assertions covers also the cost of the comparisons with extents discussed in the proof of Theorem 6). If either assertion fails, we have to resort to the standard naive trie search to look for the exit node. The naive search requires, of course, time $O(m\sigma)$. Summing up:

▶ **Theorem 7.** *Under a full randomness assumption, let $c \geq 2$ and assume that $Z(-)$ stores $((c+\varepsilon)\log n)$-bit hash values without collisions. Then, in time $O((m \log \sigma)/w + \log m + \sigma)$ and with $O((m \log \sigma)/B + \log m + \sigma)$ I/Os Algorithm 2 returns the correct result with probability at least $1 - o(1/n^c)$; otherwise, it detects an assertion error, in which case it returns the correct result by resorting to the standard naive search on the trie, which requires $O(m\sigma)$ time and I/Os.*

4 Suffix trees and suffix arrays

For a given string $s \in \Sigma^*$ of length $|s| = n$, let $\operatorname{Suff}(s\$)$ be the set of all the nonempty suffixes of $s\$$. We write $T(s)$ as an abbreviation for $T(\operatorname{Suff}(s\$))$. The trie $T(s)$ is called the *suffix tree* of s. As an example, in Figure 2 we show $T(\text{ABRACADABRA})$.

Observe that $T(s)$ is a trie over the alphabet $\hat{\Sigma}$ (the addition of \$ is needed to make the language of suffixes prefix-free). It is worth noticing that in most papers on suffix trees there are no labels on nodes; instead, arcs are labelled with a nonempty strings (and the arcs to the children of a node have strings that differ in the first character). Our trees can be transformed into this (perhaps more standard) representation by removing all node labels and changing the label of each arc (α, α') to $c_{\alpha,\alpha'} c_{\alpha'}$. For the sake of comparison, we show in Figure 3 the alternative representation of the same trie of Figure 2.

Although in theory one could build the suffix tree of a string *explicitly*, much more (space- and time-) efficient approaches are available, based on suffix arrays. The *suffix array* [11] sa of s is the permutation of $\{0, 1, \ldots, n\}$ such that

$$s\$[\mathrm{sa}[0]\,..] < s\$[\mathrm{sa}[1]\,..] < \cdots < s\$[\mathrm{sa}[n]\,..].$$

Let us write s_i as an abbreviation of $s\$[\mathrm{sa}[i]\,..]$. It is also convenient to define the *lcp array* lcp defined by letting $\mathrm{lcp}[0] = \mathrm{lcp}[n+2] = 0$ and $\mathrm{lcp}[i+1]$, $0 \leq i \leq n$, as the length of the longest common prefix between s_i and s_{i+1}. In Figure 4, we show the suffix array and the lcp array for our running example $s = \text{ABRACADABRA}$.

Every node α in the suffix tree can be identified with the interval $[\ell_\alpha \,..\, r_\alpha]$ of indices such that $k \in [\ell_\alpha \,..\, r_\alpha]$ if and only if s_k is the extent of one of the leaves that are descendants of α. The interval $[\ell_\alpha \,..\, r_\alpha]$ is called the *lcp-interval* of node α [1]. It is easy to see that:

- the lcp-interval of the root is $[0 \,..\, n]$;
- leaves are the only nodes whose lcp-interval is a singleton;

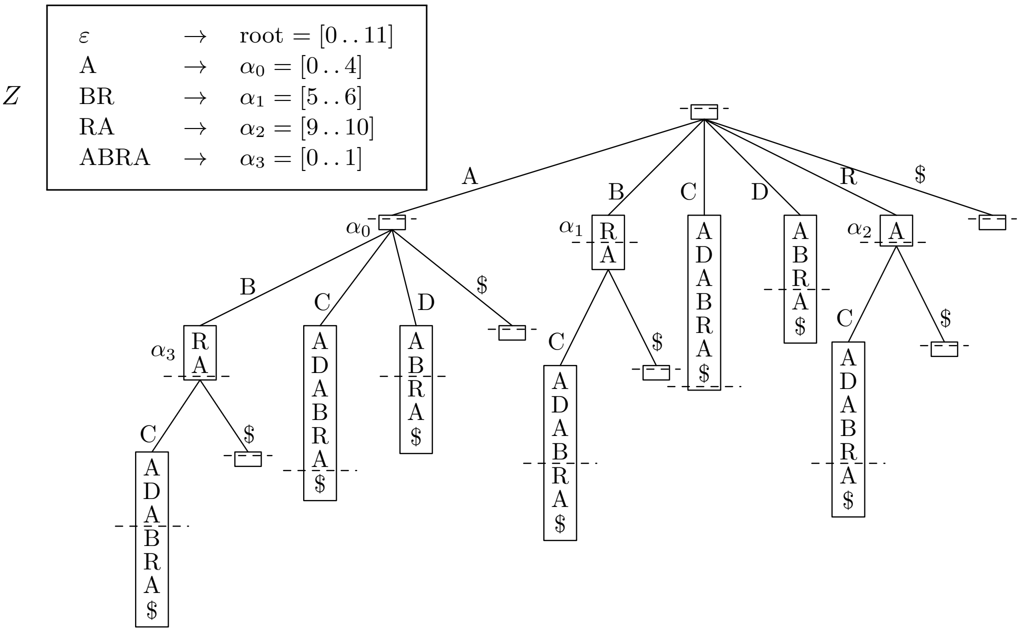


Figure 2 The zuffix tree T(ABRACADABRA) and its z-map (with lcp-intervals).

- the lcp-intervals of the children α_0, α_1, ..., α_k of a node α are an ordered partition of the lcp-interval of their parent, where the ordering is established by the lexicographic order of the characters c_{α,α_i}.

Not only there are efficient algorithms to build the suffix array (and the associated lcp array) [10, 7, 12]: at the price of one additional (and very compressible) array of integers the suffix array can be made into a so-called *enhanced suffix array*, and then used to represent and navigate implicitly the suffix tree [1]. More precisely, we can enumerate the lcp-intervals $[\ell\,..\,\ell_1 - 1]$, $[\ell_1\,..\,\ell_2 - 1]$, ..., $[\ell_k\,..\,r]$ corresponding to the children of the node whose lcp-interval is $[\ell\,..\,r]$. Each child is enumerated in constant time.

It is easy to check that given a node with lcp-interval $[\ell\,..\,r]$ we can compute all the data we need (name, extent, etc.) using only the suffix and the lcp arrays:

$$\begin{aligned}
\left|n_{[\ell,r]}\right| &= \begin{cases} 0 & \text{if } \ell = 0 \text{ and } r = n \text{ (root)} \\ 1 + \max(\operatorname{lcp}[\ell], \operatorname{lcp}[r+1]) & \text{otherwise} \end{cases} \\
\left|e_{[\ell,r]}\right| &= \begin{cases} n - \operatorname{sa}[\ell] + 1 & \text{if } \ell = r \text{ (leaf)} \\ \operatorname{lcp}[a] & \text{otherwise} \end{cases} \\
\left|h_{[\ell,r]}\right| &= \text{the 2-fattest number in } \left[\left|n_{\ell,r}\right|\,..\,\left|e_{\ell,r}\right|\right] \\
n_{[\ell,r]} &= s\left[\operatorname{sa}[\ell]\,..\,\operatorname{sa}[\ell] + \left|n_{[\ell,r]}\right| - 1\right] \\
e_{[\ell,r]} &= s\left[\operatorname{sa}[\ell]\,..\,\operatorname{sa}[\ell] + \left|e_{[\ell,r]}\right| - 1\right]
\end{aligned}$$

where a is the left extreme of the lcp-interval of the second child of $[\ell\,..\,r]$.

Getting back again to the example depicted in Figure 2, consider the leftmost grandchild of the root, corresponding to the lcp-interval $[0\,..\,1]$; its children are both leaves and correspond to $[0]$ and $[1]$. Its name has length $1+\max(\operatorname{lcp}[0], \operatorname{lcp}[2]) = 2$ (see Figure 4) and its extent has length $\operatorname{lcp}[1] = 4$. Since $\operatorname{sa}[0] = 0$, name and extent are $s[0\,..\,1] = AB$ and $s[0\,..\,3] = ABRA$, respectively. Its first child $[0]$ has name of length $1 + \max(\operatorname{lcp}[0], \operatorname{lcp}[1]) = 5$ (its name is in fact ABRAC), and extent of length $n - \operatorname{sa}[0] + 1 = 12$ (which is in fact ABRACADABRA$).

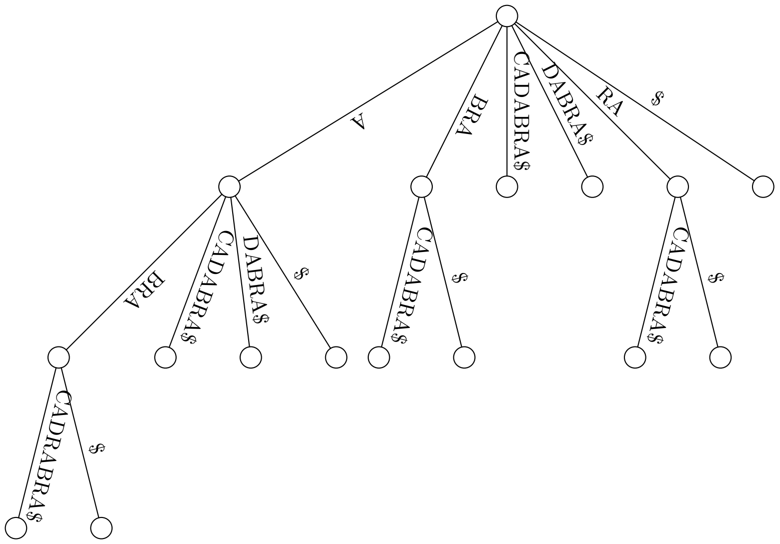


Figure 3 The suffix tree T(ABRACADABRA) in the alternative representation used in most papers about suffix trees.

5 Zuffification

As we explained, there are very efficient (in fact, even very well engineered) algorithms to build a suffix tree or an (enhanced) suffix array in linear time. What we want to do is adding to them a z-map gadget (in the case of suffix arrays, the z-map is that of the suffix tree that the suffix array implicitly represents). We call this process *zuffification* (which looks really nice for a spell), and resulting data structures are called the *zuffix tree* and the *(enhanced) zuffix array*.

The idea is very simple: we perform a depth-first visit of the suffix tree. For each internal node, we can compute its handle (as explained in the previous section), the corresponding hash, and store the correspondence between the hash and the node. We then check for collisions in the hash values of handles, possibly restarting with a different hash function if some collision is found, and finally build a static constant-time dictionary mapping the hashes to the nodes.

Of course, computing a hash for x needs time $O(1 + (|x| \log \sigma)/w)$. However, we can make hashing time constant by resorting to a particular kind of *rolling hashing*: in particular, we want the two following properties to be true:

1. Given a string $x \in \Sigma^*$ and $c \in \Sigma$, we have $H(xc) = f(H(x), c)$, where f can be computed in constant time.
2. Given strings $x, y \in \Sigma^*$, we have $H(y) = g(H(xy), H(x))$, where g can be computed in constant time.

If H is chosen in this way, we can build in linear time (by the first property) a table recording the hashes of the prefixes of the text s. At that point, computing the hash of a(ny) substring of s requires constant time by the second property (the values of $H(x)$ and $H(y)$ being found in the table). Several types of hash functions have the properties described above: the list includes hashing based on cyclic polynomials [15], Karp–Rabin hashing [8] and hashing based on the remainder of the division by a general irreducible polynomial [14] (the string is mapped to a string of bits and then interpreted as a polynomial over $\mathbf{F}_2$). Thus,

i	sa$[i]$	lcp$[i]$	$s_i = s\$[\mathrm{sa}[i]\,.\,.]$
0	0	0	ABRACADABRA$
1	7	4	ABRA$
2	3	1	ACADABRA$
3	5	1	ADABRA$
4	10	1	A$
5	1	0	BRACADABRA$
6	8	3	BRA$
7	4	0	CADABRA$
8	6	0	DABRA$
9	2	0	RACADABRA$
10	9	2	RA$
11	11	0	$
12		0	

Figure 4 The suffix array and lcp array for the string ABRACADABRA.

▶ **Theorem 8.** *Zuffification can be performed in expected linear time (in fact, with high probability) both for suffix trees and suffix arrays.*

A practical improvement that slows down the construction time, but reduces significantly the space usage, is that of recording the hashes only for prefixes whose length is multiple of some value $O(w)$. In the cases above, it possible to still compute the hash of a substring quickly, albeit in some case $O(w)$ operations might be necessary (this depends on which instructions are considered to be atomic: for example, modern processor have constant-time multiplication of polynomial on $\mathbf{F}_2$). However, we have to store much fewer prefix hashes.

5.1 Searching with zuffix arrays

We can finally completely rebuild the spell of the fat sorceress. Given a string s, we build in linear time its enhanced suffix array, and simulate in linear time a visit of the associated suffix tree to build the z-map, thus obtaining the zuffix array. The latter operation works in expected linear time, because there might be collisions.

To search the zuffix array, we apply Theorem 7 to our setting, obtaining the same bounds. More importantly, since the nodes of the simulated suffix tree are exactly the intervals of the suffix array containing the starting points of the suffixes we are looking for, at the end of Algorithm 2 we have found all the locations of the search pattern:

▶ **Theorem 9.** *Under a full randomness assumption, given a pattern p, a zuffix array (in time $O((m\log\sigma)/w + \log m + \sigma)$ and with $O((m\log\sigma)/B + \log m + \sigma)$ I/Os) returns the interval of the underlying suffix array containing the positions at which p appears, with probability at least $1 - o(1/n^c)$; otherwise, it returns the same result in time $O(m\sigma)$.*

Significant practical improvements to the space used by the z-map can be obtained using the following theorem, which can be proved by adapting the proof of Theorem 5:

▶ **Theorem 10.** *Let L be a prefix-free language, and S a parent-closed set of nodes of $T(L)$. Consider a map Z sending $h_\alpha \to \alpha$, $\alpha \in S$. Then, Algorithm 1 returns the lowest ancestor in S of the exit node of the pattern, or its parent.*

The theorem above opens the door to a number of interesting space-time tradeoffs: for example, eliminating c levels of leaves in Definition 4 one obtains a much smaller map, but now locating the actual exit node (Algorithm 2) may require to go down c levels in the trie, using time $O((m \log \sigma)/w + \log m + c\sigma)$ and with $O(m \log \sigma / B + \log m + c\sigma)$ I/Os.

An even more interesting application is in DNA searching: usually the databases are very large, but the patterns are several orders of magnitude shorter: by building a z-map containing only handles shorter than ℓ, the search for patterns shorter than ℓ will not be affected, but the map will be much smaller (longer patterns will still be searchable, possibly using time $O(m\sigma)$).

6 Conclusions

The fat sorceress was actually quite good at spells: given a pattern p, after a search that is logarithmic in $|p|$, a zuffix array tests p against s scanning from at most σ positions. In practice, this means at most σ cache misses in the test phase (this fact is only partially expressed by the I/Os in the cache-oblivious model, as the model does not consider prefetching). The result needs an uncompressed original text, but in the context of the rise of modern low-cost fast storage, like solid-state drives, this limitation does not seem so serious. We remark that in our description we used enhanced suffix arrays, but nothing prevents zuffification of compressed suffix trees [16], compressed suffix arrays [6] or even of the FM-index [4].

References

1 Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, 2(1):53–86, 2004.

2 Djamal Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna. Monotone minimal perfect hashing: Searching a sorted table with $O(1)$ accesses. In *Proceedings of the 20th Annual ACM-SIAM Symposium On Discrete Mathematics (SODA)*, pages 785–794, New York, 2009. ACM Press.

3 Djamal Belazzougui, Paolo Boldi, and Sebastiano Vigna. Dynamic z-fast tries. In Edgar Chávez and Stefano Lonardi, editors, *String Processing and Information Retrieval - 17th International Symposium, SPIRE 2010, Los Cabos, Mexico, October 11-13, 2010. Proceedings*, volume 6393 of *Lecture Notes in Computer Science*, pages 159–172. Springer, 2010.

4 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, jul 2005.

5 Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *ACM Trans. Algorithms*, 8(1):4:1–4:22, 2012.

6 Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

7 Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *Journal of the ACM (JACM)*, 53(6):918–936, 2006.

8 Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

9 Donald E. Knuth. *The Art of Computer Programming*. Addison–Wesley, 1973.

10 Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2-4):143–156, 2005.

11 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

12 Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09.*, pages 193–202. IEEE, 2009.

13 Ge Nong, Sen Zhang, and Wai Hong Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

14 W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, 1961.

15 Eugene Prange. Cyclic error-correcting codes in two symbols. Technical note AFCRC-TN-57-103, Air Force Cambridge Research Center, 1957.

16 Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

Computational Complexity of Generalized Push Fight

Jeffrey Bosboom
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
jbosboom@csail.mit.edu

Erik D. Demaine
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
edemaine@mit.edu

Mikhail Rudoy[1]
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
mrudoy@gmail.com

Abstract

We analyze the computational complexity of optimally playing the two-player board game Push Fight, generalized to an arbitrary board and number of pieces. We prove that the game is PSPACE-hard to decide who will win from a given position, even for simple (almost rectangular) hole-free boards. We also analyze the *mate-in-1* problem: can the player win in a single turn? One turn in Push Fight consists of up to two “moves” followed by a mandatory “push”. With these rules, or generalizing the number of allowed moves to any constant, we show mate-in-1 can be solved in polynomial time. If, however, the number of moves per turn is part of the input, the problem becomes NP-complete. On the other hand, without any limit on the number of moves per turn, the problem becomes polynomially solvable again.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases board games, hardness, mate-in-one

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.11

Related Version `https://arXiv.org/abs/1803.03708`

Acknowledgements This work grew out of an open problem session originally started during an MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.890) in Fall 2014.

1 Introduction

Push Fight [10] is a two-player board game, invented by Brett Picotte around 1990, popularized by Penny Arcade in 2012 [9], and briefly published by Penny Arcade in 2015 [8]. Players take turns moving and pushing pieces on a square grid until a piece gets pushed off the board or a player is unable to push on their turn. Figure 1 shows a Push Fight game in progress, and Section 2 details the rules.

In this paper, we study the computational complexity of optimal play in Push Fight, generalized to an arbitrary board and number of pieces, from two perspectives:

[1] Now at Google Inc.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 11; pp. 11:1–11:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

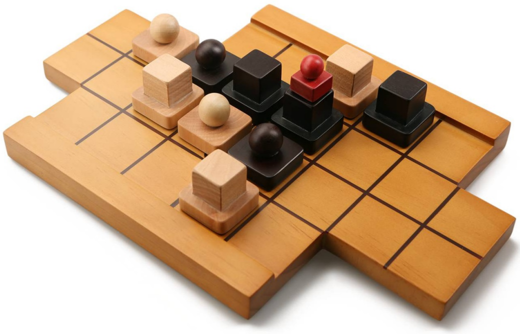


Figure 1 A Push Fight game in progress. Photo by Brettco, Inc., used with permission.

Table 1 Summary of our results.

	Computational complexity of. . .	
Moves per turn	**Mate-in-1**	**Who wins?**
≤ 2	P	PSPACE-hard, in EXPTIME
$\leq c$ constant	P	open
$\leq k$ input	NP-complete	open
unlimited	P	open

1. **Who wins?** The typical complexity-of-games problem is to determine which player wins from a given game configuration.
2. **Mate-in-1**: Can the current player win *in a single turn*?

Table 1 summarizes our results.

Generalized Push Fight is a two-player game played on a polynomially bounded board for a potentially exponential number of moves, so we conjecture the "who wins?" decision problem to be EXPTIME-complete, as with Checkers [11] and Chess [4]. (Certainly the problem is in EXPTIME, by building the game tree.) In Section 4, we prove that the problem is at least PSPACE-hard, using a proof patterned after the NP-hardness proof of Push-$*$ [7]. Our proof uses a simple, nearly rectangular board, in the spirit of the original game; in particular, the board we use is hole-free and x-monotone (see Figure 8). It remains open whether Push Fight is in PSPACE, EXPTIME-hard, or somewhere in between.

Our mate-in-1 results are perhaps most intriguing, showing a wide variability according to whether and how we generalize the "up to two moves per turn" rule in Push Fight. If we leave the rule as is, or generalize to "up to c moves per turn" where c is a fixed constant (part of the problem definition), then we show that the mate-in-1 problem is in P, i.e., can be solved in polynomial time. However, if we generalize the rule to "up to k moves per turn" where k is part of the input, then we show that the mate-in-1 problem becomes NP-complete. On the other hand, if we remove the limit on the number of moves per turn, then we show that the mate-in-1 problem is in P again. Section 3 proves these results.

The mate-in-1 problem has been studied previously for other board games. The earliest result is that mate-in-1 Checkers is in P, even though a single turn can involve a long sequence of jumps [3]. On the other hand, Phutball is a board game also featuring a sequence of jumps in each turn, yet its mate-in-1 problem is NP-complete [2].

For omitted proofs, see the full version of the paper [1].

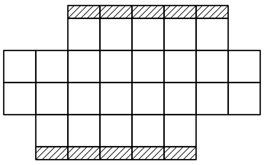


Figure 2 Original Push Fight board. Shaded regions represent side rails.

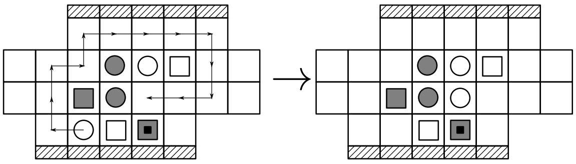


Figure 4 An example move.

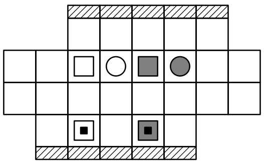


Figure 3 Our notation for pieces, in reading order: a white king, a white pawn, a black king, a black pawn; and white and black anchored kings (in an actual game, there is only one anchor).

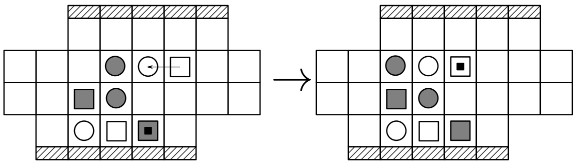


Figure 5 An example push.

2 Rules

The original Push Fight board is an oddly shaped square grid containing 26 squares; see Figure 2. Part of the boundary of this board has *side rails* which prevent pieces from being pushed off across those edges. We generalize Push Fight by considering arbitrary polyomino boards, with each boundary edge possibly having a side rail.

Push Fight is played with two types of pieces, each of which takes up a square of the board: *pawns* (drawn as circles) and *kings* (drawn as squares). Each piece is colored either black or white, denoting which player the piece belongs to. Standard Push Fight is played with three kings and two pawns per player. Additionally, there is a single *anchor* that is placed on top of a king after it pushes (but is never placed directly on the board). Figure 3 shows our notation for the pieces.

Push Fight gameplay consists of the two players alternating *turns*. During a player's turn, the player makes up to two optional *moves* followed by a mandatory *push*.

To make a move, a player moves one of their pieces along a simple path of orthogonally adjacent empty squares; see Figure 4.

To push, a player moves one of their kings into an occupied adjacent square. The piece occupying that square is pushed one square in the same direction, and this continues recursively until a piece is pushed into an unoccupied square or off the board. If this process would push a piece through a side rail, or would push the anchored king, the push cannot be made. Pushes always move at least one other piece. When the push is complete, the pushing king is anchored (the anchor is placed on top of that king). Figure 5 shows a valid push.

A player loses if any of their pieces are pushed off the board (even by their own push) or if they cannot push on their turn.

▶ **Definition 1.** A *Push Fight game state* is a description of the board's shape, including which board edges have side rails, and for each board square, what type of piece or anchor occupies it (if any).

Note that the position of the anchor encodes which player's turn it is: if the anchor is on a white king, it is black's turn, and vice versa. If the anchor has not been placed (no turns have been taken), it is white's turn.

3 Mate-in-1

We consider three variants of mate-in-1 Push Fight, varying in how the number of moves is specified: as a constant in the problem definition, as part of the input, or without a limit.

3.1 c-Move Mate-in-1

▶ **Problem 2.** c-Move Push Fight Mate-in-1: *Given a Push Fight game state, can the player whose turn it is win this turn by making up to c moves and one push?*

The standard Push Fight game has $c = 2$.

▶ **Theorem 3.** c-Move Push Fight Mate-in-1 *is in P.*

Proof Sketch. The number of possible turns is $\leq A^{2c+4}$ on a board of area A. ◀

3.2 k-Move Mate-in-1 is in NP

▶ **Problem 4.** k-Move Push Fight Mate-in-1: *Given a Push Fight game state and a positive integer k, can the player whose turn it is win this turn by making up to k moves and one push?*

In this section, we prove the following upper bound on the number of useful moves in a turn:

▶ **Theorem 5.** *Given a Push Fight game state on a board having n squares, if the current player can win this turn, they can do so using at most n^6 moves followed by a push.*

Proof Sketch. We divide the reachable game states into $\leq n^4$ equivalence classes, and show that two equivalent configurations can be reached via $\leq n^2$ moves within that class. ◀

Our bound directly implies an NP algorithm for k-Move Push Fight Mate-in-1:

▶ **Corollary 6.** k-Move Push Fight Mate-in-1 *is in* NP.

A turn consists of making some number of moves followed by a single push. For the purpose of analyzing a single turn, kings other than the single king that pushes are indistinguishable from pawns, so we can assume the current player first chooses a king, then replaces all of their other kings with pawns before making their moves and push. The following definitions are based on this assumption.

▶ **Definition 7.** Given a single-king game state, a *board configuration* is a placement of pieces reachable by the current player making a sequence of moves.

▶ **Definition 8.** The *pawnspace* of a board configuration is the (possibly disconnected) region of the board consisting of the empty squares and the squares containing the current player's pawns. Equivalently, the pawnspace is the region consisting of all squares not occupied by the current player's king or the other player's pieces.

▶ **Definition 9.** The *signature* of a board configuration is a list of nonnegative integers, where each integer is a count of the current player's pawns in a connected component of the configuration's pawnspace, ordered according to row-major order on the leftmost topmost square in the corresponding connected component.

▶ **Definition 10.** Given two board configurations C_1 and C_2 derived from the same game state, we say that $C_1 \equiv C_2$ if and only if

1. C_1 and C_2 have the same pawnspace (that is, the current player's only king occupies the same square in C_1 and C_2) and
2. C_1 and C_2 have the same signature (that is, each connected component of the pawnspace contains the same number of the current player's pawns in C_1 and C_2).

Relation $\equiv$ is clearly reflexive, symmetric, and transitive, so it is an equivalence relation inducing a partition of the set of board configurations derived from a given game state into equivalence classes. We need the following two lemmas about $\equiv$ for our proof of Theorem 5:

▶ **Lemma 11.** *For a given game state on a board with n squares, there are at most n^4 equivalence classes of board configurations.*

▶ **Lemma 12.** *If $C_1 \equiv C_2$, then C_2 can be reached from C_1 in at most $n^2 - 1$ moves without leaving the equivalence class of C_1.*

We are now ready to prove Theorem 5:

▶ **Theorem 5.** *Given a Push Fight game state on a board having n squares, if the current player can win this turn, they can do so using at most n^6 moves followed by a push.*

Proof. By our assumption that the current player can win this turn, there exists a sequence of moves for the current player after which they can immediately win with a push, corresponding to a sequence of board configurations $C_1, C_2, \ldots, C_l$. Configuration C_1 is obtained from the initial game state by replacing all of the current player's kings, except the one that ends up pushing, with pawns. Each C_{i+1} can be reached from C_i in one move, and C_l is a configuration from which the current player can win with a push.

We now define *simplifying* a sequence of board configurations over an equivalence class E. If the sequence contains no configurations from E, then simplifying the sequence over E leaves it unchanged. Otherwise, let A_i be the first configuration in the sequence in E and A_j be the last configuration in the sequence in E. By Lemma 12, there exists a sequence of fewer than $n^2 - 1$ moves that transforms A_i into A_j, corresponding to a sequence of board configurations $A_i = D_0, D_1, \ldots, D_u = A_j$ with $u \leq n^2 - 1$. Then simplifying over E consists of replacing all configurations between and including A_i and A_j with the replacement sequence $D_0, D_1, \ldots, D_u$.

Notice that simplifying a sequence (over any class) never changes the first or last configuration in the sequence, and each configuration in the resulting sequence remains reachable in one move from the previous configuration in the resulting sequence. After simplifying over a class E, the only configurations in the resulting sequence in E are those in the replacement sequence, so the number of configurations in the sequence in E is at most n^2. Furthermore, all configurations in the replacement sequence are in E, so simplifying over E never increases (but may decrease) the number of configurations falling in other classes.

Let $C'_1, C'_2, \ldots, C'_l$ be the result of simplifying $C_1, C_2, \ldots, C_l$ over every equivalence class. By Lemma 11, there are at most n^4 such classes, and by the above paragraph there are at most n^2 configurations from each class in $C'_1, C'_2, \ldots, C'_l$, so the length of $C'_1, C'_2, \ldots, C'_l$ is at most n^6. Each configuration in $C'_1, C'_2, \ldots, C'_l$ is reachable in one move from the previous configuration, and that sequence of at most n^6 moves leaves the current player in position to win with a push, as desired. ◀

3.3 Unbounded-Move Mate-in-1

▶ **Problem 13.** UNBOUNDED-MOVE PUSH FIGHT MATE-IN-1: *Given a Push Fight game state, can the player whose turn it is win this turn by making any number of moves and one push?*

▶ **Theorem 14.** UNBOUNDED-MOVE PUSH FIGHT MATE-IN-1 *is in P.*

We can of course solve UNBOUNDED-MOVE PUSH FIGHT MATE-IN-1 by trying all possible sequences of moves to find a board configuration from which the current player can win with a push, but there are exponentially many board configurations, so such an algorithm takes exponential time. Instead, we can use the fact that any two configurations in the same equivalence class are reachable from each other in a polynomial number of moves (from Lemma 12) to search over equivalence classes of board configurations instead of searching over board configurations. There are at most n^4 equivalence classes (by Lemma 11), so they can be searched in polynomial time.

We will make use of the following definitions:

▶ **Definition 15.** Two equivalence classes of board configurations C_1 and C_2 are *neighbors* if there exist board configurations $b_1 \in C_1$ and $b_2 \in C_2$ such that b_1 can be reached from b_2 with a king move of exactly one square. The *equivalence class graph* is a graph whose vertices are equivalence classes of board configurations and whose edges connect neighboring equivalence classes.

An equivalence class of board configurations C is a *winning equivalence class* if there exists a board configuration $b \in C$ such that the player whose turn it is can win with a push.

The key idea for our algorithm is the following:

▶ **Lemma 16.** *There exists a path in the equivalence class graph from the equivalence class of the initial board configuration to a winning equivalence class if and only if there exists a winning move sequence.*

The size of the equivalence class graph is polynomial in n (by Lemma 11), so provided the graph can be constructed and the winning equivalence classes identified, this type of path in the equivalence class graph, if it exists, can be found in polynomial time.

Recall from Definition 10 that equivalence classes of board configurations are defined by the pawnspace and signature, and that, for configurations derived from the same game state (i.e., having the other player's pieces in the same positions), the pawnspace is defined by the position of the current player's king. Thus we can uniquely name a class using the king position and signature.

▶ **Definition 17.** The *class descriptor* of an equivalence class of board configurations for a given game state is the ordered pair of the position of the current player's king and the signature defining that class.

To prove Theorem 14, we need to give polynomial-time algorithms to compute the neighbors of an equivalence class and to decide whether a class is a winning equivalence class.

▶ **Lemma 18.** *Given an initial game state and a class descriptor for some class C, we can compute in polynomial time the equivalence classes (as class descriptors) neighboring C.*

▶ **Lemma 19.** *Given an initial game state and a class descriptor for some class C, we can decide in polynomial time whether C is a winning equivalence class.*

We are now ready to prove Theorem 14:

▶ **Theorem 14.** UNBOUNDED-MOVE PUSH FIGHT MATE-IN-1 *is in P.*

Proof. First, compute the class descriptor for the equivalence class of the initial board configuration. Then perform a breadth- or depth-first search of the equivalence class graph, using the algorithm given in the proof of Lemma 18 to compute the neighboring class descriptors and the algorithm given in the proof of Lemma 19 to decide if the search has found a winning equivalence class. Each of these procedures takes polynomial time. By Lemma 11, there are only polynomially many equivalence classes, so the search terminates in polynomial time. By Lemma 16, there exists a winning move sequence if and only if this search finds a path to a winning equivalence class. ◀

The key idea of the above proof is that, if we do not care how many moves we make inside an equivalence class, then it is sufficient to search the graph of equivalence classes. Thus the above proof does not apply to k-MOVE PUSH FIGHT MATE-IN-1, and in the next section, we prove k-MOVE PUSH FIGHT MATE-IN-1 is NP-hard.

3.4 *k*-Move Mate-in-1 is NP-hard

To prove k-MOVE PUSH FIGHT MATE-IN-1 hard, we reduce from the following problem, proved strongly NP-hard in [5]:

▶ **Problem 20.** INTEGER RECTILINEAR STEINER TREE: *Given a set of points in* $\mathbb{R}^2$ *having integer coordinates and a length* ℓ*, is there a tree of horizontal and vertical line segments of total length at most* ℓ *containing all of the points?*

▶ **Theorem 21.** k-MOVE PUSH FIGHT MATE-IN-1 *is strongly* NP*-hard.*

Proof Sketch. The basic idea of our reduction is to create a game state mostly full of the current player's pawns, but with a few empty squares (*holes*). The player must "move" the holes (by moving pawns into them, creating a new hole at the pawn's former square) to free a king that can push one of the other player's pieces off the board. Initially each pawn can only travel one square (into an adjacent hole) per move, but once two holes have been brought together, a pawn can travel two squares per move, and so on. Bringing the holes together optimally amounts to finding a Steiner tree covering the holes' initial positions.

Reduction: Suppose we are given an instance of INTEGER RECTILINEAR STEINER TREE consisting of points $p_i = (x_i, y_i)$ with $i = 1, \ldots, n$ and length ℓ. For convenience, and without affecting the answer, we first translate the points so that $\min x_i = 2$ and $\min y_i = 4$ and reorder the points such that $y_1 = 4$.

We then build a Push Fight game state with a rectangular board with a height of $\max y_i$ and a width of $n + \max x_i$, indexed using 1-based coordinates with the origin in the bottom-left square; refer to Figure 6. The entire boundary of the board has side rails except the edge adjacent to square $(x_1, 1)$. There is a white king in square $(x_1 + n, 2)$ and a black king with the anchor in square $(x_1 - 1, 2)$. There is a black pawn in square (x, y) if any of the following are true:

1. $y = 3$ and $x \neq x_1$,
2. $y = 2$ and either $x < x_1 - 1$ or $x > x_1 + n$, or
3. $y = 1$.

The squares (x_i, y_i) with $1 \leq i \leq n$ (corresponding to the points in the INTEGER RECTILINEAR STEINER TREE instance) are empty. All remaining squares are filled with white pawns. The output of the reduction is this Push Fight board together with $k = \ell + 3$. ◀

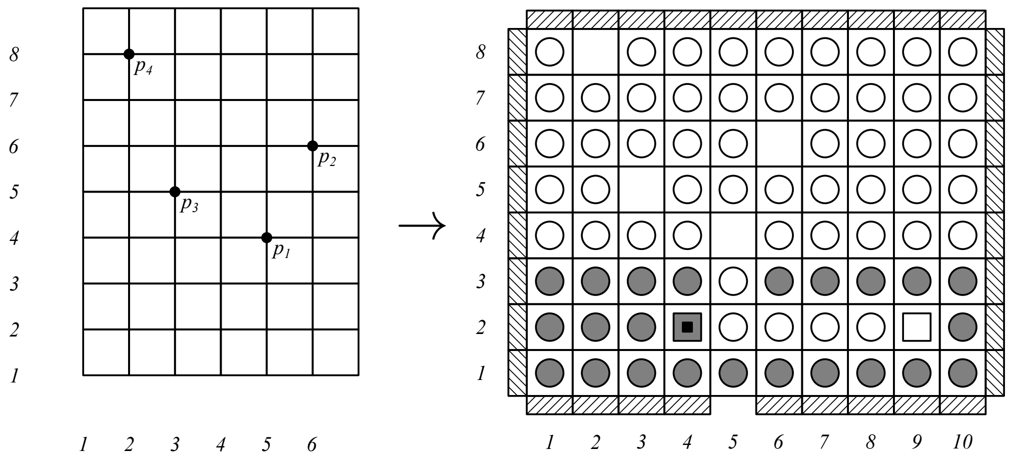


Figure 6 A Push Fight board (right) produced during the reduction from the points in an example rectilinear Steiner tree instance (left).

4 Push Fight is PSPACE-hard

In this section, we analyze the problem of deciding the winner of a Push Fight game in progress.

▶ **Problem 22.** Push Fight: *Given a Push Fight game state, does the current player have a winning strategy (where players make up to two moves per turn)?*

▶ **Theorem 23.** Push Fight *is* PSPACE*-hard.*

To prove PSPACE-hardness, we reduce from Q3SAT, proved PSPACE-complete in [12, 6]:

▶ **Problem 24.** Q3SAT: *Given a fully quantified boolean formula in conjunctive normal form with at most three literals per clause, is the formula true?*

Our proof parallels the NP-hardness proof of Push-$*$ in [7]. Push-$*$ is a motion-planning problem in which a robot (agent) traverses a rectangular grid, some squares of which contain blocks. The robot can push any number of consecutive blocks when moving into a square containing a block, provided no blocks would be pushed over the boundary of the board. The Push-$*$ decision problem asks, given a initial placement of blocks and a target location, can the robot reach the target location by some sequence of moves? In our proof, the white king takes the place of the Push-$*$ robot[2] and white pawns function as blocks. Our proof has the additional complication that Black sets the universally quantified variables, and that White's moves and Black's push must be forced at all times to keep the other gadgets intact.

Figure 7 shows an overview of the reduction. The sole white king begins at the bottom-left of the *variable gadget I* block, setting existentially quantified variables as it pushes up and right. The *variable gadget II* block contains black pawns and holes that allow Black to set the universally quantified variables. After all the variables have been set, the white king traverses the *bridge* to the *clause gadget* block. The variable and clause gadgets interact via a pattern of holes in the *connection block* encoding the literals in each clause. The white

[2] The Push-$*$ robot can move without pushing blocks, so the correspondence is not exact.

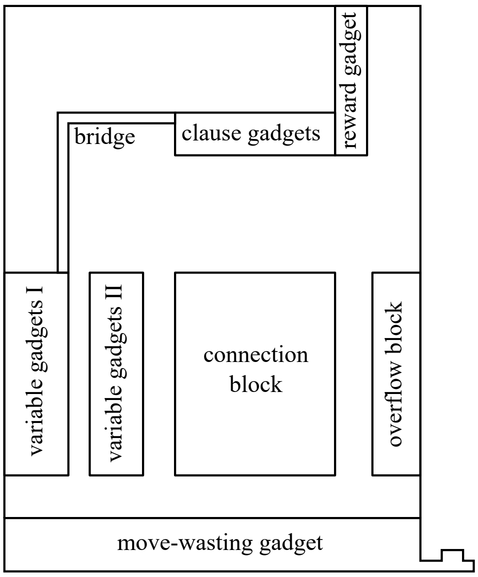


Figure 7 An overview of the Push Fight board produced by our reduction.

king can traverse the clause gadgets only if the variable gadgets were traversed in a way corresponding to a satisfying assignment of the variables. The *reward gadget* contains a boundary square without a side rail, such that the white king can push a black pawn off the board if the white king reaches the reward gadget. The *overflow block* contains empty squares needed by the variable gadgets that were not used in the connection block (for variables appearing in few clauses). The *move-wasting gadget* forces White's moves and Black's push, ensuring the integrity of the other gadgets. Finally, all other squares on the board are filled with white pawns, and the boundary has side rails except at specific locations in the reward and move-wasting gadgets. Figure 8 shows an example output of the reduction.

We first prove the behavior of each of the gadgets, then describe how the gadgets are assembled.

4.1 Move-wasting gadget

The move-wasting gadget requires White to use both moves to prevent Black from winning on the next turn (unless White can win in the current turn). The move-wasting gadget contains the only black king, thus consuming (and allowing) Black's push each turn. When analyzing the other gadgets, we can thus assume White can only push and Black can only move. The move-wasting gadget comprises the entire bottom three rows of the board, but pieces only move in the far-right portion. Figure 9a shows the initial state of the gadget. Throughout this analysis, we assume White cannot win in one turn; Section 4.5, which analyzes the reward gadget, describes the position in which White can immediately win in one turn, and can therefore disregard the threat from Black in the move-wasting gadget.

In the initial state, the anchor is on the black king, so it is White's turn. White must move the pawn above the black king to avoid losing next turn. There are only two reachable empty squares, both in the column left of the black king. If the other square in that column remains empty, Black can move the black king into it and push the white pawn in that column off the board. Thus White must fill the other square in that column, and the only way to do so is to move the pawn two columns left of the white king one square right. Figure 9b shows the resulting position (after White pushes elsewhere in the board).

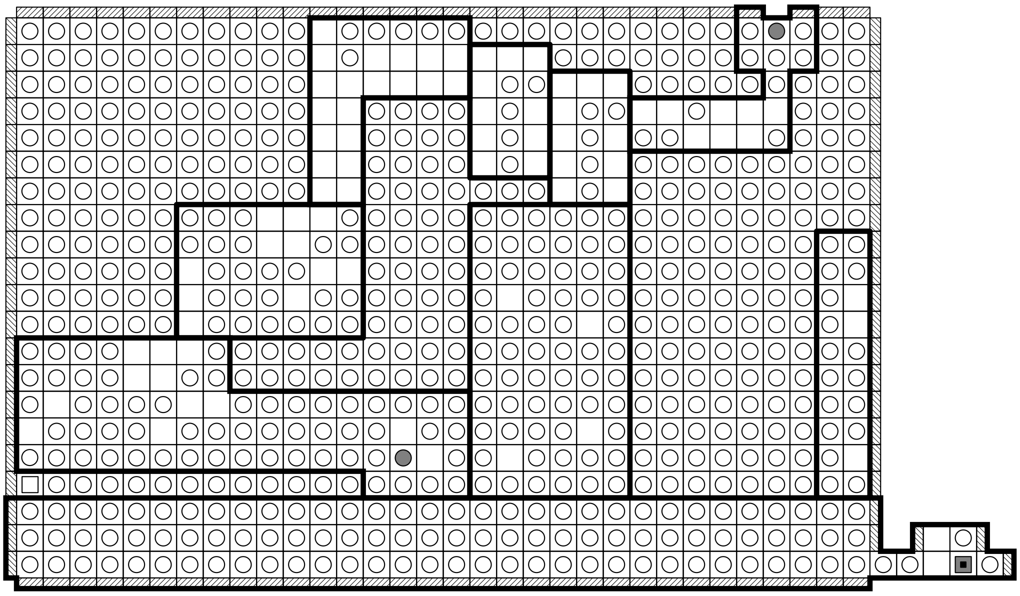
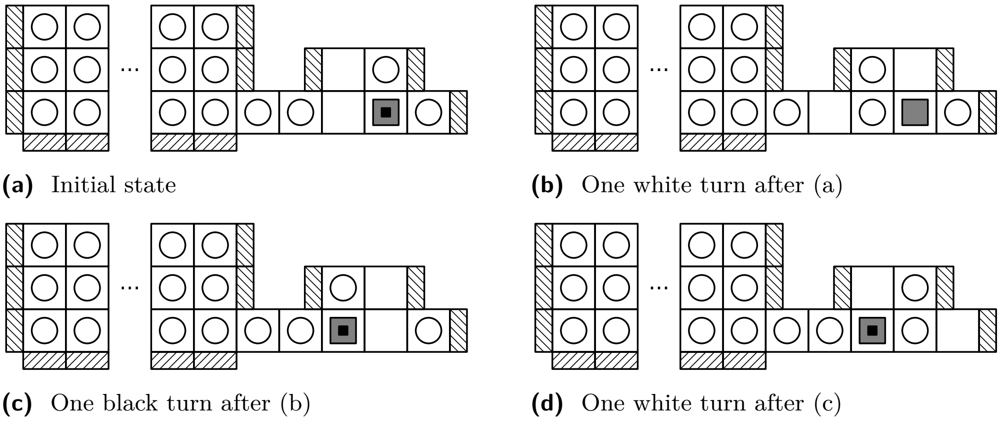


Figure 8 The result of performing the reduction on the formula $\forall x \exists y\ (x \vee \neg y) \wedge (\neg x \vee y)$. Gadgets and blocks are outlined.



(a) Initial state

(b) One white turn after (a)

(c) One black turn after (b)

(d) One white turn after (c)

Figure 9 The move-wasting gadget.

Black's only legal push is to the left, resulting in the position shown in Figure 9c.

The rightmost four columns in Figure 9c are simply the reflection of those columns in Figure 9a, so by the same argument White must fill the column to the right of the black king, resulting in Figure 9d.

Again, the rightmost four columns of Figures 9d and 9b are reflections of each other. Black's only legal push is to the right, restoring the gadget to the initial state shown in Figure 9a. Thus until White can win in one turn, White must use both moves in the move-wasting gadget, and at all times Black must (and can) push in the move-wasting gadget. In the analysis of the remaining gadgets, if the white king reaches a position from which it cannot push, we conclude that White immediately loses, because if White moves a pawn or the king into position to push, Black can win on the next turn as explained above.

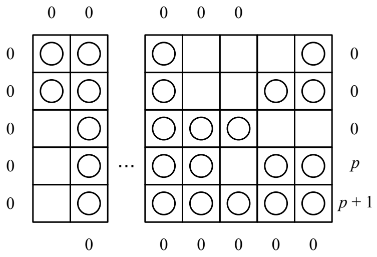


Figure 10 Existential variable gadget.

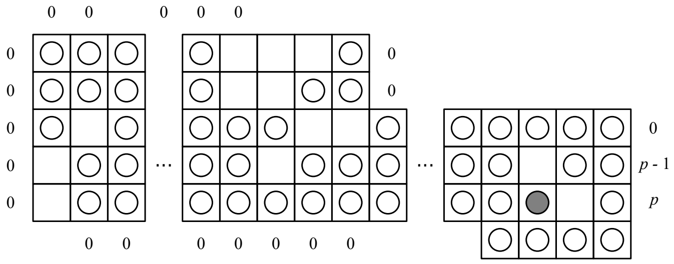


Figure 11 Universal variable gadget.

4.2 Variable gadgets

The existential variable gadget forces White to fill all empty squares in one row of the connection block, corresponding to setting the value of that variable. The universal variable gadget allows Black to choose the value of the corresponding variable, then forces White to similarly fill a row of empty squares. We first analyze a core gadget; the existential variable gadget is a minor variant of the core gadget and its correctness follows directly, while the universal variable gadget has an additional component to allow Black to choose the variable's value. Throughout our analysis, we take advantage of the board being filled with white pawns to limit the number of pieces that can leave the gadget.

The core gadget occupies a rectangle of width $p + 5$ and height 5. When instantiated in the reduction, the gadget lies entirely within the *variable gadget I* block. Integer p is one more than the maximum number of occurrences of a literal in the input formula. The initial state of the core gadget is shown in Figure 12. Each number along the boundary of the figure gives the number of empty squares outside the gadget in that direction, and thus an upper bound on the number of pieces that can leave the gadget via that edge.

The following lemma summarizes the constraints we prove about the core gadget.

▶ **Lemma 25.** *Starting from the position in Figure 12, and assuming the white king does not push down or left from this position,*

(i) *the white king leaves in the second-rightmost column, and*

(ii) *when the white king leaves either*

  (a) *the gadget is as shown in Figure 13 and $p + 1$ white pawns have been pushed out along the bottom row of the gadget, or*

  (b) *the gadget is as shown in Figure 14 and p white pawns have been pushed out along the second-to-bottom row of the gadget,*

(iii) *and no other pieces have left the gadget.*

We will construct the existential and universal variable gadgets from the core gadget such that the assumption holds. Lemma i ensures we can chain variable gadgets together

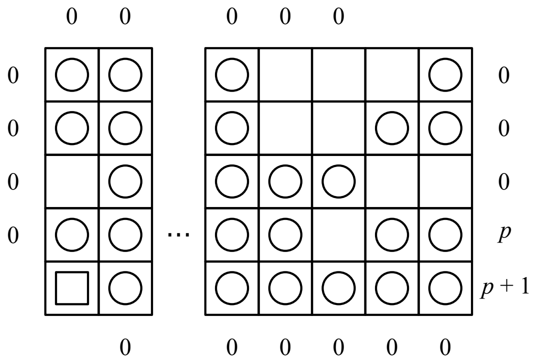


Figure 12 The initial configuration of the core gadget together with upper bounds on the number of pushes out of the gadget at each boundary edge. Omitted columns do not have a given upper bound.

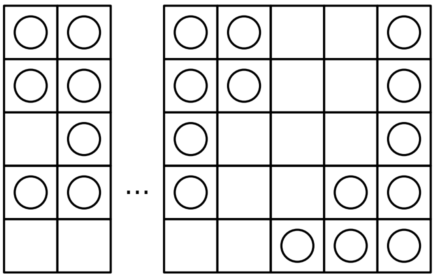


Figure 13 The final configuration of the core gadget after setting the variable to true.

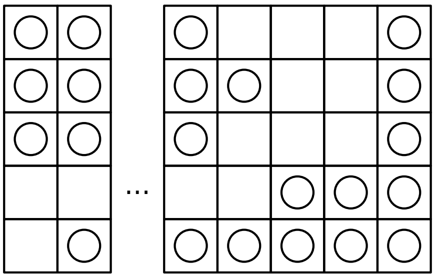


Figure 14 The final configuration of the core gadget after setting the variable to false.

in sequence without the white king escaping. The outcomes implied by Lemma iia and iib correspond to setting the variable to true or false (respectively) by filling in the empty squares in the connection block that could be used to satisfy a clause gadget for a clause containing the opposite literal; that is, pushing pawns out along the bottom row of a gadget prevents all negative literals from being used to satisfy a clause, and similarly for the second-to-bottom row and positive literals.

Proof. We proceed by case analysis starting from Figure 12. The move-wasting gadget consumes White's moves, and there are no black pieces in the core gadget, so we need only analyze the sequence of White's pushes.

Suppose the white king first pushes right. Because of the upper bounds along the top and bottom edges of the gadget, the only legal push in the resulting configuration is to the right, and this remains the case until the white king reaches the fourth column from the right of the gadget. At this point $p + 1$ pawns have been pushed off the right edge along the bottom row of the gadget, so there are no empty squares remaining in that row, so pushing right is no longer possible and the only legal push is up. Then the only legal push is again up because of the constraints on the left edge of the gadget. Figure 15 shows the result of this sequence of pushes.

If the white king pushes left from this position, the only possible next push is down, after which there are no legal pushes, resulting in a loss for White. Figure 16 shows this sequence of pushes.

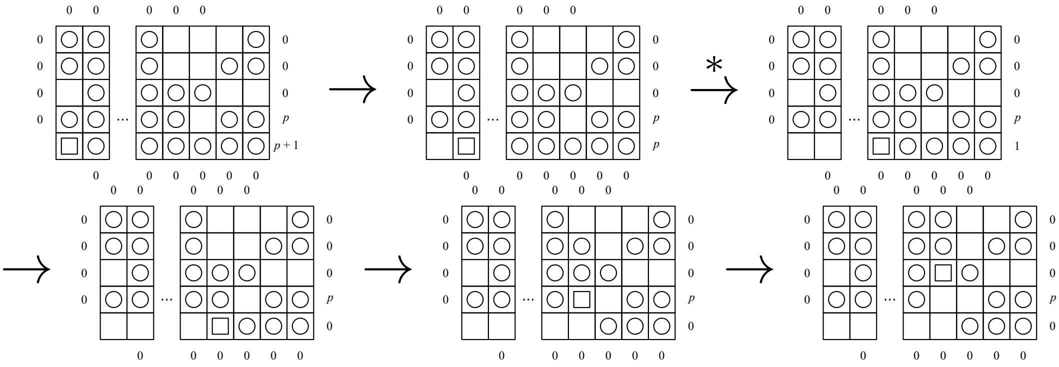


Figure 15 One possible push sequence starting from the initial state of the core gadget. The starred arrow elides a series of pushes to the right.

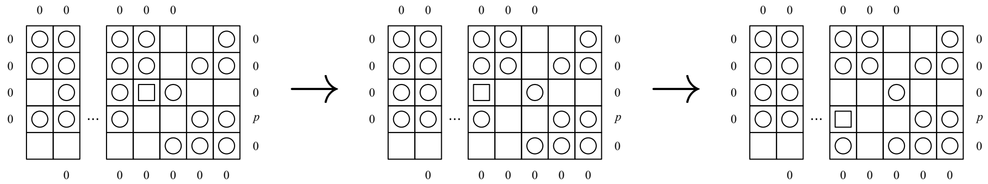


Figure 16 The result of pushing left and down from the last position in Figure 16. White has no legal pushes in the final position.

The only other legal push from the last position in Figure 15 is to the right, after which pushes right, up, up and up again are the only legal pushes. This sequence results in the white king, preceded by a white pawn, exiting the top of the gadget in the second-rightmost column, as desired by Lemma i. Figure 17 shows the positions resulting from this sequence. The final position reached is the position in Figure 13, $p+1$ pawns were pushed out of the gadget to the right along the bottom row, as desired by Lemma iia, and and no other pieces were pushed out of the gadget, as desired by Lemma iii.

Now suppose that the white king pushes up from the initial configuration. Because of the constraints on the gadget boundary, the only legal push is to the right until the white king reaches the fourth column from the right of the gadget. At this point p pawns have been pushed off the right edge along the second-to-bottom row of the gadget, so there are no empty squares remaining in that row, so pushing right is no longer possible and the only legal push is up. Then the only legal push is again up because of the constraints on the left edge of the gadget. Figure 18 shows the result of this sequence of pushes.

If the white king pushes up from this position, there are no legal pushes in the resulting position, resulting in a loss for White. Figure 19 shows this push and the resulting losing position.

The only other legal push from the last position in Figure 18 is to the right, after which pushes right, up, up and up again are the only legal pushes. This sequence results in the white king, preceded by a white pawn, exiting the top of the gadget in the second-rightmost column, as desired by Lemma i. Figure 20 shows the positions resulting from this sequence. The final position reached is the position in Figure 14, and p pawns were pushed out of the gadget to the right along the second-to-bottom row, as desired by Lemma iib. No other pieces were pushed out of the gadget, as desired by Lemma iii.

This completes the case analysis. ◀

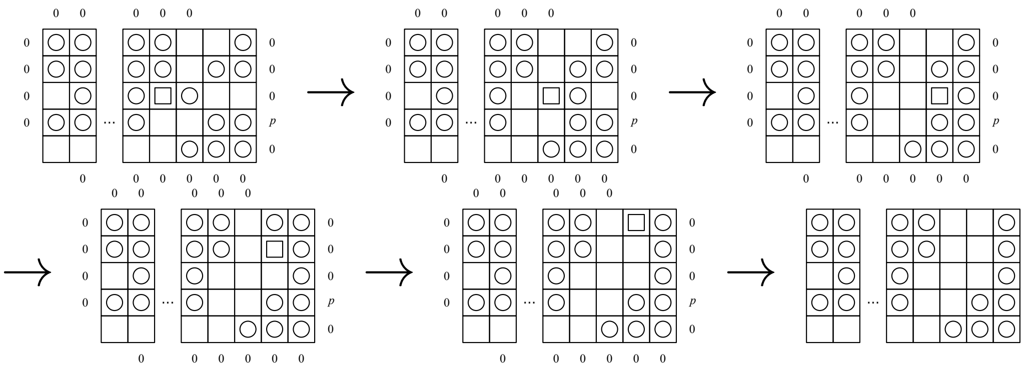


Figure 17 The result of pushing right from the last position in Figure 15, reaching the position in Figure 13.

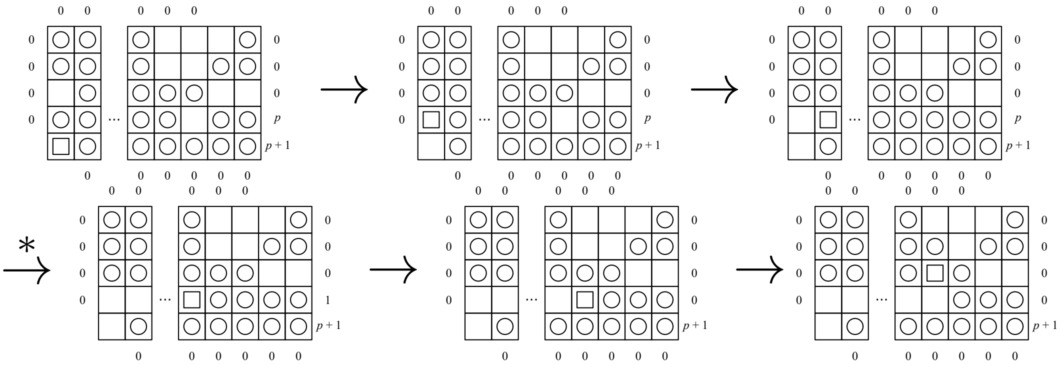


Figure 18 The other possible push sequence starting from the initial state of the core gadget. The starred arrow elides a series of pushes to the right.

Existential variable gadget:

The existential variable gadget, shown in Figure 10, is nearly the same as the core gadget, differing only in the bottom of the leftmost column. When instantiated in the reduction, the white king enters the gadget by pushing a white pawn up into the leftmost column, becoming exactly the core gadget. From the position immediately after the white king enters the gadget, the white king cannot push left (because there are no empty spaces in the row to the left) nor down (because it just pushed up, leaving an empty space in its former position), satisfying the assumption in Lemma 25. Thus by Lemma i, the white king leaves the existential variable gadget in the second-rightmost column with a white pawn above it, and by either Lemma iia or iib, all empty squares in one of two rows of the connection block are now filled by pawns pushed out of the existential variable gadget.

Universal variable gadget:

The universal variable gadget consists of two disconnected regions. The left subregion of the gadget occupies a $(p + 6) \times 5$ rectangle in the *variable gadget I* block. As the white king proceeds through the left region of the gadget, a subregion of the gadget reaches the initial state of the core gadget. The right region of the gadget occupies a 4×4 rectangle in the *variable gadget II* block and contains a black pawn to allow Black to control the value of the variable. The bottom of the right region is one row lower than the bottom of the left

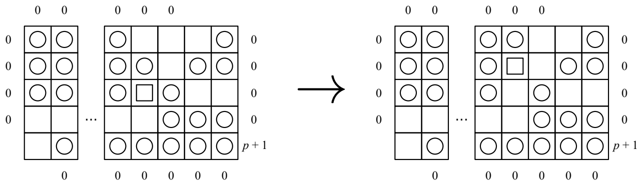


Figure 19 The result of pushing up from the last position in Figure 18. White has no legal pushes in the final position.

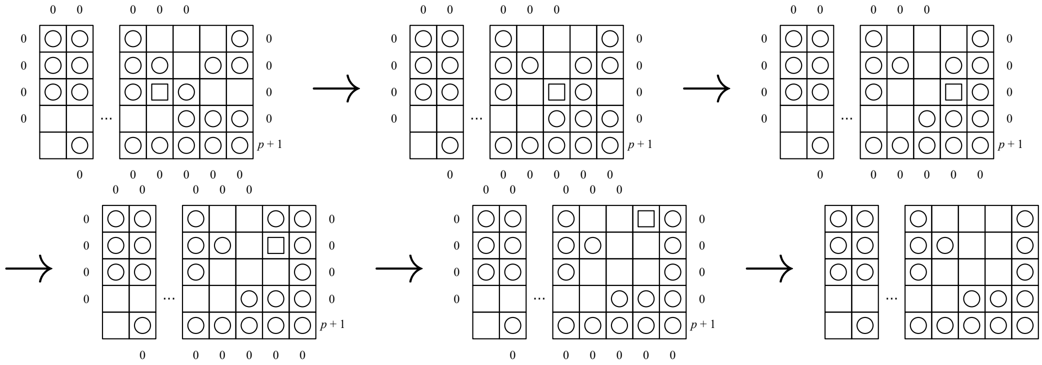


Figure 20 The result of pushing right from the last position in Figure 18, reaching the position in Figure 14.

region. The area between the two regions of the gadget (in the three rows shared by both) is entirely filled by white pawns. Figure 11 shows the universal variable gadget, including the pawn-filled area between the regions.

As with the existential variable gadget, when instantiated in the reduction, the white king enters the universal variable gadget by pushing a white pawn up into the leftmost column. Figure 21 shows the resulting position. Regardless of Black's move, White's only legal push is to the right. By moving the black pawn, Black can choose between the two positions in Figure 22, depending on which of the two rows the black pawn is in when White pushes.

In both of the resulting positions, the black pawn is surrounded, so Black can no longer influence events in this gadget. The left region of the gadget, without the leftmost column, is identical to the initial position of the core gadget. In both positions, the white king cannot push left (empty space) or down (no empty spaces down in the column), satisfying the assumption in Lemma 25. Thus either Lemma iia or Lemma iib holds. Because of the edge constraints, in Figure 22a, only Lemma iia is possible, resulting in Figure 23a. Similarly, in Figure 22b, only Lemma iib is possible, resulting in Figure 23b. By moving the black pawn to select one of these two cases, Black sets the value of the corresponding variable. Then by Lemma i, the white king leaves in the second-rightmost column of the left region (in the *variable gadget I* block) of the gadget. In both cases, the black pawn remains surrounded by white pawns in the right region of the gadget.

4.3 Bridge gadget

The bridge gadget, shown in Figure 24, brings the white king from the exit of the last variable gadget to the entrance of the first clause gadget. When instantiated in the reduction, the white king enters the bridge gadget from the bottom of the leftmost column, preceded by a

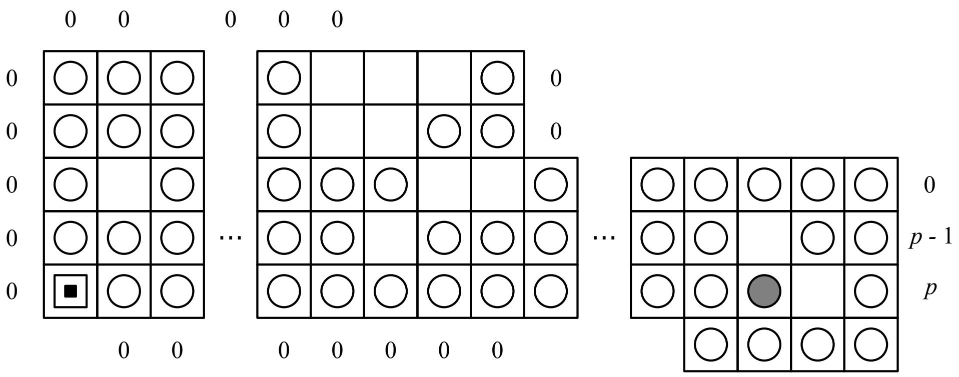


Figure 21 The universal variable gadget after the white king enters.

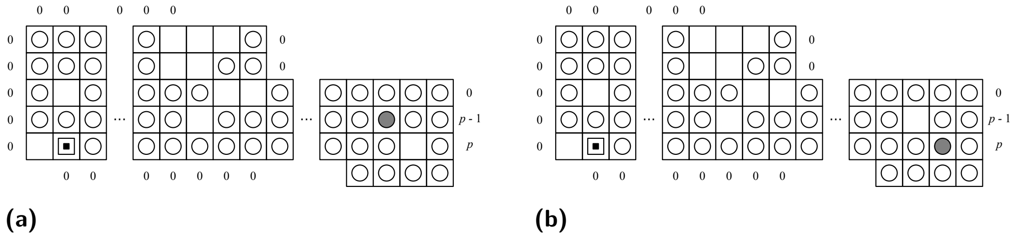


Figure 22 The two possible configurations of the universal variable gadget one white turn after the configuration from Figure 21.

white pawn. The white king's traversal of the bridge gadget is entirely forced. The white king leaves the gadget by pushing a white pawn out to the right in the second-to-top row.

4.4 Clause gadget

The clause gadget, shown in Figure 25, verifies that a column below the gadget contains at least one empty square. When instantiated in the reduction, the white king enters the gadget from the left in the top row, preceded by a white pawn. The resulting sequence of forced pushes includes a push down in the central column of the gadget; if there are no empty squares below the gadget in that column, the white king has no legal pushes and White loses. If there are more empty squares, White can continue to push down, but (when instantiated in the reduction) there are at most three total empty squares in that column, and once those squares are filled, White cannot push. Thus the white king must push right instead and leave the gadget by pushing a white pawn out to the right in the second-to-top row.

4.5 Reward gadget

The reward gadget, shown in Figure 26, allows White to win if the white king reaches the gadget. The black pawn in this gadget cannot move because it is surrounded. When instantiated in the reduction, the white king enters the gadget from the left in the top row, preceded by a white pawn. After pushing right until the white king is in the third column of Figure 26, White can win by moving a white pawn and the white king, then pushing upwards to push the black pawn off the board, as shown in Figure 27. (Recall that the move-wasting gadget no longer binds White once White can win in one turn; Black loses before Black can win using the move-wasting gadget.)

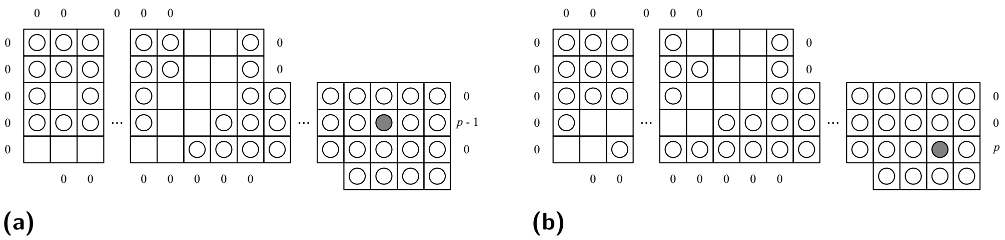


Figure 23 The two possible final positions of the universal variable gadget after the white king exits.

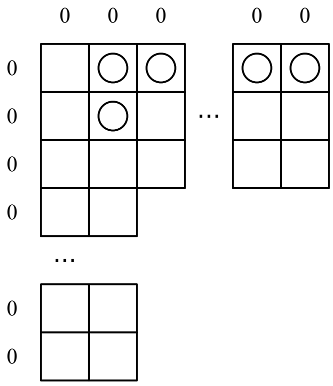


Figure 24 The bridge gadget.

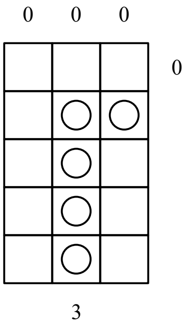


Figure 25 The clause gadget.

4.6 Layout

Having described the gadgets, it remains to show how to instantiate them in a Push Fight game state for a given quantified 3-CNF formula. We first place gadgets with respect to each other, remembering which squares should be left empty, then define the board as the bounding box of the gadgets and fill any squares not recorded as empty with white pawns. The resulting board is mostly rectangular with side rails on all boundary edges, with two exceptions: one edge along the top of the rectangle lacks a side rail as part of the reward gadget, and the board is extended in the bottom-right to accomodate the move-wasting gadget along the bottom of the board.

We begin by building the *variable gadget I* block containing the existential variable gadgets and the left portion of the universal variable gadgets. Gadgets are stacked from bottom to top in the order of the quantifiers in the input formula (using the gadget corresponding to the quantifier), with the leftmost column of each gadget aligned with the second-to-right column of the previous gadget. (Recall that the width of the variable gadgets is defined based on p, one more than the maximum number of occurrences of a literal in the input formula.) This alignment allows (and requires) the white king to traverse the gadgets in sequence as specified by Lemma 25. Figure 29 shows the relative layout of these variable gadgets.

We place the white king one square below the first variable gadget aligned with its leftmost column, and place a white pawn one square above the white king. The white king will push upwards into the first gadget on White's first turn. (If the king was instead placed directly in the variable gadget, if the first variable is universally quantified, Black would not have a move with which to choose the value of the variable before White commits it.)

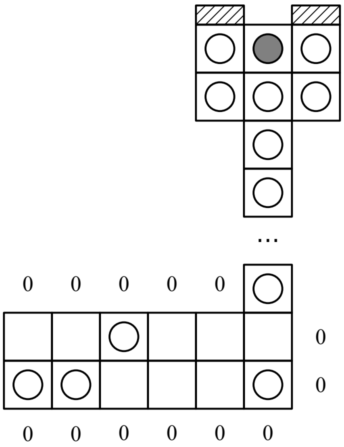


Figure 26 The reward gadget.

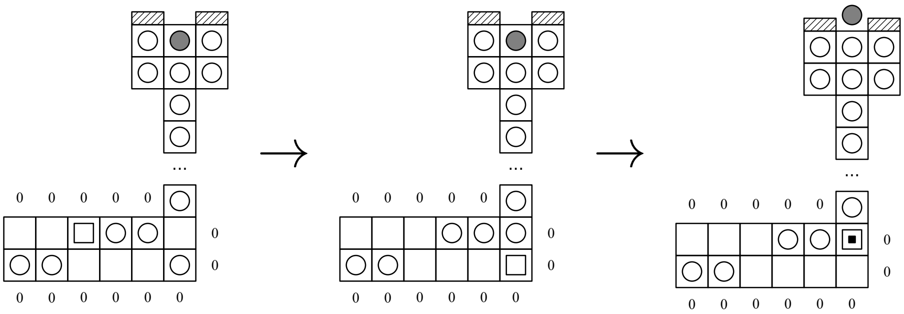


Figure 27 Once the White king reaches the third column of the reward gadget, White can win in a single turn.

We then build the *variable gadget II* block by placing the right regions of the universal variable gadgets to the right of the corresponding left regions in a single column (further right than any part of the variable gadget I section).

Next we place one clause gadget for each clause in the input formula. Each clause gadget is directly to the right of and one square lower than the previous clause gadget. The entire clause gadget block is further right of and above the *variable gadget II* block. Figure 30 shows the relative layout of the clause gadgets. Then we place a bridge gadget such that the entrance of the bridge gadget aligns with the exit of the last variable gadget and the exit of the bridge gadget aligns with the entrance of the first clause.

We place the reward gadget so that its entrance aligns with the exit of the last clause gadget.

We leave empty squares in the connection block to encode the literals in each clause in the input formula. When traversing each variable gadget, the white king pushes pawns to the right in one of two rows. The lower (upper) row corresponds to setting the variable to true (false), or equivalently, preventing negative (positive) literals from satisfying clauses. Associate each row with the literal it prevents from satisfying clauses. Each clause gadget enforces that at least one empty square remains below its middle column, corresponding to at least one of its literals not having been ruled out by the truth assignment. To realize this relation, for each literal in a clause, we leave an empty square at the intersection of the

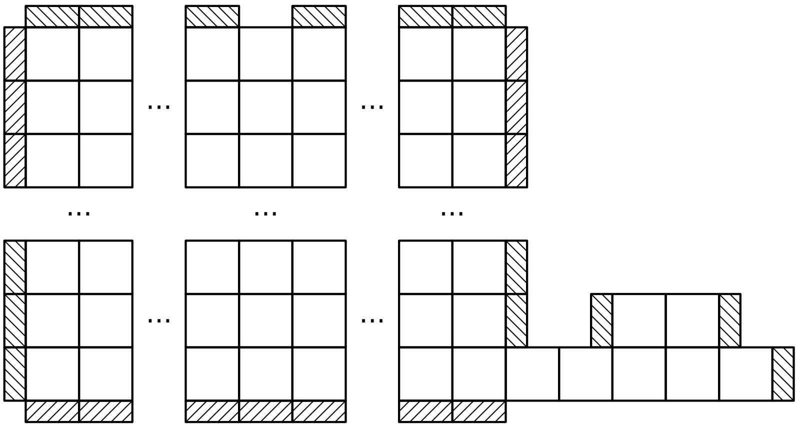


Figure 28 The shape of the Push Fight board produced by the reduction.

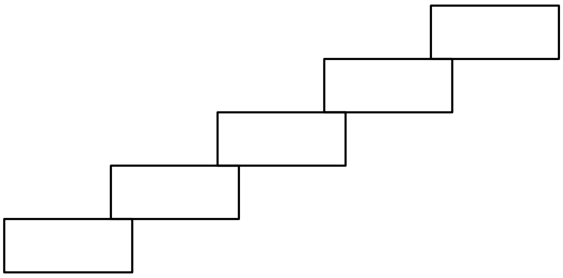


Figure 29 The layout of variable gadgets in the *variable gadget I* block.

column checked by the clause gadget and the row associated with that literal. All other squares in the connection block are filled with white pawns (as are all squares in the board whose contents are not otherwise specified).

The variable gadgets require each row associated with a literal to contain exactly $p-1$, p or $p+1$ empty squares (depending on the type of gadget and whether the row is the upper or lower row). This is at least the number of occurrences of that literal (by the definition of p), but it may be greater. We place any remaining empty squares in each row in columns further right than the reward gadget, forming the overflow block.

The boundary of the board is the bounding box of all the gadgets placed thus far with a move-wasting gadget appended to the bottom of the board. The left column of the move-wasting gadget is aligned with the leftmost column of the first (leftmost) variable gadget and the sixth-from-right column (the rightmost column having height 3) is aligned with the rightmost column of the overflow block. We then fill all squares not part of a gadget nor recorded as empty with white pawns and place side rails on all boundary edges except as described in the move-wasting and reward gadgets. The anchor is on the black king as part of the initial state of the move-wasting gadget.

4.7 Analysis

Our analysis of gadget behavior in the preceding sections constrains the white king's pushes under the assumption that there are a specific number of empty spaces (often 0) in a particular row or column on a side of the gadget. We have already discharged the assumptions regarding the rows associated with literals by our layout of the connection and overflow blocks. For

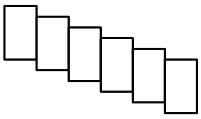


Figure 30 The layout of clause gadgets in the clause gadget block.

every other gadget except the variable gadgets, none of the constrained rows or columns intersects with another gadget, so the constraints on the edges are implied by the dense sea of white pawns outside the gadgets. For the variable gadgets, we assumed that pushing down in the second-to-left column of a variable gadget is not possible, but that column contains the previous variable gadget's rightmost column. We discharge this assumption by noting that in the final state of each variable gadget (after the white king has left the gadget), the rightmost column of that gadget is filled with white pawns, so pushing down in that column is indeed not possible.

Thus the white king must traverse the variable gadgets, setting the value of each variable, then traverse through the bridge gadget to the clause gadgets, where at least one empty space must remain in each checked column for the king to reach the reward gadget. If the choices made while traversing the variable gadgets results in filling all of the empty spaces in a checked column (i.e., the clause is false under the corresponding truth assignment), then White can only push by using a move outside the move-wasting gadget and Black wins on the next turn. If the white king successfully traverses every clause gadget (i.e., every clause is true under the truth assignment), then White wins when the white king pushes the black pawn off the board in the reward gadget. Thus White has a winning strategy for this Push Fight game state if and only if the input quantified 3-CNF formula is true.

References

1 Jeffrey Bosboom, Erik D. Demaine, and Mikhail Rudoy. Computational Complexity of Generalized Push Fight. arxiv:1803.03708, 2018. `https://arxiv.org/abs/1803.03708`.

2 Erik D. Demaine, Martin L. Demaine, and David Eppstein. Phutball endgames are NP-hard. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 351–360. Cambridge University Press, 2002.

3 Aviezri S. Fraenkel, M. R. Garey, David S. Johnson, T. Schaefer, and Yaacov Yesha. The complexity of checkers on an N * N board - preliminary report. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 55–64. IEEE Computer Society, 1978. `doi:10.1109/SFCS.1978.36`.

4 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. `doi:10.1016/0097-3165(81)90016-9`.

5 M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is *NP*-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977. `doi:DOI:10.1137/0132071`.

6 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.

7 Michael Hoffmann. Motion planning amidst movable square blocks: Push-* is NP-hard. In *Proceedings of the 12th Canadian Conference on Computational Geometry*, pages 205–210, 2000.

8 Jerry Holkins. Exposition. `https://www.penny-arcade.com/news/post/2015/12/14/exposition`, 2015.

9 Ben Kuchera. Push Fight is the best board game you've never heard of. https://web.archive.org/web/20131211190946/http://penny-arcade.com/report/article/push-fight-is-the-best-board-game-youve-never-heard-of, 2012.

10 Brett Picotte. Push Fight game. http://pushfightgame.com/, 2016. Accessed: 2017-06-22.

11 J. M. Robson. N by N checkers is exptime complete. *SIAM J. Comput.*, 13(2):252–267, 1984. doi:10.1137/0213018.

12 L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 1–9, 1973. URL: https://dl.acm.org/citation.cfm?id=804029.

SUPERSET: A (Super)Natural Variant of the Card Game SET

Fábio Botler
Universidad de Valparaíso, Valparaíso, Chile
fbotler@dii.uchile.cl

Andrés Cristi
Universidad de Chile, Santiago, Chile
andres.cristi@ing.uchile.cl

Ruben Hoeksma
Universität Bremen, Bremen, Germany
hoeksma@uni-bremen.de

Kevin Schewior
Universidad de Chile, Santiago, Chile
kschewior@gmail.com

Andreas Tönnis
Universidad de Chile, Santiago, Chile
atoennis@uni-bonn.de

Abstract

We consider Superset, a lesser-known yet interesting variant of the famous card game set. Here, players look for Supersets instead of Sets, that is, the symmetric difference of two Sets that intersect in exactly one card. In this paper, we pose questions that have been previously posed for set and provide answers to them; we also show relations between Set and Superset.

For the regular Set deck, which can be identified with $\mathbb{F}_3^4$, we give a proof for the fact that the maximum number of cards that can be on the table without having a Superset is 9. This solves an open question posed by McMahon et al. in 2016. For the deck corresponding to $\mathbb{F}_3^d$, we show that this number is $\Omega(1.442^d)$ and $O(1.733^d)$. We also compute probabilities of the presence of a superset in a collection of cards drawn uniformly at random. Finally, we consider the computational complexity of deciding whether a multi-value version of Set or Superset is contained in a given set of cards, and show an FPT-reduction from the problem for set to that for Superset, implying W[1]-hardness of the problem for Superset.

2012 ACM Subject Classification Mathematics of computing → Combinatoric problems, Theory of computation → Problems, reductions and completeness, Theory of computation → Fixed parameter tractability

Keywords and phrases SET, SUPERSET, card game, cap set, affine geometry, computational complexity

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.12

Funding This research was supported by Millenium Nucleus Information and Coordination in Networks (ICM/FIC RC 130003). Moreover, the first author was supported by CONICYT/FONDECYT/POSTDOCTORADO 3170878, and the last two authors acknowledge support from CONICYT grant PCI PII 20150140.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 12; pp. 12:1–12:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

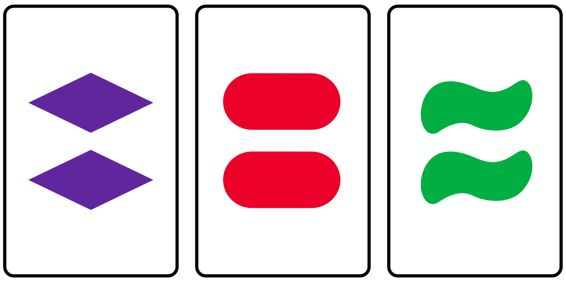


Figure 1 A SET is a collection of three cards that, in each of the properties, are either identical or distinct.

1 Introduction

The famous [1, 20] game SET[1] [5, 28] is played with cards that have four attributes each: The number, type, color, and shading of displayed shape(s). Each of these attributes can take three values, and each of the possible $3^4 = 81$ combinations of these values is contained exactly once as a card in the deck. A SET is a collection of three cards that, in each of the properties, are either identical or distinct (see Fig. 1). Among the cards that are laid out on the table, all players have to simultaneously find SETs as fast as possible. While possibly not evident from this description, we can assure you that the game is fun, even for a wider audience [31], including cats [24, Figure I.5].

However, as players get better and faster, the game becomes quite fidgety and arguably less fun. One straightforward way of making the game more difficult and thus slowing it down is adding more properties to the cards. Unfortunately, this creates decks increasing exponentially in size and possibly odor [14]. Other variants have been proposed [6, 14, 23, 24], but full-contact SET [6] seems only remotely related to mathematics, and projective SET [14] requires a completely different deck and seems incompatible with our gerontocracy [14].

Instead, we started playing a variant that is sufficiently more difficult, shows a rich mathematical structure, and can be played with the same, typically odor-free, deck of cards as SET: It is easy to see that for any pair of cards there exists exactly one *missing* card that completes the pair to a SET. Any single card, however, serves as missing card for (actually 40) different pairs. In the variant of SET considered here, the four cards from two pairs with the same missing card form the object (see Fig. 2) that the players look for instead of SETs. Until recently, when it was published in a book [24], this variant seems to have been spread mostly by word of mouth (as it was to one of the authors [7]), and appeared under different names on the Internet [6, 11, 13, 23, 32].

To continue the (admittedly short) tradition of overloading[2] mathematical terms, we choose to call the new object and the emerging variant of the game SUPERSET. We consider natural questions regarding SUPERSET: How many cards can be on the table without having a SUPERSET? More generally, what is the probability of having a SUPERSET in a collection of cards of a certain size chosen uniformly at random? What is the computational complexity of finding a SUPERSET? Are there any further connections between SUPERSET and SET?

[1] SET is a registered trademark of Cannei, LLC. The distinctive SET symbols and cards are copyrights of Cannei, LLC. All rights reserved. Used with permission from Set Enterprises, Inc.

[2] Since we typeset the two central objects of this paper as SET and SUPERSET, at least in written language we do not overload the corresponding mathematical terms. We however avoid using these mathematical terms (and thereby hopefully confusion) in this paper.

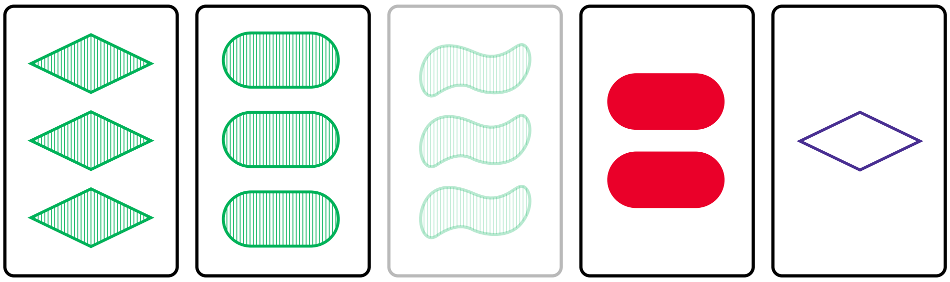


Figure 2 A SUPERSET is defined to be the symmetric difference of two SETs that intersect in exactly one card. Here, cards 1–3 and 3–5 form the two SETs; all but the third card form the SUPERSET.

These questions have, in fact, been considered for SET already, which is partly due to the applications of its study to affine spaces [14, 24], Fourier analysis [4], and error-correcting codes [19]. Clearly, the study of SUPERSET has the corresponding superapplications.

Related work. We briefly survey results related to SET. For a very accessible and lovely introduction to the mathematics of SET, we refer to McMahon et al. [24]. An also very well-written and at the time fairly comprehensive survey for mathematically more versed readers was published by Davis and Maclagan in 2003 [14].

When playing SET, one typically deals 12 cards and then looks for SETs among them. Sometimes, however, there turns out to be no SET, naturally leading to the question: How many cards have to be dealt to *guarantee* that there is a SET on the table? This question was in fact answered *before* the invention of the game SET in 1974, which is due to the following connection: One can naturally identify the deck of cards with the vector space $\mathbb{F}_3^4$ (by identifying the components with the properties) and then SETs in the deck of cards simply correspond to lines in $\mathbb{F}_3^4$. So the above question is equivalent to asking what is the maximum size of a *cap*, that is, a line-free collection of elements, in $\mathbb{F}_3^4$. This number was settled to be 20 in 1971 [26]. The most elegant proof known to date is based on counting so-called marked hyperplanes in two different ways, making use of the symmetries of the vector space [14].

It is natural to ask the same question for $\mathbb{F}_3^d$ with different d, which translates to a restricted or extended deck of cards. While this question for $d < 4$ can be easily answered using the same techniques as for $d = 4$ [14], only in 2002 did two breakthrough papers [4, 15] settle the maximum cap size for $d = 5$ to be 45 by relating the problem to the Fourier transform. For $d = 6$, the maximum cap size is 112 [27], as shown by the techniques similar to those used for $d \leq 4$ [14] along with a computer search, but the same paper claims that the Fourier-transform techniques could be used instead. Interestingly, at least up to $d = 6$, all maximum caps are from the same affine equivalence class, i.e., between any two of them there is an affine transformation that maps one to the other [15, 18, 24, 27].

Finding maximum supercaps for increasingly larger fixed integers d will probably keep (parts of) humanity busy for a while, but yet more forward-looking works have considered the asymptotic behavior of the maximum cap size as $d \to \infty$. While $\{0, 1\}^d$ is easily seen to be a cap of size 2^d, more sophisticated product constructions [8, 16] yield caps of size $\Omega(2.217^d)$. On the upper-bound side, Fourier transforms yield $O(3^d/d)$ [25] and further far-from-trivial insights about the spectrum yield $O(3^d/d^{1+\varepsilon})$ for some $\varepsilon > 0$ [3]. However, truly improving (i.e., in terms of the base of the exponential) upon the trivial upper bound of $O(3^d)$ has been a famous open problem [29] until recently. Only in 2016, the so-called polynomial method [12] was utilized [17, 30] to show quite compactly that the maximum cap size is $O(2.756^d)$.

Table 1 Bounds on the maximum cap and supercap sizes.

dimension	1	2	3	4	5	$\cdots$	d
maximum cap size	2	4	9	20	45	$\cdots$	$\Omega(2.217^d) \cap O(2.756^d)$
maximum supercap size	3	4	6	9	14	$\cdots$	$\Omega(1.442^d) \cap O(1.733^d)$

One may also wonder what the probability of the presence of a SET is in the initial layout of cards or, more generally, in a k-element collection chosen uniformly at random from $\mathbb{F}_3^d$. This question has been answered exactly for small values of k and d [24] and has been considered computationally for arbitrary values of k and d [21, 24]. An overview of the vast amount of interesting probabilistic questions, for example, the one for the expected number of SETs, can be found in the book of MacMahon et al. [24].

Let us look at the problem of deciding whether a given set of cards has a SET. As this question is boring in terms of asymptotic running time for $\mathbb{F}_3^4$, we first consider $\mathbb{F}_3^d$ where d is part of the input. Saving all cards in a dictionary and then checking the dictionary for the missing card of each pair yields an $O(n^2)$-time randomized algorithm and an $O(n^2 \log n)$-time deterministic algorithm [9]. Despite obvious resemblance to the 3SUM problem (three elements $a, b, c \in \mathbb{F}_3^d$ form a SET iff $a+b+c = 0$), 3SUM-hardness has only been conjectured [9].

To still be able to write computational-complexity papers about the problem, $\mathbb{F}_v^d$ has been considered, where v is part of the input as well. Note that a SET for a larger number of values per property can be defined either through lines (line SETs, at least for v prime) or by asking for identical or distinct values in each component (combinatorial SETs), but both variants are *different* [14] (see Section 2). Indeed, defining SETs through lines only adds a factor of v in the running time (because all v elements on the line have to be checked), but the problem is NP-hard for combinatorial SETs, as shown by a reduction from a multi-dimensional matching problem [9]. This result has been subsequently improved to W[1]-hardness for parameter v [22].

Unfortunately, none of these results is super enough yet. Yet, to date SUPERSET has not been rigorously studied. There are only a few Internet sources: experiments showing that there is a collection of 9 cards for $d = 4$ that does not have a SUPERSET [32] and some providing estimates for the probabilities of the presence of a superset in random collections [13, 32].

Our contribution. Analogous to the study of caps, we initiate the (rigorous) study of *supercaps*, that is, collections of cards that do not contain SUPERSETs. The same as for caps, we are interested in the maximum size of supercaps, but our techniques are different. For $d = 2$, by simply counting the number of pairs within the supercap and then using the pigeonhole principle, one can easily show an upper bound of 4 on the size of any supercap. A bound that can be easily matched from below by hand. In three dimensions, the same upper-bound technique allows us to prove an upper bound of 7. Yet, constructed lower-bound examples imply a maximum size of 6. To prove an upper bound of 6, we develop a refined counting technique that is based on the following observation [2, 11, 24]. If two pairs of points are disjoint and their induced vectors are parallel, then they form a SUPERSET (see Lemma 1). In $\mathbb{F}_3^4$, which corresponds to the actual SET deck, we use the same counting technique and a relatively short case distinction to show an upper bound of 9, which is tight [2, 32] and thus solves an open problem [24, Question 8]. For $d = 5$, using the same techniques, we can narrow down the maximum supercap size to at most 16 and at least 14, but an exhaustive computer search shows that the maximum supercap size is indeed 14.

Regarding asymptotic results, we utilize the simple pigeonhole-principle technique to obtain a non-trivial upper bound of $O(3^{d/2}) \subset O(1.733^d)$ on the cardinality of any supercap. To obtain a non-trivial lower bound, we essentially analyze an algorithm that greedily adds elements to the supercap. The critical observation is that each card that we cannot add is excluded by a triple of cards with which it would form a SUPERSET. Counting the number of triples and the number of elements that each triple can exclude, we obtain a lower bound of $\Omega(3^{d/3}) \subset \Omega(1.442^d)$. We summarize our results on maximum supercaps along with those known for maximum caps in Table 1.

The preceding structural insights about supercaps suffice to compute the probability of the presence of the a SUPERSET in a k-element collection of elements from $\mathbb{F}_3^d$ when $k = 4$ and d is arbitrary (or, alternatively, the probability of the collection being a supercap). We also manage to compute the same probability for $d = 3$ and $k = 5, 6$, which is based on a complete characterization of the corresponding supercaps. Since this characterization is already fairly complicated, we do not compute the exact probabilities for $d = 4$ and $k > 5$ but experimentally determine estimates (10^7 samples for every k).

To consider the computational complexity of deciding whether a given collection of cards has a SUPERSET, we define a SUPERSET (more generally) to be the symmetric difference of two (line or combinatorial) SETs that intersect in exactly one element. We note that the polynomial-time algorithm known for deciding if a given collection of cards from $\mathbb{F}_v^d$ (for v possibly fixed) has a SET can be essentially generalized to the corresponding problem for SUPERSET: One iterates through the pairs. If at least $v - 3$ other elements of the emerging line are present, one then checks a dictionary for entries of the corresponding missing card(s) and, if there are none, one saves the missing card(s) there. We then establish a close relation between the two problems by providing an FPT reduction from the problem for combinatorial SET to the corresponding one for SUPERSET, so that the W[1]-hardness [9] carries over.

Overview of this paper. In Section 2, we give formal definitions and preliminary observations. Then, we provide bounds on the maximum supercap sizes for various d and the asymptotic case in Section 3. In Section 4, we use insights from the previous section and new structural properties to obtain probabilities for the presence of supersets in the random collections of cards. We obtain results on the computations complexity in Section 5 and conclude the paper in Section 6.

2 Preliminaries

Definitions. We begin with considering $\mathbb{F}_3^d$. A collection $S = \{a, b, c\} \subseteq \mathbb{F}_3^d$ is called a SET if $a + b + c = 0$. Further, a collection of elements $S = \{a, b, c, d\} \subset \mathbb{F}_3^d$ is called a SUPERSET if for some element $z \in \mathbb{F}_3^d$ and for some $x, y \in S$, both $\{x, y, z\}$ and $(S \setminus \{x, y\}) \cup \{z\}$ are SETs. We say that two pairs $\{a_1, b_1\}, \{a_2, b_2\} \subset \mathbb{F}_3^d$ are *parallel* if $a_2 - b_2 = r(a_1 - b_1)$ for some $r \in \mathbb{F}_3 \setminus \{0\}$. Note that if $S = \{a_1, b_1, a_2, b_2\}$ contains a SET, then S is (ironically) not a SUPERSET.

Moreover, a collection S of elements from $\mathbb{F}_3^d$ is a *cap* if no SET is contained in it; it is a *supercap* if no SUPERSET is contained in it. We will be looking at both maxim*um* and maxim*al* (super)caps: The first type of (super)cap has largest-possible size among all (super)caps; the addition of any card to the second type of (super)cap revokes its (super)cap property. Note that any maximum (super)cap is maximal.

To consider the complexity of determining of a given collection of cards contains a SET or SUPERSET, we define two different, yet equally natural, generalizations of a SET and a SUPERSET. For an element $a \in \mathbb{F}_v^d$, we denote with $a[i]$ the value of the i-th dimension of a.

Given a collection $S = \{c_1, \dots, c_v\} \subseteq \mathbb{F}_v^d$ of v elements, we say it is a *combinatorial* SET (or just SET when not stated differently) if for all dimensions $i \in \{1, \dots, d\}$, either $c_1[i] = \dots = c_v[i]$ or the values $c_1[i], \dots, c_v[i]$ are distinct. For prime v, we say that a collection $S \subseteq \mathbb{F}_v^d$ is a *line* SET if it is a line on $\mathbb{F}_v^d$. It is not hard to see that these two generalizations are equivalent only for $v \leq 3$.

We obtain two generalizations for a SUPERSET in straightforward manner from the generalizations of a SET. A collection $S \subseteq \mathbb{F}_v^d$ of $2(v-1)$ elements is a *combinatorial (line)* SUPERSET if there is an element $z \in \mathbb{F}_v^d$ and a partition $S = A \cup B$ such that $A \cup \{z\}$ and $B \cup \{z\}$ are both *combinatorial (line)* SETs.

Preliminary observations. The following observation is used frequently in the technical part of the paper. Given two elements $a, b \in \mathbb{F}_3^d$, there is a unique third element in $\mathbb{F}_3^d$, namely $-(a+b)$, that completes $\{a, b\}$ to a SET. More generally, for $a, b \in \mathbb{F}_v^d$ there are $v-2$ unique elements $x_1, x_2, \dots x_{v_2} \in \mathbb{F}_v^d$ such that $\{a, b, x_1, x_2, \dots x_{v_2}\}$ is a line SET, but there are various ways of completing $\{a, b\}$ to a combinatorial SET. Regarding SUPERSETs, consider any three elements $a, b, c \in \mathbb{F}_3^d$: If they form a SET, there is no element that completes them to a SUPERSET; if they do not form a SET, they can be extended to precisely the SUPERSETS $\{a, b, c, -(a+b)\}$, $\{a, b, c, -(a+c)\}$, and $\{a, b, c, -(b+c)\}$. The situation for SUPERSETs is a bit more complicated in $\mathbb{F}_v^d$ but irrelevant for this paper.

The following lemma contains the formal statement of a fairly well-known fact for those that have concerned themselves with SUPERSETs [2, 11, 24]. For completeness, we still provide a proof.

▶ **Lemma 1.** *A collection $S \subset \mathbb{F}_3^d$ with four distinct elements is a* SUPERSET *if and only if $\{x, y\}$ and $S \setminus \{x, y\}$ are parallel pairs for some $x, y \in S$.*

Proof. Let $S = \{a, b, c, d\}$ be as in the statement. Suppose, without loss of generality, that $\{a, c\}$ and $\{b, d\}$ are parallel pairs. In this case, $x_{a,b} = -(a+b)$ and $x_{c,d} = -(c+d)$ are the (unique) elements that complete the SETs $S_{a,b} = \{a, b, x_{a,b}\}$ and $S_{c,d} = \{x, d, x_{c,d}\}$. Now, since $\{a, c\}$ and $\{b, d\}$ are parallel, we can assume that $b - d = -(a-c)$ (otherwise $b - d = a - c$, and we switch a and c). Thus we have $-(c+d) = -(a+b)$, and hence $x_{a,b} = x_{c,d}$, which implies that S is a SUPERSET.

Now, suppose that S is a SUPERSET. We may assume, without loss of generality, that there is an element $z \in \mathbb{F}_3^d$ such that $\{a, b, z\}$ and $\{c, d, z\}$ are SETs. Thus, we have $a + b + z = c + d + z = 0$, and hence $a + b = c + d$, which implies $a - c = d - b$. Therefore $\{a, c\}$ and $\{b, d\}$ are parallel. ◀

3 Bounds for supercaps

In this section we exactly determine the maximum sizes of supercaps of $\mathbb{F}_3^2$, $\mathbb{F}_3^3$, and $\mathbb{F}_3^4$. We also prove non-trivial upper and lower bounds on the asymptotic behavior of the maximum supercap size as $d \to \infty$.

3.1 Bounds for small d

In this subsection we present some auxiliary structural results along with the exact maximum sizes of supercaps of $\mathbb{F}_3^d$, for $d = 2, 3, 4$.

▶ **Proposition 2.** *A collection of four elements of $\mathbb{F}_3^2$ is a supercap if and only if it contains a* SET.

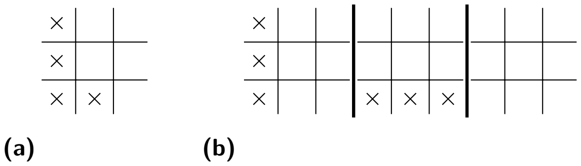


Figure 3 Maximum supercaps in $\mathbb{F}_3^2$ and $\mathbb{F}_3^3$.

Proof. Let S be a collection with precisely four distinct elements of $\mathbb{F}_3^2$. For every pair of elements $a, b \in S$, there is a (unique) third element $x_{ab} \in \mathbb{F}_3^2$ such that $\{a, b, x_{ab}\}$ is a SET. If S does not contain a SET, then $x_{ab} \notin S$, for every pair of elements of S. Since there are precisely 6 such pairs, and $|\mathbb{F}_3^2 - S| = 3^3 - 4 = 5$, there must be two different pairs, say $\{a, b\}$ and $\{c, d\}$, such that $x_{ab} = x_{cd}$. Therefore, S contains a SUPERSET. Now, suppose that S contains a SET, say $\{a, b, c\}$, and another element d. If S is a SUPERSET, then we may suppose that there is an element w in $\mathbb{F}_3^2$ such that, without loss of generality, $\{a, b, w\}$ and $\{c, d, w\}$ are SETs. Since there is a unique w such that $\{a, b, w\}$ is a SET, we have $w = c$, which implies that $\{c, d, w\} = \{c, d\}$ is not a SET. ◀

This proposition immediately implies the lower-bound part of the following theorem.

▶ **Theorem 3.** *A maximum supercap in $\mathbb{F}_3^2$ has four elements.*

Proof. By Proposition 2, there exists a supercap of size 4 in $\mathbb{F}_3^2$. We illustrate one in Figure 3a.

We now prove that any collection S of elements of $\mathbb{F}_3^2$ of size 5 contains a SUPERSET. First, note that, if S contains two SETS S_1 and S_2, they need to intersect, because S has only size 5. Since S_1 and S_2 are non-identical, they intersect exactly in one element w, so $(S_1 \cup S_2) \setminus \{w\}$ is a SUPERSET. Thus, if S does not contain a SUPERSET, then S contains at most one SET. If S contains a SET, say P, then let x be an element of P, otherwise, let x be any element of S. Now, note that $S \setminus \{x\}$ contains four elements, and no SET. Therefore, by Proposition 2, $S \setminus \{x\}$ is a SUPERSET. ◀

Next, note that if $\varphi \colon \mathbb{F}_3^d \to \mathbb{F}_3^d$ is an invertible affine transformation and $S \subset \mathbb{F}_3^d$, then $\varphi(S)$ is a SET (resp. SUPERSET) if and only if S is a SET (resp. SUPERSET), because φ preserves addition. The following result implies a lower bound for the size of a maximum supercap of $\mathbb{F}_3^3$.

▶ **Proposition 4.** *If S is a collection of elements of $\mathbb{F}_3^3$ consisting of two skew (disjoint non-parallel)* SET*s, then S is a supercap.*

Proof. Let S be as in the statement. Since these SETs are skew, their two direction vectors and an arbitrary vector connecting them are linearly independent. So we can construct an invertible linear transformation that maps these vectors into $v_1 = (1, 0, 0)$, $v_2 = (0, 1, 0)$, and $(0, 0, 1)$, respectively. We can further determine a translation such that the emerging invertible affine transformation φ maps the SETs into $P_1 = \{iv_1 : i \in \mathbb{F}_3\}$ and $P_2 = \{(0, 0, 1) + jv_2 : j \in \mathbb{F}_3\}$. Therefore, the element $(-i, -j, 2)$ is the unique element that forms a SET with $iv_1 \in P_1$ and $(0, 0, 1) + jv_2 \in P_2$. Since there are precisely nine pairs consisting of a vertex of P_1 and a vertex of P_2, no element in $\{(-i, -j, 2) : i, j \in \mathbb{F}_3\}$ may complete to two different such pairs to SETs. This implies that $P_1 \cup P_2$ is a supercap of $\mathbb{F}_3^3$, so $S = \{\varphi^{-1}(s) : s \in P_1 \cup P_2\}$ is as well. ◀

Let S be a collection of elements of $\mathbb{F}_3^d$, for a fixed integer d. Each pair $a, b \in S$ defines a direction $a - b$. We say that S *generates* a vector $v \in \mathbb{F}_3^d$ if there is a pair $a, b \in S$ such that $v = a - b$. In this case, we also say that v is generated by the pair $\{a, b\}$. By Lemma 1, if there are distinct $a, b, c, d \in S$ such that $a - b$ is parallel to $c - d$, then S contains a SUPERSET. So, to obtain an upper bound on the size of a supercap S, one can compare the number of parallel vectors that S generates and the number of equivalence classes of parallel vectors in the entire space. The following lemma formalizes this idea and will be used for the next upper-bound proofs.

▶ **Lemma 5.** *Let S be a supercap in $\mathbb{F}_3^d$ with s elements and r SETs. Then*

$$r \geq \left\lceil \frac{s^2 - s - 3^d + 1}{4} \right\rceil .$$

Proof. Let S, s, and r be as in the statement. The number of pairs of elements of S is $\binom{s}{2}$. Note that each SET in S generates exactly 3 parallel vectors without creating a SUPERSET, but there are r SETs in S. Only considering one vector per SET, S still generates $\binom{s}{2} - 2r$ vectors. Note that, since we only consider one vector per SET and all SETs are pairwise disjoint (otherwise there would be a SUPERSET), any two pairs that generate parallel vectors need to be disjoint. So, by Lemma 1, any two of the $\binom{s}{2} - 2r$ must not be parallel. On the other hand, we give an upper bound on the equivalence classes of parallel vectors in $\mathbb{F}_3^d$ by counting the SETs that go through the origin $0 = (0, \ldots, 0)$: Since, for any other $a \in \mathbb{F}_3^d$, there is a unique SET containing 0 and a, and each SET has size 3, there are exactly $(3^d - 1)/2$ such SETs. Thus

$$\binom{s}{2} - 2r \leq \frac{3^d - 1}{2},$$

and the result follows by solving for r. ◀

We now apply Proposition 4 and Lemma 5 to get the following theorem.

▶ **Theorem 6.** *A maximum supercap in $\mathbb{F}_3^3$ has six elements.*

Proof. By Proposition 2, there exists a supercap of size 6 in $\mathbb{F}_3^3$. We illustrate one in Figure 3b.

Now assume that $S \subset \mathbb{F}_3^3$ is a supercap of size 7. By Lemma 5, the number of SETs in S is at least 4 but there are at most two non-intersecting SETs in S, a contradiction. ◀

The proof of the next theorem goes one step further. In this case, the application of Lemma 5 does not directly imply a tight upper bound.

▶ **Theorem 7.** *A maximum supercap in $\mathbb{F}_3^4$ has nine elements.*

Proof. A supercap of $\mathbb{F}_3^4$ of size 9 was previously known [32, 2] and is illustrated in Figure 4.

For the upper bound, let S be a supercap with precisely ten different elements of $\mathbb{F}_3^4$. By Lemma 5, the number of SETs in S is at least $\lceil (100 - 10 - 81 + 1)/4 \rceil = 3$. Analogously to the proof of Proposition 4, by applying a certain invertible affine transformation, we can suppose that two of these SETs are $P_1 = \{kv_1 : k \in \mathbb{F}_3\}$ and $P_2 = \{(0, 0, 1, 0) + kv_2 : k \in \mathbb{F}_3\}$, where $v_1 = (1, 0, 0, 0)$ and $v_2 = (0, 1, 0, 0)$.

Now, let $P_3 = \{(a, b, c, d) + kv_3 : k \in \mathbb{F}_3\}$. We first show that $v_3 = (e_1, e_2, 0, 0)$, where $e_1, e_2 \in \mathbb{F}_3 \setminus \{0\}$. Let $v_3 = (e_1, e_2, e_3, e_4)$. If $e_4 \neq 0$, then P_3 has an element q of the form $(x, y, z, 0)$. Thus, the restriction S' of S to the affine subspace $F_0 = \{(x, y, z, 0) : x, y, z \in$

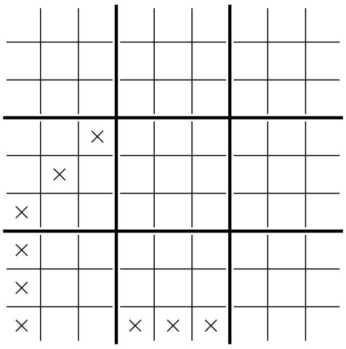


Figure 4 Maximum supercap in $\mathbb{F}_3^4$.

$\mathbb{F}_3\}$ contains the seven elements $P_1 \cup P_2 \cup \{q\}$. Since F_0 is isomorphic to $\mathbb{F}_3^3$, by Theorem 6, S' contains a SUPERSET, a contradiction. In fact, S may not contain any element of the form $(\cdot,\cdot,\cdot,0)$ different from the elements in $P_1 \cup P_2$. Now, suppose that $e_3 \neq 0$. Then P_3 contains elements q_1 and q_2 of the form $(\cdot,\cdot,0,d)$ and $(\cdot,\cdot,2,d)$, with $d \neq 0$. Let $A_1 = (0,0,0,0)$ and $A_2 = (0,0,1,0)$, and for $i = 1,2$, consider the SETs P_1' and P_2' defined by

$$\begin{aligned} P_i' &= \{r \colon s + q_i + r = 0, s \in P_i\} \\ &= \{-(s+q_i) \colon s \in P_i\} \\ &= \{-(A_i + kv_i + q_i) \colon k \in \mathbb{F}_3\} \\ &= \{-(A_i + q_i) + 2kv_i \colon k \in \mathbb{F}_3\} \\ &= \{-(A_i + q_i) + kv_i \colon k \in \mathbb{F}_3\}. \end{aligned}$$

Note that hence P_i' is parallel to P_i, for $i = 1,2$. Moreover, since the vertices of P_1 and P_2 are of the form $(\cdot,\cdot,\cdot,0)$, the vertices of P_1' and P_2' are of the form $(\cdot,\cdot,\cdot,3-d)$. Also, since the vertices of P_1 and q_1 are of the form $(\cdot,\cdot,0,\cdot)$, the vertices of P_1' are of the form $(\cdot,\cdot,0,\cdot)$; and since the vertices of P_2 are of the form $(\cdot,\cdot,1,\cdot)$, and q_2 is of the form $(\cdot,\cdot,2,\cdot)$, the vertices of P_2' are of the form $(\cdot,\cdot,0,\cdot)$. We conclude that P_1' and P_2' belong to the 2-dimensional affine subspace $F_{0,3-d} = \{(x,y,0,3-d) \colon x,y \in \mathbb{F}_3\}$. Note that P_1' and P_2' are not disjoint because they are parallel, respectively, to P_1 and P_2. Thus, there is a vertex q^* in $P_1' \cap P_2'$ such that $s_1 + q_1 + q^* = s_2 + q_2 + q^* = 0$, for some $s_1 \in P_1$ and $s_2 \in P_2$. Therefore, S contains a SUPERSET, a contradiction. Now, if $e_1 = 0$ or $e_2 = 0$, then P_3 is parallel to either P_1 or P_2, a contradiction.

Now, let $F_{i,j} = \{(x,y,i,j) \colon x,y \in \mathbb{F}_3\}$, for $i,j \in \mathbb{F}_3$. Since $v_3 = (e_1,e_2,0,0)$, the SET P_3 must be contained in some affine subspace F_{i^*,j^*}. Further, we must have $j \neq 0$ since otherwise there are again seven elements of the form $(\cdot,\cdot,\cdot,0)$. Assume, by adapting the invertible affine transformation accordingly, that $i^* = 0$ and $j^* = 1$. Analogously to the proof of Proposition 4, each element of $F_{2,0}$ is the unique element that forms a SET with a vertex of P_1 and of P_2; each element of $F_{0,2}$ is the unique element that forms a SET with a vertex of P_1 and of P_3; and each element of $F_{2,2}$ is the unique element that forms a SET with a vertex of P_2 and of P_3; Recall that S has ten elements, i.e., there is an element q in $S \setminus (P_1 \cup P_2 \cup P_3)$. Note that the collections $F_{0,0} \cup F_{1,0} \cup F_{2,0}$, $F_{0,0} \cup F_{0,1} \cup F_{0,2}$, and $F_{1,0} \cup F_{0,1} \cup F_{2,2}$ are affine subspaces isomorphic to $\mathbb{F}_3^3$, and, by Theorem 6, q may not belong to any of these collections. Now, suppose that $q \in F_{1,1}$. For each $q_1 \in P_1 \subset F_{0,0}$, there is a vertex $q_{2,2} \in F_{2,2}$ such that $q_1 + q + q_{2,2} = 0$. As noted above, each element of $F_{2,2}$ forms a SET with a vertex of P_2 and of P_3, say q_2, q_3. Therefore $\{q_1, q_2, q_3, q\}$ is a SUPERSET. Analogously, if q belongs

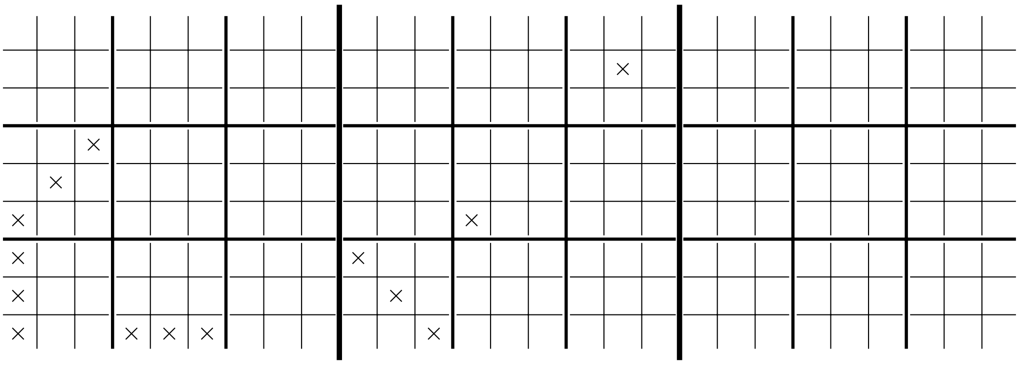


Figure 5 Maximum supercap in $\mathbb{F}_3^5$.

to $F_{2,1}$ or $F_{1,2}$, we can find elements $q_1 \in P_1$, $q_2 \in P_2$, and $q_3 \in P_3$ such that $\{q_1, q_2, q_3, q\}$ is a SUPERSET. This contradicts the assumption that S is a supercap and concludes the proof. ◀

We conclude the subsection with a discussion of open questions, preliminary answers, and fascinating phenomena. In $\mathbb{F}_3^5$, the largest supercap we can construct has size 14 (see Figure 5), but Lemma 5 only shows an upper bound of 16 on the size of supercaps. While an exhaustive computer search shows that 14 is indeed the right answer, we still believe in (super) elegant proofs. Indeed, looking at the lower bounds in this section, one may notice that, interestingly, all of them contain the maximum number of SETs possible. Also, Lemma 5 gives the loosest upper bound when the maximum number of SETs are present. So one may conjecture that, for each d, a maximum supercap is attained by a union of SETs and at most 2 additional points.

On the other hand, it has been observed [24] that the maximum supercap in four dimensions can be partitioned into ten pairs each of which is completed to a SET by the same element. So it seems that caps and supercaps are somewhat complementary in that maximum supercaps are far from being caps and vice versa. Unfortunately, however, we need to push back on this line of thought a bit. As we will see in Section 4, already in $\mathbb{F}_3^3$ there are maximum supercaps with only one (instead of two) SETs. Also, there is a maximum supercap in $\mathbb{F}_3^4$ that does not have a SET at all. On a slightly different matter, this situation is somewhat different from the one for caps in that, for any $d \in \{1, \ldots, 6\}$, there is an affine transormation that takes any maximum cap to any other maximum cap [15, 18, 24, 27].

As all of the phenomena pointed at here may simply be due to the (small) dimensions we are working with, we now look at the asymptotic case.

3.2 Asymptotic supercaps

In this section we present upper and lower bounds for the size of a maximum supercap in $\mathbb{F}_3^d$. The next theorem gives the upper bound; its proof is analogous to the proof of Theorems 6 and to some of the cases of the proof of Theorem 7.

▶ **Theorem 8.** *A maximum supercap in $\mathbb{F}_3^d$ has less than $2 \cdot 3^{\frac{d}{2}}$ elements.*

Proof. It is sufficient to prove for $d \geq 2$ that, if a collection $S \subseteq \mathbb{F}_3^d$ has size $s = 2 \cdot 3^{\frac{d}{2}}$, then it contains a SUPERSET. Let S be such a collection, and suppose that S is a supercap. By

Lemma 5, the number of non-intersecting SETs in S is at least

$$\left\lceil \frac{4 \cdot 3^d - 2 \cdot 3^{d/2} - 3^d + 1}{4} \right\rceil .$$

On the other hand, there are at most $\lfloor s/3 \rfloor$ non-intersecting SETs in S. Thus, we have

$$\left\lceil \frac{4 \cdot 3^d - 2 \cdot 3^{d/2} - 3^d + 1}{4} \right\rceil \leq \left\lfloor \frac{2 \cdot 3^{d/2}}{3} \right\rfloor .$$

Note that for any $d \geq 2$ we have

$$\frac{4 \cdot 3^d - 2 \cdot 3^{d/2} - 3^d + 1}{4} > \frac{3 \cdot 3^d - 2 \cdot 3^{d/2}}{4} > \frac{3^d}{4} > \frac{2 \cdot 3^{d/2}}{3}.$$

Therefore, we have

$$\left\lceil \frac{4 \cdot 3^d - 2 \cdot 3^{d/2} - 3^d + 1}{4} \right\rceil \geq \frac{4 \cdot 3^d - 2 \cdot 3^{d/2} - 3^d + 1}{4} > \frac{2 \cdot 3^{d/2}}{3} \geq \left\lfloor \frac{2 \cdot 3^{d/2}}{3} \right\rfloor ,$$

a contradiction. Therefore, for any SUPERSET S in $\mathbb{F}_3^d$ we have $|S| < 2 \cdot 3^{d/2}$. ◀

The next theorem gives a lower bound for the size of a maximum supercap in $\mathbb{F}_3^d$.

▶ **Theorem 9.** *A maximum supercap in $\mathbb{F}_3^d$ has more than $3^{\frac{d}{3}}$ elements.*

Proof. We prove that every maximal supercap has size at least $3^{\frac{d}{3}}$. Given a supercap S in $\mathbb{F}_3^d$, let $\bar{S}$ be the collection of elements v of $\mathbb{F}_3^d - S$ for which there is at least one triple T in S such that $T \cup \{v\}$ is a SUPERSET. Note that if $x \in \mathbb{F}_3^d \setminus (S \cup \bar{S})$, then $S \cup \{x\}$ is a supercap. Thus, if S is a maximal supercap, then $S \cup \bar{S} = \mathbb{F}_3^d$. Given $a, b \in \mathbb{F}_3^d$, let x_{ab} be the (unique) element of $\mathbb{F}_3^d$ such that $\{a, b, x_{ab}\}$ is a SET; and given a triple $\{a, b, c\} \subset \mathbb{F}_3^d$ that is not a SET, let y_c be the (unique) element of $\mathbb{F}_3^d$ such that $\{c, x_{ab}, y_c\}$ is a SET. Note that for every such triple $\{a, b, c\}$ in a supercap S, we have $y_a, y_b, y_c \in \bar{S}$. Moreover, if $\{a, b, c, y\}$ is a SUPERSET, then $y = y_z$ for some $z \in \{a, b, c\}$. Therefore, $|\bar{S}| \leq 3\binom{|S|}{3}$ for every supercap S in $\mathbb{F}_3^d$.

Now, suppose that S is a maximal supercap and that $|S| = s \leq 3^{\frac{d}{3}}$. Since S is maximal, we have $3^d = |\mathbb{F}_3^d| \leq |S| + |\bar{S}|$. Thus, we have

$$s^3 \leq 3^d \leq s + 3\binom{s}{3} .$$

Yet, $s^3 > s + 3\binom{s}{3}$ for all $s > 1$, contradicting our assumption, since a maximal supercap has at least three elements. We conclude that if S is maximal, then $|S| > 3^{\frac{d}{3}}$. ◀

4 Probabilities of the presence of a superset in random collections

In the section, we compute probabilities of k-element collections in $\mathbb{F}_3^d$ being supercaps. Using structural insights from Section 3, we get the following result, settling the question for $d = 2$.

▶ **Theorem 10.** *A collection of four elements drawn uniformly at random without replacement from $\mathbb{F}_3^d$ is a supercap with probability $\frac{3^d-5}{3^d-2}$.*

Proof. Let $S = \{a, b, c, d\}$ be a collection of four elements drawn uniformly at random without replacement from $\mathbb{F}_3^d$. Consider the four elements of S in (alphabetical) order. As noted earlier in Proposition 2, if S contains a SET, then it is a supercap. Without loss of generality, fix the first two elements $\{a, b\}$. The third element, c, completes a SET with probability $\frac{1}{3^d-2}$, since exactly one of the remaining $3^d - 2$ elements from $\mathbb{F}_3^d$ forms a SET with $\{a, b\}$. If $\{a, b, c\}$ does not form a SET, then there are three pairs, $\{a, b\}$, $\{b, c\}$, and $\{a, c\}$ that define different elements with which they form a SET. Thus, there are exactly three elements that can complement $\{a, b, c\}$ into a SUPERSET. Therefore

$$\Pr(S \text{ is a supercap}) = \frac{1}{3^d-2} + \frac{3^d-3}{3^d-2} \cdot \frac{3^d-6}{3^d-3} = \frac{3^d-5}{3^d-2}. \qquad \blacktriangleleft$$

For $d = 3$, we require new structural insights.

▶ **Proposition 11.** *Let S be a collection with five elements in $\mathbb{F}_3^3$. Then S is a supercap if and only if either*

- *S contains a SET P and the elements not in P form a pair skew with P*
- *or S does not contain a SET and there is no hyperplane in $\mathbb{F}_3^3$ containing at least four elements of S.*

Proof. Let S be a collection of five elements with a SET P. It is clear that if $S \setminus P$ forms a pair not skew with P, then S contains a SUPERSET either by the intersection of P and the SET containing $S \setminus P$, or by P being parallel to $S \setminus P$. Now, suppose that $S \setminus P = \{a, b\}$ is skew with P. It is not hard to check that a SUPERSET admits three partitions into two pairs, and one of these partitions consists of two pairs that miss the same third element to complete a SET; and the other two of these partitions consist of two parallel pairs (see Lemma 1). Since $\{a, b\}$ is skew with P, for any $c, d \in P$, the pair $(\{a, b\}, \{c, d\})$ forms a partition of $\{a, b, c, d\}$ that does not consist of two pairs with a common missing third element, and does not consist of two parallel pairs. Thus, $\{a, b, c, d\}$ is not a SUPERSET. Suppose now that S does not contain a SET. Since a hyperplane in $\mathbb{F}_3^3$ is isomorphic to $\mathbb{F}_3^2$ and every SUPERSET is in a hyperplane, by proposition 2, S contains a SUPERSET if and only if there is a set of four vertices contained in a hyperplane. ◀

▶ **Proposition 12.** *Let S be a collection of six elements in $\mathbb{F}_3^3$. Then S is a supercap if and only if either*

- *S contains two SETs that are skew, or*
- *there are three parallel planes H_1, H_2, H_3 that partition $\mathbb{F}_3^3$ such that $S \cap H_1 = \{a, b, c, d\}$, $S \cap H_2 = \{e\}$, and $S \cap H_3 = \{f\}$ where $\{a, b, c\} = P$ is a SET and $f \notin \{-(x + e) : x \in S \cap H_1\} \cup \{x + d - e : x \in P\}$.*

Proof. Let S be as in the statement and first suppose that S is a supercap. First note that Lemma 5 implies that S must contain at least one SET. If S contains two SETs, they must be skew, because otherwise the two SETs (and thus at least five elements) are within a two-dimensional affine subspace, contradicting Theorem 3. If S contains precisely one SET $\{a, b, c\} = P$, we can find a plane H_1 that contains P and any fourth element $d \in S$. Note that H_1 may not contain any other element of S, because this would be a contradiction to Theorem 3 again. Next, consider the case that there is a plane H' parallel to H_2 such that $|S \cap H'| = 2$. But this is not possible: Since H_1 and H' generate 5 vectors parallel to H_1 and, among the vectors parallel to H_1, there are only 4 equivalence classes of parallel vectors, we get a contradiction to Lemma 1. Hence, there are planes H_2 and H_3 parallel to H_1 with $S \cap H_2 = \{e\}$ and $S \cap H_3 = \{f\}$ for some $e, f \in \mathbb{F}_3^3$. Now, since S contains only the set P, $x + e + f \neq 0$ for all $x \in H_1$, so $f \notin \{-(x + e) : x \in S \cap H_1\}$. Similarly, since S is a supercap $x + d \neq e + f$ for all $x \in P$, so $f \notin \{x + d - e : x \in P\}$. Thus we are in the second situation.

If S contains two SETs that are skew, then Proposition 4 shows that S is a supercap. If the second condition is fulfilled, then let $a, \dots, f$ and H_1, H_2, H_3 be as in the statement. Note that H_1 does not contain a SUPERSET by Proposition 2. If for a SUPERSET $Q \subset S$, we have $|Q \cap H_1| = 3$, then, for any pair $x_1, x_2 \in H_1$, $x_1 + x_2 \in H_1$, but $x_3 + x_4 \notin H_1$ where $\{x_3, x_4\} = Q \setminus \{x_1, x_2\}$, a contradiction. So, if S contains a SUPERSET Q, then $Q = \{e, f, y_1, y_2\}$ for some $y_1, y_2 \in S \cap H_1$. But $y_i + e \in H_3$ while $y_i + f \in H_2$ for any $i \in \{1, 2\}$. If $d \notin Q$, then $-(y_1 + y_2) \in P$, but $f \neq -(x + e)$ for all $x \in P$ by the choice of f; a contradiction. So $Q = \{e, f, d, x\}$ for some $x \in P$. But then we must have $x + d = e + f$; a contradiction to the choice of f. ◀

Using these insights and counting the numbers of the corresponding objects yields the following theorem, which settles the central question of this section for $d = 3$.

▶ **Theorem 13.** *A collection of five (six) elements drawn uniformly at random without replacement from $\mathbb{F}_3^3$ is a supercap with probability $\frac{54}{115} \approx 46.96\%$ ($\frac{18}{253} \approx 7.11\%$).*

Proof. We count the number of supercaps of five elements using Proposition 11. The ones that contain a SET and a pair skew with it can be constructed as follows. Choose a SET, then any of the remaining $(3^d - 3)$ cards and finally any of the $(3^d - 9)$ cards that do not complete an intersecting SET or creates a parallel vector with the SET. Since the last pair is counted twice this way, the total number is

$$N^5_{\text{SET,skew}} = \frac{3^3(3^3-1)}{6}(3^3-3)(3^3-9) \cdot \frac{1}{2} = 25272$$

For the ones that do not contain a SET and in which no four elements are in a hyperplane, we count first the number of collections with a SET: pick first one of the $\frac{1}{3}\binom{3^3}{2}$ possible SETs, then pick any pair on the remaining cards. With this procedure we double count the collections composed by two intersecting SETs, so the total number of collections of five elements with a SET is

$$N^5_{\text{SET}} = \frac{1}{3}\binom{3^3}{2} \cdot \binom{3^3-3}{2} - \frac{1}{3}\binom{3^3}{2} \cdot \frac{1}{4}\left(3^3-3\right) \cdot 3 = 30186$$

We now compute the number of collections without a SET but with a hyperplane. It is clear that only one hyperplane contains four points of such a collection. Pick then the first four elements to be the ones in the same hyper plane. There are 3^3 options for the first, $(3^3 - 1)$ for the second, $(3^3 - 3)$ for the third without forming a SET, and only 3 for the fourth so it lies in the same hyperplane and does not form a SET. We divide by the number of permutations of 4 elements to avoid multiple counting. For the fifth element the only condition is that it is outside the hyperplane, so there are $3^{3-1} \cdot 2$ options. The number of such collections is then

$$N^5_{\text{!SET,HP4}} = \frac{1}{4!}3^3(3^3-1)(3^3-3) \cdot 3 \cdot (3^{3-1}2) = 37908$$

Now, the number of collections of five elements without a SET is $N^5_{\text{!SET}} = \binom{3^3}{5} - N^5_{\text{SET}} = 50544$, and the amount of collections without a SET and with no four elements in a hyperplane is $N^5_{\text{!SET,!HP4}} = N^5_{\text{!SET}} - N^5_{\text{!SET,HP4}} = 12636$. Finally, the number of supercaps of size five is $N^5_{\text{SET,skew}} + N^5_{\text{!SET,!HP4}} = 37908$, which divided by $\binom{3^3}{5}$ gives the probability that a random collection S of five elements in $\mathbb{F}_3^3$ is a supercap, so

$$\Pr(S \text{ is a supercap}) = \frac{37908}{80730} = \frac{54}{115} \approx 46.96\%.$$

Table 2 Probabilities (and estimates thereof) of a k-element collection from $\mathbb{F}_3^d$ being a supercap, expressed as percentages rounded to two decimals. We used a computer to estimate the probability where indicated by an asterisk (10^7 samples for each corresponding cell); the other probabilities are exact.

	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$
$d=2$	57.14%	0	0	0	0	0
$d=3$	88.00%	46.96%	7.11%	0	0	0
$d=4$	96.20%	81.68%*	52.08%*	19.25%*	2.34%*	0.01%*

We count the number of supercaps S of six elements using Proposition 12. Note that the number of two skew SETs is exactly a sixth the number of five cards formed by a SET and a pair that is skew with it, so

$$N^6_{2\text{SETs,skew}} = \frac{\binom{3^3}{2}}{3} \cdot (3^3-3) \cdot (3^3-9) \cdot \frac{1}{12} = 4212.$$

Now consider the second situation in Proposition 12, and let $a, \ldots, f$ and H_1, H_2, H_3 be as in the statement. First note that $f \notin \{-(x+e) : x \in S \cap H_1\}$ ensures that there is exactly one SET in S, so the two situations cannot happen simultaneously. We count the number of collections S that fall into this situation the following way: First fix any set $\{a,b,c\} = P$ and any fourth point d. There are 9 choices for e. Since $\{-(x+e) : x \in S \cap H_1\}$ and $\{x+d-e : x \in P\}$ are both contained in H_3 and are disjoint, there are 2 choices left for f. As we count each collection three times (any of the points outside the SET can be the fourth point), the total number of collections S that fall into the second situation is

$$N^6_{\text{Case }2} = \frac{\binom{3^3}{2}}{3} \cdot (3^3-3) \cdot 9 \cdot 2 \cdot \frac{1}{3} = 16848.$$

In total, we get

$$\Pr(S \text{ is a supercap}) = \frac{N^6_{2\text{SETs,skew}} + N^6_{\text{Case }2}}{\binom{3^3}{6}} = \frac{4212+16848}{296010} = \frac{18}{253} \approx 7.11\%.$$

This concludes the proof. ◀

Considering the name of this conference, we leave proving similar statements for $d=4$ to future work. To not disappoint the reader, we however provide probabilities that were determined experimentally with the computer. We summarize the results in Table 2.

5 Algorithms and complexity

Chaudhuri et al. [9] as well as Lampis and Mitsou [22] consider decision problem versions for SET and show complexity results for them. In these decision problems, we are given a collection of n elements from $\mathbb{F}_v^d$ and ask if the collection contains a SET[3]. In this section, we obtain similar results for decision problem versions of SUPERSET.

We define the problems COMBINATORIAL SUPERSET and LINE SUPERSET as follows: Given a collection of elements $C \subseteq \mathbb{F}_v^d$ for any $v, d > 0$, is there a combinatorial (line)

[3] Previously only these decision versions of Combinatorial SETs were considered [9, 22]. More restricted versions with given number of values (k-VALUE SET) or given number of dimensions (k-DIMENSIONAL SET) have also been considered by the same authors.

Superset $S \subseteq C$? We define k-Value Combinatorial (Line) Superset as the restricted versions where $v = k$ is fixed, and k-Dimensional Combinatorial (Line) Superset as the restricted versions where $d = k$ instead.

Note that, in order to get interesting complexity questions, the number of possible values v of each attribute needs to be variable, as we see from the following result.

▶ **Theorem 14.** *The problem k-Value Combinatorial Superset and k-Value Line Superset can be solved in $\tilde{O}(dkn^{k-1})$ time, for any given $k > 0$.*

Proof. Consider the algorithm that iteratively checks all $\binom{n}{k-1}$ subsets of size $k - 1$ and keeps an AVL tree [10] of missing k-th elements to complete a set, if such an element exists. Then checking if the ordered list contains any duplicates decides if there is a superset in the given collection of elements. This algorithm works independent of the considered type of superset (combinatorial or line) and runs in $\tilde{O}(dkn^{k-1})$ time. ◀

▶ **Theorem 15.** *The problem Line Superset can be solved in $\tilde{O}(dvn^2)$ time.*

Proof. Each pair of elements defines exactly one line, so it suffices to check for each pair if the collection contains $v - 1$ elements of the line. If so, the missing element is stored in an ordered list. ◀

Note that, by similar reasoning, Line Set can also be solved in $\tilde{O}(dvn^2)$ time.

▶ **Theorem 16.** *There is a $O(k^2n)$-time reduction from k-Dimensional Combinatorial Set to $(k+1)$-Dimensional Combinatorial Superset. Furthermore, it sends an instance on v values to an instance on $v + 1$ values.*

Proof. Let $S \in \mathbb{F}_v^k$ be an instance of k-Dimensional Combinatorial Set. That is, S is a collection of n elements in k dimensions, each with v possible values. Through the following procedure, we construct an instance of $k + 1$-Dimensional Combinatorial Superset consisting of collection of elements $S' \in \mathbb{F}_{v+1}^{k+1}$ of size at most $(v + 1) \cdot n$, in $k + 1$ dimensions, each with $v + 1$ possible values.

Create a copy S_0 of S on $k + 1$ dimensions, filling the $(k + 1)$-th dimension of every element with the value $v + 1$. Then, create $(k + 1)$-dimensional copies $S_1, \ldots, S_k$ of S, where the $(k + 1)$-th dimension of elements in S_i have value equal to the i-th dimension of that element elements. That is, for an element $c' \in S_i$ there is an element $c \in S$, such that $c' = (c[1], \ldots, c[k], c[i])$.

Now, we show that S contains a set in $\mathbb{F}_v^k$ if and only if $S' = \bigcup_{i=0}^k S_i$ contains a superset in $\mathbb{F}_{v+1}^{k+1}$ (note that the union might be non disjoint). Suppose S contains a set in $\mathbb{F}_v^k$, say A, and let $A_i \subseteq S_i$ be the corresponding copy of A for all $i \in \{0, \ldots, v\}$. Let $z \in \mathbb{F}_{v+1}^{k+1}$ be the element that for all $j \in \{1, \ldots, k\}$ has $z[j] = a_1[j]$, if $a_1[j] = a_2[j]$, and $z[j] = v+1$, otherwise, and $z[k + 1] = v + 1$. Then $A_0 \cup \{z\}$ is a set in $\mathbb{F}_{v+1}^{k+1}$. Since the elements $a_1, \ldots, a_v \in A$ are distinct and A is a set in $\mathbb{F}_v^k$, there must be at least one dimension $1 \leq j \leq k$ such that the values $a_1[j], \ldots, a_v[j]$ are all distinct. Then $A_j \cup \{z\}$ forms a set in $k + 1$ dimensions and $v + 1$ values, because the first k dimensions are the same as $A_0 \cup \{z\}$, and dimension $k + 1$ has the same values as dimension j, so the missing value in A_j is $v + 1$. Since by construction we have $A_0 \cap A_j = \emptyset$, we conclude that $A_0 \cup A_j$ is a superset in $\mathbb{F}_{v+1}^{k+1}$.

Next, let $A \cup B \subseteq S'$ be a superset in $\mathbb{F}_{v+1}^{k+1}$ such that $A \cap B = \emptyset$ and there is an element z such that $A \cup \{z\}$ and $B \cup \{z\}$ are sets in $\mathbb{F}_{v+1}^{k+1}$. This implies that $|A| = |B| = v$. Let $A = \{a_1, \ldots, a_v\}$ and let $a'_j \in S$ denote the projection of a_j onto its first k dimensions, which is the original of a_j in S. If all elements $a'_1, \ldots, a'_v$ are different, then they form a set in $\mathbb{F}_v^k$. Now, assume that there are two elements a_k and a_ℓ in A, such that $a'_k = a'_\ell$. Since $A \cup \{z\}$

is a SET, this implies that $a'_1 = a'_2 = \ldots = a'_v$. Moreover, the projection of z onto the first k dimensions is equal to a'_1. Therefore, if the same holds for B, then the projections of each of the elements of B onto the first k dimensions are all equal and these projections are also equal to a'_1. However, this is a contradiction, as S' contains at most $v+1$ copies of the same element in S (and $|A \cup B| = 2v$). Thus, without loss of generality, we can assume that the elements in A are copies of different elements in S and the projection of A onto the first k dimensions is a SET in S. ◀

Chaudhuri et al. [9] prove that k-Dimensional Combinatorial Set is NP-complete for $k \geq 3$, and Lampis and Mitsou [22] prove that Combinatorial Set parametrized by the number of values is W[1]-hard. These two results, together with Theorem 16, yield the following hardness results for SUPERSET.

▶ **Corollary 17.** *The problem k-Dimensional Combinatorial Superset for $k \geq 4$ and Combinatorial Superset are NP-complete.*

▶ **Corollary 18.** *The problem Combinatorial Superset parametrized by the number of values v is W[1]-hard.*

6 Conclusion

While it is plausible that we have exhausted (hopefully not gone beyond) the reader's tolerance of jokes including "super" in this paper, we believe that we have not done so to their curiosity regarding SUPERSET. In fact, while we have made progress on many natural questions in this paper, a few remain open: As for caps, the gaps for the maximum supercap size for larger fixed dimensions and its asymptotic behavior would be interesting to investigate. Also figuring out whether a subquadratic algorithm for deciding the presence of a SET or SUPERSET exists in $\mathbb{F}_3^d$ seems to be an interesting open problem.

Just like SET became too easy one day, we will eventually demand a variant of SET more difficult than SUPERSET. In fact, note that the term *powerset* is yet to be overloaded. For instance, a POWERSET could be the union of three (or more) pairs that are all completed to a SET by a same element [24] or, alternatively, the symmetric difference between two SUPERSETs that intersect in exactly one element.

References

1 Gary Antonick. The problem with SET. News Article. `https://www.nytimes.com/2016/08/22/crosswords/the-problem-with-set.html`.

2 Mark Baker, Jane Beltran, Jason Buell, Brian Conrey, Tom Davis, Brianna Donaldson, Jeanne Detorre-Ozeki, Leila Dibble, Tom Freeman, Robert Hammie, Julie Montgomery, Avery Pickford, and Justine Wong. Sets, planets, and comets. *The College Mathematics Journal*, 44(4):258–264, 2013.

3 Michael Bateman and Nets Katz. New bounds on cap sets. *Journal of the American Mathematical Society*, 25(2):585–613, 2012.

4 Jürgen Bierbrauer and Yves Edel. Bounds on affine caps. *Journal of Combinatorial Designs*, 10:111–115, 2000.

5 BoardGameGeek. Set. `https://www.boardgamegeek.com/boardgame/1198/set`.

6 BoardGameGeek community. Variations on a Set. Forum thread. `https://boardgamegeek.com/thread/92588/variations-set`.

7 Sebastian Brandt. Personal communcation, 2014.

8 A. R. Calderbank and P. C. Fishburn. Maximal three-independent subsets of $\{0,1,2\}^n$. *Designs, Codes and Cryptography*, 4(4):203–211, 1994.

9 Kamalika Chaudhuri, Brighten Godfrey, David Ratajczak, and Hoeteck Wee. On the complexity of the game of Set, 2003. Manuscript.

10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

11 Brian Corney and Brianna Donaldson. SET. Classroom material. `https://www.mathteacherscircle.org/assets/session-materials/BConreyBDonaldsonSET.pdf`.

12 Ernie Croot, Vsevolod Lev, and Peter Pach. Progression-free sets in $\mathbb{Z}_4^n$ are exponentially small. *Annals of Mathematics*, 185:331–337, 2017.

13 Daniel. Answer at Mathematics Stack Exchange. `https://math.stackexchange.com/a/202863`.

14 Benjamin Lent Davis and Diane Maclagan. The card game SET. *The Mathematical Intelligencer*, 25(3):33–40, 2003.

15 Y. Edel, S. Ferret, I. Landjev, and L. Storme. The classification of the largest caps in AG(5,3). *Journal of Combinatorial Theory, Series A*, 99(1):95–110, 2002.

16 Yves Edel. Extensions of generalized product caps. *Designs, Codes and Cryptography*, 31(1):5–14, 2004.

17 Jordan S Ellenberg and Dion Gijswijt. On large subsets of $\mathbb{F}_q^n$ with no three-term arithmetic progression. *Annals of Mathematics*, 185:339–343, 2017.

18 R. Hill. On Pellegrino's 20-caps in $S_{4,3}$. In *Combinatorics '81 in honour of Beniamino Segre*, volume 78 of *North-Holland Mathematics Studies*, pages 433–447. 1983.

19 Raymond Hill. Caps and codes. *Discrete Mathematics*, 22(2):111–137, 1978.

20 Erica Klarreich. A simple proof from the pattern-matching card game Set stuns mathematicians. News article. `https://www.wired.com/2016/06/simple-proof-card-game-set-stuns-mathematicians/`.

21 Donald Knuth. SETSET-RANDOM. CWEB program. `https://www-cs-faculty.stanford.edu/~knuth/programs/setset-random.w`.

22 Michael Lampis and Valia Mitsou. The computational complexity of the game of set and its theoretical applications. In *Latin American Theoretical Informatics Symposium (LATIN)*, pages 24–34, 2014.

23 Tom Magliery. Set variants. Personal Homepage. `http://magliery.com/Set/SetVariants.html`.

24 L. McMahon, G. Gordon, H. Gordon, and R. Gordon. *The Joy of SET: The Many Mathematical Dimensions of a Seemingly Simple Card Game*. Princeton University Press, 2016. URL: `https://books.google.cl/books?id=8JojDQAAQBAJ`.

25 Roy Meshulam. On subsets of finite abelian groups with no 3-term arithmetic progressions. *Journal of Combinatorial Theory, Series A*, 71(1):168–172, 1995.

26 Gu Pellegrino. Sul massimo ordine delle calotte in $S_{4,3}$. *Matematiche (Catania)*, 25:149–157, 1971. In Italian.

27 Aaron Potechin. Maximal caps in AG(6,3). *Designs, Codes and Cryptography*, 46(3):243–259, 2008.

28 SET Enterprises, Inc. SET instructions. `https://www.setgame.com/set/puzzle_rules`.

29 Terence Tao. Open question: best bounds for cap sets. Blog post. `https://terrytao.wordpress.com/2007/02/23/`.

30 Terence Tao. A symmetric formulation of the Croot-Lev-Pach-Ellenberg-Gijswijt capset bound. Blog post. `https://terrytao.wordpress.com/2016/05/18/`.

31 The New York Times. Daily SET feature. `https://www.nytimes.com/crosswords/game/set`.

32 Henrik Warne. SET card game variation – complementary pairs. `https://henrikwarne.com/2013/04/07/set-card-game-variation-complementary-pairs/`.

A Cryptographer's Conspiracy Santa

Xavier Bultel
LIMOS, University Clermont Auvergne, Campus des Cézeaux, Aubière, France
xavier.bultel@uca.fr
https://orcid.org/0000-0002-8309-8984

Jannik Dreier
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France
jannik.dreier@loria.fr
https://orcid.org/0000-0002-1026-3360

Jean-Guillaume Dumas
Université Grenoble Alpes, Laboratoire Jean Kuntzmann, UMR CNRS 5224, 700 avenue centrale, IMAG - CS 40700, 38058 Grenoble cedex 9, France
Jean-Guillaume.Dumas@univ-grenoble-alpes.fr
https://orcid.org/0000-0002-2591-172X

Pascal Lafourcade
LIMOS, University Clermont Auvergne, Campus des Cézeaux, Aubière, France
pascal.lafourcade@uca.fr
https://orcid.org/0000-0002-4459-511X

Abstract

In Conspiracy Santa, a variant of Secret Santa, a group of people offer each other Christmas gifts, where each member of the group receives a gift from the other members of the group. To that end, the members of the group form conspiracies, to decide on appropriate gifts, and usually divide the cost of each gift among all participants of that conspiracy. This requires to settle the shared expenses per conspiracy, so Conspiracy Santa can actually be seen as an aggregation of several shared expenses problems.

First, we show that the problem of finding a minimal number of transaction when settling shared expenses is NP-complete. Still, there exists good greedy approximations. Second, we present a greedy distributed secure solution to Conspiracy Santa. This solution allows a group of people to share the expenses for the gifts in such a way that no participant learns the price of his gift, but at the same time notably reduces the number of transactions with respect to a naive aggregation. Furthermore, our solution does not require a trusted third party, and can either be implemented physically (the participants are in the same room and exchange money using envelopes) or, virtually, using a cryptocurrency.

2012 ACM Subject Classification Security and privacy → Privacy-preserving protocols, Security and privacy → Formal security models, Theory of computation → Problems, reductions and completeness

Keywords and phrases Secret Santa, Conspiracy Santa, Secure Multi-Party Computation, Cryptocurrency, Physical Cryptography

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.13

Funding This research was conducted with the support of the FEDER program of 2014-2020, the region council of Auvergne-Rhône-Alpes, the support of the "Digital Trust" Chair from the University of Auvergne Foundation, the Indo-French Centre for the Promotion of Advanced Research (IFCPAR), the Center Franco-Indien Pour La Promotion De La Recherche Avancée (CEFIPRA)

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 13; pp. 13:1–13:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

through the project DST/CNRS 2015-03 under DST-INRIA-CNRS Targeted Programme, and the OpenDreamKit Horizon 2020 European Research Infrastructures project (#676541).

Acknowledgements Many thanks to Marie-Béatrice, Anne-Catherine, Marc, Jacques and Luc for such great conspiracy Santas! A big thanks also to the Gilbert family for having big instances of the Shared Expenses Problem problem regularly, and to Cyprien for asking the question of its complexity. More thanks go to Mathilde and Gwénaël for the discussions on and propositions of efficient algorithms.

1 Introduction

Secret Santa is a Christmas tradition, where members of a group are randomly assigned to another person, to whom they have to offer a gift. The identity of the person offering the present is usually secret, as well as the price of the present.

In *Conspiracy Santa*, a variant of Secret Santa, for each participant, the other members of the group collude and jointly decide on an appropriate gift. The gift is then usually bought by one of the colluding participants, and the expenses are shared among the colluding participants.

In this setting, the price of the gift must remain secret and, potentially, also who bought the present. At the same time, sharing the expenses usually results in numerous transactions. Existing results in the literature (e.g., [3, 4, 5, 12]) aim at minimizing the number of transactions, but they assume that all expenses are public, that all participants are honest, and that communications are safe. Our goal is to propose a secure Conspiracy Santa algorithm for cryptographers that do not want to disclose the prices.

1.1 Contributions

We provide the following contributions:

- We show that the general problem of finding a solution with a minimal number of transactions when sharing expenses is NP-complete.
- We provide a secure protocol for Conspiracy Santa. The algorithm ensures that no participant learns the price of his gift, nor who bought it. Moreover, the algorithm reduces the number of transactions necessary compared to a naive solution (although the solution in general is not optimal, as this could leak information).
- Our secure algorithm is entirely distributed and does not require any trusted third party. To also realize the payments in a distributed fashion, a secure peer-to-peer cryptocurrency can be used. We also discuss a physical payment solution, using envelopes and bank notes.

Our algorithm can also be used in the case where expenses are shared within multiple groups. There, some people belong to several of these groups and the goal is to reduce the number of transactions while still ensuring privacy: all participants only learn about the expenses of their groups, not the other groups. One can also see this problem as a variant of the dining cryptographers [7]. However, instead of respecting the cryptographers' right to anonymously invite everybody, we here want to respect the cryptographers' right to privately share expenses of multiple diners with different groups.

1.2 Outline

The remainder of the paper is structured as follows: in Section 2, we analyze the complexity of the general problem of sharing expenses. In Section 3, we present our protocol to solve the problem of privately sharing expenses in Conspiracy Santa, in a peer-to-peer setting. We also discuss further applications of our solution, and how to realize the anonymous payments required by the algorithm. We then conclude in Section 4.

2 The Shared Expenses Problem and its Complexity

Before analyzing the Conspiracy Santa problem in more detail, we now discuss the more general problem of settling shared expenses with a minimal number of transactions. This problem frequently arises, for example when a group of security researchers attends a FUN conference and wants to share common expenses such as taxis, restaurants etc. Reducing the overall number of transactions might then reduce the overall currency exchange fees paid by the researchers.

In such a case, each participant covers some of the common expenses, and in the end of the conference, some transactions are necessary to ensure that all participants payed the same amount. Note for this first example, there are no privacy constraints, as all amounts are public.

▶ **Example 1.** Alice, Bob, and Carole attended FUN'16. The first night, Alice payed the restaurant for 155 €, and Bob the drinks at the bar for 52 €. The second day Carole payed the restaurant and drinks for a total of 213 €.

The total sum is then $155 + 52 + 213 = 420$ €, meaning 140 € per person. This means that Alice payed $140 - 155 = -15$ € too much, Bob needs to pay $140 - 52 = 88$ € more, and Carole has to receive $140 - 213 = -73$ €. In this case, the optimal solution uses two transactions: Bob gives 15 € to Alice, and 73 € to Carole.

There are numerous applications implementing solutions to this problem (e.g., [3, 4, 5]), but it is unclear how they compute the transactions. Moreover, in these applications all expenses are public, making them unsuitable for Conspiracy Santa.

David Vávra wrote a master's thesis [12] about a similar smartphone application that allows to settle expenses within group. He discusses a greedy approximation algorithm (see below), and conjectures that the problem is $\mathcal{NP}$-complete, but without giving a formal proof.

We start by formally defining the problem.

▶ **Definition 2.** Shared Expenses Problem (SEP). Given a multiset of values $K = \{k_1, \ldots, k_n\}$ such that $\sum_{i=1}^{n} k_i = 0$ (where a positive k_i means that participant i has to pay money, and a negative k_i means that i has to be reimbursed), is there a way to do all reimbursements using (strictly) less than $n - 1$ transactions?

Note that there is always a solution using $n-1$ transactions using a greedy approach: given the values in $K = \{k_1, \ldots, k_n\}$, let i be the index of the maximum value of K ($i = \arg\max_i(k_i)$) and let j be the index of the minimum value of K ($j = \arg\min_j(k_j)$), we use one transaction between i and j such that after the transaction either the participant i or j ends up at 0. I.e., if $|k_i| - |k_j| > 0$, then the participant j ends up at 0, otherwise the participant i ends up at 0. By then recursively applying the same procedure on the remaining $n - 1$ values, we can do all reimbursements. Overall, this greedy solution uses $n - 1$ transactions in the worst case.

It is easy to see that $SEP \in \mathcal{NP}$: guess a list of (less than $n-1$) transactions, and verify for each participant that in the end there are no debts or credits left.

We show that SEP is $\mathcal{NP}$-complete, for this we use a reduction from the *Subset Sum Problem* [10] which can be seen as a special case of the well known knapsack problem [9].

▶ **Definition 3.** Subset Sum Problem (SSP) Given a multiset of values $K = \{k_1, \dots, k_n\}$, is there a subset $K' \subseteq K$ such that $\sum_{k' \in K'} k' = 0$?

The Subset Sum Problem is known to be $\mathcal{NP}$-complete (see, e.g., [8]).

▶ **Theorem 4.** *The Shared Expenses Problem is $\mathcal{NP}$-complete.*

Proof. Consider the following reduction algorithm:

Given a Subset Sum Problem (SSP) instance, i.e., a multiset of values $K = \{k_1, \dots, k_n\}$, compute $s = \sum_{k \in K} k$. If $s = 0$, return yes, otherwise let $K' = K \cup \{-s\}$ and return the answer of an oracle for the Shared Expenses Problem for K'.

It is easy to see that the reduction is polynomial, as computing the sum is in $\mathcal{O}(n)$.

We now need to show that the reduction is correct. We consider the two following cases:

- Suppose the answer to the SSP is yes, then there is a subset $K'' \subseteq K$ such that $\sum_{k \in K''} k = 0$. If $K'' = K$, then the check in the reduction is true, and the algorithm returns yes. If $K'' \neq K$, then we can balance the expenses in the sets K'' and $K' \setminus K''$ independently using the greedy algorithm explained above. This results in $|K''| - 1$ and $|K'| - |K''| - 1$ transactions respectively, for a total of $|K'| - |K''| - 1 + |K''| - 1 = |K'| - 2 < |K'| - 1$ transactions. Thus there is a way to do all reimbursements using strictly less than $|K'| - 1$ transactions, hence the answer will be yes.
- Suppose the answer to the SSP is no, then there is no subset $K'' \subseteq K$ such that $\sum_{k \in K''} k = 0$. This means that there is no subset $K_3 \subseteq K'$ such that the expenses within this set can be balanced independently of the other expenses. To see this, suppose it were possible to balance the expenses in K_3 independently, then we must have $\sum_{k \in K_3} k = 0$, contradicting the hypothesis that there is no such subset (note that w.l.o.g. $K_3 \subseteq K$, if it contains the added value one can simply choose $K' \setminus K_3$).
  Hence any way of balancing the expenses has to involve all n participants, but building a connected graph with n nodes requires at least $n-1$ edges. Thus there cannot be a solution with less than $n-1$ transactions, and the oracle will answer no. ◀

3 Cryptographer's Conspiracy Santa

Consider now the problem of organizing Conspiracy Santa, where no participant shall learn the price of his gift. Obviously we cannot simply apply, e.g., the greedy algorithm explained above on all the expenses, as this would imply that everybody learns all the prices.

More formally, an instance of Conspiracy Santa with n participant consists of n shared expenses problem (sub-SEP), each with $n-1$ participants and with non-empty intersections of the participants. In each sub-SEP, the $n-1$ participants freely discuss, decide on a gift, its value v_i and who pays it; then agree that their share for this gift is $v_i/(n-1)$. Overall the share of each participant j is

$$\frac{\sum_{i=1, i \neq j}^{n} v_i}{n-1}.$$

A participants *balance* p_j is this share minus the values of the gifts she bought.

A simple solution would be to use a trusted third party, but most cryptographers are paranoid and do not like trusted third parties. A distributed solution would be to settle the expenses for each gift within the associated conspiracy group individually, but this then results in n instances of the problem, with $n-2$ transactions each (assuming that only one person bought the gift), for a total of $n \times (n-2)$ transactions.

Moreover, the problem becomes more complex if several groups with non-empty intersections want to minimize transactions all together while preserving the inter-group privacy.

▶ **Example 5.** *Example 1 continued.* For the same conference, FUN'16, Alice, Bob and Dan shared a taxi from the airport and Bob paid for a total of 60€, that is 20€ per person. There are two possibilities. Either Alice and Dan make two new transactions to reimburse Bob. Or, to minimize the overall number of transactions, they aggregate both accounts, i.e. those from Example 1 with those of the taxi ride. That is $[-15, 88, -73, 0] + [20, -40, 0, 20] = [5, 48, -73, 20]$. Overall Alice thus gives 5 € to Carole, Bob reduces his debt to Carole to only 48€ and Dan gives 20 € to Carole. The security issue, in this second case, is that maybe Alice and Bob did not want Dan to know that they were having lunch with Carole, nor that they had a debt of more than 20 €, etc.

In the next part we present our solution for the generalization of Conspiracy Santa as the aggregation of several shared expenses problems with non-empty intersections between the participants. This solution uses $3n$ transactions, preserves privacy, and does not require a trusted third party.

3.1 A Distributed Solution using Cryptocurrencies

We suppose that all participants know a fixed upper bound B for the value of any gift. Apart from the setup, the protocol has 3 rounds, each one with n transactions, and one initialization phase.

Note that we consider *semi-honest* participants in the sense that the participants follow *honestly* the protocol, but they try to exploit all intermediate information that they have received during the protocol to break privacy.

Initialization Phase

In the setup phase, the participants learn the price of the gifts in which they participate and can therefore compute their overall balance, p_i. They also setup several anonymous addresses in a given public transaction cryptocurrency like Bitcoin [1], ZCash [6] or Monero [2].

Finally the participants create one anonymous address which is used as a piggy bank. They all have access to the secret key associated to that piggy bank address. For instance, they can exchange encrypted emails to share this secret key. Protocol 1 presents the details of this setup phase.

First Round

The idea is that the participants will round their debts or credits so that the different amounts become indistinguishable. For this, the participants perform transactions to adjust their balance to either 0, B or a negative multiple of B. The first participant randomly selects an initial value between 1 and B €, and sends it to the second participant. This transaction is realized via any private payment channel between the two participants (physical payment, bank transfer, cryptocurrency payment, ..., as long as no other participant learns the transferred amount). Then the second participant adds his balance to the received amount modulo B, and forwards the money (up to B, or such that its credit becomes a multiple of

Protocol 1 SEP broadcast setup

```
Require: An upper bound $B$ on the value of any gift;
Require: All expenses.
Ensure: Each participant learns his balance $p_i$.
Ensure: Each participant creates 1 or several anonymous currency addresses.
Ensure: A shared anonymous currency address.
One anonymous currency address is created and the associated secret key is shared
  among all participants.
for each exchange group do
    for each payment within the group do
        broadcast the amount paid to all members of the group;
    end for
    for each participant in the group do
        Sum all the paid amounts of all the participants;
        Divide by the number of participants in the group;
        This produces the in-group share by participant.
    end for
end for
for each overall participant do
    Add up all in-group shares;
    Subtract all own expenses to get $p_i$;
    if $p_i < 0$ then
        Create $\lfloor \frac{p_i}{B} \rfloor$ anonymous currency addresses.
    end if
end for
```

B) to the next participant, and so on. The last participant also adds his balance and sends the resulting amount to the first participant. In the end, all participants obtain a balance of a multiple of B, and the random amount chosen by the first participant has hidden the exact amounts. The details are described in Protocol 2.

Second Round

The second and third rounds of the protocol require anonymous payments, for which we use anonymous cryptocurrency addresses. These two rounds are presented in Protocol 3. In the second round, every participant makes one public transaction of B € to the piggy bank.

Third Round

Each creditor recovers their assets via $\lfloor \frac{p_i}{B} \rfloor$ public transactions of B € from the piggy bank. Note that if a participant needs to withdraw more than B € he needs to perform several transactions. To ensure anonymity, he needs to use a different anonymous address for each transaction. In the end, the account is empty and the number of transactions corresponds exactly to the number of initial transactions used to credit the piggy bank's account.

▶ **Theorem 6.** *For n participants, Protocols 1, 2, 3 are correct and require $3n$ transactions.*

Proof. Including the piggy bank, all the transactions are among participants, therefore the sum of all the debts and credits is invariant and zero. There remains to prove that in the

Protocol 2 Secure rounding to multiple of the bound

```
Require: An upper bound $B$ on the value of any gift;
Require: Each one of $n$ participants knows his balance $p_i$;
Require: $\sum_{i=1}^{n} p_i = 0$.
Ensure: Each one of $n$ participants has a new balance $p_i$, either 0, $B$ or a negative multiple of $B$;
Ensure: $\sum_{i=1}^{n} p_i = 0$;
Ensure: Each transaction is between 1 and $B$ €;
Ensure: The protocol is zero-knowledge.
$P_1$: $t_1 \xleftarrow{\$} [1..B]$ uniformly sampled at random;
$P_1$: $p_1 = p_1 - t_1$;
$P_1$ sends $t_1$ € to $P_2$;            ▷ Random transaction 1..$B$ on a secure channel
$P_2$: $p_2 = p_2 + t_1$;
for $i = 2$ to $n - 1$ do
    $P_i$: $t_i = p_i \mod B$;
    $P_i$: if $t_i = 0$ then $t_i = t_i + B$; end if            ▷ $1 \leq t_i \leq B$
    $P_i$: $p_i = p_i - t_i$;
    $P_i$ sends $t_i$ € to $P_{i+1}$;            ▷ Random transaction 1..$B$ on a secure channel
    $P_{i+1}$: $p_{i+1} = p_{i+1} + t_i$;
end for
$P_n$: $t_n = p_n \mod B$;
$P_n$: if $t_n = 0$ then $t_n = t_n + B$; end if            ▷ $1 \leq t_n \leq B$
$P_n$: $p_n = p_n - t_n$;
$P_n$ sends $t_n$ € to $P_1$;            ▷ Random transaction 1..$B$ on a secure channel
$P_1$: $p_1 = p_1 + t_n$;
```

end of the protocol all the debts and credits are also zero. The value of any gift is bounded by B, thus any initial debt for any gift is at most $B/(n-1)$. As participants participate to at most $n-1$ gifts, the largest debt is thus lower than B €. Then, during the first round, all participants, except P_1, round their credits or debts to multiples of B. But then, by the invariant, after the first round, the debt or credit of P_1 must also be a multiple of B. Furthermore, any debtor will thus either be at zero after the first round or at a debt of exactly B €. After the second round any debtor will then be either at zero or at a credit of exactly B €. Thus after the second round only the piggy bank has a debt. Since the piggy bank received exactly nB €, exactly n transactions of B € will make it zero and the invariant ensures that, after the third round, all the creditors must be zero too. ◀

▶ Remarks. It is important to use a cryptocurrency such as Bitcoin, Monero or ZCash in order to hide both the issuer and the receiver of each transaction in the third round. This ensures that nobody can identify the users.

Note that when using Bitcoin, users can potentially be tracked if the addresses are used for other transactions. Using Monero or Zcash can offer more privacy since the exchanged amount can also be anonymized. Moreover, to avoid leaking the fact that some persons need to withdraw B€ multiple times, and are thus doing multiple transaction at the same time, all the withdrawals should be synchronized. If exact synchronization is difficult to achieve, one can decide on a common time interval, e.g., an hour, and all the transactions have to be done at random time points during this interval, independently, whether they are executed from the same or a different participant.

Protocol 3 Peer-to-peer secure debt resolution

```
Require: An upper bound $B$ on the value of any gift;
Require: $n$ participants each with a balance $p_i$, either 0, $B$ or a negative multiple of $B$.
Ensure: All balances are zero;
Ensure: The protocol is zero-knowledge.
parfor $i = 1$ to $n$ do                                   ▷ Everybody sends $B$ to the piggy bank
    $P_i$: $p_i$ -= $B$;
    $P_i$ sends $B$ € to the shared anonymous address;     ▷ Public transaction of $B$
end parfor
parfor $i = 1$ to $n$ do
    if $p_i < 0$ then                                      ▷ Creditors recover their assets
        parfor $j = 1$ to $\frac{-p_i}{B}$ do
            $P_i$ makes the shared anonymous address pay $B$€ to one of his own anonymous
            addresses;                                     ▷ Public transaction of $B$
        end parfor
        $P_i$: $p_i = 0$.
    end if
end parfor
```

▶ **Example 7.** We now have a look at the algorithm for our example with Alice, Bob, Carole and Dan. As in Example 5, the initial balance vector is $[5, 48, -73, 20]$. They decide on an upper bound of $B = 50$ € (note that to provably *ensure* exactly $3n = 12$ transactions they should take an upper bound larger than any expense, that is larger than 213 €, but 50 is sufficient for our example here). For the first round, Alice randomly selects $1 \leq t_1 = 12 \leq 50$ and makes a first private transaction of $t_1 = 12$ € to Bob. Bob then makes a private transaction of $t_2 = 12 + 48 \mod 50 = 10$ € to Carole; Carole makes a private transaction of $t_3 = 10 - 73 \mod 50 = 37$ € to Dan; who makes a private transaction of $t_4 = 37 + 20 \mod 50 = 7$ € to Alice. All these transactions are represented in Figure 1. The balance vector is thus now $[0, 50, -100, 50]$, because for instance Bob had a balance of 48 €, received 12 € from Alice and sends 10 € to Carole, hence his new balance is $48 + 12 - 10 = 50$ €. Everybody sends 50 € to the piggy bank address, so that the balance vector becomes $[-50, 0, -150, 0]$. Finally there are four 50 € transactions, one to an address controlled by Alice and three to (different) addresses controlled by Carole. These two last rounds are illustrated in Figure 2. Note that we have exactly $n = 4$ transactions per round.

3.2 Security Proof

We now provide a formal security proof for our protocol. We use the standard multi-party computations definition of security against semi-honest adversaries [11]. As stated above, we consider *semi-honest* adversaries in the sense that the entities run *honestly* the protocols, but they try to exploit all intermediate information that they have received during the protocol.

We start by formally defining the *indistinguishability* and the *view* of an entity.

▶ **Definition 8** (Indistinguishability). Let η be a security parameter and X_η and Y_η two distributions. We say that X_η and Y_η are *indistinguishable*, denoted $X_\eta \equiv Y_\eta$, if for every probabilistic distinguisher $\mathcal{D}$ we have:

$$\Pr[x \leftarrow X_\eta : 1 \leftarrow \mathcal{D}(x)] - \Pr[y \leftarrow Y_\eta : 1 \leftarrow \mathcal{D}(y)] = 0$$

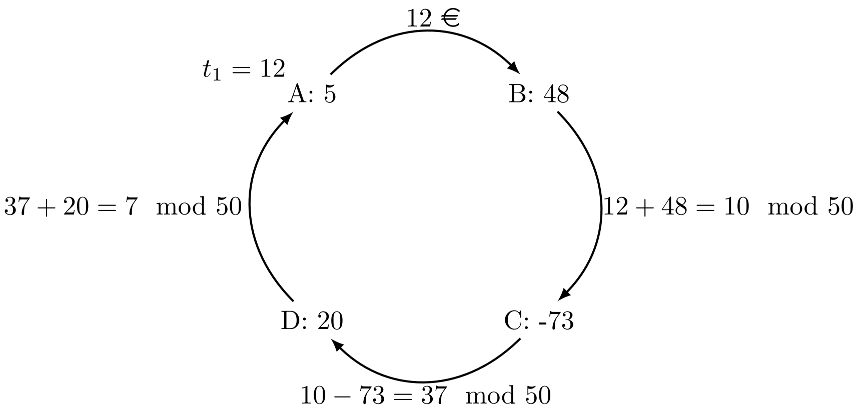


Figure 1 First round of Example 7.

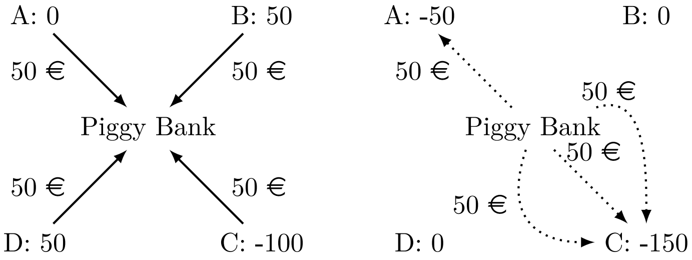


Figure 2 On the left: second round of Example 7. On the right: third round of Example 7. Dotted arrows represent anonymous transactions, in particular Carole uses three different anonymous addresses.

▶ **Definition 9** (view)**.** Let $\pi(I)$ be an n-parties protocol for the entities $(P_i)_{1\leq i\leq n}$ using inputs $I = (I_i)_{1\leq i\leq n}$. The view of a party $P_i(I_i)$ (where $1 \leq i \leq n$) during an execution of π, denoted $\text{VIEW}_{\pi(I)}(P_i(I_i))$, is the set of all values sent and received by P_i during the protocol.

To prove that a party P learns nothing during execution of the protocol, we show that P can run a *simulator* algorithm that simulates the protocol, such that P (or any polynomially bounded algorithm) is not able to differentiate an execution of the simulator and an execution of the real protocol. The idea is the following: since the entity P is able to generate his view using the simulator without the secret inputs of other entities, P cannot extract any information from his view during the protocol. This notion is formalized in Definition 10.

▶ **Definition 10** (Security with respect to semi-honest behavior)**.** Let $\pi(I)$ be an n-parties protocol between the entites $(P_i)_{1\leq i\leq n}$ using inputs $I = (I_i)_{1\leq i\leq n}$. We say that π is *secure in the presence of semi-honest adversaries* if for each P_i (where $1 \leq i \leq n$) there exists a protocol $\mathsf{Sim}_i(I_i)$ where P_i interacts with a polynomial time algorithm $S_i(I_i)$ such that:

$$\text{VIEW}_{\mathsf{Sim}_i(I_i)}(P_i(I_i)) \equiv \text{VIEW}_{\pi(I)}(P_i(I_i))$$

▶ **Theorem 11.** *Our conspiracy santa protocol is secure with respect to semi-honest behavior.*

Proof. We denote our protocol by $\mathsf{SCS}_n(I)$ (for *Secure Conspiracy Santa*). For all $1 \leq i \leq n$, each entity P_i has the input $I_i = (n, B, p_i)$, where $I = (I_i)_{1\leq i\leq n}$. For all $1 \leq i \leq n$, we show how to build the protocol Sim_i such that:

$$\text{VIEW}_{\mathsf{Sim}_i(I_i)}(P_i(I_i)) \equiv \text{VIEW}_{\mathsf{SCS}_n(I)}(P_i(I_i))$$

Simulator 4 Algorithm S_1 of the protocol $\mathsf{Sim}_1(I_1)$.

```
Require: $S_1$ knows $I_1 = (n, B, p_1)$
 1: $S_1$ receives $t_1$ € from $P_1$;
 2: if $0 \leq (p_1 - t_1)$ then
 3:     $S_1$ sends $(B - (p_1 - t_1))$ € to $P_1$;
 4: else if $(p_1 - t_1) < 0$ then
 5:     $S_1$ sends $(B - ((t_1 - p_1) \mod B))$ € to $P_1$;
 6: end if
 7: for $j = 1$ to $n - 1$ do
 8:     $S_1$ sends $B$ € to the shared anonymous address;
 9: end for
10: if $0 \leq (p_1 - t_1)$ then
11:     $x = n$;
12: else if $(p_1 - t_1) < 0$ then
13:     $x = n + \frac{(p_1 - t_1) - ((t_1 - p_1) \mod B)}{B}$;
14: end if
15: for $j = 1$ to $x$ do
16:     $S_1$ makes the shared anonymous address pay $B$ € to an anonymous address;
17: end for
```

Simulator 5 Algorithm S_i of the protocol $\mathsf{Sim}_i(I_i)$, where $1 < i \leq n$.

```
Require: $S_i$ knows $I_1 = (n, B, p_i)$
 1: $t_{i-1} \xleftarrow{\$} [1..B]$ ;
 2: $S_i$ sends $t_{i-1}$ € to $P_i$;
 3: $S_i$ receives $t_i$ € from $P_i$;
 4: for $j = 1$ to $n - 1$ do
 5:     $S_i$ sends $B$ € to the shared anonymous address;
 6: end for
 7: $x = n + \frac{p_i + t_{i-1} - t_i - B}{B}$;
 8: for $j = 1$ to $x$ do
 9:     $S_i$ makes the shared anonymous address pay $B$ € to an anonymous address;
10: end for
```

Sim_1 is given in Simulator 4, and Sim_i for $1 < i \leq n$ is given in Simulator 5.

We first show that the view of P_1 in the real protocol SCS_n is the same as in the protocol Sim_1:

- At Instruction 1 of Simulator 4, S_1 receives t_1 € from P_1 such that $1 \leq t_1 \leq B$, as at Instruction 3 of Protocol 2.
- At Instruction 15 of Protocol 2, P_n sends t_n € to P_1 such that:
  - $1 \leq t_n \leq B$
  - The balance of P_1 is a multiple of B.

  We show that these two conditions hold in the simulator. At Instruction 2 of Protocol 2, the balance of P_1 is $(p_1 - t_1)$.
  1. If the balance is positive, then $0 \leq (p_1 - t_1) < B$ and S_1 sends $B - (p_1 - t_1)$ € to P_1. We then have:
     - $1 \leq B - (p_1 - t_1) \leq B$
     - The balance of P_1 is $B - (p_1 - t_1) + (p_1 - t_1) = B$ which is multiple of B.

2. If the balance is negative, then S_1 sends $(B - ((t_1 - p_1) \mod B))$ € to P_1. We then have:
   - $1 \leq B - ((t_1 - p_1) \mod B) \leq B$
   - The balance of P_1 is: $B - ((t_1 - p_1) \mod B) + (p_1 - t_1) = B + \left\lfloor \frac{p_1 - t_1}{B} \right\rfloor \cdot B = \left(\left\lfloor \frac{p_1 - t_1}{B} \right\rfloor + 1\right) \cdot B$, which is a multiple of B.

- At Instruction 8 of Simulator 4, S_1 sends B € to the shared anonymous address $(n - 1)$ times, and P_1 sends B € to the shared anonymous address 1 time, so together they send B € n times to the shared anonymous address, as at Instruction 3 of Protocol 3.
- At Instruction 8 of Protocol 3, the users make the shared anonymous address pay B € to n anonymous addresses. At Instruction 16 of Simulator 4, the balance of P_1 is:
  - 0 if $0 \leq (p_1 - t_1)$ (because P_1 had B € and sent B € to the shared address).
  - Otherwise, the balance of P_1 is $B - ((t_1 - p_1) \mod B) + (p_1 - t_1) - B = ((t_1 - p_1) \mod B) + (p_1 - t_1)$. Hence P_1 receives B € from the shared anonymous address $\left| \frac{((t_1 - p_1) \mod B) + (p_1 - t_1)}{B} \right|$ times, and S_1 receives B € from the shared anonymous address $n + \frac{((t_1 - p_1) \mod B) + (p_1 - t_1)}{B}$ times. We note that $((t_1 - p_1) \mod B) + (p_1 - t_1) \leq 0$ because $(p_1 - t_1) \leq 0$ and $((t_1 - p_1) \mod B) \leq -(p_1 - t_1)$. Finally, P_1 and S_1 make the shared anonymous address pay B € to n anonymous addresses because:

$$n + \frac{((t_1 - p_1) \mod B) + (p_1 - t_1)}{B} + \left| \frac{((t_1 - p_1) \mod B) + (p_1 - t_1)}{B} \right| = n$$

Finally, we deduce that the view of P_1 in the real protocol SCS_n is the the same as in the simulator Sim_1:

$$\mathrm{VIEW}_{\mathsf{Sim}_1(I_1)}(P_1(I_1)) \equiv \mathrm{VIEW}_{\mathsf{SCS}_n(I)}(P_1(I_1))$$

We then show that the view of P_i in the real protocol SCS_n is the same as in the protocol Sim_1 for any $1 \leq i \leq n$:

- At instruction 3 and 9 of Protocol 2, each user P_i receives t_{i-1} € from P_{i-1} for any $1 \leq i \leq n$ such that $1 \leq t_{i-1} \leq B$. We note that each t_{i-1} depends on the value t_1 chosen by P_1. Moreover, t_1 comes form a uniform distribution and acts as a one-time pad on the values t_{i-1}, i.e., it *randomizes* t_{i-1} such that P_i cannot distinguish whether t_{i-1} was correctly generated or comes from the uniform distribution on $\{1, \dots, B\}$. At instruction 1 of Simulator 5, S_i chooses t_{i-1} at random in the uniform distribution on $\{1, \dots, B\}$ and sends t_{i-1} to P_i.
- At Instruction 3 of Simulator 5, S_i receives t_i € from P_i such that $1 \leq t_1 \leq B$, like at Instruction 9 of Protocol 2.
- At Instruction 5 of Simulator 5, S_i sends B € to the shared anonymous address $(n - 1)$ times, and P_i sends B € to the shared anonymous address 1 time, so together they send B € n times to the shared anonymous address, as at Instruction 3 of Protocol 3.
- At Instruction 8 of Protocol 3, the users make the shared anonymous address pay B € to n anonymous addresses. At Instruction 9 of Simulator 5, the balance of P_i is $p_i + t_{i-1} - t_i - B$. Hence P_i receives B € from the shared anonymous address $\left| \frac{p_i + t_{i-1} - t_i - B}{B} \right|$ times, and S_i receives B € from the shared anonymous address $n + \frac{p_i + t_{i-1} - t_i - B}{B}$ times. We note that $p_i + t_{i-1} - t_i - B \leq 0$; indeed, we have $t_i = (p_i + t_{i-1}) \mod B$ (Instruction 6 of Protocol 2). Since $p_i \leq B$ and $t_{i-1} \leq B$, then we have $(p_i + t_{i-1}) - t_i \leq B$, so we have $p_i + t_{i-1} - t_i - B \leq 0$. Finally, P_i and S_i make the shared anonymous address pay B € to n anonymous addresses because:

$$n + \frac{p_i + t_{i-1} - t_i - B}{B} + \left| \frac{p_i + t_{i-1} - t_i - B}{B} \right| = n$$

Finally, to conclude the proof, we deduce that for all $1 \leq i \leq n$ the view of P_i in the real protocol SCS_n is the the same as in the simulator Sim_i:

$$\text{VIEW}_{\mathsf{Sim}_i(I_i)}(P_i(I_i)) \equiv \text{VIEW}_{\mathsf{SCS}_n(I)}(P_i(I_i)). \qquad ◀$$

3.3 Physical Variant

If one does not wish to use cryptocurrencies, one can use the following physical variant of the protocol. In the first round each participant needs to transfer some money to another participant using a private channel. A simple physical solution is that they meet and perform the transfer face to face, while ensuring that nobody spies on them. For the second round, the balance of all participants is a multiple of B €. During the first part of this algorithm, everyone puts an envelope containing B € onto a stack that is in a secure room. By *secure room*, we mean a place where no other participants can spy what is going on inside. In the second part all participants enter this secure room one after the other and do the following according to their balance:

- If the balance is 0 then the participant does nothing.
- If the balance is a multiple k of B €, the participant takes k envelopes from the top of the stack, opens them and collects the corresponding $k * B$ €. Then he places, in each of the now empty k envelopes, a piece of paper that have the same shape and weight as a the B €. These envelopes are placed under the stack of envelopes.

This method allows everyone to collect his money without revealing to the other ones how much they have taken.

We show that this protocol is secure with respect to semi-honest behavior. For this, we physically simulate the protocol for any participant. We first note that the first round of the protocol is the same as Protocol 2, so this round can be simulated exactly as in the proof of Theorem 11. We simulate the second round for any participant as follows. During the first part of the algorithm, the simulator enters $n-1$ times the secure room and puts an envelope containing B € onto the stack. When it is his turn, the participant enters the room and puts an envelope containing B € onto the stack. Finally, there are n envelopes containing B € on a stack. In the second part the simulator enters the room $n-1$ times and does nothing. When it is his turn, the participant enters the room and takes k envelopes from the top of the stack, opens them and collects the corresponding $k * B$ € as in the real protocol, where $0 \leq k \leq n$. Since each of the n envelopes contains B €, the simulation works for any $0 \leq k \leq n$.

We deduce that the view of the participant during the simulation is the same as during the real protocol, which implies that our physical protocol is secure with respect to semi-honest behavior.

▶ Remark. This physical protocol mimics exactly the solution using cryptocurrencies. One advantage, though, of the physical world is that it is easier to perform transactions with 0 €. Therefore there exists a simpler solution for the second round, where creditors do not have to give B € in advance: if the participant is in debt he puts an envelope containing B € onto the stack, otherwise he puts an envelope containing a piece of paper under the stack.

The first and third rounds are not modified, and the simulator for the security proof is not modified either.

4 Conclusion

In this paper we showed that the Shared Expenses Problem (SEP) is $\mathcal{NP}$-complete. Moreover, we devised a privacy-preserving protocol to share expenses in a Conspiracy Santa setting where members of a group offer each other gifts.

Our protocol ensures that no participant learns the price of his gift, while reducing the number of transactions compared to a naive solution, and not relying on a trusted third party. We formally prove the security of our protocol and propose two variants, one relying on cryptocurrencies for anonymous payments, the other one using physical means, such as envelopes, to achieve anonymous payments.

Our protocol can also be used to share expenses among different groups with non-empty intersections, while still ensuring that each participant only learns the expenses of his group(s).

References

1 Bitcoin. https://bitcoin.org/. Accessed: 2018-02-13.

2 Monero. https://getmonero.org/. Accessed: 2018-02-13.

3 Settle up. https://settleup.io/. Accessed: 2018-02-13.

4 Splitwise. https://www.splitwise.com/. Accessed: 2018-02-13.

5 Tricount. https://www.tricount.com/. Accessed: 2018-02-13.

6 Zcash. https://z.cash/. Accessed: 2018-02-13.

7 David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *J. Cryptology*, 1(1):65–75, 1988. doi:10.1007/BF00206326.

8 Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

9 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, 1990.

10 Richard M. Karp. Reducibility among combinatorial problems. In Michael Jünger, Thomas M. Liebling, Denis Naddef, George L. Nemhauser, William R. Pulleyblank, Gerhard Reinelt, Giovanni Rinaldi, and Laurence A. Wolsey, editors, *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*, pages 219–241. Springer, Berlin, Heidelberg, 2010.

11 Qingkai Ma and Ping Deng. Secure multi-party protocols for privacy preserving data mining. In Yingshu Li, Dung T. Huynh, Sajal K. Das, and Ding-Zhu Du, editors, *Wireless Algorithms, Systems, and Applications, Third International Conference, WASA 2008, Dallas, TX, USA, October 26-28, 2008. Proceedings*, volume 5258 of *Lecture Notes in Computer Science*, pages 526–537. Springer, 2008. doi:10.1007/978-3-540-88582-5_49.

12 David Vávra. Mobile Application for Group Expenses and Its Deployment. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, Department of Computer Graphics and Interaction, 2012.

Cooperating in Video Games? Impossible! Undecidability of Team Multiplayer Games

Michael J. Coulombe
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
mcoulomb@mit.edu

Jayson Lynch
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
jaysonl@mit.edu

Abstract

We show the undecidability of whether a team has a forced win in a number of well known video games including: Team Fortress 2, Super Smash Brothers: Brawl, and Mario Kart.To do so, we give a simplification of the Team Computation Game [7] and use that to give an undecidable abstract game on graphs. This graph game framework better captures the geometry and common constraints in many games and is thus a powerful tool for showing their computational complexity.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases computational complexity, undecidable, team games, imperfect information

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.14

1 Introduction

Multiplayer videogames account for a large portion of the video game market and yet the additional computational complexity added by coordinating different team members has not seen much study from a theoretical standpoint. We finally bridge the gap between known theoretical models where imperfect information team games are known to be much more computationally complex and popular, commonly played video games.

In a series of papers [8–11], Reif and Peterson explored the computational complexity of games of imperfect information. One surprising result was a proof that unbounded team multiplayer games with imperfect information can be undecidable, despite having a bounded configuration space in the game itself. This work has been expanded to include formula and constraint logic games [7]; however, to the best of our knowledge, no commonly played game has been shown to be undecidable using this framework.

The computational complexity of video games has started becoming a popular topic of inquiry. Past research includes the study of classic arcade games like Pac-Man [13], classic Nintendo games such as Mario and the Legend of Zelda [1], to more modern games like Candy Crush [5], Portal [4], Angry Birds [12], and Braid [6]. However, all of these papers considered single-player, perfect information versions of the game. These are both aspects that intuitively and theoretically should make the games much more computationally challenging. This paper critically utilizes these properties to show far stronger hardness results than usually appears. We are aware of only one other video game, Braid, which has been shown to be undecidable. However, it does so by the construction of a counter machine using enemy units

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 14; pp. 14:1–14:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and thus playing such a level will require unbounded computational resources. The ability for a bounded game state to be able to lead to an undecidable problem has been remarked on by others are a fascinating feature of this type of problem [7].

In addition, much of the past work on video games has focused on environmental obstacles such as toggles for moving platforms and locking doors, rather than more central mechanics of the game. An aesthetic advantage of our proofs are that they focus on player vs player interaction and use the central combat mechanics of the game as core elements in the reduction.

Organization

This paper is organized into two parts. The first half deals with abstract games and builds a framework for later reductions. In particular, Section 2 details the gadgets involved in our team multiplayer graph game. Section 3 reduces the TEAM COMPUTATION GAME to the TEAM GRAPH GAME using our simplification of the former, the TEAM DFA GAME, with further details in Appendix A. The second half, Section 4, applies this framework to show the undecidability of several popular multiplayer games.

2 Team Graph Game Components

In this section we describe the different components of our undecidability framework which will be instantiated in the TEAM GRAPH GAME which we define and show to be undecidable in Section 3. Roughly speaking, it is a multi-player game with two teams, which we will refer to as blue and red, on a graph where each team wants to get one of their players to one of the win nodes. Players take time moving from node to node and from a node other nodes may be visible, allowing the player to determine if another player is there. In addition, some nodes will allow a player to guard an edge. A player attempting to cross a guarded edge will be eliminated and no longer be able to perform any useful actions. In our reduction we want to simulate a DFA which takes input from blue and red players and changes state based on this input. The state of the DFA will be encoded in the location of one player on the blue team, called the runner, and we call the other blue team members executors. The DFA entering an accept state will correspond to the runner being on a path which leads freely to a win node. The red team will supply their inputs by guarding some of the possible paths of the executors, while the executors will provide the blue team's inputs by choosing among unguarded paths to take. Both teams' inputs will force the runner to take a certain path through the region representing the DFA transition function. This section of the paper will describe these gadgets and their function in detail and Section 3 will formalize and complete the proof.

We break this framework down into several important gadgets each given their own subsection. We require a state transition gadget to manage the state of a deterministic finite automaton. This is described in Subsection 2.3. Both teams need to set variables which are taken as input to the DFA which is done with the choice gadgets described in Subsection 2.2. We need to synchronize all of the players so that the variable choices and DFA execution all occur in the proper order. This is done with the delay gadget described in Subsection 2.1. Finally, there is an optional initializer gadget which forces players from initial locations to the pathways needed in the gadgets. This is described in Subsection 2.4. These gadgets are put together in Section 3, as shown in Figure 5.

In this paper we use the following diagram conventions. Edges and nodes in the graph potentially containing red Team players are red and use square for nodes. Edges and nodes potentially containing blue Team players except for the runner are blue with circles as nodes.

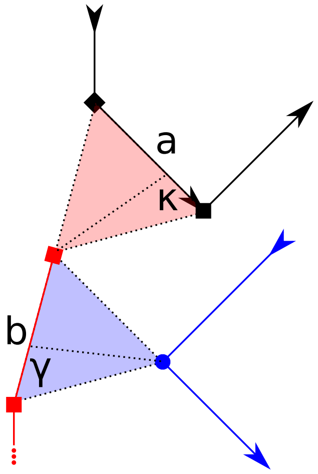


Figure 1 Gadget to delay the runner until a blue executor arrives to remove the red attacker.

Edges and nodes potentially containing the runner are black with diamonds for nodes. The graph contains both directed and undirected edges. Bold edges represent many different paths which serve similar function but are only accessed by one player. They are often accompanied by a label of how many edges are represented. Triple dots denote the continuation of a pattern, often many of the same type of edge. In contrast to bold edges, a different player will generally occupy each of these. Combat zones are pairs of nodes and edges and are denoted by a lightly colored red or blue triangle. The color dictates which team is posing a threat in the combat and always involves a node guarding an edge. If relevant, the combat zone is labeled with the length of time an enemy must spend traversing a guarded edge to be eliminated. These zones also imply visibility; however, we do not explicitly label visibility in all of our diagrams. Labeled boxes are used to refer to unrepresented gadgets, and dotted boxes are used to delineate different gadgets whose internal details are in the figure. An encircled W is a win node. Other labels and notation will hopefully be clear from context. Some of these conventions are used more liberally in the diagrams in Section 4 along side more representative pictures for the games.

2.1 Delay Gadget

The simplest gadget is the Delay Gate, as seen in Figure 1. The blue runner moves through the maze and is frequently blocked from making progress by a red player guarding a combat zone (edge) from an attack node. To progress, one of the blue executors must arrive at its own attack node which threatens the red guard, who must escape outside the combat zone (and far from its attack node) or be eliminated. As long as the red-beats-blue time $\kappa < a$ and the blue-beats-red time $\gamma < b$, the delay gadget achieves this goal.

2.2 Red Team Choice Gadget

The Red Team Choice Gadget gives the red team the ability to influence the path of a blue team player's movement. Detailed in Figure 2, a blue team member starts at node v_b and wants to exit out of v'_0 or v'_1, and a red team chooser at u_r (or its neighbors) will be able to force the outcome without fully preventing progress.

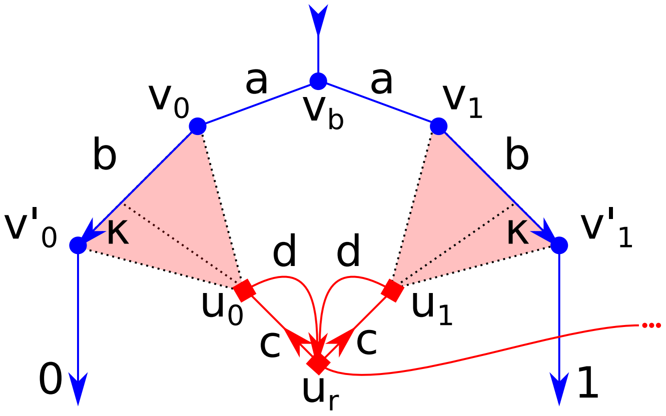


Figure 2 Gadget for a red player to force a blue player to take exit 0 or 1.

The graph is symmetric, so suppose without loss of generality that the red chooser wants the blue player to exit out of v_1'. Given their choice of where to start among the subgraph $\{u_0, u_r, u_1\}$, they can successfully block the v_0' exit by simply waiting at u_0 and attacking if the blue player tries to traverse edge (v_0, v_0'). If $c > a + b$, no starting location of the red chooser allows them to prevent the blue player from reaching both exits: the red chooser must start at least $d = a + b - \kappa$ time units away from u_0 to block v_0', which means starting $c + (c - d) > a + b$ away from u_1 which is too far to block v_1' as well.

An optimal strategy for the blue player to guarantee progress is thus to immediately move towards v_0'. Either the red chooser is blocking v_1' and the blue player will leave through the preferred exit, or red chooser is blocking v_0' and the blue player will have time to turn around and reach v_1' (the preferred exit) before the red chooser can reach u_1.

2.3 State Transition Gadget

Whereas the Red Team Choice Gadget is used to allow red team to influence a blue executor's path, the State Gate gadget is used to allow blue team executors to influence the blue runner's path. The "core" of a State Gate is essentially two Delay Gates sharing the same red guard who, unlike the Red Team Choice Gadget, is able to simultaneously block both exits for the blue runner. Depending on which of the two paths the blue executor is on, it will be able to safely open one of two exit paths for the blue runner.

Looking ahead to our undecidability proof for TGG, we generalize the core into a State Gate by first allowing for two independent hallways per blue executor "input" and second to allow for multiple independent hallways for the blue runner. Detailed in Figure 3, the first can be constructed using two cores (each with one hallway of each "input" type) or with one core modified such that the red guard's edges are the target of two blue executor attack nodes at once. The second generalization is simply constructed using multiple instances of the first in series along the blue executor's paths, one per required blue runner hallway.

The core works correctly as long as the red guard has visibility on the blue runner and executor and $\gamma < b < a - \kappa$. When safe, the red guard can mimic the blue runner's movement and always reach the closer attack node fast enough to block the path, but when the blue executor arrives on one side, the red guard must vacate the corresponding attack zone and can only safely block the opposite path. Thus, the blue runner strategy of repeatedly attempting to go in either direction until the red guard stops following to block will allow for guaranteed safe passage without visibility between the two blue team players. As a side note, the core

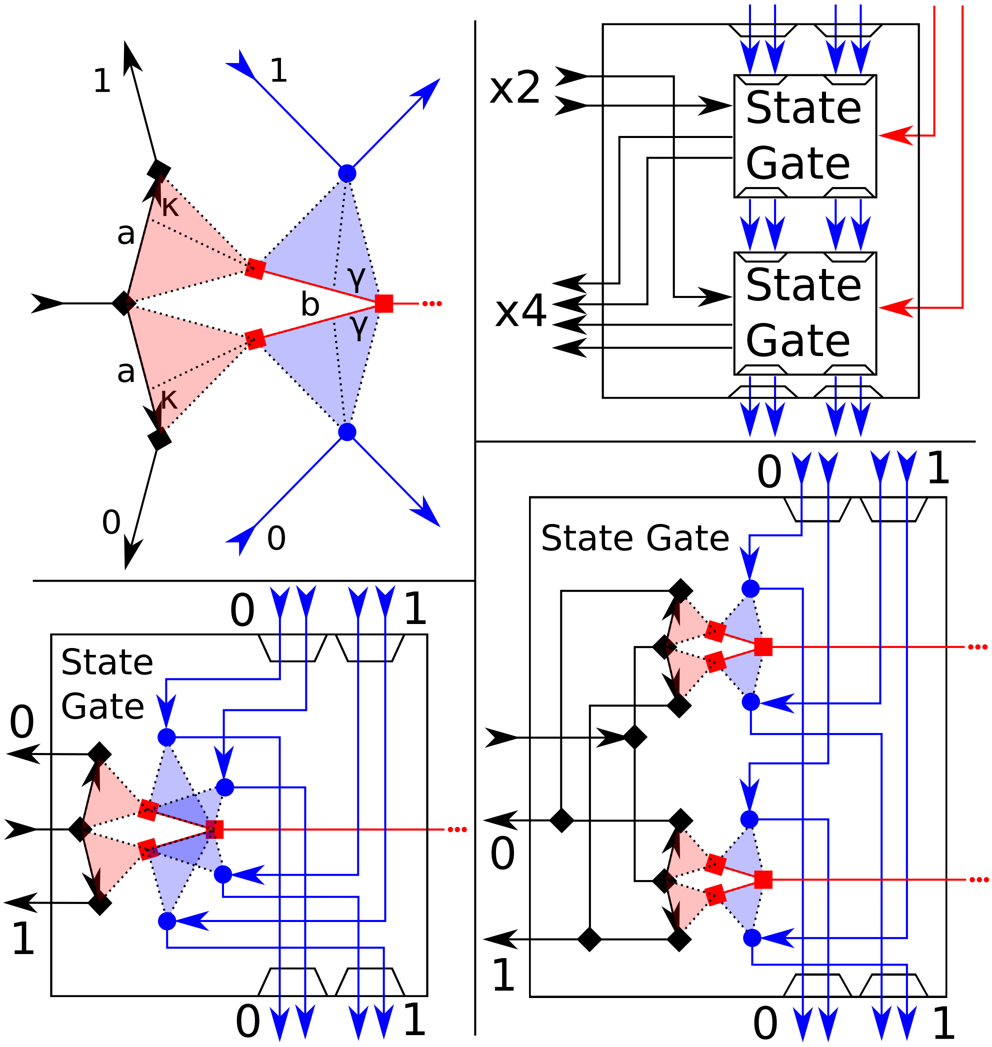


Figure 3 "State Gate" gadget schema for a blue executor to branch the blue runner. The core of player interaction (top-left) is generalized first allowing two blue paths per input (two possible constructions on bottom) then allowing multiple runner paths (top-right).

could also be implemented with two separate, unmodified Delay Gates, thus using two red guards instead of one but having no additional timing constraints.

2.4 Initialization

In many games we are modeling with TGG, all players on each team start in their team's single spawn room. In order to force the team members into separate hallways, they are coerced into guarding a set of paths, one per player (besides the runner), which all lead to the victory node w. Figure 4 shows the initializer gadget with spawn nodes s_b or s_r, where first blue must split into three hallways to block any red players from reaching w and force the red players to make progress and split up in order to block the blue runner from reaching w.

Specifically, to incentivize the blue team to fully split up, two red team "win paths" are placed and each guarded by a series of n_r blue attack zones of length $b_2 > \gamma$, so that even if the red team sends all of its players down one win path, the defending blue player could

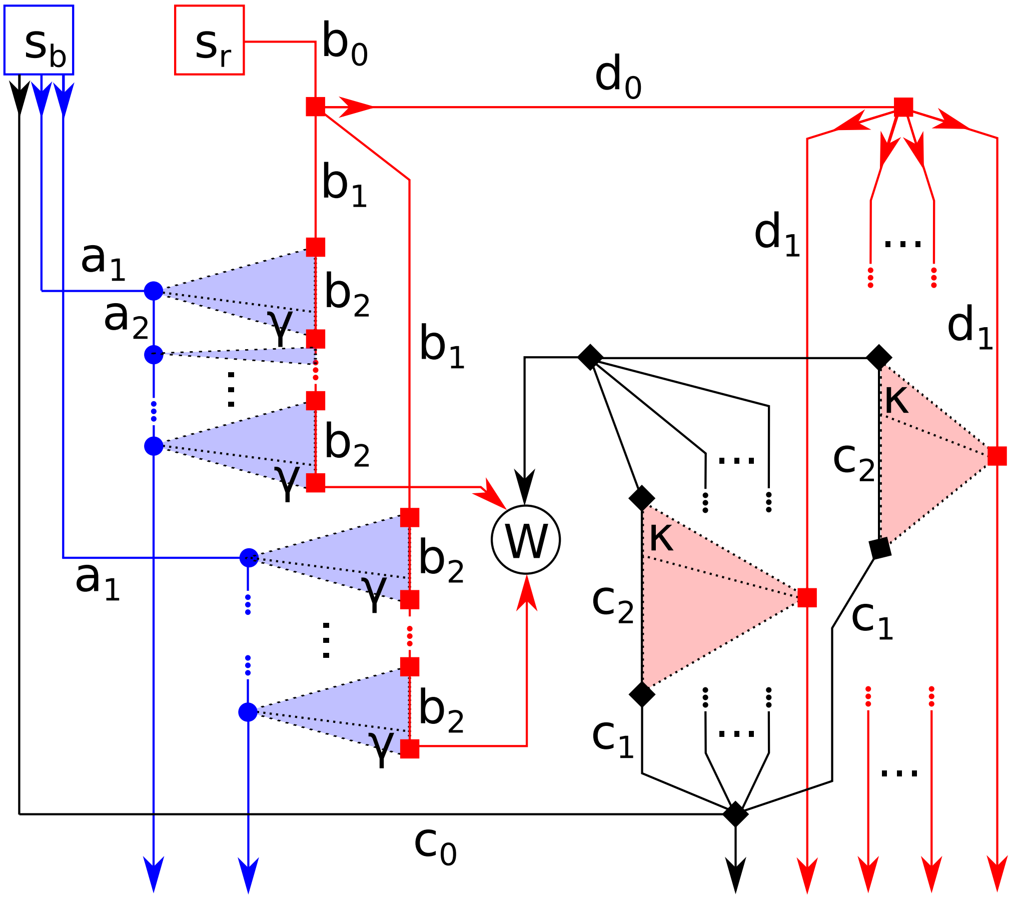


Figure 4 Initializer Gadget to separate players that must start together in team spawn rooms.

eliminate all of them by the end. If the blue team tries to send multiple players out the same hallway from s_b, they will either allow red team to win through the other win path in the initializer gadget, or have no player in the blue runner path, which is designed in our undecidability construction to be the only path to w.

If blue team does split up and guard the red team win paths, then red team must then prevent the blue runner from reaching w by going down a third path that splits into n_r branches, each responsible for guarding a different path for the blue runner. This forces the red team to separate and block every path until the blue runner gives up and exits the Initializer Gadget, at which point all other now-separated players can safely exit as well.

The constraints on the Initializer Gadget are light beyond the need for visibility so each player can learn when it is safe to stop guarding an attack zone and make progress. No information needs to be private at this point so full visibility is allowed within the gadget, although a set of hallways at the exit for the blue runner to pass within visibility range of every other player would be a sufficient signal for games being modeled by TGG with occlusion or view distance constraints. For the blue players to have time to block the red players, the attack nodes should be close enough together such that $\forall i \in [0, n_r) : a_1 + ia_2 < b_0 + b_1 + (i+1)(b_2 - \gamma)$. So that the red players have time to block the blue runner, it must be that $b_0 + d_0 + d_1 < c_0 + c_1 + c_2 - \kappa$.

3 Reductions

The TEAM COMPUTATION GAME (TCG), as defined in [3], is a game about two teams ($\exists$ and $\forall$) whose players alternate writing symbols onto certain cells of a finite-length tape of a Turing machine, which takes a fixed number of steps during each round and if it halts then the game ends and one team wins based on whether it accepts or rejects. A simplifying insight is that this Turing machine is effectively a DFA that teams are alternatively feeding input symbols into until it ends up in a final state that determines which team wins. The following modified definition will use this terminology instead for the purposes of the later reduction. Reductions establishing the equivalence of TDA with TCG and thus its undecidability can be found in Appendix A.

▶ **Definition 1.** The TEAM DFA GAME (TDG) is a two-versus-one team game. An instance of the game is a DFA $D = (\Sigma = \{0, 1\}, Q, q_0, \delta, F = F_\exists \Delta F_\forall)$. The existential team $\{\exists_1, \exists_2\}$ competes against the universal team $\{\forall\}$. The game starts with D in state q_0 and each round proceeds as follows:

1. If D's state $q \in F_\exists$ then team existential wins. If $q \in F_\forall$ then team universal wins.
2. $\forall$ learns the state q of D then inputs two bits b_1, b_2 into D.
3. $\exists_1$ learns b_1 then inputs one bit m_1 into D. $\forall$ learns m_1.
4. $\exists_2$ learns b_2 then inputs one bit m_2 into D. $\forall$ learns m_2.

We now go on to define the TEAM GRAPH GAME and show it is undecidable by a reduction from the TEAM DFA GAME.

▶ **Definition 2.** The TEAM GRAPH GAME is a team multiplayer game. Let the TGG of red team vs blue team consist of:

- Directed Graph $G = (V, E)$ with edge weights $\in \mathbb{N}$
- Designated team start nodes $s_r, s_b \in V$ and win node $w \in V$
- Directed visibility relation $S \subseteq V^2$
- (Uni)Directed attack relation $A \subseteq V^2$
- Initial number of players per team $n_r, n_b \in \mathbb{N}$

The execution of the TEAM GRAPH GAME starts with n_r red player tokens at node s_r and n_b blue player tokens at node s_b. Blue team wins if either every red token is eliminated or any blue token reaches the node w. Red team wins similarly.

The game proceeds as a sequence of time steps, or frames. Each frame, all active players simultaneously commit to their action and then all effects are triggered and handled before the frame ends. The action of a player consists of a node $n \in N[v]$ to move towards (or none to signify not moving). Once players have performed their moves, each player whose token can "see" another player's token learns of said token's position and team. Visibility zones are defined at nodes by S and on edges by union of the visibilities of the endpoints; combat zones are defined similarly.

▶ **Theorem 3.** *TDG reduces to the TEAM GRAPH GAME (TGG). Namely, $\exists h : \langle D\rangle \mapsto \langle I\rangle$ which maps instances $\langle D\rangle$ of TDG and instances $\langle I\rangle$ of TGG such that the existential team has a forced win in the TDG on D iff the blue team has a forced win in TGG on I.*

Proof. Figure 5 gives an overview of the structure of $I = h(D)$. Once the initializer gadget distributes each blue and red player into their proper hallways, each loop of the blue team in the graph simulates one round of TDG. The universal team's decisions b_1, b_2 are made (cooperatively) by the two decision-making red team members in the red choice gadgets,

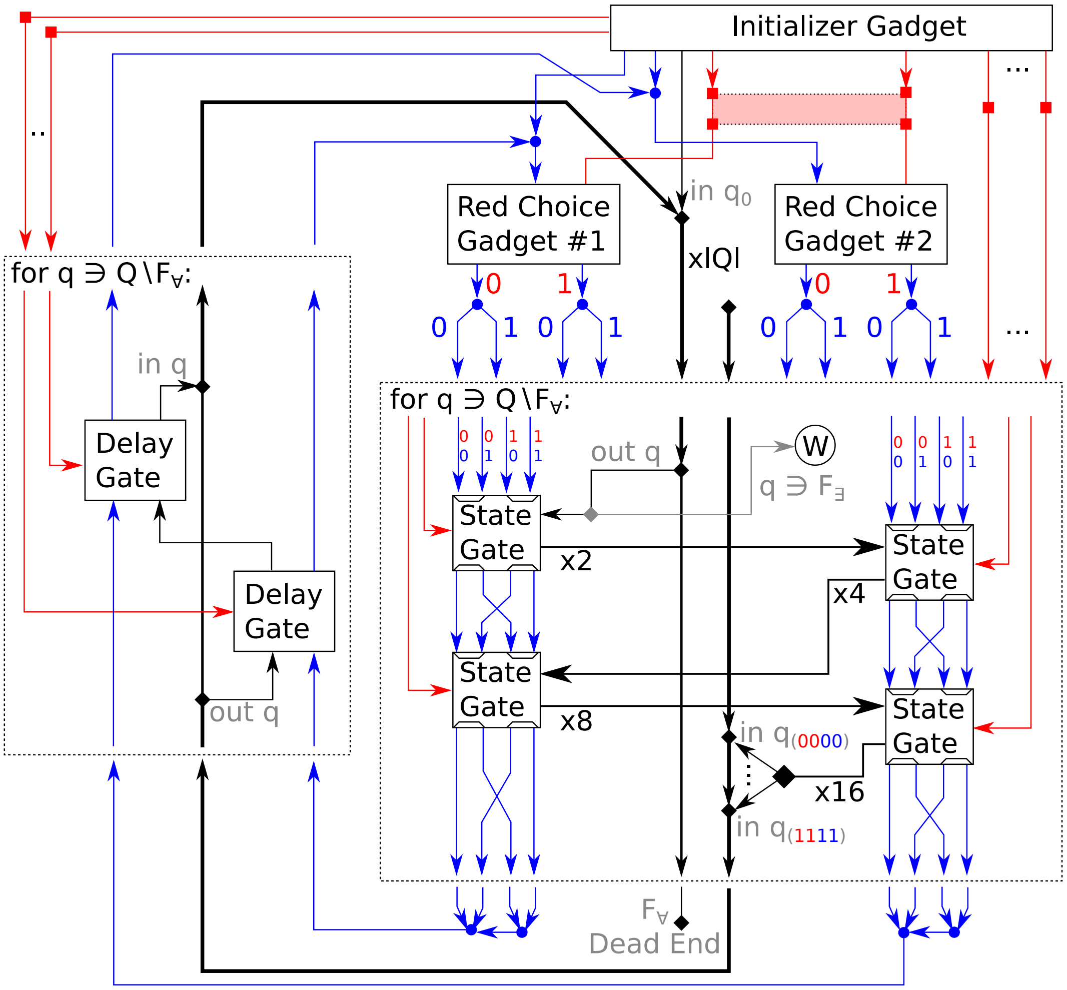


Figure 5 A diagram of how the gadgets are put together.

and the existential team's decisions m_1, m_2 are made (independently of each-other) by the decision-making blue team members directly after exiting the red choice gadgets. The blue runner's location corresponds directly to the state of the DFA, and their teammates open paths inside state gates which allows the runner to implement the DFA transition function δ.

Each state $q \in Q \setminus F_\forall$ of the DFA has an "arena" with two sides: the right side with a series of four state gates of increasing arity and a left side with a series of two Delay Gates. When the blue runner enters the right side of the arena for q before the first state gate, the DFA is in state q. If $q \in F_\exists$ then there will also be a hallway here leading directly to the win node. The four state gates encode the tree of states reachable from q in up to 4 transitions, outputting the runner in one of 16 hallways each corresponding to a state $q' = \mathsf{foldl}(\delta, q, [b_1, b_2, m_1, m_2])$ and leading to the left side of the arena for q'. Once the runner passes through the Delay Gates, they enter the right side of the arena for q'. Lastly, if $q \in F_\forall$, then all hallways entering its arena lead to a dead-end.

As we showed in Section 2.4, each team has a course of action which will prevent any players on the other team from reaching the Win node. Further, this puts every player on a path whose only way forward is out of the initializer gadget. At that point there is no incentive to stay in the initializer gadget and we may as well assume they continue into the rest of the map.

$\Longrightarrow$ Suppose the existential team has a forced win in TDG on D. This means that there are optimal strategy functions $\mathsf{s}_i : ([b_{i,1}, b_{i,2}, ..., b_{i,j-1}], [m_{i,1}, ..., m_{i,j-1}], b_{i,j}) \mapsto m_{i,j}$ which produce a win-preserving move for $\exists_i$ in round j given $\forall$'s move and what they learned in the past $j-1$ turns.

For decision-making blue player i, on the j^{th} time they pass through red choice gate i, let $b_{i,j} = 0$ if exiting the A side else $b_{i,j} = 1$ if exiting the B, let $m_{i,j} = \mathsf{s}_i([b_{i,1}, ..., b_{i,j-1}], [m_{i,1}, ..., m_{i,j-1}], b_{i,j})$, then at the upcoming branch take path $m_{i,j}$. The blue runner should follow the hallways and wait until combat zones are safe before passing through, and the decision-making blue team members should open combat zones long enough for the runner to pass through safely and to defeat the red team member there if necessary. By the structure of the graph, the path of the runner will lead to a $q \in F_\exists$ no matter what choices red team makes in the red choice gadgets, and every attack zone along the way will be opened up for the blue runner by their teammates, thus blue team has a forced win in TGG on I.

$\Longleftarrow$ Now suppose blue team has a forced win in TGG on I. Since only the blue runner can reach win node (outside the initializer gadget), any winning execution entails a path through the graph that the runner took which starts by entering the right side of the q_0 arena, passes through n arena right sides and left sides (as described earlier), and ends at the entrance of the right side of an arena for some $q_n \in F_\exists$.

In order for the runner to pass through the combat zones in the gates along the path, the decision-making blue teammates must have dealt with the attacking red team members. Since blue team has a forced win, they still have a forced win even if red team attackers always leave their attack zone before the decision-making blue team member has a chance to defeat them, thus that strategy forces the blue runner at the entrance of the right side of an arena to take a path through the state gates determined by the red and blue teams' choices at the start of the loop.

This implies the existence of functions $\mathsf{s}_i : ([b_{i,1}, b_{i,2}, ..., b_{i,j-1}], [m_{i,1}, ..., m_{i,j-1}], b_{i,j}) \mapsto m_{i,j}$ which produce a win-preserving branch for decision-making blue team member i to take on the loop j after exiting red choice gate i from exit $b_{i,j}$ and what they learned in the past $j-1$ loops. By the structure of the graph, s_i is also an optimal strategy function for $\exists_i$ in TDG on D, thus the existential team has a forced win. ◀

▶ **Corollary 4.** *The TEAM GRAPH GAME is undecidable.*

Proof. If TEAM GRAPH GAME were decidable, then TDG would be decidable using h from Theorem 3 to get a homomorphic instance, but since TDG is undecidable by Corollary 8, TEAM GRAPH GAME cannot be either. ◀

4 Applications

We now show how to apply the TEAM GRAPH game to generalized versions of several popular video games. In particular we will show that it is undecidable to determine whether a team can force a win in the following games: Team Fortress 2, Mario Kart, and Super Smash Bros. Brawl. For all of these games we generalize the map size and number of players able to participate in a single game. In addition, we assume that players on the same team have no way of communicating with each other beyond their actions in the game. This means players are not co-located, there is no screen-sharing, and any sort of team or global chat is disabled.

The following are the essential components needed in the game to fit the TGG framework. 1) The game needs a 3D map or crossover gadgets in 2D because the TGG graph used in

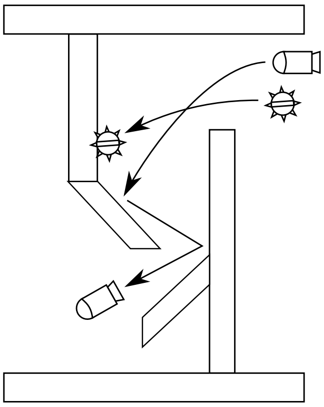


Figure 6 Grenade-only Attack Gadget (vertical 2D slice)

our reduction is non-planar. 2) One-way Doors. 3) Visibility zones such that we can have two players communicate their location without being able to reach each others path, and ways of blocking visibility so communication can only occur in specific regions. 4) Combat zones which allow the attacker a guaranteed strategy to eliminate or disable the defender and which has no path between the attacker and defender. 5) A win condition that can be activated by one player in a limited location.

4.1 Team Fortress 2 and many other team FPS games

Like many others of its kind, Team Fortress 2 is a first person shooter with 3D environments (1), one-way doorways (2), clear unbreakable glass/fences and opaque walls (3) made out of polygons, grenades and sniper rifles (4), and a capture point where one team can win by standing on it (5). These features allow TF2 and others to directly simulate TGG, leading to their undecidability. Note: only the base TF2 game with default loadouts are considered.

The nodes and edges of the graph are generally represented as hallways made of opaque walls connecting at intersections, possibly lengthened or bent-out-of-shape to enforce a required minimum traversal time. Visibility is limited by the first-person view, and visibility zones are constructed by making walls out of glass that gives a line-of-sight between desired locations and possibly additional walls to block view elsewhere.

The combat zones are constructed based on which team the attacker is on. A blue team member attacking a red team member will be faced with a room with a wall that only Demomen grenades can be shot over and succeed at damaging the defender. Figure **??** shows how to construct a hole which only physics-enabled grenades can tumble through and sticky bombs and other weapons cannot penetrate. A red team member attacking a blue team member will be faced with a small hole in the wall at Sniper-eye-level which gives a long-distance view of the defender's head such that only a Sniper's sniper rifle can kill the defender before they can pass through the attack zone at optimal speed.

In order to further enforce desired class choices, the red and blue teams are incentivized to choose the Sniper and Demoman classes (respectively) by the map design. The blue team spawn room is separated by a deadly chasm that can only be crossed using the Demoman's unique ability to sticky bomb jump long distances through the air without touching a surface (as a Soldier requires). Health pack pick-ups and distance-based fall damage may be used to force the health of players down so one sniper shot or grenade explosion will defeat any opponent.

By playing in a king-of-the-hill match with unlimited-time and with text and voice chat disabled, this map structure will exactly simulate TGG.

4.2 Super Smash Brothers

Super Smash Brothers is a popular Nintendo fighting game series. Out of the series' five releases, the most recent three (Super Smash Bros. Brawl, Super Smash Bros. for 3DS, and Super Smash Bros. for Wii U, henceforth referred to as Brawl, SSB4 3DS, and SSB4 Wii U, respectively) share a number of gameplay elements which we will shortly show result in undecidability.

We consider a generalized Super Smash Bros. game, where an arbitrary number of players on red or blue team control fighters (who are followed by the players' personal, local cameras, as in SSB4 3DS Smash Run mode) which fight on a stage (a bounded 2D plane with gravity, solid polygonal ground, and other obstacles) in Stamina mode (where each player starts with a given number of hit points and dies when they are depleted). Fighters are selected among a set of characters, each with unique traits, and can walk, run, jump off the ground and jump in the air finitely-many times before landing, and fight using aerial and ground attacks (which may create hitboxes which damage and knockback other characters, may move the attacker, and may provide defense), and defensive maneuvers such as shielding (a bubble around character which blocks attacks at the expense of temporary shrinkage), air and ground dodging (temporary invincibility at the cost of short vulnerability before and afterwards) and rolling (a ground dodge with fixed motion left or right). Due to close-quarters, we also consider obtrusive stage background music such that all character sound effects are drowned-out.

▶ **Theorem 5.** *In generalized Super Smash Bros. match between two teams of Pikachus on some stage, it is undecidable whether Player 1's team has a forced win.*

Proof. Reducing from TGG constrained to graphs constructed from DFA as in Theorem 3, we consider only the character Pikachu due to its unique Thunder attack that temporarily spawns a damaging cloud and lightning strike at a fixed position above Pikachu, even if there are obstacles in between. Instead of 3D hallways, our construction of the stage simulating the graph only needs to bound 2D areas with strings of solid blocks (as in Brawl's and SSB4 Wii U's stage builder) that are thin enough in certain areas for Thunder to attack other characters through ceilings. We also use thin floors, which allow for jumping upwards through but do not allow for falling through, to construct one-way doors.

The most striking problem for this 2D fighting game is the need for a crossover gadget. We make use of the barrel cannon stage obstacle, as seen in the Kongo Jungle stage from the first Super Smash Bros. as well as all future titles in some form, which captures a player upon contact and, when activated by the player inside, launches them along a fixed path without the player having aerial control until the end. Notably, we consider the original design of the cannon where a launched player does not hurt others via collision. By using two barrels and two one-way floors, a section of the stage as in Figure 7 can allow for crossovers without player interaction, although it does provide visibility. Because the constrained TGG graphs can be embedded in the plane where all edge crossings are either outside of the main loop before the simulation begins, same-player crossings, or between players who are allowed to know where the other's token is located, visibility does not transmit information that is useful for making red or blue team "choices."

As mentioned, attack zones are built around Pikachu's Thunder attack, which unconditionally creates a hitbox at a fixed distance high above the character. For attack zones that guard the traversal of an edge, the idea is to force the defending Pikachu to predictably position itself in a vulnerable state above the attacker, so that the attacking Pikachu can always hit them with Thunder if traversal is attempted. In Delay Gates, such as in Figure 8, where the red attacker of the blue runner is under attack themselves, the blue attacker is able to Thunder the only location at which the red attacker can use Thunder to hit the blue runner, so as to open the path safely. The Red Team Choice Gadget can be implemented in Brawl similarly to the Delay Gate, and the State Gates directly out of Delay Gates, thus the given TGG graph is fully representable.

When the blue runner reaches the win node, they can themselves open a path for the other blue Pikachus and all go into a new series of pathways that lead underneath every red team player so they can work together to eliminate them all, as properly-timed Thunders by multiple players can break shields and hit for longer than dodge invincibility. and end the match with a blue victory. This path-opening can be a Delay Gate or even compactly implemented using Brawl's Falling Block object, which is a solid obstacle that temporarily falls and disappears after a player (the blue runner, in this case) stands on it, reappearing at its original position after a short period of time. ◀

4.3 Mario Kart

In an earlier paper, two player, perfect information Mario Kart was shown to be PSPACE-complete [2]. It also did not consider the commonly enjoyed Battle game type. Here we show that a generalized version of Mario Kart in team Balloon Battle mode is undecidable by a reduction from TGG.

Mario Kart takes place in a 3D environment where each player has a personal third-person camera view of their character; when playing online or on local wireless, players cannot see other players' screens. In Balloon Battle, the players are placed in an enclosed, obstacle-filled Battle Course with a small number of balloons that pop when the player is damaged, eliminating the player if none remain. By searching the course for item boxes (in fixed, reusable spawn locations), players can get items from a given distribution to damage other players and avoid attacks against themselves. There is a blue team and a red team, and if one team is completely eliminated, the other team wins.

▶ **Theorem 6.** *In generalized Mario Kart Balloon Battle with the Bob-ombs Only item distribution, it is undecidable whether or not the blue team has a forced win.*

Proof. We reduce from TGG constrained to graphs constructed from DFA as in Theorem 3, which involves building a Battle Course that simulates the graph. Mario Kart courses are polygonal 3D environments with a finite maximum movement speed, one-way jumps, clear glass, and opaque walls, so the primary complexity is describing the attack zones and how to win.

A player using a Bob-omb item causes a Bob-omb to be thrown from the character's kart in an arc. It can bounce off walls and will explode into a large, temporary, damaging sphere on contact with another player or after a short time interval. One common obstacle in Mario Kart is the Thwomp, which are large spike-covered boxes which can move along fixed paths.

To construct an attack zone where the attacker is preventing the defending character from traversing an edge, said edge is a short, thin hallway with exits guarded by Twomps that alternate moving up and down between the ceiling and ground such that at least one is always on the ground blocking the path and the space between is smaller than the diameter of a Bob-omb explosion. The attacking character is spawned in a raised hallway with an item box and an uncrossable pit such that a Bob-omb can be thrown by the attacker and create an explosion to eliminate any player between the Thwomps but no Bob-omb can be thrown back high enough to reach the attacker. In an attack zone where the defender is itself an attacker in a dead-end hallway, there need only be one Thwomp guarding the single exit

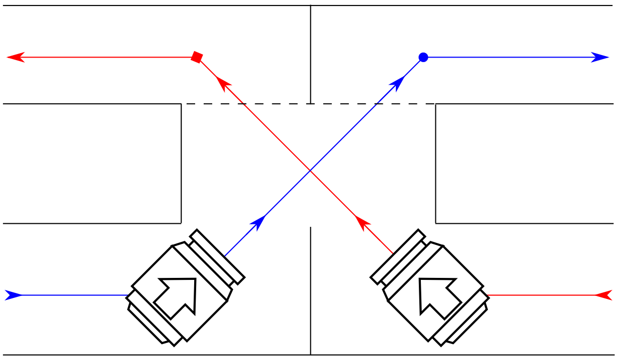


Figure 7 Super Smash Bros Crossover Gadget using Barrel Cannons

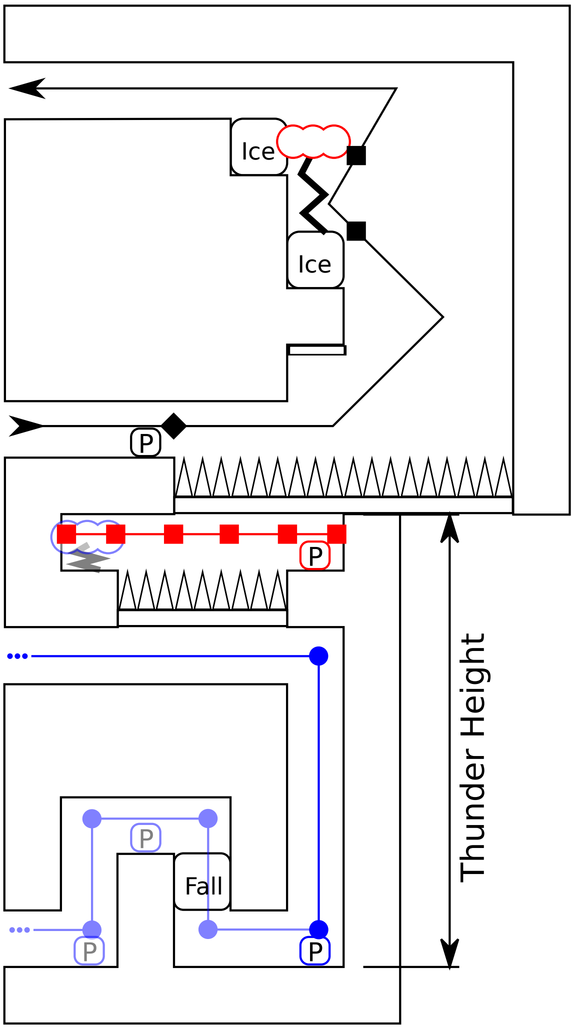


Figure 8 Delay Gate constructed using Brawl's Custom Stage Builder parts. A single player's screen is approximately 5 blocks tall, so the blue executor can never see the runner. Each "P" is an example location of a Pikachu, "Ice" is a block with no edge to hang onto, and "Fall" represents a Falling Block. Shaded blue figures are only relevant during the blue victory phase. Example Thunder clouds and associated lightning strikes are also shown.

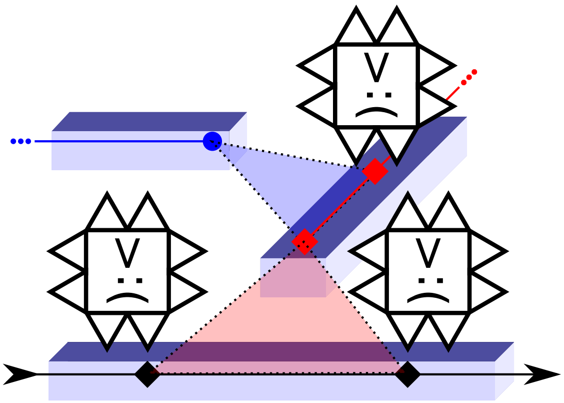


Figure 9 Mario Kart Delay Gadget's 3D Layout with Thwomps (opaque walls not shown).

and trapping the defender for a period of time such that the attacker in an even-more-raised hallway could safely throw down a Bob-omb to eliminate them. Figure 9 gives an overview of this construction.

When the Mario Kart character simulating blue runner is supposed to reach the win node, they are first able to open a path for their blue teammates (normally blocked by a red attack zone) to join them into a set of hallways above the rest of the course which lead to attack zones spanning each red team character's small region of the graph. With plentiful item boxes, the blue team characters can thus trap and eliminate each red team member using coordinated Bob-omb threats and throws, winning them the game. ◀

5 Conclusion and Open Problems

Our TEAM GRAPH GAME framework has proven useful in showing the undecidability of more natural team multi-player games, as shown in our application to various video games. We currently wonder how far this framework can go. Can we capture other popular genera of video-games with teams such as MMORPGs like World of Warcraft and Guild Wars, real time strategy games like Starcraft or Age of Empires, MoBAs like DotA and Heroes of the Storm, or others? Each of these has their own challenges in adapting to our framework, but given our success with Super Smash Brothers which was a 2D game that lacked vision blockers and a location based victory condition, we believe a lot can be done with a little work. We also pose the question of whether this framework can be used to understand the complexity of any real world multi-agent coordination scenarios.

There are also a number of interesting questions about imperfect information team multi-player games, many of which would be very useful in allowing broader application of this framework. First, can TGG be adapted to use only a constant number of players on each team? The TCG needs only three; however, we find it useful to assign different players to many of our gadgets, leading to a linear scaling. Is the TCG or TGG still undecidable if we allow a limited amount of communication between players on the same team? For example, players may be allowed to pairwise communicate or broadcast a constant number of bits per round. At what point does this problem become equivalent to a two player game of imperfect information? Finally, is there a way to adapt these abstract games to describe semi-cooperative games and are these still undecidable. For example, instead of having fixed teams and asking for a forced win, we might define optimal play to involve trying to maximize an individual player's probability of winning and ask whether a certain player has a strategy which wins with some fixed probability. If players have some chance of winning by working together but zero chance of winning otherwise, we might be able to force players to simulate teams in such a game.

References

1 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (NP-)hard. In *Proceedings of the 7th International Conference on Fun with Algorithms (FUN 2014)*, Lipari Island, Italy, July 1–3 2014.

2 Jeffrey Bosboom, Erik D Demaine, Adam Hesterberg, Jayson Lynch, and Erik Waingarten. Mario kart is hard. In *Japanese Conference on Discrete and Computational Geometry and Graphs*, pages 49–59. Springer, 2015.

3 Erik D Demaine and Robert A Hearn. Constraint logic: A uniform framework for modeling computation as games. In *Computational Complexity, 2008. CCC'08. 23rd Annual IEEE Conference on*, pages 149–162. IEEE, 2008.

4 Erik D Demaine, Joshua Lockhart, and Jayson Lynch. The computational complexity of portal and other 3d video games. *arXiv preprint arXiv:1611.10319*, 2016.

5 Luciano Guala, Stefano Leucci, and Emanuele Natale. Bejeweled, candy crush and other match-three games are (np-) hard. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

6 Linus Hamilton. Braid is undecidable. *arXiv preprint arXiv:1412.0784*, 2014.

7 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A. K. Peters, Ltd., Natick, MA, USA, 2009.

8 G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers and Mathematics with Applications*, 41(7):957–992, 2001. doi:10.1016/S0898-1221(00)00333-3.

9 Gary L Peterson and John H Reif. Multiple-person alternation. In *Foundations of Computer Science, 1979., 20th Annual Symposium on*, pages 348–363. IEEE, 1979.

10 John H Reif. Universal games of incomplete information. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 288–308. ACM, 1979.

11 John H Reif. The complexity of two-player games of incomplete information. *Journal of computer and system sciences*, 29(2):274–301, 1984.

12 Matthew Stephenson, Jochen Renz, and Xiaoyu Ge. The computational complexity of angry birds and similar physics-simulation games. 2017.

13 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

A TEAM DFA GAME is Undecidable

▶ **Lemma 7.** *TDG is reducible from and to TCG. Namely,* $\exists f : \langle D\rangle \to \langle I\rangle$ *and* $\exists g : \langle I\rangle \to \langle D\rangle$ *which map between instances* $\langle D\rangle$ *of TDG and instances* $\langle I\rangle$ *of TCG which both preserve the predicate of whether or not the existential team has a forced win.*

Proof.

$\Longleftarrow$ Consider an instance $I = \langle S, O, k, \Gamma \supset O \cup \{\mathsf{A}, \mathsf{B}\}\rangle$ of the TCG.

The TDG on the corresponding DFA D will directly simulate the TCG on I. The state space $Q(D)$ is the configurations of S as well as additional counters for input tracking. The first $\forall$ turn runs S without input from the existential team, thus $q_0(D)$ is the result of immediately applying δ_S k times (or until termination) from its initial configuration. After that, both games check for termination in the same way (accept states of S are win states of existential team, reject for universal), then begin writing to S's tape or feeding bits into D. The only significant difference is that the existential moves O must be input to D in binary over $2\lfloor\log_2 |O|\rfloor$ rounds where the universal player's moves are ignored by D. The transition function δ_D simply writes the appropriate bits of the moves from $\forall, \exists_1, \exists_2$ onto the tape of the current configuration, and once everything is input then it updates the configuration by applying δ_S k times (or until termination).

$\Longrightarrow$ Consider an instance D of the TEAM DFA COMPUTATION GAME.

The TCG on the corresponding instance $I = \langle S, O, k, \Gamma\rangle$ will similarly be a direct simulation of the TDG. Using $k = 6$ and $\Gamma = O = \{0, 1\}$, the tape of S is just the cells for each input bit b_1, b_2, m_1, m_2 plus unused space at the end. Its state space simply augments $Q(D)$ with input reading states. The first k steps, S will be in $q_0(D)$ and move nowhere, but each following time S is simulated for k steps, starting at tape position 0, S will read each bit, applying δ_D to update its DFA state for each read (unless it has entered a final state), then just return to position 0.

At the start, TCG runs S for k steps, which does nothing. The termination check for each game is the same, as before, then each player will input their move onto the appropriate cell of the tape (in the same order in both games) then run S again, which will simulate the same inputs being given to D and updating its state. ◀

▶ **Corollary 8.** *The TEAM DFA GAME is undecidable.*

Proof. If TDG were decidable, then TCG would be decidable using f from Theorem 7 to get a homomorphic instance, but since TCG is undecidable [3], TDG cannot be either. ◀

A Muffin-Theorem Generator

Guangqi Cui
Montgomery Blair High School
bestwillcui@gmail.com

John Dickerson
Department of Computer Science and UMIACS, Univ of MD at College Park
john@cs.umd.edu

Naveen Durvasula
Montgomery Blair High School
140.naveen.d@gmail.com

William Gasarch
Department of Computer Science, Univ of MD at College Park
gasarch@cs.umd.edu

Erik Metz
Department of Mathematics, Univ of MD at College Park (ugrad)
emetz1618@gmail.com

Jacob Prinz
Department of Physics, Univ of MD at College Park (ugrad)
jacobeliasprinz@gmail.com

Naveen Raman
Richard Montgomery High School
nav.j.raman@gmail.com

Daniel Smolyak
Department of Computer Science (ugrad). Univ of MD at College Park
dsmolyak@gmail.com

Sung Hyun Yoo
Bergen County Academies (a High School)
sunnyyoo812@gmail.com

Abstract

Consider the following FUN problem. Given m, s you want to divide m muffins among s students so that everyone gets $\frac{m}{s}$ muffins; however, you want to maximize the minimum piece so that nobody gets crumbs. Let $f(m, s)$ be the size of the smallest piece in an optimal procedure.

We study the case where $\left\lceil \frac{2m}{s} \right\rceil = 3$ because (1) many of our hardest open problems were of this form until we found this method, (2) we have used the technique to *generate muffin-theorems*, and (3) we conjecture this can be used to solve the general case. We give (1) an algorithm to find an upper bound for $f(m, s)$ when $\left\lceil \frac{2m}{s} \right\rceil = 3$ (and some ways to speed up that algorithm if certain conjectures are true), (2) an algorithm that uses the information from (1) to try to find a lower bound on $f(m, s)$ (a procedure) which matches the upper bound, (3) an algorithm that uses the information from (1) to generate muffin-theorems, and (4) an algorithm that we think works well in practice to find $f(m, s)$ for any m, s.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization

Keywords and phrases Fair Division, Theorem Generation

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 15; pp. 15:1–15:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.15

Related Version `https://arxiv.org/abs/1709.02452`

Acknowledgements We thank Nancy Blachman who compiled the list of problems that introduced us to this problem, Alan Frank who first came up with the problem. We would also like to thank Alan Frank, James Propp, and Sam Zbarsky for stimulating discussions of the topic.

1 Introduction

Consider the following FUN problem. Given m, s you want to divide m muffins among s students so that everyone gets $\frac{m}{s}$ muffins; however, you want to maximize the minimum piece so that nobody gets crumbs. Let $f(m, s)$ be the size of the smallest piece in an optimal procedure.

We give an example:
You have 47 muffins and 36 students. You want to divide the muffins evenly, but no student wants a small piece. Find a protocol that maximizes the smallest piece. We show in Section 5 that there is a procedure for this with smallest piece $\frac{31}{90}$ *and that this is optimal. Hence* $f(47, 36) = \frac{31}{90}$.

Convention. When discussing a muffin being cut we refer to *pieces*. When discussing a student receiving we refer to *shares*. They are the same; however, it will be good to have different terminologies to focus on what's important. We treat a piece, a share, and its value as the same thing. So we may say *let* $x \geq \frac{1}{3}$ *be given to a student.*

▶ **Definition 1.** Let $m, s \in \mathbb{N}$. An (m, s)-*protocol* is a protocol to cut m muffins into pieces and then distribute them to the s students so that each student gets $\frac{m}{s}$ muffins. An (m, s)-protocol is *optimal* if it has the largest smallest piece of any protocol. $f(m, s)$ is the size of the smallest piece in an optimal (m, s)-protocol.

Clearly, for all $a \in \mathbb{N}$, $f(am, as) \geq f(m, s)$. All of our theorems indicate that $f(am, as) = f(m, s)$. We have not been able to prove this; however, we will only consider the cases where m, s are relatively prime.

We came upon this problem in a pamphlet *Julia Robinson Mathematics Festival: A Sample of Mathematical Puzzles* compiled by Nancy Blachman. On Page 2 was *The Muffin Puzzle* which asked about the problem for several particular cases. Nancy Blachman attributes the problem to Alan Frank and points out that it was described by Jeremy Copeland [3]. We are the first ones to consider this problem seriously for general m, s with one caveat: There was some discussion of this problem in the math-fun email list in 2009. We have obtained a copy of their arxives and discovered that they already had Theorem 3 and 11. We will credit the individuals when we get to those theorems.

Given m, s how hard is it to compute $f(m, s)$? Computing $f(m, s)$ can be rephrased as a mixed integer program on $O(ms)$ variables (the proof is in the Section A). Since the input is of size $O(\log m + \log s)$ this result does not even put the problem into NP. One of the upshots of this paper will be a procedure that we conjecture puts the computation of $f(m, s)$ into P.

We study the case where $\lceil \frac{2m}{s} \rceil = 3$ because (1) many of our hardest open problems were of this form until we found this method, (2) we have used the technique to *generate muffin-theorems*, (3) we conjecture this can be used to solve the general case.

We have a long paper [2] and some programs [1] for computing $f(m,s)$. For $1 \le s \le 50$, $1 \le m \le 60$ we have computed $f(m,s)$. In this paper we focus on a subset of the material that lends itself to generating theorems about muffins via an algorithm.

2 Summary of Results

In Sections 3,4 we give basic theorems and definitions used throughout the paper. In Section 5 we illustrate the *Buddy-Match techniques* by proving $f(47,36) \le \frac{31}{90}$. In Section 6 we illustrate how to obtain lower bounds and present the result $f(47,36) \ge \frac{31}{90}$.

In Sections 7 we discuss how to generate theorems from the Buddy-Match Technique. These theorems are of the form:

If $d \in \mathbb{N}$ and $1 \le a \le 3d-1$, a,d relatively primes, then

$$(\forall k \ge 1)\left[f(3dk+a+d, 3dk+a) \le \frac{dk+X}{3dk+a}\right]$$

where X is a constant which can depend on a,d but not on k. In Section 8 we discuss how to generate theorems that are more general. Here is an example:

If $1 \le a \le \frac{5d}{7}$ and $a \ne \frac{2d}{3}$ then $f(3dk+a+d, 3dk+a) \le \frac{X}{360}$ where $X = \max\{\frac{2a}{5}, \frac{a+d}{6}\}$.

In Sections 10, 11 we show how, assuming certain conjectures, one can speed up the Buddy-Match Technique. In Section 12 we give an algorithm that we conjecture puts $f(m,s)$ into P. In Section 13 we speculate about that algorithm and other muffin-issues.

In the appendix we state and sometimes prove theorems that are needed to fill in some of the gaps in our narrative. We also give some examples of the theorems we generated.

3 Basic Theorems

In this section we prove two theorems that will enable us, for the rest of the paper, to only consider m,s and protocols such that (1) $m > s \ge 3$, (2) s does not divide m, and (4) every muffin is cut into exactly two pieces.

The following theorem takes care of the cases $s=1$ and $s=2$. The proofs are easy and left to the reader.

▶ **Theorem 2.**

1. $(\forall m)[f(m,1)=1]$
2. $(\forall m)[m \equiv 0 \pmod 2 \rightarrow f(m,2)=1]$
3. $(\forall m)[m \equiv 1 \pmod 2 \rightarrow f(m,2)=\frac{1}{2}]$
4. $(\forall m,s)[s \textit{ divides } m \rightarrow f(m,s)=1]$.

The following theorem shows that if you know $f(m,s)$ then you know $f(s,m)$. Combined with Theorem 2 we need only consider $m > s \ge 3$. This theorem was independently discovered by Erich Friedman, within the math-fun email list, in 2009.

▶ **Theorem 3.** *Let $m,s \in \mathbb{N}$. Then $f(s,m) = \frac{s}{m} f(m,s)$.*

Proof. Assume $f(m,s) \ge \alpha$. We show $f(s,m) \ge \frac{s}{m}\alpha$. Let $M_1,\ldots,M_m$ be the muffins. Let $S_1,\ldots,S_s$ be the students. The protocol that achieves $f(m,s) \ge \alpha$ must be of the following form:

1. For each $1 \le i \le m$ divide M_i into pieces $(a_{i1}, a_{i2}, \ldots, a_{im_i})$ where $\sum_{j=1}^{m_i} a_{ij} = 1$.
2. For each $1 \le j \le s$ give S_j the shares $[b_{1j}, b_{2j}, \ldots, b_{s_j j}]$ where $\sum_{i=1}^{s_j} b_{ij} = \frac{m}{s}$.

The following hold:

- $\bigcup_{i=1}^{m} \bigcup_{j=1}^{m_i} \{a_{ij}\} = \bigcup_{j=1}^{s} \bigcup_{i=1}^{s_j} \{b_{ij}\}$
- The min over all of the a_{ij} is α.

The following protocol shows that $f(s,m) \geq \frac{s}{m}\alpha$. Let $M'_1, \ldots, M'_s$ be the muffins. Let $S'_1, \ldots, S'_m$ be the students.

1. For each $1 \leq j \leq s$ divide M'_j into $(\frac{s}{m}b_{1j}, \frac{s}{m}b_{2j}, \ldots, \frac{s}{m}b_{s_j j})$. Note that $\sum_{i=1}^{s_j} \frac{s}{m}b_{ij} = \frac{s}{m}\sum_{i=1}^{s_j} b_{ij} = \frac{s}{m} \times \frac{m}{s} = 1$.
2. For each $1 \leq i \leq m$ give S'_j $[\frac{s}{m}a_{i1}, \frac{s}{m}a_{ij}, \ldots, \frac{s}{m}a_{im_i}]$. Note that $\sum_{j=1}^{m_i} \frac{s}{m}a_{ij} = \frac{s}{m}\sum_{j=1}^{m_i} a_{ij} = \frac{s}{m} \times 1 = \frac{s}{m}$.

Clearly this is a correct protocol and the minimum piece is of size $\frac{s}{m}\alpha$.

We now show that $f(s,m) = \frac{s}{m}f(m,s)$. By the above we have both (1) $f(s,m) \geq \frac{s}{m}f(m,s)$, and (2) $f(m,s) \geq \frac{m}{s}f(s,m)$. Hence

$$f(s,m) \geq \frac{s}{m}f(m,s) \geq \frac{s}{m}\frac{m}{s}f(s,m) = f(s,m).$$

Therefore $f(s,m) = \frac{s}{m}f(m,s)$. ◀

▶ **Theorem 4.** *Let $m, s \in \mathbb{N}$.*

1. *If $f(m,s) \geq \alpha$ and $\alpha > \frac{1}{3}$ via protocol P then protocol P cuts every muffin into 1 or 2 pieces.*
2. *$f(m,s) \geq \alpha$ and $\alpha \leq \frac{1}{2}$ via protocol P then there is a protocol P' such that (1) P' also yields $f(m,s) \geq \alpha$, and (2) P' cuts every muffin into 2 or more pieces.*

Proof.

a) If any muffin is cut into ≥ 3 pieces then there is a piece $\leq \frac{1}{3} < \alpha$.

b) If any muffin is uncut and given to (say) Alice then we can add a step where we cut the muffin into $(\frac{1}{2}, \frac{1}{2})$ and give both $\frac{1}{2}$-sized pieces to Alice. Since $\alpha \leq \frac{1}{2}$ adding in some pieces of size $\frac{1}{2}$ does not affect the smallest piece. ◀

By Theorem 4 we have the following convention.

Convention: When trying to show that $f(m,s) \leq \alpha$ where $\frac{1}{3} < \alpha < \frac{1}{2}$ we will assume, by way of contradiction, that there is a protocol showing $f(m,s) > \alpha$ where every muffin is cut into exactly 2 pieces.

4 Basic Definitions

▶ **Definition 5.** Let $m, s \in \mathbb{N}$. Assume there is an (m,s)-protocol.

1. The two pieces that come from the same muffin are called *buddies*. $B(x)$ is the buddy of x. Note that $B(x) = 1 - x$.
2. A student that gets A shares is an *A-student*. A share given to an A-student is an *A-share*.
3. 2-Shares that are given to the same 2-student are *matched*. $M(x)$ is the match of 2-share x. Note that $M(x) = \frac{m}{s} - x$.
4. If x is a share given to a 3-student then $M_S(x)$ is the smallest share (not including x) that the student has, and $M_L(x)$ is the largest. Note that $M_S(x) \leq \frac{(m/s)-x}{2}$. Hence $B(M_S(x)) \geq 1 - \frac{(m/s)-x}{2}$.

Notation: (a,b) will mean the set of shares that have size strictly between a and b. Hence $|(a,b)|$ will be the number of such shares. We use similar notation for $[a,b]$.

5 An Example is Worth A Thousand Theorems: 43 muffins, 39 Students

The method we demonstrate in this section is called *The Buddy-Match Method.*

▶ **Theorem 6.** $f(47,36) \le \frac{31}{90} = \frac{124}{360}$.

Proof. To make the notation easier we write all fractions as having denominator 360.

Assume there is an $(47,36)$-procedure. We show that there is a piece $\le \frac{124}{360}$. Note that $\frac{47}{36} = \frac{470}{360}$.

Case 1: Some student gets ≥ 4 shares. Then some students has a share $\le \frac{47}{36\times 4} < \frac{124}{360}$.

Case 2: Some student gets ≤ 1 share. $1 < \frac{47}{36}$, so this is impossible.

Case 3: Every muffin is cut in 2 pieces and every student gets either 2 or 3 shares. The total number of shares is 94. Let s_2 (s_3) be the number of 2-students (3-students).

$$\begin{aligned} 2s_2 + 3s_3 &= \ 94 \\ s_2 + s_3 &= \ 36 \end{aligned}$$

So $s_2 = 14$ and $s_3 = 22$.

Case 3.1: There is a 2-share $x \le \frac{234}{360}$. $M(x) \ge \frac{470}{360} - \frac{234}{360} = \frac{236}{360}$ so $B(M(x)) \le 1 - \frac{236}{360} = \frac{124}{360}$

Case 3.2: There is a 3-share $x \ge \frac{222}{360}$. $B(M_S(x)) \le 1 - \frac{\frac{470}{360} - \frac{222}{360}}{2} = \frac{124}{360}$.

Case 3.3: There is a 2-share $x \ge \frac{236}{360}$. $B(x) \le 1 - \frac{236}{360} = \frac{124}{360}$

Case 3.4: There is a 3-share $x \le \frac{124}{360}$. This one is self-explanatory.

Case 3.5: All 3-shares are in $(\frac{124}{360}, \frac{222}{360})$ and all 2-shares are in $(\frac{234}{360}, \frac{236}{360})$.

The following picture captures what we know so far.

(	− − −	)[	− − −	](	− − −	)
$\frac{124}{360}$	3-shs	$\frac{222}{360}$	No shs	$\frac{234}{360}$	2-shs	$\frac{236}{360}$

Since there are no shares in $[\frac{222}{360}, \frac{234}{360}]$, there are no shares in $B([\frac{222}{360}, \frac{234}{360}]) = [\frac{126}{360}, \frac{138}{360}]$ The following picture captures what we know so far.

(	− − −	)[	− − −	](	− − −	)[	− − −	](	− − −	)
$\frac{124}{360}$	S3-shs	$\frac{126}{360}$	No shs	$\frac{138}{360}$	L3-shs	$\frac{222}{360}$	No shs	$\frac{234}{360}$	2-shs	$\frac{236}{360}$

S3-shs stands for *short 3-shares* and L3-shs stands for *large 3-shares*. There are $2s_2 = 28$ 2-shares so there are 28 S3-shares (B is a bijection between 2-shares and S3-shares). Since there are $3s_3 = 66$ 3-shares total that leaves 38 S3 shares.

Since the midpoint of L3-shs is $\frac{360}{2}$, the Buddy function is a bijection from $(\frac{138}{360}, \frac{180}{360})$ to $(\frac{180}{360}, \frac{222}{360})$, Hence these two intervals have the same number of shares.

Since the midpoint of 2-shs is $\frac{470}{2}$, the Match function is a bijection from $(\frac{234}{360}, \frac{235}{360})$ to $(\frac{235}{360}, \frac{236}{360})$. Hence these two intervals have the same number of shares. Applying the Buddy function to both these intervals we obtain that $(\frac{124}{360}, \frac{125}{360})$ and $(\frac{125}{360}, \frac{126}{360})$ have the same number of shares.

In the scenarios above there are an even number of shares of size the midpoint. We arbitrarily assign half to the left and half to the right.

We define the following intervals.

▶ **Definition 7.**

1. $I_1 = (\frac{124}{360}, \frac{125}{360})$
2. $I_2 = (\frac{125}{360}, \frac{126}{360})$ $(|I_1| = |I_2|,\ |I_1 \cup I_2| = 28)$
3. $I_3 = (\frac{138}{360}, \frac{180}{360})$
4. $I_4 = (\frac{180}{360}, \frac{222}{360})$ $(|I_3| = |I_4|,\ |I_3 \cup I_4| = 38)$

Henceforth all of the students considered will be 3-students. We now look at the students in a more detailed way than 2-students and 3-students.

▶ **Definition 8.** Let $1 \le i_1 \le \cdots \le i_3 \le 4$. An $e(i_1, i_2, i_3)$-student is a student who has, for each $1 \le j \le 3$, a share in I_{i_j}. For example, an $e(1,1,4)$-students has two shares in I_1 and one share in I_4.

▶ **Claim 1.**

1. *The only possible students are:*
   a. $e(1,1,4)$
   b. $e(1,2,4)$
   c. $e(1,3,3)$
   d. $e(1,3,4)$
   e. $e(2,2,4)$
   f. $e(2,3,3)$
   g. $e(2,3,4)$
   h. $e(3,3,3)$
   i. $e(3,3,4)$
2. *There are no shares in* $[\frac{208}{360}, \frac{218}{360}]$
3. *There are no shares in* $[\frac{142}{360}, \frac{152}{360}]$ *(this follows from the prior part and buddying).*

Proof of Claim 1.

1) We establish that some students are impossible.
   A $e(1,4,4)$-student has more than $\frac{124}{360} + 2 \times \frac{180}{360} = \frac{484}{360}$
   A $e(2,2,3)$-student has less than $2 \times \frac{126}{360} + \frac{180}{360} = \frac{432}{360}$
   The result follows from these two statements, though the proof is tedious.
2) We look at which I_4-shares are used
   A $e(1,1,4)$ student uses I_4-share $> \frac{470}{360} - 2 \times \frac{125}{360} = \frac{220}{360}$
   A $e(1,2,4)$ student uses I_4-shares $> \frac{470}{360} - \frac{125}{360} - \frac{126}{360} = \frac{219}{360}$
   A $e(1,3,4)$ student uses I_4-shares $< \frac{470}{360} - \frac{124}{360} - \frac{138}{360} = \frac{208}{360}$
   A $e(2,2,4)$ student uses I_4-shares $> \frac{470}{360} - 2 \times \frac{126}{360} = \frac{218}{360}$
   A $e(2,3,4)$ student uses I_4-shares $< \frac{470}{360} - \frac{125}{360} - \frac{138}{360} = \frac{207}{360}$
   A $e(3,3,4)$ student uses I_4-shares $< \frac{470}{360} - 2 \times \frac{138}{360} = \frac{194}{360}$
   Hence the only shares in I_4 that can be used are those $< \frac{208}{360}$ or $> \frac{218}{360}$. The result follows. ◀

We redefine the intervals.

▶ **Definition 9.**

1. $I_1 = (\frac{124}{360}, \frac{125}{360})$
2. $I_2 = (\frac{125}{360}, \frac{126}{360})$ $(|I_1| = |I_2|), |I_1 \cup I_2| = 28)$
3. $I_3 = (\frac{138}{360}, \frac{142}{360})$
4. $I_4 = (\frac{152}{360}, \frac{180}{360})$
5. $I_5 = (\frac{180}{360}, \frac{208}{360})$ $(|I_4| = |I_5|)$
6. $I_6 = (\frac{218}{360}, \frac{222}{360})$ $(|I_3| = |I_6|,\ |I_3 \cup I_4 \cup I_5 \cup I_6| = 38)$

By a proof similar to that of Claim 1 we obtain the following:

▶ **Claim 2.**

1. *The only possible students are:* $e(1,1,6)$, $e(1,2,6)$, $e(1,3,5)$, $e(1,4,4)$, $e(1,4,5)$, $e(2,2,6)$, $e(2,3,5)$, $e(2,4,4)$, $e(2,4,5)$, $e(3,3,5)$, $e(3,4,4)$, *and* $e(4,4,4)$.
2. *There are no shares in* $[\frac{194}{360}, \frac{202}{360}]$
3. *There are no shares in* $[\frac{158}{360}, \frac{166}{360}]$ *(this follows from the prior part and buddying).*

We define the following intervals.

▶ **Definition 10.**

1. $I_1 = (\frac{124}{360}, \frac{125}{360})$
2. $I_2 = (\frac{125}{360}, \frac{126}{360})$ $(|I_1| = |I_2|, |I_1 \cup I_2| = 28)$
3. $I_3 = (\frac{138}{360}, \frac{142}{360})$
4. $I_4 = (\frac{152}{360}, \frac{158}{360})$
5. $I_5 = (\frac{166}{360}, \frac{180}{360})$
6. $I_6 = (\frac{180}{360}, \frac{194}{360})$ $(|I_5| = |I_6|)$
7. $I_7 = (\frac{202}{360}, \frac{208}{360})$ $(|I_4| = |I_7|)$
8. $I_8 = (\frac{218}{360}, \frac{222}{360})$ $(|I_3| = |I_8|, |I_3 \cup \cdots \cup I_8| = 38)$

By a proof similar to that of Claim 1 we obtain:

▶ **Claim 3.** *The only possible students are:* $e(1,1,8)$, $e(1,2,8)$, $e(1,3,7)$, $e(1,4,6)$, $e(1,5,5)$, $e(2,2,8)$, $e(2,3,7)$, $e(2,4,6)$, $e(2,5,5)$, $e(3,3,6)$, *and* $e(4,4,4)$.

Let

1. $|e(1,1,8)| = a$
2. $|e(1,2,8)| = b$
3. $|e(1,3,7)| = c$
4. $|e(1,4,6)| = d$
5. $|e(1,5,5)| = e$
6. $|e(2,2,8)| = f$
7. $|e(2,3,7)| = g$
8. $|e(2,4,6)| = h$
9. $|e(2,5,5)| = i$
10. $|e(3,3,6)| = j$
11. $|e(4,4,4)| = k$

Since $|I_1| = |I_2|$, $2a+b+c+d+e = b+2f+g+h+i$, *so* $2a+c+d+e = 2f+g+h+i$
Since $|I_3| = |I_8|$, $c+g+2j = a+b+f$
Since $|I_4| = |I_7|$, $d+h+3k = c+g$
Since $|I_5| = |I_6|$, $2e+2i = d+h+j$
Since $|I_1 \cup I_2| = 28$, $2a+2b+c+d+e+2f+g+h+i = 28$
Since there are 22 3-students, $a+b+c+d+e+f+g+h+i+k = 22$
From the last two equations we obtain $a+b+f = 6$
We combine I_1 *and* I_2 *into a single interval. This reduces the system to 6 variables, resulting in the equation*

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 1 & 1 & 1 & 0 & 0 \\ -1 & 1 & 0 & 0 & 2 & 0 \\ 0 & -1 & 1 & 0 & 0 & 3 \\ 0 & 0 & -1 & 2 & -1 & 0 \end{bmatrix} \begin{bmatrix} p \\ q \\ r \\ s \\ t \end{bmatrix} = \begin{bmatrix} 22 \\ 28 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

However, one can check that eliminating the bottom 3 rows requires the top 2 rows to be in the ratio 7 : 9. 22 : 28 $\neq$ 7 : 9, so there is no solution. ◀

The above proof used that $\lceil \frac{2m}{s} \rceil = 3$ since that is the condition that leads to having 2-shares and 3-shares. This is usually important since it gives us symmetry from matches, not just from buddying; however, in this case we just so happened to not need that symmetry.

6 Finding a Procedure

We now describe the program that finds the procedure showing $f(47, 36) \geq \frac{124}{360}$. We *guess* that all shares are of the form $\frac{x}{360}$ where $124 \leq x \leq 236$. But we can cut down those variables *a lot* based on the proof. For example, by modifying the proof slightly, we can deduce that there are no share of size $\frac{127}{360}, \frac{128}{360}, \ldots, \frac{137}{360}$. This is a key factor in speeding up the program. We can also use the symmetries of where shares can be.

For every way to split a muffin we have a variable for how many muffins are split that way, as follows: $(\frac{124}{360}, \frac{236}{360})$ is associated to the variables $y_{124,236}$, $(\frac{125}{360}, \frac{235}{360})$ is associated with the variable $y_{125,235}$, etc. This variable is the *number of muffins* that are split that way.

For every way to give muffin shares to a student we have a variable for how many students get that set of shares, as follows: $[\frac{87}{360}, \frac{79}{360}, \frac{69}{360}]$ is associated to the variable $z_{87,79,69}$, $[\frac{118}{360}, \frac{117}{360}]$ is associated to the variables $z_{118,117}$, etc. This variable is the *number of students* who get that share-size.

For each size we express how many pieces are of that size in two ways.

- The number of pieces of that size based on the muffins. For example, the number of pieces of size $\frac{131}{360}$ is $y_{131,256}$. The number of pieces of size $\frac{180}{360}$ is $2 \times y_{180,180}$.
- The number of shares of that size based on the students. For example, the number of shares of size $\frac{131}{360}$ is

$$z_{124,131,215} + \cdots + z_{130,131,209} + 2z_{131,131,208} + z_{132,131,207} + \cdots + z_{215,131,124}$$

For each size we get an equation by equating the muffin-based and student-based expressions. We have more equations based on the number of pieces and the number in each interval which falls out of the proof of the upper bound. This leads to a set of linear equations whose solution leads to a procedure.

Here is the procedure for $f(47, 36) \geq \frac{124}{360} = \frac{117}{180}$ we obtained with this method:

1. Divide 1 muffin $(\frac{90}{180}, \frac{90}{180})$
2. Divide 2 muffins $(\frac{93}{180}, \frac{87}{180})$
3. Divide 2 muffins $(\frac{101}{180}, \frac{79}{180})$
4. Divide 2 muffins $(\frac{104}{180}, \frac{76}{180})$
5. Divide 6 muffins $(\frac{109}{180}, \frac{71}{180})$
6. Divide 6 muffins $(\frac{111}{180}, \frac{69}{180})$
7. Divide 14 muffins $(\frac{117}{180}, \frac{63}{180})$
8. Divide 14 muffins $(\frac{118}{180}, \frac{62}{180})$
9. Give 2 students $[\frac{87}{180} \frac{79}{180} \frac{69}{180}]$
10. Give 2 students $[\frac{90}{180} \frac{76}{180} \frac{69}{180}]$
11. Give 2 students $[\frac{93}{180} \frac{71}{180} \frac{71}{180}]$
12. Give 2 students $[\frac{101}{180} \frac{71}{180} \frac{63}{180}]$
13. Give 2 students $[\frac{104}{180} \frac{69}{180} \frac{62}{180}]$
14. Give 6 students $[\frac{109}{180} \frac{63}{180} \frac{63}{180}]$
15. Give 6 students $[\frac{111}{180} \frac{62}{180} \frac{62}{180}]$
16. Give 14 students $[\frac{118}{180} \frac{117}{180}]$

The reader should be able to see how to generalize the method outlined above.

What is described above is not quite what we have coded up (though we will). The Interval Method (see Section B) is another method to find lower bounds that gives information that can be used to cut down the time to find a procedure. We have coded up a version of what is outlined above with the interval method.

We denote the algorithm given above (the one using Buddy-Match) VLOWER(m, s, α) where one finds a procedure showing $f(m, s) \ge \alpha$, hence verifying that $f(m, s) \ge \alpha$.

7 The Proof that $f(47, 36) \le \frac{31}{90}$ Reveals Much More

The proof that $f(47, 36) \le \frac{31}{90}$ can be modified very slightly (just notation) to obtain the following result (which we write in a strange way for later exposition):

$$(\forall k \ge 1)\left[f(3 \times 11 \times k + 11 + 3, 33k + 3) \le \frac{11k + \frac{7}{5}}{3 \times 11 \times 3k + 3}\right]$$

More generally the following seems to be true empirically:
for all d (d stands for difference and is $m - s$), for all $1 \le a \le 3d - 1$ (a, d relatively primes), there exists X:

$$(\forall k \ge 1)\left[f(3dk + d + a, 3dk + a) \le \frac{dk + X}{3dk + a}\right]$$

For $d = 1$ to 8, for all relevant a, we have found X. In many concrete cases we have shown that it is also an upper bound. In Section C we present the results for the $d = 7$ case.

Note that we need $k \ge 1$ since if $k = 0$ then we no longer have $\lceil \frac{2m}{s} \rceil = 3$.

8 Generating More General Theorems

The techniques discussed in Section 7 generate theorems of the form

$$(\forall k \ge 1)\left[f(3dk + a + d, 3dk + a) \le \frac{dk + X}{3dk + a}\right].$$

However, the program can be modified to obtain more general theorems. As noted in Section 7 our program finds interesting values of X. That is, the program may find that (say) if $X \le \frac{7}{6}$ then there are no $e(1, 3, 4)$-students. What is it about $X \le \frac{7}{6}$ that makes this happen? It may be that (say) $1 \le a \le \frac{5d}{7}$ and $a \ne \frac{2d}{3}$ makes this work, and it may be that $X = \max\{\frac{2a}{5}, \frac{a+d}{6}\}$.

We have taken the results from the program and, with the help of additional programs and our own ingenuity generated many theorems (we hope to fully automate it soon). These theorems are a great time saver since often the result we want falls out of them directly. We present a sample of such theorems in the Section D.

9 How to find X

The proof of Theorem 6 can be summarizes as follows: The assumption $f(47, 36) > \frac{31}{90}$ implies that a certain system of linear equations have a solution where all of the variables are natural numbers between 0 and $s_3 = 22$. The system had no such solution, hence a contradiction.

Imagine that we want an upper bound on $f(47, 36)$ but do not know what it is ahead of time. Following the line of reasoning in Section 7 we seek X such that

$$f(33 + 3 + 11, 33 + 3) \le \frac{11 + X}{33 + 3}.$$

We use a program to simulate the proof of Theorem 6 but with X instead of the actual numbers. This program will produce many values of X where something interesting happens, such as a type of student no longer being allowed. The program looks at the (finite) set of interesting values of X and finds the least one that causes the resulting system of linear equations to be unsolvable using natural numbers between 0 and 22. Hence we have a value of X. We then use VLOWER$(47, 36, \frac{11+X}{36})$ to find the matching lower bound (if this does not work then the algorithm failed to find $f(m, s)$).

For the values $47, 36$ it was easy to find the value of X. For larger m, s it may be that verifying $f(m, s) \le \alpha$ is faster than finding the α. In the next two sections we examine how to speed up finding X.

We leave it to the reader to generalize the algorithm to any m, s where $\lceil \frac{2m}{s} \rceil = 3$; however, we give the following picture which represents intervals where 3-shares can be. In the picture each nonempty interval has the number of 3-shares in it (though y is not known) and a label such as I_1 so we can refer to it. This picture is the result of many buddy-match sequences.

$$\begin{array}{ccccccc}
(& a+d & (I_1) & | & a+d & (I_2) &)[\quad 0 \quad] \\
\frac{dk+X}{3dk+a} & & & \frac{dk+\frac{a}{2}}{3dk+a} & & & \frac{dk+a-X}{3dk+a} \qquad \frac{dk+2X}{3dk+a}
\end{array}$$

$$\begin{array}{ccccc}
(& y \quad (I_3) &)[& 0 &] \\
\frac{dk+2X}{3dk+a} & & \frac{dk+a+d-3X}{3dk+a} & & \frac{dk+d-a+2X}{3dk+a}
\end{array}$$

$$\begin{array}{ccccc}
(& 2d-a-y \; (I_4) & | & 2d-a-y \; (I_5) &) \\
\frac{dk+d-a+2X}{3dk+a} & & \frac{dk+\frac{a+d}{2}}{3dk+a} & & \frac{dk+2a-2X}{3dk+a}
\end{array}$$

$$\begin{array}{ccccc}
)[& 0 &](& y \; (I_6) &) \\
\frac{dk+2a-2X}{3dk+a} & & \frac{dk+3X}{3dk+a} & & \frac{dk+a+d-2X}{3dk+a}
\end{array}$$

Facts and Caveats:

1. $|I_1| = |I_2|$
2. $|I_4| = |I_5|$
3. In the picture it is unclear if the endpoint of I_1 is included in I_1. We do not include it; however, we take the even number of shares that are at that endpoint and arbitrary assign half to I_1 and half to I_2.
4. There is a similar comment for I_2, I_4, and I_5.

We denote the version where you do not already have upper bound to check BUDMAT(m, s) and the version where you do BUDMAT(m, s, α) where α is the bound. We will avoid using BUDMAT(m, s) unless m, s are small since it may be slow.

10 How to find X Cheating a Little

Say you want to find $f(213, 200)$. Since $\lceil \frac{2 \times 213}{200} \rceil = 3$ you could run BUDMAT$(213, 200)$. But the numbers are large! Following the line of reasoning in Section 7 we note that $d = 213 - 200 = 13$ and generalize the problem to finding an X such that

$$f(39k + 5 + 13, 39k + 5) \le \frac{13k + X}{39k + 5}.$$

Lets look at the $k=1$ case: $f(57,44)$. Since $\lceil \frac{2\times 57}{44} \rceil = 3$ you could run BUDMAT$(57,44)$. But the numbers are small! Oh, thats a good thing! Lets say the answer is α. Run VLOWER$(57,44,\alpha)$ to verify that its a lower bound. If it is then solve $\alpha = \frac{13+X}{39+5}$ to find X. The proof you did for $f(57,44) \le \frac{13+X}{39+5}$ can be modified to show $(\forall k \ge 1)[f(39k+5+13, 39k+5) \le \frac{13k+X}{39k+5}]$. In particular $f(213,200) \le \frac{13\times 5+X}{39\times 5+5} = \beta$. Run VLOWER$(213,300,\beta)$ to verify the lower bound (if this does not work then the algorithm failed to find $f(57,44)$).

This is cheating a little since we don't really know that the such an X exists. But it has so far. And we do verify in the end.

We leave it to the reader to generalize this procedure. We denote this algorithm CHEATALITTLE(m,s).

11 How to find X Cheating a Lot

Say you want to find $f(1717,1650)$. Since $\lceil \frac{2\times 1717}{1650} \rceil = 3$ you could run BUDMAT$(1717,1650)$. But the numbers are really large! Following the line of reasoning in Section 7 we note that $d = 1717 - 1650 = 67$ and generalize the problem to finding an X such that

$$f(201k+42+67, 201k+42) \le \frac{67k+X}{201k+42}.$$

Lets look at the $k=1$ case: $f(310,243)$. These numbers are still big!

Lets look at the $k=0$ case: $f(109,42)$. These numbers are small! Since $\lceil \frac{2\times 109}{42} \rceil \ge 4$ you cannot run BUDMAT$(109,42)$). But the situation is worse than that. Even if we bound $f(109,42)$ the proof will not use BUDMAT and hence cannot be modified to get an upper bound for $f(201k+42+67, 201k+42)$. In fact, the answer for $f(109,42)$ should have no bearing on our problem.

Except for one thing. Empirically it does. In all cases that we looked at the X obtained from knowing an upper bound on the $k=0$ case of $f(3dk+a+d, 3dk+a)$ was the correct X for $k \ge 1$. We proceed as if this is always true.

We cannot use BUDMAT$(109,42)$; however, there are other techniques that to find an upper bound on $f(m,s)$. They summarized in Section B. Use them. Lets say the answer is α. Run VLOWER$(109,42,\alpha)$ to verify that its a lower bound. If it is then solve $\alpha = \frac{X}{42}$ to find X. The proof you did for $f(109,42) \le \frac{X}{42}$ *cannot* be modified to show $(\forall k \ge 1)[f(201k+42+67, 201k+42) \le \frac{67k+X}{201+42}]$. But you have a very good conjecture. Run BUDMAT$(109,42,\frac{67+X}{201+42})$. If it returns YES and a proof then modify the proof to obtain $(\forall k \ge 1)[f(201k+42+67, 201k+42) \le \frac{67k+X}{201+42}]$ (if this does not work then the algorithm failed to find $f(1717,1650)$). In particular $f(1717,1658) \le \frac{67\times 5+X}{201\times 5+5} = \beta$. Run VLOWER$(1717,1658,\beta)$ to verify the lower bound (if this does not work then the algorithm failed to find $f(1717,1650)$).

This is cheating a lot since we don't really know that the $k=0$ case has any bearing on the $k \ge 1$ case. But it has so far, and we verify in the end.

We leave it to the reader to generalize this procedure. We denote this algorithm CHEATALOT(m,s).

12 A General Algorithm

We present an algorithm that we conjecture always finds $f(m,s)$ and operates in polynomial time.

The reader should read Section B since we will be using FC, INT, and BUD which are explained there. They are other methods to find or verify upper bounds on $f(m,s)$.

1. Input(m, s).
2. If $m = s$ output 1. If $gcd(m, s) = d \geq 1$ then call the algorithm recursively with $f(m/d, s/d)$. If $s = 2$ then output $\frac{1}{2}$. If $m < s$ then call the algorithm recursively to find $f(s, m)$ and output $\frac{m}{s} f(s, m)$.
3. Compute $\alpha = \text{FC}(m, s)$. Compute VLOWER(m, s, α) to see if α is a matching lower bound. If it is then output α and stop.
4. Compute $\alpha = \text{INT}(m, s)$. Compute VLOWER$(m, s, \alpha)$ to see if α is a matching lower bound. If it is then output α and stop.
5. If $\lceil \frac{2m}{s} \rceil = 3$ then:
   a. Compute $\alpha = \text{CHEATALOT}(m, s)$. Compute VLOWER$(m, s, \alpha)$ to see if α is a matching lower bound. If it is then output α and stop. (This might fail if the methods of Section B do not work on the input they are given.)
   b. Compute $\alpha = \text{CHEATALITTLE}(m, s)$. Compute VLOWER$(m, s, \alpha)$ to see if α is a matching lower bound. If it is then output α and stop.
6. If $\lceil \frac{2m}{s} \rceil \geq 4$ then let $a = s$ and $d = m - a$. We seek $f(3d \times 0 + a + d, 3d \times 0 + a)$. Recursively call $f(3d + a + d, 3d + a)$ (we could tell it to not bother with CHEATALOT(m, s) since that just asks to compute $f(a + d, a)$ using FC and INT). If the computation succeeds and returns α then run BUD(m, s, α) to verify that $f(m, s) \leq \alpha$. If this is verified then compute VLOWER(m, s, α) to see if α is a matching lower bound. If it is then output α and stop.
7. If nothing above works then output **FAILED!**

This can be sped up by, upon first seeing m, s, see if any of the general theorems such as those in Sections C and D apply to get an upper bound α and then run VLOWER(m, s, α).

13 Open Problems and Speculation

We would like to think that the algorithm in the last section will always work and hence computing $f(m, s)$ is in P. But we've been down this road before where we think we can compute all $f(m, s)$ only to come to a troublesome case which leads to a new technique and more co-authors. The following are possible outcomes: (1) we prove that the algorithm always works, (2) we keep running the algorithm and it always works but when the numbers get too big we can't tell, (3) we come across a value the algorithm does not work on and this leads to a a new technique and more co-authors.

We believe that computing $f(m, s)$ is in P. One piece of evidence for this is that for all s, for all $m \geq s^3$, $f(m, s) = \text{FC}(m, s)$. Hence if you fix s then for large enough s the problem is *very easy*. One might call this Fixed Parameter *very tractable*.

We believe that $f(m, s)$ only depends on $\frac{m}{s}$. This seems provable.

References

1 Guangiqi Cui, John Dickerson, Naveen Durvasula, William Gasarch, Erik Metz, Jacob Prinz, Naveen Raman, Daniel Smolyak, and Sung Hyun Yoo. Code for muffin problems, 2017. `https://github.com/jeprinz/MuffinProblem`.

2 Guangiqi Cui, John Dickerson, Naveen Durvasula, William Gasarch, Erik Metz, Jacob Prinz, Naveen Raman, Daniel Smolyak, and Sung Hyun Yoo. The muffin problem, 2017. `https://arxiv.org/abs/1709.02452`.

3 Alan Frank. The muffin problem, 2013. Described to Jeremy Copeland and in the New York Times Numberplay Online Blog `wordplay.blogs.nytimes.com/2013/08/19/cake`.

A A Mixed Integer Program for $f(m, s)$

The following theorem shows that $f(m, s)$ always exists (as opposed to having better and better algorithms), is rational, and is computable. This theorem was independently discovered by Veit Elser, within the math-fun email list, in 2009.

▶ **Theorem 11.** *Let $m, s \geq 1$.*

1. *There is a mixed integer program with $O(ms)$ binary variables, $O(ms)$ real variables, $O(ms)$ constraints, and all coefficients integers of absolute value $\leq \max\{m, s\}$ such that, from the solution, one can extract $f(m, s)$ and a protocol that achieves this bound. This MIP can easily be obtained given m, s.*
2. *$f(m, s)$ is always rational. This follows from part 1.*
3. *In every optimal protocol for m muffins and s students all of the pieces are of rational size. This follows from part 1.*
4. *The problem of, given m, s, determine $f(m, s)$, is decidable. This follows from part 1.*

Proof. Consider the following (failed) attempt to solve the problem using linear programming.

1. The variables are x_{ij} where $1 \leq i \leq m$ and $1 \leq j \leq s$. The intent is that x_{ij} is the fraction of muffin i that student j gets.
2. For all $1 \leq i \leq m$, $1 \leq j \leq s$, $0 \leq x_{ij} \leq 1$.
3. For each $1 \leq i \leq m$, $\sum_{j=1}^{s} x_{ij} = 1$.
   This says that the amount of muffin i that student 1 gets, students 2 gets, . . ., student s gets all adds up to 1.
4. For each $1 \leq j \leq s$, $\sum_{i=1}^{m} x_{ij} = \frac{m}{s}$.
   This says that the amount that student j gets from muffin 1, muffin 2, . . ., muffin m all adds up to $\frac{m}{s}$.
5. For all $1 \leq i \leq m$, $1 \leq j \leq s$, $x_{ij} \geq z$.
6. Maximize z.

This does not work. The problem is that (say) x_{13} could be 0. In fact it is likely that some x_{ij} is 0. This makes $z = 0$. What we really want is

$$x_{ij} \neq 0 \implies x_{ij} \geq z$$

It is easy to show that $f(m, s) \geq \frac{1}{s}$. Hence every nonzero x_{ij} is $\geq \frac{1}{s}$. We will use this in our proof.

For $1 \leq i \leq m$, $1 \leq j \leq s$ modify the linear program above as follows.

1. Add variable y_{ij} which is in $\{0, 1\}$.
2. Add the constraint $x_{ij} + y_{ij} \leq 1$. Note that
   - $x_{ij} = 0 \implies x_{ij} + y_{ij} \leq 1$, so the constraint imposes no condition on y_{ij}.
   - $x_{ij} > 0 \implies y_{ij} < 1 \implies y_{ij} = 0 \implies x_{ij} + y_{ij} = x_{ij}$.
3. Add the constraint $x_{ij} + y_{ij} \geq \frac{1}{s}$. Note that
   - $x_{ij} = 0 \implies y_{ij} \geq \frac{1}{s} \implies y_{ij} = 1 \implies x_{ij} + y_{ij} = 1$
   - $x_{ij} > 0 \implies x_{ij} \geq \frac{1}{s}$ (since we know all non-zero pieces are $\geq \frac{1}{s}$) $\implies x_{ij} + y_{ij} \geq \frac{1}{s}$, so the constraint imposes no condition on y_{ij}.
4. Replace the constraint $z \leq x_{ij}$ with $z \leq x_{ij} + y_{ij}$.

If $x_{ij} = 0$ then the constraint

$$z \leq x_{ij} + y_{ij} = 1$$

is always met and hence is (as it should be) irrelevant. If $x_{ij} > 0$ then the constraint

$$z \le x_{ij} + y_{ij} = x_{ij}$$

is the constraint we want.

Solve the resulting mixed integer program. Since all of the coefficients are rational the answer will be rational. ◀

B Other Methods

We discuss three methods for finding an upper bound on $f(m,s)$.

The method from the following theorem is called *The Floor Ceiling Method* or just FC-method. Note that it is very fast and gives you the upper bound.

▶ **Theorem 12.** *Assume that $m, s \in \mathbb{N}$ and $\frac{m}{s} \notin \mathbb{N}$.*

$$f(m,s) \le \max\left\{\frac{1}{3}, \min\left\{\frac{m}{s\lceil 2m/s\rceil}, 1 - \frac{m}{s\lfloor 2m/s\rfloor}\right\}\right\}.$$

Proof. Assume we have an optimal (m,s) protocol. Since $\frac{m}{s} \notin \mathbb{N}$ we can assume every muffin is cut into at least 2 pieces.
Case 1: Some muffin is cut into $u \ge 3$ pieces. Then some piece is $\le \frac{1}{3}$.
Case 2: All muffins are cut into 2 pieces.
Since there are $2m$ shares and s students both of the following happen:

- Some student gets $t \ge \lceil 2m/s\rceil$ shares, so some share is $\le \frac{m}{s\lceil 2m/s\rceil}$.
- Some student gets $t \le \lfloor 2m/s\rfloor$ shares, so some share x is $\ge \frac{m}{s\lfloor 2m/s\rfloor}$. $B(x)) \le 1 - \frac{m}{s\lfloor 2m/s\rfloor}$.

Putting together Cases 1 and 2 yields the theorem. ◀

We denote the function from Theorem 12 FC(m,s).

The other two methods are to long to describe fully here so we just sketch.

The *Interval Method* is a primitive version of the Buddy-Match method where we do not use symmetry and (since we have shares other than 2-shares and 3-shares) cannot use the Match in Buddy-Match. This method is fast and can be used to derive the answer. We denote the result INT(m,s).

The *Buddy Method* is like the Buddy-Match Method only we do not use the Match part since we have shares other than 2-shares and 3-shares. And like the Buddy-Match Method this one is faster if you already have the answer. We denote the version where you do not already an upper bound to check BUD(m,s) and the version where you do BUD(m,s,α) where α is the bound.

C Everything You Ever Wanted to Know About $f(s+7,s)$

By either cheating a little (Section 10) or cheating a lot (Section 11) we have obtained formulas for $f(3dk+a+d, 3dk+a)$ for $1 \le d \le 50$ and $1 \le a \le 3d-1$ (a,d relatively primes). We present the results for $d=7$. Note that for most of the formulas the formula which is supposed to only hold for $k \ge 1$ also holds for $k=0$ (with a different proof).

▶ **Theorem 13.**

1. a. $f(8,1) = 1$. *For all $k \ge 1$, $f(21k+8, 21k+1) \le \frac{7k+X}{21k+1}$ where $X = \frac{1}{2}$.*

b. *For all $k \ge 0$, $f(21k+9, 21k+2) \le \frac{7k+X}{21k+2}$ where $X = 1$.*

2. *For all* $k \geq 0$, $f(21k+10, 21k+3) \leq \frac{7k+X}{21k+3}$ *where* $X = \frac{4}{3}$.
3. *For all* $k \geq 0$, $f(21k+11, 21k+4) = \frac{7k+X}{21k+4}$ *where* $X = \frac{9}{5}$.
4. *For all* $k \geq 0$, $f(21k+12, 21k+5) \leq \frac{7k+X}{21k+5}$ *where* $X = 2$.
5. *For all* $k \geq 0$, $f(21k+13, 21k+6) \leq \frac{7k+X}{21k+6}$ *where* $X = \frac{13}{5}$.
6. *For all* $k \geq 0$, $f(21k+15, 21k+8) \leq \frac{7k+X}{21k+8}$ *where* $X = 3$.
7. *For all* $k \geq 0$, $f(21k+16, 21k+9) \leq \frac{7k+X}{21k+9}$ *where* $X = \frac{11}{3}$.
8. *For all* $k \geq 0$, $f(21k+17, 21k+10) \leq \frac{7k+X}{21k+10}$ *where* $X = 4$.
9. *For all* $k \geq 0$ $f(21k+18, 21k+11) \leq \frac{7k+X}{21k+11}$ *where* $X = \frac{9}{2}$.
10. *For all* $k \geq 0$ $f(21k+19, 21k+12) \leq \frac{7k+X}{2ak+12}$ *where* $X = \frac{19}{4}$.
11. *For all* $k \geq 0$ $f(21k+20, 21k+13) \leq \frac{7k+X}{21k+13}$ *where* $X = 5$.
12. *For all* $k \geq 0$: $f(21k+22, 21k+15) = \frac{1}{3}$,
13. *For all* $k \geq 0$: $f(21k+23, 21k+16) = \frac{1}{3}$,
14. *For all* $k \geq 0$: $f(21k+24, 21k+17) = \frac{1}{3}$,
15. *For all* $k \geq 0$: $f(21k+25, 21k+18) = \frac{1}{3}$,
16. *For all* $k \geq 0$: $f(21k+26, 21k+19) = \frac{1}{3}$,
17. *For all* $k \geq 0$: $f(21k+27, 21k+20) = \frac{1}{3}$.

Note that the last few answers were $\frac{1}{3}$ and there is an equality. The $\frac{1}{3}$ follows from Theorem 14. The equality holds since we have proven that, for all $m > s$, $f(m,s) \geq \frac{1}{3}$.

D A Sample of General Theorems

In all cases a, d are relatively prime.

▶ **Theorem 14.** *If* $a \in \{2d+1, \dots, 3d-1\}$ *then* $f(3dk+a+d, 3dk+a) \leq \frac{dk+X}{3dk+a}$ *where* $X = \frac{a}{3}$, *so* $f(3dk+a+d, 3dk+a) \leq \frac{1}{3}$.

▶ **Theorem 15.** *If* $a \in \{1, \dots, 3d-1\}$, $a \neq d$, *then* $f(3dk+a+d, 3dk+a) \leq \frac{dk+X}{3dk+a}$ *where* $X = \max\{\frac{a}{3}, \frac{a+d}{5}, \frac{2a-d}{3}\}$.

▶ **Theorem 16.** *If* $1 \leq a \leq 3d-1$ *and* $5a \neq 7d$ *then* $f(3dk+a+d, 3dk+a) \leq \frac{dk+X}{3dk+a}$ *where* $X = \max\{\frac{a}{3}, \frac{a+d}{5}, \frac{a+2d}{6}, \frac{3a-2d}{4}\}$.

▶ **Theorem 17.** *If* $1 \leq a \leq \frac{5d}{7}$ *and* $a \neq \frac{2d}{3}$ *then* $f(3dk+a+d, 3dk+a) \leq \frac{dk+X}{3dk+a}$ *where* $X = \max\{\frac{2a}{5}, \frac{a+d}{6}\}$.

▶ **Theorem 18.** *If* $\frac{5d}{7} \leq a \leq d-1$ *then* $f(3dk+a+d, 3dk+a) \leq \frac{dk+X}{3dk+a}$ *where* $X = \max\{\frac{2a}{5}, \frac{3a-d}{4}\}$.

▶ **Theorem 19.** *If* $\frac{5d}{13} \leq a \leq \frac{13d}{29}$ *and* $a \neq \frac{2}{5}d$ *then* $f(3dk+a+d, 3dk+a) \leq \frac{dk+X}{3dk+a}$ *where* $X = \max\{\frac{5a-d}{6}, \frac{a+d}{8}, \frac{3a}{7}\}$.

E If $m \geq s$ then $f(m,s) \geq 1/3$

Before showing the general technique we give an example.

▶ **Example.** $f(19, 17) \geq \frac{1}{3}$.

We express $\frac{19}{17}$ as $\frac{57}{51}$ since other fractions will have a denominator of 51.

We initially divide the 19 muffins $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. There are now 57 pieces $\frac{1}{3}$-shares. We initially give 11 students 3 $\frac{1}{3}$-shares and 6 students 4 $\frac{1}{3}$-shares. (In the proof below $W = 3$,

$s_W = s_3 = 11$, and $s_{W+1} = s_4 = 6$.) A student who gets 3 (4) shares is called a *3-student (4-student).*

We describe a process whereby students give pieces of muffins, called gifts, to other students so that, in the end, all students have $\frac{57}{51}$. Each gift leads to a change in how the muffins are cut in the first place; however, there will never be a muffin of size $< \frac{1}{3}$.

Each 4-student has $\frac{4}{3} = \frac{68}{51}$ and hence has to give (perhaps in several increments) $\frac{68}{51} - \frac{57}{51} = \frac{11}{51}$ to get *down to* $\frac{57}{51}$. Realize that if a 4-student gives $\frac{11}{51}$ to a 3-student, then the 3-student now has $\frac{51}{51} + \frac{11}{51} = \frac{62}{51} > \frac{57}{51}$.

Each 3-student has $\frac{51}{51}$ and hence has to receive $\frac{57}{51} - \frac{51}{51} = \frac{6}{51}$ to get *up to* $\frac{57}{51}$.

Call the 11 3-students $g_1, \ldots, g_{11}$.

Call the 6 4-students $f_1, \ldots, f_6$.

We use a notation that we just give an example of:

f_1 gives x to g_1 by taking two $\frac{1}{3}$-pieces, combining them, cutting off a piece of size x, giving it to g_1 while keeping the rest. g_1 takes the piece given to him and combines it with a $\frac{1}{3}$ piece. Notice that in terms of pieces we are taking three pieces of size $\frac{1}{3}$ (2 from f_1 and 1 from g_1) and turning them into 1 piece of size $\frac{2}{3} - x$ and one of size $\frac{1}{3} + x$. Hence we can easily rearrange how the muffins are cut.

$x(f_1 \to g_1)$

We need to make sure this procedure never results in a piece that is $< \frac{1}{3}$. In the above example (1) f_1 now has a piece of size $\frac{2}{3} - x$, hence we need $x \le \frac{1}{3}$, (2) g_1 now has a piece of size $\frac{1}{3} + x$, which is clearly $\ge \frac{1}{3}$. Hence the only restriction is $x \le \frac{1}{3}$.

1. $\frac{11}{51}(f_1 \to g_1)$. Now f_1 has $\frac{57}{51}$. YEAH. However, g_1 has $\frac{62}{51}$.
2. $\frac{5}{51}(g_1 \to g_2)$. Now g_1 has $\frac{62}{51} - \frac{5}{51} = \frac{57}{51}$. YEAH. However, g_2 has $\frac{51}{51} + \frac{5}{51} = \frac{56}{51}$.
3. $\frac{1}{51}(f_2 \to g_2)$. Now g_2 has $\frac{57}{51}$. YEAH. However, f_2 has $\frac{67}{51}$.
4. $\frac{10}{51}(f_2 \to g_3)$. Now f_2 has $\frac{57}{51}$. YEAH. However, g_3 has $\frac{61}{51}$.
5. $\frac{4}{51}(g_3 \to g_4)$. Now g_3 has $\frac{57}{51}$. YEAH. However, g_4 has $\frac{55}{51}$.
6. $\frac{2}{51}(f_3 \to g_4)$. Now g_4 has $\frac{57}{51}$. YEAH. However, f_3 has $\frac{66}{51}$.
7. $\frac{9}{51}(f_3 \to g_5)$. Now f_3 has $\frac{57}{51}$. YEAH. However, g_5 has $\frac{60}{51}$.
8. $\frac{3}{51}(g_5 \to g_6)$. Now g_5 has $\frac{57}{51}$. YEAH. However, g_6 has $\frac{54}{51}$.
9. $\frac{3}{51}(f_4 \to g_6)$. Now g_6 has $\frac{57}{51}$. YEAH. However, f_4 has $\frac{65}{51}$.
10. $\frac{8}{51}(f_4 \to g_7)$. Now f_4 has $\frac{57}{51}$. YEAH. However, g_7 has $\frac{59}{51}$.
11. $\frac{2}{51}(g_7 \to g_8)$. Now g_7 has $\frac{57}{51}$. YEAH. However, g_8 has $\frac{53}{51}$.
12. $\frac{4}{51}(f_5 \to g_8)$. Now g_8 has $\frac{57}{51}$. YEAH. However, f_5 has $\frac{64}{51}$.
13. $\frac{7}{51}(f_5 \to g_9)$. Now f_5 has $\frac{57}{51}$. YEAH. However, g_9 has $\frac{58}{51}$.
14. $\frac{1}{51}(g_9 \to g_{10})$. Now g_9 has $\frac{58}{51}$. YEAH. However, g_{10} has $\frac{52}{51}$.
15. $\frac{5}{51}(f_6 \to g_{10})$. Now g_{10} has $\frac{57}{51}$. YEAH. However, f_6 has $\frac{63}{51}$.
16. $\frac{6}{51}(f_6 \to g_{11})$. Now f_6 has $\frac{57}{51}$. YEAH. However, g_{11} has $\frac{57}{51}$. OH. thats a good thing!

YEAH- we are done.

Note that the first x was $\frac{11}{51} \le \frac{1}{3}$ and the remaining x were all $\le \frac{11}{51} \le \frac{1}{3}$. Hence all pieces in the final protocol are $\ge \frac{1}{3}$.

▶ **Theorem 20.** *For all $m \ge s$, $f(m,s) \ge \frac{1}{3}$.*

Proof. Divide all the muffins into $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. Initially distribute them as evenly as possible among the students. There will be a number W such that some students get W shares and some get $(W+1)$-shares. Let s_W (s_{W+1}) be the number of students who get W $(W+1)$ shares.

We do not need the following but are noting it anyway. If s does not divide $3m$ then $W = \frac{3m}{s}$ and s_W, s_{W+1} are unique and determined by:

$$\begin{aligned} Ws_W + (W+1)s_{W+1} &= 3m \\ s_W + s_{W+1} &= s \end{aligned}$$

(Technically, if $s \mid 3m$ there are two possible values of W.)

A student who gets W $(W+1)$ shares we call a W-student ($(W+1)$-student). All W-students get $\frac{W}{3}$. All $(W+1)$-students get $\frac{W+1}{3}$.

A W-student must get $< \frac{m}{s}$: if a W-student got $> \frac{m}{s}$ then all students would get $> \frac{m}{s}$ and hence there would be $> s\frac{m}{s} = m$ muffins total. A $(W+1)$-student must get $> \frac{m}{s}$: if a $(W+1)$-student got $< \frac{m}{s}$ then all students would get $< \frac{m}{s}$ and hence there would be $< s\frac{m}{s} = m$ muffins total.

Hence we have:

$$\frac{m}{s} - \frac{W}{3} \leq \frac{1}{3} \tag{1}$$

$$\frac{W+1}{3} - \frac{m}{s} \leq \frac{1}{3} \tag{2}$$

Now we will need to smooth out the distribution so that everyone receives $\frac{m}{s}$. We will do this by doing a sequence of moves of the form $x(f_i \to g_j)$ or $x(g_i \to g_j)$. as defined in the example.

We will assume s_{W+1} and s_W are relatively prime (this only comes up in Claim 3 below). This is fine because if they have a common factor d, we can just use the procedure for the $\frac{s_{W+1}}{d}$, $\frac{s_W}{d}$ case repeated d times.

▶ **Claim 1.**

1. *If* $s_{W+1} < s_W$ *then* $\frac{W+1}{3} - \frac{m}{s} > \frac{m}{s} - \frac{W}{3}$.
2. *If* $s_W < s_{W+1}$ *then* $\frac{W+1}{3} - \frac{m}{s} > \frac{m}{s} - \frac{W}{3}$.

Proof of Claim 1.

$$s_{W+1} \times \frac{W+1}{3} + s_W \times \frac{W}{3} = m$$

$$s_{W+1} \times \left(\frac{m}{s} + \frac{W+1}{3} - \frac{m}{s}\right) + s_W\left(\frac{m}{s} + \frac{W}{3} - \frac{m}{s}\right) = m.$$

$$\left(s_{W+1} + s_W\right)\frac{m}{s} + s_{W+1}\left(\frac{W+1}{3} - \frac{m}{s}\right) + s_W\left(\frac{W}{3} - \frac{m}{s}\right) = m$$

$$s \times \frac{m}{s} + s_{W+1}\left(\frac{W+1}{3} - \frac{m}{s}\right) + s_W\left(\frac{W}{3} - \frac{m}{s}\right) = m$$

$$\frac{W+1}{3} - \frac{m}{s} = \frac{s_W}{s_{W+1}}\left(\frac{m}{s} - \frac{W}{3}\right)$$

Both parts follow. ◀

We give the procedure to obtain $f(m,s) \leq \frac{1}{3}$. There are two cases.

Case 1: $s_{W+1} < s_W$. Hence by Claim 1 $\frac{W+1}{3} - \frac{m}{s} > \frac{m}{s} - \frac{W}{3}$.
Call the s_W W-students $g_1, \ldots, g_{s_W}$.
Call the s_{W+1} $(W+1)$-students $f_1, \ldots, f_{s_{W+1}}$.

1. Let $x = \frac{W+1}{3} - \frac{m}{s}$. Note that $x \le \frac{1}{3}$. Do $x(f_1 \rightarrow g_1)$. Now f_1 has $\frac{m}{s}$. YEAH. However, g_1 has $\frac{W}{3} + \frac{W+1}{3} - \frac{m}{s} > \frac{m}{s}$. (This is where we use $s_{W+1} < s_W$, or more accurately the consequence of that from Claim 1.)
2. Let $x = \frac{2W+1}{3} - 2\frac{m}{s}$. Do $x(g_1 \rightarrow g_2)$. Now g_1 has $\frac{m}{s}$. YEAH.
3. If g_2 has $> \frac{m}{s}$ then g_2 gives enough to g_3 so that g_2 has $\frac{m}{s}$. Keep up this chain of $g_1, g_2, g_3, \ldots$ until there is a g_i such that g_i end up with $< \frac{m}{s}$ (though more than the $\frac{W}{3}$ that g_i had originally).
4. Do $x(f_2 \rightarrow g_i)$ where x is such that g_i will now have $\frac{m}{s}$.
5. Do $x(f_2 \rightarrow g_{i+1})$ where x is such that f_2 will now have $\frac{m}{s}$. Repeat the same chain of g_i's as in step 3.
6. Repeat the above steps until you are done.

We need to show that (1) there is never a piece of size $< \frac{1}{3}$, and (2) the process ends with every student getting $\frac{m}{s}$.

▶ **Claim 2.** *The first gift is $\le \frac{1}{3}$ and no gift is larger.*

Proof of Claim 2. Let $C = \frac{W+1}{3} - \frac{m}{s}$ which is the size of the first gift. By equation (2) $C \le \frac{1}{3}$.

Assume that all gifts so far have been $\le C$. We analyze the three kinds of gifts and show that in all cases the gift is $\le C$.

- $x(f_i \rightarrow g_j)$ where (1) initially f_i has $> \frac{m}{s}$, g_j has $< \frac{m}{s}$, and (2) after the gift f_i has $\frac{m}{s}$. When this occurs it is f_i's first or second gift giving. (This happens in steps 1 and 5 above, and later as well.) Before the gift f_i has at least $\frac{m}{s}$ but at most $\frac{W+1}{3}$, so this gift has size at most $\frac{W+1}{3} - \frac{m}{s} = C$.
- $x(g_i \rightarrow g_{i+1})$ where (1) initially g_i has $> \frac{m}{s}$, g_j has $< \frac{m}{s}$, and (2) after the gift g_i has $\frac{m}{s}$. When this occurs g_i has received a gift once and this is g_i's first time giving. (This happens in steps 2 and in the chain referred to in step 5.) Since g_i just received a gift of size $\le C$ she has $\le \frac{W}{3} + C$. Hence the gift is $\le \frac{W}{3} - \frac{m}{s} + C \le C$.
- $x(f_i \rightarrow g_j)$ where (1) initially f_i has $> \frac{m}{s}$, g_j has $< \frac{m}{s}$, and (2) after the gift g_j has $\frac{m}{s}$. This will be f_i's first time giving. (This happens in step 4 above.) Before the gift f_i has at least $\frac{W}{3}$ but at most $\frac{m}{s}$, so this gift has size at most $\frac{m}{s} - \frac{W}{3} \le C$ (by Claim 1). ◀

▶ **Claim 3.** *If s_W and s_{W+1} are relatively prime then the process terminates with all students having $\frac{m}{s}$.*

Proof of Claim 3. In each step all of the f_i have at least $\frac{m}{s}$. In each step the number of students who have the correct amount of muffin goes up. One may be worried that at some point we will try to do step 4 (for example) of the procedure and there will be no g_i left who need more muffin. But this is not possible because until the process terminates the f's always have more muffin than they need, so there is always a g with insufficient muffin.

One may also be worried that eventually we will get all of the f's to have $\frac{m}{s}$, but the g's will not all have $\frac{m}{s}$. This is not possible either, because whenever we only make gifts from f to g when there is no g with more than $\frac{m}{s}$.

Finally, if s_W and s_{W+1} are not relatively prime, it is possible that the procedure will terminate early because in step 5 the size of the donation x is 0. If this occurred it would mean that there is some subset of F f's and G g's each of which having exactly $\frac{m}{s}$, who only made donations amongst themselves. But then $\frac{F}{G} = \frac{s_{W+1}}{s_W}$, a contradiction. ◀

Case 2: $s_W < s_{W+1}$. This is similar to Case 1 except that instead of f_1 giving g_1 so that f_1 has $\frac{m}{s}$, f_1 gives to g_1 so that g_1 has $\frac{m}{s}$. Hence we have a chain of f_i's instead of a chain of g_i's. ◀

God Save the Queen

Jurek Czyzowicz[1]
Université du Québec en Outaouais, Gatineau, Québec, Canada
jurek.czyzowicz@uqo.ca

Konstantinos Georgiou[2]
Department of Mathematics, Ryerson University, Toronto, Ontario, Canada
konstantinos@ryerson.ca

Ryan Killick[3]
School of Computer Science, Carleton University, Ottawa, Ontario, Canada
ryankillick@cmail.carleton.ca

Evangelos Kranakis[4]
School of Computer Science, Carleton University, Ottawa, Ontario, Canada
kranakis@scs.carleton.ca

Danny Krizanc
Department of Mathematics & Comp. Sci., Wesleyan University, Middletown, CT, USA
dkrizanc@wesleyan.edu

Lata Narayanan[5]
Department of Comp. Sci. and Software Eng., Concordia University, Montreal, Québec, Canada
lata@encs.concordia.ca

Jaroslav Opatrny
Department of Comp. Sci. and Software Eng., Concordia University, Montreal, Québec, Canada
opatrny@cs.concordia.ca

Sunil Shende
Department of Computer Science, Rutgers University, Camden, NJ, USA
shende@camden.rutgers.edu

Abstract

Queen Daniela of Sardinia is asleep at the center of a round room at the top of the tower in her castle. She is accompanied by her faithful servant, Eva. Suddenly, they are awakened by cries of "Fire". The room is pitch black and they are disoriented. There is exactly one exit from the room somewhere along its boundary. They must find it as quickly as possible in order to save the life of the queen. It is known that with two people searching while moving at maximum speed 1 anywhere in the room, the room can be evacuated (i.e., with both people exiting) in $1 + \frac{2\pi}{3} + \sqrt{3} \approx 4.8264$ time units and this is optimal [Czyzowicz et al., DISC'14], assuming that the first person to find the exit can directly guide the other person to the exit using her voice. Somewhat surprisingly, in this paper we show that if the goal is to save the queen (possibly leaving Eva behind to die in the fire) there is a slightly better strategy. We prove that this "priority" version of evacuation can be solved in time at most 4.81854. Furthermore, we show that any strategy for saving the queen requires time at least $3 + \pi/6 + \sqrt{3}/2 \approx 4.3896$ in the worst case. If one or both of the queen's other servants (Biddy and/or Lili) are with her, we show that the time bounds can be improved to 3.8327 for two servants, and 3.3738 for three servants.

[1] NSERC
[2] NSERC
[3] OGS
[4] NSERC
[5] NSERC

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 16; pp. 16:1–16:20
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Finally we show lower bounds for these cases of 3.6307 (two servants) and 3.2017 (three servants). The case of $n \geq 4$ is the subject of an independent study by Queen Daniela's Royal Scientific Team.

2012 ACM Subject Classification Computing methodologies → Distributed artificial intelligence,Computing methodologies → Mobile agents

Keywords and phrases Algorithm, Evacuation, Exit, Disk, Wireless Communication, Queen, Priority, Robots, Search, Servants, Trajectory

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.16

Related Version Full version hosted on arXiv `http://arxiv.org/abs/1804.06011`

1 Introduction

In traditional search, a group of searchers (modeled as mobile autonomous agents or robots) may collaboratively search for an exit (or target) placed within a given search domain [1, 2, 20]. Although the searchers may have differing capabilities (communication, perception, mobility, memory) search algorithms, previously employed, generally make no distinction between them as they usually play identical roles throughout the execution of the search algorithm and with respect to the termination time (with the exception of faulty robots, which also do not contribute to searching). In this work we are motivated by real-life safeguarding-type situations where a number of agents have the exclusive role to facilitate the execution of the task by a distinguished entity. More particularly, we introduce and study *Priority Evacuation*, a new form of search , under the wireless communication model, in which the search time of the algorithm is measured by the time it takes a special searcher, called the queen, to reach the exit. The remaining searchers in the group, called servants, are participating in the search but are not required to exit.

1.1 Problem Definition of Priority Evacuation (PE$_n$)

A target (exit) is hidden in an unknown location on the unit circle. The exit can be located by any of the $n+1$ robots (searchers) that walks over it ($n = 1, 2, 3$). Robots share the same coordinate system, start from the center of the circle, and have maximum speed 1. Among them there is a distinguished robot, called the *queen*, and the remaining n robots are referred to as *servants*. All servants are known to the queen by their identities. Robots may run asymmetric algorithms, and can communicate their findings wirelessly and instantaneously (each message is composed by an identity and a location). Only the queen is required to be able to receive messages. Feasible solutions to this problem are *evacuation algorithms*, i.e. robots' movements (trajectories) that guarantee the finding of the hidden exit. The cost of an evacuation algorithm is the *evacuation time* of the queen, i.e., the worst case total time until the queen reaches the exit. None of the n servants needs to evacuate.

1.2 Related work

Related to our work is linear search which refers to search in an infinite line. There have been several interesting studies attempting to optimize the search time which were initiated with the influential works of Bellman [7] and Beck [6]. A long list of results followed for numerous variants of the problem, citing which is outside the scope of this work. For a comprehensive study of seminal search-type problems see [2, 3].

The problem of searching in the plane by one or more searchers, has been considered by [4, 5]. The unit disk model considered in our present paper is a form of two-dimensional search that was initiated in the work of [10]. In this paper the authors obtained evacuation algorithms in the wireless and face-to-face communication models both for a small number of robots as well optimal asymptotic results for a large number of robots. Additional evacuation algorithms in the face-to-face communication model were subsequently analyzed for two robots in [14] and later in [8]. Other variations of the problem include the case of more than one exit, see [9] and [19], triangular and square domains in [15], robots with different moving speeds [18], and evacuation in the presence of crash or byzantine faulty robots [11].

A priority evacuation-type problem has been previously considered in [16, 17] but with different terminology. Using the jargon of the current paper, an immobile queen is hidden somewhere on the unit disk, and a number of robots try to locate her, and fetch (evacuate) her to an exit which is also hidden. The performance of the evacuation algorithm is measured by the time the queen reaches the exit.

Apart from the results in [16, 17], all relevant previous work in search-type problems considered the objective of minimizing the time it takes either by the first or the last agent to reach the hidden target. In contrast, this paper considers an evacuation (search-type) problem where the completion time is defined with respect to a distinguished mobile agent, the *queen*, while the remaining n servants are not required to evacuate. Our current focus is to design efficient algorithms for $n = 1, 2, 3$ servants, as well as give strong lower bounds. Notably, the algorithms we propose significantly improve upon evacuation costs induced by naive trajectories, and in fact the trajectories we propose are non-trivial. Our main contribution concerns priority evacuation for each of the cases of $n = 1, 2, 3$ servants, all of which require special treatment. Moreover, all our algorithms are characterized by the fact that the queen does contribute effectively to the search of the hidden item. In sharp contrast, the independent and concurrent work of [13] studies the same problem for $n \geq 4$ servants where the queen never contributes to the search. More importantly, the proposed algorithms of [13] admit a unified description and analysis that does not intersect with the current work.

1.3 Our Results & Paper Organization

Section 2 introduces necessary notation and terminology and discusses preliminaries. Section 3 is devoted to upper bounds for PE_n for $n = 1, 2, 3$ servants (see Subsections 3.1, 3.2, and 3.3, respectively). All our upper bounds are achieved by fixing optimal parameters for families of parameterized algorithms. In Section 4 we derive lower bounds for PE_n, $n = 1, 2, 3$. An interesting corollary of our positive results is that priority evacuation with $n = 1, 2, 3$ servants (i.e. with $n + 1$ searchers) can be performed strictly faster than ordinary evacuation with $n + 1$ robots where all robots have to evacuate. Indeed, an argument found in [10] can be adjusted to show that the evacuation problem with $n + 1$ robots cannot be solved faster than $1 + \frac{4\pi}{3(n+1)} + \sqrt{3}$. Surprisingly, when one needs to evacuate only one designated robot, the task can provably (due to our upper bounds) be executed faster. All our results, together with the comparison to the lower bounds of [10], are summarized in Table 1. We conclude the paper in Section 5 with a discussion of open problems. Whenever we omit proofs, due to space limitations, we provide an outline of our arguments. The interested reader may consult the full version of our paper [12] for the missing details.

Table 1 Upper and lower bounds for priority evacuation.

# of Servants	Upper Bounds for PE_n	Lower Bounds for PE_n	Lower Bounds for Ordinary Evacuation
$n = 1$	4.8185 (Theorem 8)	4.3896 (Theorem 17)	4.826445 (see [10])
$n = 2$	3.8327 (Theorem 10)	3.6307 (Theorem 19)	4.128314 (see [10])
$n = 3$	3.3738 (Theorem 14)	3.2017 (Theorem 19)	3.779248 (see [10])

2 Notation and Preliminaries

We use n to denote the number of servants, and we set $[n] = \{1, \dots, n\}$. Queen and servant i will be denoted by $\mathcal{Q}$ and $\mathcal{S}_i$, respectively, where $i \in [n]$. We assume that all robots start from the origin $O = (0, 0)$ of a unit circle in $\mathbb{R}^2$. As usual, points in $A \in \mathbb{R}^2$ will be treated, when it is convenient, as vectors from O to A, and $\|A\|$ will denote the euclidean norm of that vector.

2.1 Problem Reformulation & Solutions' Description

Robots' trajectories will be defined by parametric functions $\mathcal{F}(t) = (f(t), g(t))$, where $f, g : \mathbb{R} \mapsto \mathbb{R}$ are continuous and piecewise differentiable. In particular, search algorithms for all robots will be given by trajectories

$$\mathbb{S}_n := \left\{ \mathcal{Q}(t), \{\mathcal{S}_i(t)\}_{i \in [n]} \right\},$$

where $\mathcal{Q}(t), \mathcal{S}_i(t)$ will denote the position of $\mathcal{Q}$ and $\mathcal{S}_i$, respectively, at time $t \geq 0$.

▶ **Definition 1** (Feasible Trajectories). We say that trajectories $\mathbb{S}_n$ are *feasible* for PE_n if:

(a) $\mathcal{Q}(0) = \mathcal{S}_i(0) = O$, for all $i \in [n]$,

(b) $\mathcal{Q}(t), \{\mathcal{S}_i(t)\}_{i \in [n]}$ induce speed-1 trajectories for $\mathcal{Q}, \{\mathcal{S}_i\}_{i \in [n]}$ respectively, and

(c) there is some time $t_0 \geq 1$, such that each point of the unit circle is visited (searched) by at least one robot in the time window $[0, t_0]$. We refer to the smallest such t_0 as the *search time* of the circle.

Note that feasible trajectories do indeed correspond to robots' movements for PE_n in which, eventually the entire circle is searched, and hence the search time is bounded. We will describe all our search/evacuation algorithms as feasible trajectories, and we will assume that once the target is reported, $\mathcal{Q}$ will go directly to the location of the exit.

For feasible trajectories $\mathbb{S}_n$ with search time t_0, and for any trajectory $\mathcal{F}(t)$ (either of the queen or of a servant), we denote by $\mathbb{I}(\mathcal{F})$ the subinterval of $[0, t_0]$ that contains all $x \in [0, t_0]$ such that $\|\mathcal{F}(x)\| = 1$ (i.e. the robot is on the the circle) and no other robot has been to $\mathcal{F}(x)$ before. Since robots start from the origin, it is immediate that $\mathbb{I}(\mathcal{F}) \subseteq [1, t_0]$. With this notation in mind, note that the exit can be discovered by some robot $\mathcal{F}$, say at time x, only if $x \in \mathbb{I}(\mathcal{F})$. In this case, the finding is instantaneously reported, so $\mathcal{Q}$ goes directly to the exit, moving along the corresponding line segment between her current position $\mathcal{Q}(x)$ and the reported position of the exit $\mathcal{F}(x)$. Hence, the total time that $\mathcal{Q}$ needs to evacuate equals

$$x + \|\mathcal{Q}(x) - \mathcal{F}(x)\|.$$

Therefore, the *evacuation time* of feasible trajectories $\mathbb{S}_n$ to PE_n is given by expression

$$\max_{\mathcal{F} \in \mathbb{S}_n} \sup_{x \in \mathbb{I}(\mathcal{F})} \{x + \|\mathcal{Q}(x) - \mathcal{F}(x)\|\}.$$

Notice that for "non-degenerate" search algorithms for which the last point on the circle is not searched by $\mathcal{Q}$ alone, the previous maximum can be simply computed over the servants, i.e the evacuation cost will be

$$\max_{i\in[n]} \sup_{x\in\mathbb{I}(\mathcal{S}_i)} \left\{x + \|\mathcal{Q}(x) - \mathcal{S}_i(x)\|\right\}. \tag{1}$$

In other words, we can restate PE$_n$ as the problem of determining feasible trajectories $\mathbb{S}_n$ so as to minimize (1).

2.2 Useful Trajectories' Components

Feasible trajectories induce, by definition, robots that are moving at (maximum) speed 1. The speed restriction will be ensured by the next condition.

▶ **Lemma 2.** *An object following trajectory $\mathcal{F}(t) = (f(t), g(t))$ has unit speed if and only if*

$$\left(f'(t)\right)^2 + \left(g'(t)\right)^2 = 1, \quad \forall t \geq 0.$$

Proof. For any $t \geq 0$, the velocity of $\mathcal{F}$ is given by $\mathcal{F}'(t) = (df(t)/dt, dg(t)/dt)$, and its speed is calculated as $\|\mathcal{F}'(t)\|$. ◀

Robots' trajectories will be composed by piecewise smooth parametric functions. In order to describe them, we introduce some further notation. For any $\theta \in \mathbb{R}$, we introduce abbreviation C_θ for point $\{\cos(\theta), \sin(\theta)\}$. Next we introduce parametric equations for moving along the perimeter of a unit circle (Lemma 3), and along a line segment (Lemma 4).

▶ **Lemma 3.** *Let $b \in [0, 2\pi)$ and $\sigma \in \{-1, 1\}$. The trajectory of an object moving at speed 1 on the perimeter of a unit circle with initial location C_b is given by the parametric equation*

$$\mathcal{C}(b, \sigma t) := (\cos(\sigma t + b), \sin(\sigma t + b)).$$

If $\sigma = 1$ the movement is counter-clockwise (ccw), and clockwise (cw) otherwise.

Proof. Clearly, $\mathcal{C}(b, 0) = C_b$. Also, it is easy to see that $\|\mathcal{C}(b, t)\| = 1$, i.e. the object is moving on the perimeter of the unit circle. Lastly,

$$\left(\frac{d}{dt}\cos(\sigma t + b)\right)^2 + \left(\frac{d}{dt}\sin(\sigma t + b)\right)^2 = \sigma^2\left(-\sin(\sigma t + b)\right)^2 + \sigma^2\left(\cos(\sigma t + b)\right)^2 = 1,$$

so the claim follows by Lemma 2. ◀

▶ **Lemma 4.** *Consider distinct points $A = (a_1, a_2), B = (b_1, b_2)$ in $\mathbb{R}^2$. The trajectory of a speed 1 object moving along the line passing through A, B and with initial position A is given by the parametric equation*

$$\mathcal{L}(A, B, t) := \left(\frac{b_1 - a_1}{\|A - B\|}t + a_1, \frac{b_2 - a_2}{\|A - B\|}t + a_2\right).$$

Proof. It is immediate that the parametric equation corresponds to a line. Also, it is easy to see that $\mathcal{L}(A, B, 0) = A$ and $\mathcal{L}(A, B, \|A - B\|) = B$, i.e. the object starts from A, and eventually visits B. As for the object's speed, we calculate

$$\left(\frac{d}{dt}\left(\frac{b_1 - a_1}{\|A - B\|}t + a_1\right)\right)^2 + \left(\frac{d}{dt}\left(\frac{b_2 - a_2}{\|A - B\|}t + a_2\right)\right)^2 = \left(\frac{b_1 - a_1}{\|A - B\|}\right)^2 + \left(\frac{b_2 - a_2}{\|A - B\|}\right)^2 = 1$$

so, by Lemma 2, the speed is indeed 1. ◀

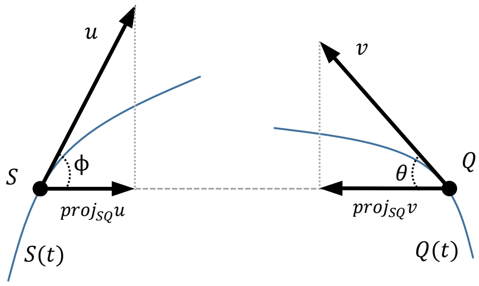


Figure 1 An illustration of trajectories $\mathcal{S}(t)$, $\mathcal{Q}(t)$, and their critical angles at some fixed time τ, with $\mathcal{S}(\tau) = S$, $\mathcal{Q}(\tau) = Q$, $\mathcal{S}'(\tau) = u$, $\mathcal{Q}'(\tau) = v$.

Robots trajectories will be described in phases. In each phase, robot, say $\mathcal{F}$, will be moving between two explicit points, and the corresponding trajectory $\mathcal{F}(t)$ will be implied by the previous description, using most of the times Lemma 3 and Lemma 4. We will summarize the details in tables of the following format.

Robot	#	*Description*	*Trajectory*	*Duration*
$\mathcal{F}$	0		$\mathcal{F}(t)$	t_0
	1		$\mathcal{F}(t)$	t_1
	$\vdots$			$\vdots$

Phase 0 will usually correspond to the deployment of $\mathcal{F}$ from the origin to some point of the circle. Also, for each phase we will summarize it's duration. With that in mind, trajectory $\mathcal{F}(t)$ during phase i, with duration t_i, will be valid for all $t \geq 0$ with $|t-(t_0+t_1+\ldots t_{i-1})| \leq t_i$.

Lastly, the following abbreviation will be useful for the exposition of the trajectories. For any $\rho \in [0,1]$ and $\theta \in [0,2\pi)$, we introduce notation

$$K(\theta,\rho) := (1-\rho)C_{\pi-\theta} + \rho C_{-\theta}.$$

In other words, $K(\theta,\rho)$ is a convex combination of antipodal points $C_{\pi-\theta}, C_{-\theta}$ of the unit circle, i.e. it lies on the diameter of the unit circle passing through these two points. Moreover, it is easy to see that $\|C_{\pi-\theta} - K(\theta,\rho)\| = 2\rho$, and hence

$$\|K(\theta,\rho) - C_{-\theta}\| = 2 - 2\rho.$$

As it will be handy later, we also introduce abbreviation

$$AK(\theta,\rho) := \|C_\pi - K(\theta,\rho)\|.$$

The choice of the abbreviation is clear, if the reader denotes $C_\pi = (-1,0)$ by A.

2.3 Critical Angles

The following definition introduces a key concept. In what follows, abstract trajectories will be assumed to be continuous and differentiable, which in particular implies that corresponding velocities are continuous.

▶ **Definition 5** (Critical Angle)**.** Let $\mathcal{S}(t) \in \mathbb{R}^2$ denote the trajectory of a speed-1 object, where $t \geq 0$. For some point $Q \in \mathbb{R}^2$, we define the $(\mathcal{S}, Q)$-critical angle at time $t = \tau$ to be the angle between the velocity vector $\mathcal{S}'(\tau)$ and vector $\overrightarrow{\mathcal{S}(\tau)Q}$, i.e. the vector from $\mathcal{S}(\tau)$ to Q.

We make the following critical observation, see also Figure 1.

▶ **Theorem 6.** *Consider trajectories $\mathcal{S}(t), \mathcal{Q}(t)$ of two speed-1 objects $\mathcal{S}, \mathcal{Q}$, where $t \geq 0$. Let also ϕ, θ denote the $(\mathcal{S}, \mathcal{Q}(t))$-critical angle and the $(\mathcal{Q}, \mathcal{S}(t))$-critical angle at time t, respectively. Then $t + \|\mathcal{Q}(t) - \mathcal{S}(t)\|$ is strictly increasing if $\cos(\phi) + \cos(\theta) < 1$, strictly decreasing if $\cos(\phi) + \cos(\theta) > 1$, and constant otherwise.*

Theorem 6 is an immediate corollary of the following lemma.

▶ **Lemma 7.** *Consider trajectories $\mathcal{S}(t), \mathcal{Q}(t)$ and their critical angles π, θ, as in the statement of Theorem 6. Then*

$$\frac{d}{dt}\|\mathcal{Q}(t) - \mathcal{S}(t)\| = \cos(\phi) + \cos(\theta).$$

Proof. For any fixed t, let d denote $D(t)$, and S, Q denote points $\mathcal{S}(t), \mathcal{Q}(t)$, respectively. Denote also by u, v the velocities of $\mathcal{S}, \mathcal{Q}$ at time t, respectively, i.e. $u = \mathcal{S}'(t), v = \mathcal{Q}'(t)$. See also Figure 1.

With that notation, observe that $\left\|\overrightarrow{SQ}\right\| = d$. Since $\|u\| = \|v\| = 1$, we see that

$$\text{proj}_{SQ}u = \frac{\cos(\phi)}{d}\overrightarrow{SQ}$$

and

$$\text{proj}_{SQ}v = \frac{\cos(\theta)}{d}\overrightarrow{QS}.$$

Now consider two imaginary objects $\overline{\mathcal{S}}, \overline{\mathcal{Q}}$, with corresponding velocities $\overline{\mathcal{S}}'(t) = \text{proj}_{SQ}u$ and $\overline{\mathcal{Q}}'(t) = \text{proj}_{SQ}v$. It is immediate that $\|\mathcal{Q}(t) - \mathcal{S}(t)\| = \left\|\overline{\mathcal{Q}}(t) - \overline{\mathcal{S}}(t)\right\|$.

In particular, $\text{proj}_{SQ}u - \text{proj}_{SQ}v$ is the projection of the relative velocities of $\mathcal{S}, \mathcal{Q}$ on the line segment connecting $\mathcal{S}(t), \mathcal{Q}(t)$. As such, the distance between $\mathcal{S}, \mathcal{Q}$ changes at a rate determined by velocity

$$\text{proj}_{SQ}u - \text{proj}_{SQ}v = \frac{\cos(\phi) + \cos(\theta)}{d}\overrightarrow{SQ},$$

where $\left\|\text{proj}_{SQ}u - \text{proj}_{SQ}v\right\| = |\cos(\phi) + \cos(\theta)|$. Moreover, $\text{proj}_{SQ}u, \text{proj}_{SQ}v$ are antiparallel iff and only if $\cos(\phi), \cos(\theta) > 0$, in which case the two objects come closer to each other. ◀

3 Upper Bounds

3.1 Evacuation Algorithm for PE$_1$

This subsection is devoted in proving the following.

▶ **Theorem 8.** *Consider the real function $f(x) = x + \sin(x)$, and denote by $\alpha_0 > 0$ the solution to equation*

$$f(f(\alpha - \sin(\alpha))) = \sin(\alpha),$$

with $\alpha_0 \approx 1.14193$. Then PE$_1$ can be solved in time $1 + \pi - \alpha_0 + 2\sin(\alpha_0) \approx 4.81854$.

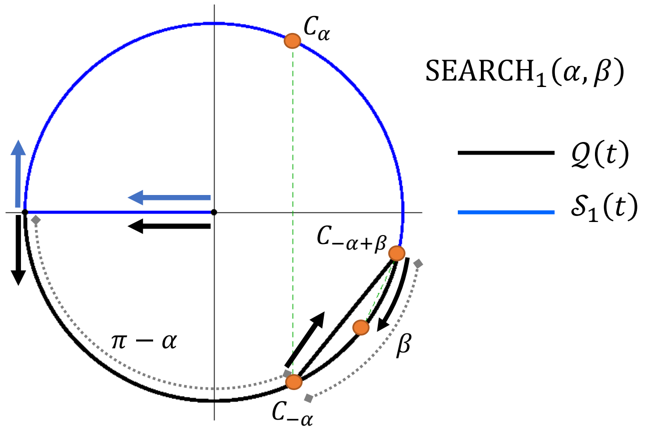


Figure 2 Algorithm Search$_1(\alpha,\beta)$ depicted for the optimal parameters of the algorithm. In all subsequent figures, as well as here, the orange points on the perimeter of the disc correspond to the worst adversarial placements of the treasure, which due to our optimality conditions induce the same evacuation cost. The orange points in $\mathcal{Q}$'s trajectories correspond to the $\mathcal{Q}$'s positioning when the treasures are reported, in the worst cost induced cases. The green dashed line depict $\mathcal{Q}$'s trajectory after $\mathcal{Q}$ abandons her trajectory and moves toward the reported exit following a straight line.

The value of α_0 is well defined in the statement of Theorem 8. Indeed, by letting $g(x) = f(f(x - \sin(x))) - \sin(x)$, we observe that g is continuous, while $g(1) \approx -0.213934$ and $g(\pi/2) \approx 1.00729$, hence there exists $\alpha_0 \in (1, \pi/2)$ with $g(\alpha_0) = 0$.

In order to prove Theorem 8, and given parameters α, β, we introduce the family of trajectories Search$_1(\alpha,\beta)$, see also Figure 2.

Algorithm Search$_1(\alpha,\beta)$

Robot	#	*Description*	*Trajectory*	*Duration*
$\mathcal{Q}$	0	Move to point C_π	$\mathcal{L}(O, C_\pi, t)$	1
	1	Search circle ccw till point $C_{-\alpha}$	$\mathcal{C}(\pi, t-1)$	$\pi-\alpha$
	2	Move to point $C_{-\alpha+\beta}$,	$\mathcal{L}(C_{-\alpha}, C_{-\alpha+\beta}, t-(1+\pi-\alpha))$	$2\sin(\beta/2)$
	3	Search circle cw till point $C_{-\alpha}$	$\mathcal{C}(\beta-\alpha, 1+\pi-\alpha+2\sin(\beta/2)-t)$	β
$\mathcal{S}_1$	0	Move to point C_π	$\mathcal{L}(O, C_\pi, t)$	1
	1	Search circle cw till point $C_{\beta-\alpha}$	$\mathcal{C}(\pi, -t+1)$	$\pi+\alpha-\beta$

Partitioning the circle clockwise, we see that the arc with endpoints $C_\pi, C_{\pi+\alpha-\beta}$ is searched by $\mathcal{S}_1$, while the remaining of the circle is searched by $\mathcal{Q}$. Therefore, robots' trajectories in Search$_1(\alpha,\beta)$ are feasible, and it is also easy to see that they are continuous as well. The search time equals $1+\pi+\max\{\alpha-\beta, 2\sin(\beta/2)+\beta-\alpha\}$, as well as

$$\mathbb{I}(\mathcal{Q}) = [1, 1+\pi-\alpha] \cup [1+\pi-\alpha+2\sin(\beta/2), 1+\pi-\alpha+2\sin(\beta/2)+\beta], \mathbb{I}(\mathcal{S}_1) = [1, 1+\pi+\alpha-\beta].$$

An illustration of the above trajectories for certain values of α, β can be seen in Figure 2.

First we make some observations pertaining to the monotonicity of the evacuation cost.

▶ **Lemma 9.** *Assuming that $\alpha > \pi/3$ and that $\cos(\alpha) + \cos(\alpha - \beta/2) > 1$, the evacuation cost of* Search$_1(\alpha,\beta)$ *is monotonically increasing if the exit is found by $\mathcal{S}_1$ during $\mathcal{Q}$'s phase 1 and monotonically decreasing if the exit is found by $\mathcal{S}_1$ during $\mathcal{Q}$'s phase 2.*

Proof. Suppose that the exit is found by $\mathcal{S}_1$ during $\mathcal{Q}$'s phase 1, i.e. at time x after robots start searching for the first time, where $0 \le x \le \pi - \alpha$. It is easy to see that the critical angles between $\mathcal{Q}, \mathcal{S}_1$ are both equal to $\pi - x$. But then $2\cos(\pi - x) \geq 2\cos(\alpha) > 2\cos(\pi/3) = 1$. Hence, by Theorem 6, the evacuation cost is decreasing in this case.

Now suppose that the exit is found by $\mathcal{S}_1$ during $\mathcal{Q}$'s phase 2, i.e. at time x after $\mathcal{Q}$ starts moving along the chord with endpoints $C_{-\alpha}, C_{-\alpha+\beta}$, where $0 \leq x \leq 2\sin(\beta/2)$. If ϕ_x, θ_x denote the $\mathcal{S}_1, \mathcal{Q}$ critical angles, then it is easy to see that $\phi_0 = \cos(\alpha)$ and that $\theta_0 = \alpha - \beta/2$. Since $\cos(\phi_0) + \cos(\theta_0) > 1$, Theorem 6 implies that the evacuation cost is initially decreasing in this phase. For the remaining of $\mathcal{Q}$'s phase 2, it is easy to see that both ϕ_x, θ_x are decreasing in x, hence $\cos(\phi_x) + \cos(\theta_x)$ is increasing in x, hence, the evacuation cost will remain decreasing in this phase. ◀

Now we can prove Theorem 8 by fixing certain values for parameters α, β of $\textsc{Search}_1(\alpha, \beta)$. In particular, we set α_0 as in the statement of Theorem 8, and $\beta_0 = 2f(\alpha_0 - \sin(\alpha_0)) \approx 0.925793$. The trajectories of the robots, for the exact same values of the parameters, can be seen in Figure 2.

Proof of Theorem 8. Let f, α_0 be as in the statement of Theorem, and set $\beta_0 = 2f(\alpha_0 - \sin(\alpha_0)) \approx 0.925793$. We argue that the worst evacuation time of $\textsc{Search}_1(\alpha_0, \beta_0)$ is $1 + \pi - \alpha_0 + 2\sin(\alpha_0)$. Note that for the given values of the parameters, we have that $\alpha_0 > \pi/3$, that $\alpha_0 - \sin(\beta_0/2) \leq \beta_0$, and that $\cos(\alpha_0) + \cos(\alpha_0 - \beta_0/2) > 1$.

First we observe that if the exit if found by $\mathcal{Q}$, then the worst case evacuation time $E_0(\alpha_0, \beta_0)$ is incurred when the exit is found just before $\mathcal{Q}$ stops searching, that is

$$E_0(\alpha_0, \beta_0) = 1 + \pi - \alpha_0 + 2\sin(\beta_0/2) + \beta_0.$$

Next we examine some cases as to when the exit is found by $\mathcal{S}_1$. If the exit is found by $\mathcal{S}_1$ during the 1st phase of $\mathcal{Q}$, then the evacuation time is, due to Lemma 9, given as

$$E_1(\alpha_0, \beta_0) = \sup_{1 \leq x \leq 1+\pi-\alpha_0} \{x + \|\mathcal{Q}(x) - \mathcal{S}_1(x)\|\} = 1 + \pi - \alpha_0 + 2\sin(\alpha_0).$$

Recall that $\cos(\alpha_0) + \cos(\alpha_0 - \beta_0/2) > 1$, and so, again by Lemma 9 we may omit the case that the exit is found by $\mathcal{S}_1$ while $\mathcal{Q}$ is at phase 2. The end of $\mathcal{Q}$'s phase 2 happens at time $\tau := 1+\pi-\alpha_0+2\sin(\beta_0/2)$, when have that $\mathcal{Q}(\tau) = C_{-\alpha+\beta}$, and $\mathcal{S}_1(\tau) = C_{\alpha - 2\sin(\beta_0/2)}$, and both robots are intending to search ccw. Condition $\alpha_0 - \sin(\beta_0/2) \leq \beta_0$ says that $\mathcal{S}_1$ will finish searching prior to $\mathcal{Q}$, and this happens when $\mathcal{S}_1$ reaches point $C_{-\alpha+\beta}$. During this phase, the distance between $\mathcal{Q}, \mathcal{S}_1$ stays invariant and equal to $2\alpha_0 - \beta_0 - 2\sin(\beta_0/2)$. We conclude that the cost in this case would be

$$E_2(\alpha_0, \beta_0) = 1 + \pi + \alpha_0 - \beta_0 + 2\sin(\alpha_0 - \beta_0/2 - \sin(\beta_0/2)).$$

Then, we argue that that the choice of α_0, β_0 guarantees that $E_0(\alpha_0, \beta_0) = E_1(\alpha_0, \beta_0) = E_2(\alpha_0, \beta_0)$, as wanted.

Indeed, $E_0(\alpha_0, \beta_0) = E_1(\alpha_0, \beta_0)$ implies that $\sin(\beta_0/2) + \beta_0/2 = \sin(\alpha_0)$. But then, we can rewrite $E_2(\alpha_0, \beta_0)$ as

$$E_2(\alpha_0, \beta_0) = 1 + \pi + \alpha_0 - \beta_0 + 2\sin(\alpha_0 - \sin(\alpha_0)).$$

Equating the last expression with $E_1(\alpha_0, \beta_0)$ implies that

$$\beta_0/2 = \alpha_0 - \sin(\alpha_0) + \sin(\alpha_0 - \sin(\alpha_0)) = f(\alpha_0 - \sin(\alpha_0)).$$

Substituting twice $\beta_0/2$ in the already derived condition $\sin(\beta_0/2) + \beta_0/2 = \sin(\alpha_0)$ implies that

$$f(f(\alpha - \sin(\alpha_0))) = \sin(\alpha_0).$$

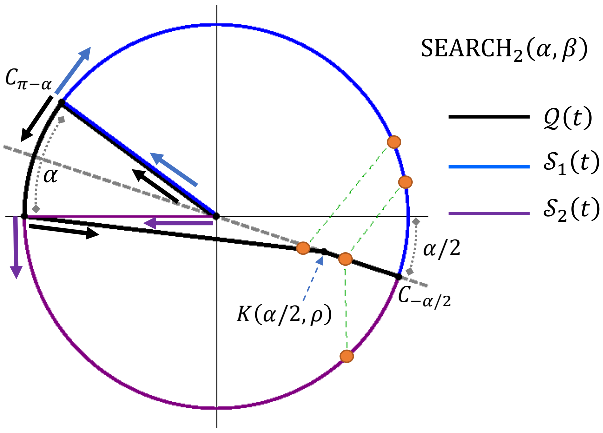


Figure 3 Algorithm Search$_2(\alpha, \beta)$ depicted for the optimal parameters of the algorithm.

Figure 2 depicts the worst placements of the exit, along with the trajectories of the queen (in dashed green lines) after the exit is reported. ◀

It should be stressed that $\mathcal{Q}$'s Phases 2,3 are essential for achieving the promised bound. Indeed, had we chosen $\alpha = \beta = 0$, the worst case evacuation time would have been

$$\sup_{1 \le x \le 1+\pi} \{x + \|\mathcal{Q}(x) - \mathcal{S}_1(x)\|\} = \sup_{0 \le x \le \pi} \{1 + x + 2\sin(x)\}.$$

The maximum is attained at $x_0 = 2\pi/3$ (and indeed, both critical angles in this case are $\pi/3$ and in particular $2\cos(\pi/3) = 1$), inducing cost $1 + 2\pi/3 + \sqrt{3} \approx 4.82645$. The latter is the cost of the evacuation algorithm for two robots without priority of [10].

3.2 Evacuation Algorithm for PE$_2$

In this subsection we prove the following theorem.

▶ **Theorem 10.** PE$_2$ *can be solved in time 3.8327.*

Given parameters α, ρ, we introduce the family of trajectories Search$_2(\alpha, \rho)$, see also Figure 3.

Algorithm Search$_2(\alpha, \rho)$

Robot	#	*Description*	*Trajectory*	*Duration*
$\mathcal{Q}$	0	Move to point $C_{\pi-\alpha}$	$\mathcal{L}(O, C_{\pi-\alpha}, t)$	1
	1	Search the circle ccw till point C_π	$\mathcal{C}(\pi - \alpha, t - 1)$	α
	2	Move to point $K(\alpha/2, \rho)$	$\mathcal{L}(C_\pi, K(\alpha/2,\rho), t - (1+\alpha))$	$AK(\alpha/2, \rho)$
	3	Move to point $C_{-\alpha/2}$	$\mathcal{L}(K(\alpha/2,\rho), C_{-\alpha/2})$	$2 - 2\rho$
$\mathcal{S}_1$	0	Move to point $C_{\pi-\alpha}$	$\mathcal{L}(O, C_{\pi-\alpha})$	1
	1	Search the circle cw till point $C_{-\alpha/2}$	$\mathcal{C}(\pi - \alpha, -t + 1)$	$\pi - \alpha/2$
$\mathcal{S}_2$	0	Move to point C_π	$\mathcal{L}(O, C_\pi)$	1
	1	Search the circle cw till point $C_{-\alpha/2}$	$\mathcal{C}(\pi, t - 1)$	$\pi - \alpha/2$

Notice that, by definition of Search$_2(\alpha, \rho)$, robots' trajectories are continuous and feasible, meaning that the entire circle is eventually searched. Indeed, partitioning the circle clockwise, we see that: the arc with endpoints $C_\pi, C_{\pi-\alpha}$ is searched by $\mathcal{Q}$, the arc with endpoints $C_{\pi-\alpha}, C_{-\alpha/2}$ is searched by $\mathcal{S}_1$, and the arc with endpoints $C_{-\alpha/2}, C_\pi$ is searched by $\mathcal{S}_2$.

It is immediate from the description of the trajectories that the search time is $1+\pi-\alpha/2$. Moreover

$$\mathbb{I}(\mathcal{Q}) = [1, 1+\alpha],\ \mathbb{I}(\mathcal{S}_1) = \mathbb{I}(\mathcal{S}_2) = [1, 1+\pi-\alpha/2].$$

An illustration of the above trajectories for certain values of α, ρ can be seen in Figure 3. Now we make some observations, in order to calculate the worst case evacuation time.

▶ **Lemma 11.** *Suppose that $\pi - \alpha/2 \geq \alpha + AK(\alpha/2, \rho) + 2 - 2\rho$. Then $\|\mathcal{Q}(x) - \mathcal{S}_1(t)\|$ is continuous and differentiable in the time intervals I_1, I_2, I_3 of $\mathcal{Q}$'s phases 1,2,3, respectively. Moreover, the worst case evacuation time of* Search$_2(\alpha, \rho)$ *can be computed as*

$$\max \left\{ \begin{array}{l} 1+\alpha+2\sin(\alpha), \\ \sup_{t\in I_2}\{t + \|\mathcal{Q}(t)-\mathcal{S}_1(t)\|\} \\ \sup_{t\in I_3}\{t + \|\mathcal{Q}(t)-\mathcal{S}_1(t)\|\} \\ 1+\pi-\alpha/2 \end{array} \right\}$$

where

$$I_2 = [1+\alpha, 1+\alpha+AK(\alpha/2,\rho)], I_3 = [1+\alpha+AK(\alpha/2,\rho), 3-2\rho+\alpha+AK(\alpha/2,\rho)].$$

Proof. Note that the line passing through O and $C_{-\alpha/2}$, call it ϵ, has the property that each point of it, including $K(\alpha/2, \rho)$ is equidistant from $\mathcal{S}_1, \mathcal{S}_2$. Moreover, in the time window $[1+\alpha, 1+\alpha+AK(\alpha/2,\rho)]$ that only $\mathcal{S}_1, \mathcal{S}_2$ are searching, $\mathcal{Q}$ stays below line ϵ. At time $1+\alpha+AK(\alpha/2,\rho)$, $\mathcal{Q}$ is, by construction, equidistant from $\mathcal{S}_1, \mathcal{S}_2$, a property that is preserved for the remaining of the execution of the algorithm. As a result, the evacuation time of Search$_2(\alpha,\rho)$ is given by

$$\sup_{1\leq t\leq 1+\pi-\alpha/2} \{t + \|\mathcal{Q}(t)-\mathcal{S}_1(t)\|\}.$$

Now note that condition $\pi - \alpha/2 \geq \alpha + AK(\alpha/2,\rho) + 2 - 2\rho$ guarantees that $\mathcal{Q}$ reaches point $C_{-\alpha/2}$ no later than $\mathcal{S}_1$. Moreover, in each time interval I_1, I_2, I_3, $\mathcal{Q}$'s trajectory is differentiable (and so is $\mathcal{S}_1$'s trajectory). ◀

Now Theorem 10 can be proven by fixing parameters α, ρ for Search$_2(\alpha,\rho)$, in particular, $\alpha = 0.6361, \rho = 0.7944$. Notably, the performance of Search$_2(\alpha,\rho)$ is provably improvable (slightly) using a technique we will describe in the next section.

3.3 Evacuation Algorithm for PE$_3$

3.3.1 A Simple Algorithm

In this section we prove the following preliminary theorem (to be improved in Section 3.3.2).

▶ **Theorem 12.** PE$_3$ *can be solved in time 3.37882.*

Given parameters α, β, ρ, we introduce the family of trajectories Search$_3(\alpha,\beta,\rho)$, corresponding to robots $\mathcal{Q}, \mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$, see also Figure 4.

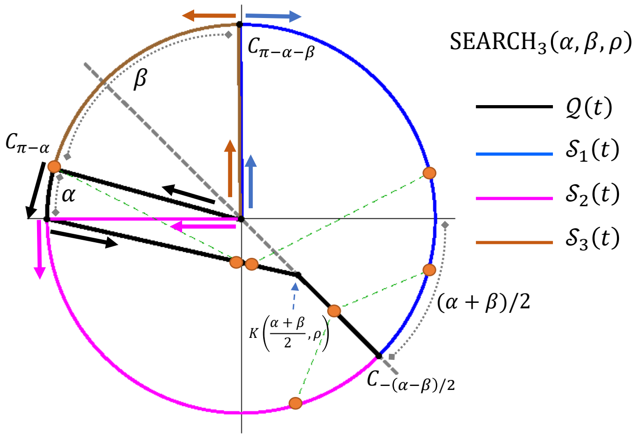


Figure 4 Algorithm Search3(α, β, ρ) depicted for the optimal parameters of the algorithm.

Algorithm Search$_3(\alpha, \beta, \rho)$

Robot	#	*Description*	*Trajectory*	*Duration*
$\mathcal{Q}$	0	Move to point $C_{\pi-\alpha}$	$\mathcal{L}(O, C_{\pi-\alpha}, t)$	1
	1	Search the circle ccw till point C_π	$\mathcal{C}(\pi-\alpha, t-1)$	α
	2	Move to point $K(\frac{\alpha+\beta}{2}, \rho)$	$\mathcal{L}(C_\pi, K(\frac{\alpha+\beta}{2}, \rho), t-(1+\alpha))$	$AK(\frac{\alpha+\beta}{2}, \rho)$
	3	Move to point $C_{-\frac{\alpha+\beta}{2}}$	$\mathcal{L}(K(\frac{\alpha+\beta}{2}, \rho), C_{-\frac{\alpha+\beta}{2}})$	$2-2\rho$
$\mathcal{S}_1$	0	Move to point $C_{\pi-\alpha-\beta}$	$\mathcal{L}(O, C_{\pi-\alpha-\beta})$	1
	1	Search the circle cw till point $C_{-\frac{\alpha+\beta}{2}}$	$\mathcal{C}(\pi-\alpha-\beta, -t+1)$	$\pi-\frac{\alpha+\beta}{2}$
$\mathcal{S}_2$	0	Move to point C_π	$\mathcal{L}(O, C_\pi)$	1
	1	Search the circle ccw till point $C_{-\frac{\alpha+\beta}{2}}$	$\mathcal{C}(\pi, t-1)$	$\pi-\frac{\alpha+\beta}{2}$
$\mathcal{S}_3$	0	Move to point $C_{\pi-\alpha-\beta}$	$\mathcal{L}(O, C_{\pi-\alpha-\beta})$	1
	1	Search the circle ccw till point $C_{-\alpha}$	$\mathcal{C}(\pi-\alpha-\beta, -t+1)$	β

As before, it is immediate that, in Search3(α, β, ρ), robots' trajectories are continuous and feasible, meaning that the entire circle is eventually searched. In particular, the arc with endpoints $C_\pi, C_{\pi-\alpha}$ is searched by $\mathcal{Q}$, the arc with endpoints $C_{\pi-\alpha-\beta}, C_{-\frac{\alpha+\beta}{2}}$ is searched by $\mathcal{S}_1$, the arc with endpoints $C_{-\pi}, C_{-\frac{\alpha+\beta}{2}}$ is searched by $\mathcal{S}_2$, and the arc with endpoints $C_{\pi-\alpha}, C_{\pi-\alpha-\beta}$ is searched by $\mathcal{S}_3$. Also, the search time is $1+\pi-\frac{\alpha+\beta}{2}$, and

$$\mathbb{I}(\mathcal{Q}) = [1, 1+\alpha],\ \mathbb{I}(\mathcal{S}_1) = \mathbb{I}(\mathcal{S}_2) = [1, 1+\pi-\frac{\alpha+\beta}{2}],\ \mathbb{I}(\mathcal{S}_3) = [1, 1+\beta].$$

An illustration of the above trajectories for certain values of α, β, ρ can be seen in Figure 4.

Before we prove Theorem 12, we need to make some observation, in order to calculate the worst case evacuation time.

▶ **Lemma 13.** *Suppose that $\alpha \le \beta$, $\alpha + AK(\frac{\alpha+\beta}{2}, \rho) \ge \beta$, and $\pi - \frac{\alpha+\beta}{2} \ge \alpha + AK(\frac{\alpha+\beta}{2}, \rho) + 2 - 2\rho$. Then the following functions are continuous and differentiable in each associated time intervals: $\|\mathcal{Q}(x) - \mathcal{S}_3(t)\|$ in $I_1 = \{t \ge 0 :\ \alpha \le t-1 \le \beta\}$, $\|\mathcal{Q}(x) - \mathcal{S}_1(t)\|$ in $I_2 = \{t \ge 0 :\ |t-1-\alpha| \le AK(\frac{\alpha+\beta}{2}, \rho)\}$ and in $I_3 = \{t \ge 0 :\ |t-1-\alpha-AK(\frac{\alpha+\beta}{2}, \rho)| \le 2-2\rho\}$. Moreover, the worst case evacuation time of* Search3(α, β, ρ) *can be computed as*

$$\max \left\{ \begin{array}{l} \sup_{t \in I_1} \{t + \|\mathcal{Q}(t) - \mathcal{S}_3(t)\|\} \\ \sup_{t \in I_2} \{t + \|\mathcal{Q}(t) - \mathcal{S}_1(t)\|\} \\ \sup_{t \in I_3} \{t + \|\mathcal{Q}(t) - \mathcal{S}_1(t)\|\} \\ 1 + \pi - \frac{\alpha+\beta}{2} \end{array} \right\}$$

Proof. Conditions $\alpha \leq \beta$ and $\alpha + AK(\frac{\alpha+\beta}{2}, \rho) \geq \beta$ mean that $\mathcal{Q}$ stops searching no later than $\mathcal{S}_3$, and that when $\mathcal{S}_3$ stops searching $\mathcal{Q}$ is still in her phase 2, respectively.

The line passing through O and $C_{-(\alpha+\beta)/2}$, call it ϵ, has the property that each point of it, including $K(\frac{\alpha+\beta}{2}, \rho)$ is equidistant from $\mathcal{S}_1, \mathcal{S}_2$. Moreover, while $\mathcal{S}_1, \mathcal{S}_2$ are searching, $\mathcal{Q}$ never goes above line ϵ. At time $1 + \alpha + AK(\frac{\alpha+\beta}{2}, \rho)$, $\mathcal{Q}$ is, by construction, equidistant from $\mathcal{S}_1, \mathcal{S}_2$, a property that is preserved for the remaining of the execution of the algorithm. As a result, $\mathcal{S}_2$ can be ignored in the performance analysis, and when it comes to the case that $\mathcal{S}_1$ finds the exit, the evacuation cost is given by the supremum of $t + \|\mathcal{Q}(t) - \mathcal{S}_1(t)\|$ in the time interval I_2 or in the interval I_3. Note that in both intervals, the evacuation cost is continuous and differentiable, by construction.

If the exit is reported by $\mathcal{S}_3$ then the evacuation cost is $t + \|\mathcal{Q}(t) - \mathcal{S}_3(t)\|$ for $t \in [1, 1+\beta]$. However, it is easy to see that the cost is strictly increasing for all $t \in [1, 1+\alpha]$ (in fact it is linear). Since the evacuation cost is also continuous, we may restrict the analysis in interval I_1.

Lastly, observe that $\pi - \frac{\alpha+\beta}{2} \geq \alpha + AK(\frac{\alpha+\beta}{2}, \rho) + 2 - 2\rho$ implies that $\mathcal{S}_1, \mathcal{S}_2$ reach point $C_{-(\alpha+\beta)/2}$ no earlier than $\mathcal{Q}$. Hence $\mathcal{Q}$ waits at $C_{-(\alpha+\beta)/2}$ till the search of the circle is over, which can be easily seen to induce the worse evacuation time after $\mathcal{Q}$ reaches $C_{-(\alpha+\beta)/2}$. ◀

We prove Theorem 12 by fixing parameters α, β, ρ for Search$_3(\alpha, \beta, \rho)$, in particular $\alpha = 0.26738, \beta = 1.2949, \rho = 0.70685$.

3.3.2 Improved Search Algorithm

In this section we improve the upper bound of Theorem 12 by 0.00495 additive term.

▶ **Theorem 14.** PE$_3$ *can be solved in time 3.37387.*

The main idea can be described, at a high level, as a cost preservation technique. By the analysis of Algorithm Search$_3(\alpha, \beta, \rho)$ for the value of parameters of α, β, ρ as in the proof of Theorem 12, we know that there are is a critical time window $[\tau_2, \tau_3]$ so that the total evacuation time is the same if the exit is found by $\mathcal{S}_1$ either at time τ_2 or τ_3, and strictly less for time moments strictly in-between. In fact, during time $[\tau_2, 1 + \alpha + AK(\frac{\alpha+\beta}{2}, \rho)]$ $\mathcal{Q}$ is executing phase 2, and in the time window $[1 + \alpha + AK(\frac{\alpha+\beta}{2}, \rho), \tau_3]$ $\mathcal{Q}$ is executing phase 3 of Search$_3(\alpha, \beta, \rho)$.

From the above, it is immediate that we can lower $\mathcal{Q}$'s speed in the time window $[\tau_2, \tau_3]$ so that the evacuation time remains *unchanged* no matter when $\mathcal{S}_1$ finds the exit in the same time interval (notably, $\mathcal{S}_3$ has finished searching prior to τ_2 and $\|\mathcal{Q}(t) - \mathcal{S}_1\| \geq \|\mathcal{Q}(t) - \mathcal{S}_2\|$). But this also implies that we must be able to maintain the evacuation time even if we preserve speed 1 for $\mathcal{Q}$, that will in turn allow us to twist parameters α, β, ρ, hopefully improving the worst case evacuation time. We show this improvement is possible by using the following technical observation

▶ **Theorem 15.** *Consider point $Q = (q_1, q_2) \in \mathbb{R}^2$. Let $\mathcal{S}(t)$ be the trajectory of an object $\mathcal{S}$ moving at speed 1, where $t \geq 0$, and denote by ϕ the $(\mathcal{S}, Q)$-critical angle at time $t = 0$. Assuming that $\cos(\phi) \geq 0$, then there is some $\tau > 0$, and a trajectory $\mathcal{Q}(t) = (f(t), g(t))$ of a speed-1 object, where $t \geq 0$, so that $t + \|\mathcal{Q}(t) - \mathcal{S}(t)\|$ remains constant, for all $t \in [0, \tau]$. Moreover, $\mathcal{Q}(t)$ can be determined by solving the system of differential equations*

$$(f'(t))^2 + (g'(t))^2 = 1 \tag{2}$$

$$t + \|\mathcal{Q}(t) - \mathcal{S}(t)\| = \|\mathcal{S}(0) - Q\| \tag{3}$$

$$(f(0), g(0)) = (q_1, q_2). \tag{4}$$

Proof. An object with trajectory $(f(t), g(t))$ satisfying (2) and (4) has speed 1 (by Lemma 2), and starts from point $Q = (q_1, q_2)$. We need to examine whether we can choose f, g so as to satisfy (3).

By Lemma 7, such a trajectory $\mathcal{Q}(t)$ exists exactly when we can guarantee that $\cos(\phi) + \cos(\theta) = 1$ over time t. When $t = 0$ we are given that $\cos(\phi) > 0$, hence there exists θ satisfying $\cos(\phi) + \cos(\theta) = 1$. This uniquely determines the velocity of $\mathcal{Q}$ at $t = 0$.

By continuity of the velocities, there must exist a $\tau > 0$ such that $\cos(\phi) + \cos(\theta) = 1$ admits a solution for θ also as ϕ changes over time $t \in [0, \tau]$, in which time window the cosine of the $(\mathcal{S}, \mathcal{Q}(t))$-critical angle at time t remains non-negative. ◀

Note that condition $\cos(\phi) \geq 0$ of Theorem 15 translates to that $\|\mathcal{S}(t) - Q\|$ is not increasing at $t = \tau$, i.e. that $\mathcal{S}$ does not move away from point Q.

Now fix parameters α, β, ρ together with the trajectories of $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ as in the description of Algorithm Search$_3(\alpha, \beta, \rho)$. The description of our *new algorithm* N-Search$_3(\alpha, \beta, \rho)$ will be complete once we fix a new trajectory for $\mathcal{Q}$. Naming specific values for parameters α, β, ρ will eventually prove Theorem 14. In order to do so, we introduce some *further notation and conditions*, denoted below by *(Conditions i-iv)*, that we later make sure are satisfied.

Consider $\mathcal{Q}$'s trajectory as in Search$_3(\alpha, \beta, \rho)$. Let τ_0 denote a local maximum of

$$t + \|\mathcal{Q}(t) - \mathcal{S}_1(t)\|$$

as it reads for $t \geq 0$ with $|t - 1 - \alpha| \leq AK(\frac{\alpha+\beta}{2}, \rho)$ (recall that in this time window, expression is differentiable by Lemma 13), i.e.

$$|\tau_0 - 1 - \alpha| \leq AK(\frac{\alpha + \beta}{2}, \rho) \qquad \text{(Condition i)}$$

Set $Q = \mathcal{Q}(\tau_0)$, and assume that

"The cosine of the $(\mathcal{S}, Q)$-critical angle at time τ_0 is non-negative." (Condition ii)

Then obtain from Theorem 15 trajectory $(f(t), g(t))$ that has the property that it preserves $\tau_0 + \|\mathcal{Q}(\tau_0) - \mathcal{S}_1(\tau_0)\|$ in the time window $[\tau_0, \tau']$. Assume also that

"There is time $\tau_1 \leq \tau'$ such that point $K_1 := (f(\tau_1), g(\tau_1))$ is equidistant from $\mathcal{S}_1(\tau_1), \mathcal{S}_2(\tau_1)$," (Condition iii)

for the first time after time τ_0, such that

$$\tau_1 \leq 1 + \pi - \frac{\alpha + \beta}{2}. \qquad \text{(Condition iv)}$$

Then consider the following modification of Search$_3(\alpha, \beta, \rho)$, where the trajectories of $\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3$ remain unchanged, see also Figure 5.

Algorithm N-Search$_3(\alpha, \beta, \rho)$

Robot	#	*Description*	*Trajectory*	*Duration*
$\mathcal{Q}$	0	Move to point $C_{\pi-\alpha}$	$\mathcal{L}(O, C_{\pi-\alpha}, t)$	1
	1	Search the circle ccw till point C_π	$\mathcal{C}(\pi - \alpha, t - 1)$	α
	2	Move toward point $K(\frac{\alpha+\beta}{2}, \rho)$	$\mathcal{L}(C_\pi, K(\frac{\alpha+\beta}{2}, \rho), t - (1 + \alpha))$	$\tau_0 - 1 - \alpha$
	3	Preserve $\tau_0 + \|\mathcal{Q}(\tau_0) - \mathcal{S}_1(\tau_0)\|$	$(f(t), g(t))$	$\tau_1 - \tau_0$
	4	Move to point $C_{-\frac{\alpha+\beta}{2}}$	$\mathcal{L}(K_1, C_{-\frac{\alpha+\beta}{2}})$	$\left\|K_1 - C_{-\frac{\alpha+\beta}{2}}\right\|$

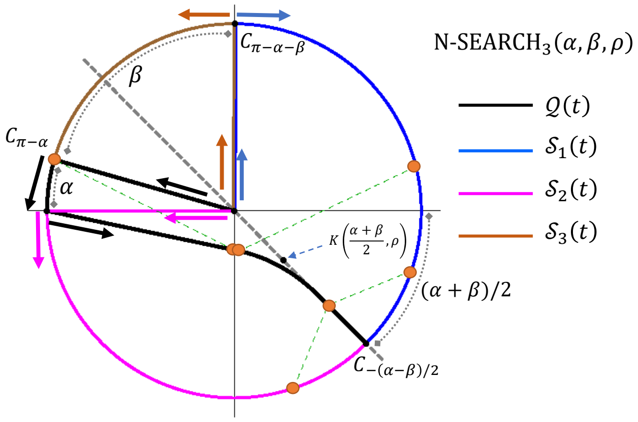


Figure 5 Algorithm SEARCH$_3(\alpha,\beta,\rho)$ depicted for the optimal parameters of the algorithm.

Note that in phase 2, $\mathcal{Q}$ is not reaching (necessarily) point K rather it moves toward it for a certain duration. The search time is still $1+\pi-\frac{\alpha+\beta}{2}$. Trajectories of $\mathcal{S}_1,\mathcal{S}_2,\mathcal{S}_3$ are continuous as before, and

$$\mathbb{I}(\mathcal{S}_1)=\mathbb{I}(\mathcal{S}_2)=[1,1+\pi-\frac{\alpha+\beta}{2}],\ \mathbb{I}(\mathcal{S}_3)=[1,1+\beta],$$

as well as $\mathbb{I}(\mathcal{Q})=[1,1+\alpha]$.

Condition i makes sure that while $\mathcal{Q}$ is at phase 2, and before it reaches $K(\frac{\alpha+\beta}{2},\rho)$, there is a time moment τ_0 when the rate of change of $t+\|\mathcal{Q}(t)-\mathcal{S}_1(t)\|$ is 0. Together with condition ii, this implies that Theorem 15 applies. In fact, for the corresponding critical angles ϕ,θ between $\mathcal{S}_1,\mathcal{Q}$ at time τ_0, we have that $\cos(\phi)+\cos(\theta)=1$ by construction. Hence trajectory $(f(t),g(t))$ of phase 3 is well defined, and indeed, $\mathcal{Q}$ jumps from phase 2 to phase 3 while $\mathcal{Q}$ is still moving toward point K. Notably, $\mathcal{Q}$'s trajectory is even differentiable at $t=\tau_0$ (but not necessarily at $t=\tau_1$). Then, Condition iii says that $\mathcal{Q}$ eventually will enter phase 4, and that this will happen before $\mathcal{S}_1,\mathcal{S}_2$ finish the exploration of the circle. Overall, we conclude that in N-SEARCH$_3(\alpha,\rho)$, robots' trajectories are continuous and feasible. An illustration of the above trajectories for certain values of α,β,ρ can be seen in Figure 5.

Now we make some observations, in order to calculate the worst case evacuation time.

▶ **Lemma 16.** *Suppose that $\alpha\le\beta$, $1+\beta\le\tau_0$, and $1+\pi-\frac{\alpha+\beta}{2}\ge\tau_1+\left\|K_1-C_{-\frac{\alpha+\beta}{2}}\right\|$ as well as Conditions i-iv are satisfied. Then the following functions are continuous and differentiable in each associated time intervals: $\|\mathcal{Q}(x)-\mathcal{S}_3(t)\|$ in $I_1=\{t\ge0:\ \alpha\le t-1\le\beta\}$, $\|\mathcal{Q}(x)-\mathcal{S}_1(t)\|$ in $I_2=\{t\ge0:\ 1+\alpha\le t\le\tau_0$ and in $I_3=\left\{t\ge0:\ |t-\tau_1|\le\left\|K_1-C_{-\frac{\alpha+\beta}{2}}\right\|\right\}$. Moreover, the worst case evacuation time of* N-SEARCH$_3(\alpha,\beta,\rho)$ *can be computed as*

$$\max\left\{\begin{array}{l}\sup_{t\in I_1}\{t+\|\mathcal{Q}(t)-\mathcal{S}_3(t)\|\}\\ \sup_{t\in I_2}\{t+\|\mathcal{Q}(t)-\mathcal{S}_1(t)\|\}\\ \sup_{t\in I_3}\{t+\|\mathcal{Q}(t)-\mathcal{S}_1(t)\|\}\\ 1+\pi-\frac{\alpha+\beta}{2}\end{array}\right\}$$

Proof. Conditions $\alpha\le\beta$ and $1+\beta\le\tau_0$ mean that $\mathcal{Q}$ stops searching no later than $\mathcal{S}_3$, and that when $\mathcal{Q}$ enters phase 3 after $\mathcal{S}_3$ is done searching, respectively.

The line passing through O and $C_{-(\alpha+\beta)/2}$, call it ϵ, has the property that each point of it, including $K(\frac{\alpha+\beta}{2},\rho)$ is equidistant from $\mathcal{S}_1,\mathcal{S}_2$. Moreover, while $\mathcal{S}_1,\mathcal{S}_2$ are searching,

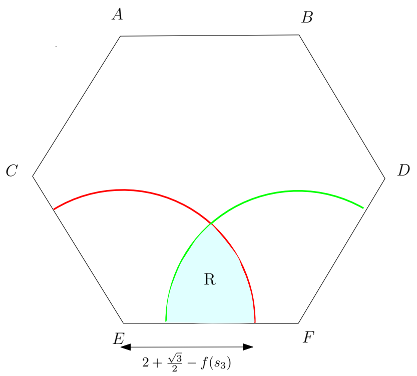


Figure 6 (Left) The queen must be in region R at time $f(s_3)$. Here $s_3 = E$ and $q_3 = F$.

$\mathcal{Q}$ never goes above line ϵ. Also, while $\mathcal{Q}$ is executing phase 3, $\mathcal{Q}$ remains equidistant from $\mathcal{S}_1, \mathcal{S}_2$ and this is preserved for the remainder of the execution of the algorithm. As a result, $\mathcal{S}_2$ can be ignored in the performance analysis, and when it comes to the case that $\mathcal{S}_1$ finds the exit, the evacuation cost is given by the supremum of $t + \|\mathcal{Q}(t) - \mathcal{S}_1(t)\|$ in the time interval I_2 or in the interval I_3. Note that in both intervals, the evacuation cost is continuous and differentiable, by construction.

If the exit is reported by $\mathcal{S}_3$ then the evacuation cost is $t + \|\mathcal{Q}(t) - \mathcal{S}_3(t)\|$ for $t \in [1, 1+\beta]$. However, it is easy to see that the cost is strictly increasing for all $t \in [1, 1+\alpha]$ (in fact it is linear). Since the evacuation cost is also continuous, we may restrict the analysis in interval I_1.

Lastly, observe that $1 + \pi - \frac{\alpha+\beta}{2} \geq \tau_1 + \left\| K_1 - C_{-\frac{\alpha+\beta}{2}} \right\|$ implies that $\mathcal{S}_1, \mathcal{S}_2$ reach point $C_{-(\alpha+\beta)/2}$ no earlier than $\mathcal{Q}$. Hence $\mathcal{Q}$ waits at $C_{-(\alpha+\beta)/2}$ till the search of the circle is over, which can be easily seen to induce the worse evacuation time after $\mathcal{Q}$ reaches $C_{-(\alpha+\beta)/2}$. ◀

We can prove now Theorem 14 by fixing parameters α, β, ρ for N-Search$_3(\alpha, \beta, \rho)$, in particular $\alpha = 0.27764, \beta = 1.29839, \rho = 0.68648$.

4 Lower Bounds

In this section we derive lower bounds for evacuation. In Section 4.1 we treat the case of $n = 1$ (see Theorem 17) and in Section 4.2 we treat the case of $n = 2$ and 3 (see Theorem 19).

4.1 Lower Bound for PE$_1$

We will derive the lower bound using an adversarial argument placing the exit at an unknown vertex of a regular hexagon.

▶ **Theorem 17.** *The worst-case evacuation time for* PE$_1$ *is at least* $3 + \pi/6 + \sqrt{3}/2 \approx 4.3896$

Proof. At time $1 + \pi/6$, at most $\pi/3$ of the perimeter of the circle can have been explored by the queen and servant. Thus, there is a regular hexagon, none of whose vertices have been explored. If the exit is at one of these vertices, by Theorem 18, it takes $2 + \sqrt{3}/2$ for the queen to evacuate. The total time is $1 + \pi/6 + 2 + \sqrt{3}/2$. ◀

Next we proceed to provide a lower bound on a unit-side hexagon. Label the vertices of the hexagon V as $A, \ldots, F$ as shown in Figure 6. Fix an evacuation algorithm $\mathcal{A}$. For any vertex v of the hexagon, we call $f(v)$ the time of *first visit* of the vertex v by either the servant or the queen, according to algorithm $\mathcal{A}$. We call $q(v)$ the time that the queen gets to the vertex v. Clearly, $q(v) \geq f(v)$, and if the queen arrives at the vertex no later than the servant, $q(v) = f(v)$.

▶ **Theorem 18.** *For any algorithm $\mathcal{A}$, the evacuation time for the queen when the exit is at one of the vertices of the hexagon is $max_{v \in V}\{q(v)\} \geq 2 + \sqrt{3}/2$.*

Proof. Suppose there is an algorithm in which the queen can always evacuate in time $< 2 + \sqrt{3}/2$. Consider the trajectories of the servant and the queen. If either the queen or the servant are the first to visit 4 vertices, then for the fourth such vertex v, we have $f(v) \geq 3$, a contradiction. Therefore, the queen is the first to visit three vertices, and the servant is the first to visit three vertices. We denote the three vertices visited first by the servant as s_1, s_2, s_3 (in the order they are visited) and the three vertices visited first by the queen as q_1, q_2, q_3, and note that they are all distinct.

Notice that neither s_3 nor q_3 can be visited before time 2, that is, $f(s_3), f(q_3) \geq 2$. If $f(q_3) \leq f(s_3)$, then we place the exit at s_3, and the queen needs time at least 1 to get to s_3, which implies that $T \geq q(s_3) \geq f(q_3) + 1 \geq 3$, a contradiction. We conclude that at time $f(s_3)$, the queen is yet to visit q_3. Since the exit can be at either s_3 or q_3, at time $f(s_3)$, the queen must be at distance $< 2 + \sqrt{3}/2 - f(s_3) \leq \sqrt{3}/2$ from *both* s_3 and q_3.

Assume without loss of generality that $s_3 = E$ (see Figure 6). Since A, B, D are all at distance at least $\sqrt{3}$ from E, we conclude that q_3 is either C or F. Assume without loss of generality that $q_3 = F$. Let R denote the lens-shaped region that is at distance $< 2 + \sqrt{3}/2 - f(s_3)$ from both E and F. Recall that at time $f(s_3)$, the queen must be inside the region R. Notice that if $f(s_3) \geq 1.5 + \sqrt{3}/2$, the region R is empty, yielding a contradiction. So it must be that $2 \leq f(s_3) < 1.5 + \sqrt{3}/2$.

We now work backwards to deduce the trajectories of the servant and the queen. Clearly $s_2 \neq F$ since $q_3 = F$. If $s_2 \neq C$, then $f(s_3) \geq \sqrt{3} + 1 > 1.5 + \sqrt{3}/2$, a contradiction. Therefore, $s_2 = C$. By the same reasoning, $s_1 = A$. Therefore, the queen is the first to visit D and B. If $q_1 = D$ and $q_2 = B$, we place the exit at E; since $f(q_2) \geq 1$ and $dist(B, E) = 2$, we have $T \geq q(E) \geq 3$, a contradiction. Thus, $q_2 = D$ and $q_1 = B$.

Consider the location of the queen at time 1. If she is at distance $\geq 1 + \sqrt{3}/2$ from C at time 1, then if the exit is at C, $q(C) \geq 2 + \sqrt{3}/2$. So at time 1, the queen must be at distance $< 1 + \sqrt{3}/2$ from C and consequently she is at distance $\geq 1 - \sqrt{3}/2$ from vertex D. Therefore $f(q_2) = f(D) \geq 2 - \sqrt{3}/2$. Also, $f(D) < 1.5$ since if the queen reaches D at or after time 1.5, she cannot reach the region R before time $1.5 + \sqrt{3}/2 > f(s_3)$. So $f(D) \leq f(s_3)$. If the exit is at $E = s_3$, the queen cannot reach the exit before time $f(D) + dist(D, E) \geq 2 - \sqrt{3}/2 + \sqrt{3} = 2 + \sqrt{3}$, concluding the proof by contradiction. ◀

We remark that the above bound is optimal, and is achieved by the algorithm depicted in Figure 7.

4.2 Lower Bounds for PE_2 and PE_3

In the case of $n = 2$ and $n = 3$ the proof is rather technical and we will only present a high level outline as to why the lower bounds hold.

▶ **Theorem 19.** *The worst-case evacuation time for* PE_2 *is at least* 3.6307 *and for* PE_3 *at least* 3.2017*.*

Throughout this section we will use $\mathcal{T}$ to refer to the evacuation time of an arbitrary algorithm and use $\mathcal{U}$ to refer to the unit circle which must be evacuated.

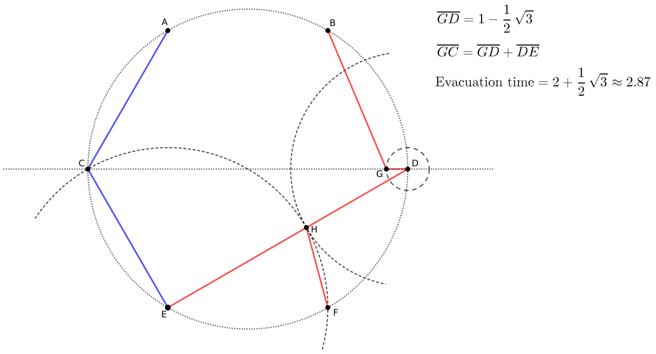

A
B
C
D
G
H
E
F
$\overline{GD} = 1 - \frac{1}{2}\sqrt{3}$
$\overline{GC} = \overline{GD} + \overline{DE}$
Evacuation time $= 2 + \frac{1}{2}\sqrt{3} \approx 2.87$

Figure 7 Blue trajectory: servant and red trajectory: queen. At point H, if the queen hears of an exit at E, she goes there, otherwise she goes to F.

The main thrust of the proof relies on a simple idea – the queen should aid in the exploration of $\mathcal{U}$. This is immediately evident for the particular case of $n = 2$ since, if the queen does not explore, it will take time at least $1 + \pi$ for the servants to search all of $\mathcal{U}$ and we already have an upper bound smaller than this (Theorem 10). Thus, a general overview of the proof is as follows: we show that in order to evacuate in time $\mathcal{T}$ the queen must explore some minimum length of the perimeter of $\mathcal{U}$. We will then demonstrate that the queen is not able to explore this minimum amount in any algorithm with evacuation time smaller than what is given in Theorem 19.

To be concrete, consider the case of $n = 2$ and assume that we have an algorithm with evacuation time $\mathcal{T} < 1 + \pi$. Then, in order for the robots to have explored all of $\mathcal{U}$ in time $\mathcal{T}$, the queen must explore a subset of the perimeter of total length at least $2(1 + \pi - \mathcal{T})$. Intuitively, this minimum length of perimeter will increase in size as $\mathcal{T}$ decreases.

Now consider that it is not possible for the queen to always remain on the perimeter (indeed, in each of the algorithms presented, the queen leaves the perimeter). To see why this is consider that, in any algorithm with evacuation time $\mathcal{T}$, it must be the case that all unexplored points of $\mathcal{U}$ are located a distance no more than $\mathcal{T} - t$ from the queen at all times $t \leq \mathcal{T}$. If the queen is on the perimeter at any time t satisfying $\mathcal{T} - t \leq 2$, then, there will be some arc $\theta(t, \mathcal{T}) \subset \mathcal{U}$ such that all points of $\theta(t, \mathcal{T})$ are at a distance at least $\mathcal{T} - t$ from the queen. Thus, if the queen is to be on the perimeter at the time t we can conclude that all of the arc $\theta(t, \mathcal{T})$ must have already been discovered. However, we will find that $\theta(t, \mathcal{T})$ will often grow at a rate much larger than the robots can collectively explore and at some point the queen will have to leave the perimeter. In fact, there will be an interval of time during which it is not possible for the queen to be exploring and this in turn implies that there is a maximum amount of perimeter that can be explored by the queen. Intuitively, the maximum length of perimeter that can be explored by the queen will decrease as $\mathcal{T}$ decreases. The lower bound will result by balancing the minimum amount of perimeter the queen needs to search and the maximum amount of perimeter that the queen is able to search.

The above argument will need a slight modification in the case of $n = 3$. In this case we will show that there is some critical time t_* before which the queen must have explored some minimum amount of perimeter. Again, the lower bound follows by balancing the maximum amount of perimeter the queen can explore by the time t_* and the minimum amount of perimeter the queen needs to explore before the time t_*.

5 Conclusion

We considered an evacuation problem concerning priority searching on the perimeter of a unit disk where only one robot (the queen) needs to find the exit. In addition to the queen, there are $n \leq 3$ other robots (servants) aiding the queen by contributing to the exploration of the disk but which do not need to evacuate. We proposed evacuation algorithms and studied non-trivial tradeoffs on the queen evacuation time depending on the number n of servants. In addition to analyzing tradeoffs and improving the bounds obtained for the wireless communication model, an interesting open problem would be to investigate other models with limited communication range, e.g., face-to-face.

References

1 R. Ahlswede and I. Wegener. *Search problems.* Wiley-Interscience, 1987.

2 S. Alpern and S. Gal. *The theory of search games and rendezvous*, volume 55. Kluwer Academic Publishers, 2002.

3 Steve Alpern, Robbert Fokkink, Leszek Gąsieniec, Roy Lindelauf, and V.S. Subrahmanian, editors. *Ten Open Problems in Rendezvous Search*, pages 223–230. Springer NY, New York, NY, 2013.

4 R. Baeza Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, 1993.

5 R. Baeza-Yates and R. Schott. Parallel searching in the plane. *Computational Geometry*, 5(3):143–154, 1995.

6 A. Beck. On the linear search problem. *Israel J. of Mathematics*, 2(4):221–228, 1964.

7 R. Bellman. An optimal search. *SIAM Review*, 5(3):274–274, 1963.

8 S. Brandt, F. Laufenberg, Y. Lv, D. Stolz, and R. Wattenhofer. Collaboration without communication: Evacuating two robots from a disk. In *Proceedings of Algorithms and Complexity - 10th International Conference, CIAC 2017, Athens, Greece, May 24-26, 2017*, pages 104–115, 2017.

9 J. Czyzowicz, S. Dobrev, K. Georgiou, E. Kranakis, and F. MacQuarrie. Evacuating two robots from multiple unknown exits in a circle. *Theor. Comput. Sci.*, 709:20–30, 2018.

10 J. Czyzowicz, L. Gasieniec, T. Gorry, E. Kranakis, R. Martin, and D. Pajak. Evacuating robots from an unknown exit located on the perimeter of a disc. In *Proceedings DISC, Austin, Texas*, pages 122–136. Springer, 2014.

11 J. Czyzowicz, K. Georgiou, M. Godon, E. Kranakis, D. Krizanc, W. Rytter, and M. Wlodarczyk. Evacuation from a disc in the presence of a faulty robot. In *Proceedings SIROCCO 2017, 19-22 June 2017, Porquerolles, France*, pages 158–173, 2018.

12 J. Czyzowicz, K. Georgiou, R. Killick, E. Kranakis, D. Krizanc, L. Narayanan, J. Opatrny, and S. Shence. God Save the Queen. *CoRR*, abs/1804.06011, 2018.

13 J. Czyzowicz, K. Georgiou, R. Killick, E. Kranakis, D. Krizanc, L. Narayanan, J. Opatrny, and S. Shende. Priority evacuation from a disk using mobile robots, 2018, Submitted.

14 J. Czyzowicz, K. Georgiou, E. Kranakis, L. Narayanan, J. Opatrny, and B. Vogtenhuber. Evacuating robots from a disk using face-to-face communication (extended abstract). In *Proceedings of Algorithms and Complexity, CIAC 2015, Paris, France, May 20-22, 2015*, pages 140–152, 2015.

15 J. Czyzowicz, E. Kranakis, D. Krizanc, L. Narayanan, J. Opatrny, and S. Shende. Wireless autonomous robot evacuation from equilateral triangles and squares. In *Proceedings of Ad-hoc, Mobile, and Wireless Networks, ADHOC-NOW, Athens, Greece, June 29 - July 1, 2015*, pages 181–194, 2015.

16 Konstantinos Georgiou, George Karakostas, and Evangelos Kranakis. Search-and-fetch with one robot on a disk - (track: Wireless and geometry). In *Algorithms for Sensor Systems - 12th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2016, Aarhus, Denmark, August 25-26, 2016, Revised Selected Papers*, pages 80–94, 2016.

17 Konstantinos Georgiou, George Karakostas, and Evangelos Kranakis. Search-and-fetch with 2 robots on a disk - wireless and face-to-face communication models. In Federico Liberatore, Greg H. Parlier, and Marc Demange, editors, *Proceedings of the 6th International Conference on Operations Research and Enterprise Systems, ICORES 2017, Porto, Portugal, February 23-25, 2017*, pages 15–26. SciTePress, 2017.

18 I. Lamprou, R. Martin, and S. Schewe. Fast two-robot disk evacuation with wireless communication. In *Proceedings DISC, Paris, France*, pages 1–15, 2016.

19 D. Pattanayak, H. Ramesh, P.S. Mandal, and S. Schmid. Evacuating two robots from two unknown exits on the perimeter of a disk with wireless communication. In *Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018*, pages 20:1–20:4, 2018.

20 L. Stone. *Theory of optimal search.* Academic Press New York, 1975.

Restricted Power – Computational Complexity Results for Strategic Defense Games

Ronald de Haan
Institute for Logic, Language and Computation, University of Amsterdam, the Netherlands
me@ronalddehaan.eu
https://orcid.org/0000-0003-2023-0586

Petra Wolf
Wilhelm-Schickard-Institut, University of Tübingen, Germany
wolfp@informatik.uni-tuebingen.de

Abstract

We study the game *Greedy Spiders*, a two-player strategic defense game, on planar graphs and show PSPACE-completeness for the problem of deciding whether one player has a winning strategy for a given instance of the game. We also generalize our results in metatheorems, which consider a large set of strategic defense games. We achieve more detailed complexity results by restricting the possible strategies of one of the players, which leads us to Σ_2^p- and Π_2^p-hardness results.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Computational complexity, generalized games, metatheorems

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.17

Funding Ronald de Haan is supported by the Austrian Science Fund (FWF), project J4047.

Acknowledgements We thank Janosch Döcker and Britta Dorn for proof-reading.

1 Introduction

With computational devices in nearly everyone's pockets nowadays, the opportunities to play puzzle games on these devices are plentiful. What makes such games so addictive that they are played every day by millions of people? One possible answer to the suggested question is that (generalized variants of) these games are computationally intractable [9, 13], which could explain why it can be so challenging to find a solution or to get a good score.

In this paper, we analyze the two-player strategic defense game *Greedy Spiders* [3] from a computational complexity perspective. In the game Player 1 must prevent Player 2 from reaching designated positions. In particular, we show that the problem of deciding whether Player 1 or 2 has a winning strategy is PSPACE-complete. We also generalize this result to state two metatheorems, which can be applied to a larger set of strategic defense games. These metatheorems additionally claim that the problem becomes Σ_2^p-hard if we restrict the possible strategies of Player 1 to those that can be specified by a polynomial-time computable algorithm that is to be submitted at the beginning of the game – the problem is Π_2^p-hard if we restrict Player 2 in a similar way. In both cases, the question is whether Player 1 has a winning strategy. We get hardness results for the complementary classes, if we ask whether Player 2 has a winning strategy.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 17; pp. 17:1–17:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Related Work

While two-player board games have been studied well from a computational complexity point of view in the 80's [15, 16], two-player computer games are rarely examined till now. Despite the fact that the first gaming console was released in 1983 [4], it took until 2000 until the first computer games where studied in terms of their computational complexity. A good survey is given by Demaine et al. [9] and Kendall et al. [13]. While the first results where obtained by examining concrete games, Demaine et al. [10] made a first approach to find more general structures in games by developing a directed graph based framework for which they showed several hardness results for different versions. The framework was intended to be "a natural problem to reduce from." That approach instantly led to complexity results for games like *Sokoban*, *Rush Hour*, *Pushing blocks*, and many more. This proposal was taken up by Forišek [12], who coined the term "metatheorem" to describe complexity results for abstracted games consisting of a combination of game elements, which are often implemented in real computer games. The reduction from a metatheorem to a concrete computer game is obtained by proving that all the elements of one metatheorem can be implemented with the mechanics provided by the game. Note that this is often much easier than finding an individual reduction from a computationally hard formal problem to a certain computer game. Viglietta [19] further developed this approach with several metatheorems that are particularly useful for platform- and puzzle-games. His metatheorems have been used to study the complexity of the best-known Nintendo games [6]. Demaine, Lockhart and Lynche also continued the study of metatheorems for platform-games [11]. To our knowledge, very few of the currently known metatheorems are suitable to describe two-player games (the only one known to us are given by Demaine and Hearn [10]) and most of the known metatheorems are applicable to single-player platform- and puzzle-games only.

3 Greedy Spiders

We describe the game *Greedy Spiders* [3] as a two-player game – in the version of the game for iOS and Android devices, the user plays as Player 1, and the moves of Player 2 are determined by the application. In this paper, we consider the most basic variant of the game. We describe the game in intuitive terms, before we give a fully detailed formal description of the game.

Informal Description of the Game

Greedy Spiders is a turn-based strategic defense two-player game played on planar graphs (that represent spider webs). Initially, some nodes of the graph are occupied by spiders, and some nodes of the graph are occupied by flies. The players alternate turns, and Player 1 plays first. In each turn of Player 1, she removes an edge from the graph, and Player 2 in each of her turns moves a subset of the spiders (possibly all) along a remaining edge of the graph to an adjacent node. The flies cannot move. Player 2 wins whenever some spider occupies the same node as some fly, and Player 1 wins whenever there is no path anymore from any of the spiders to any of the flies.

Formal Description of the Game

A *game situation* (for *Greedy Spiders*) is represented by a triple $C = (G, S, F)$, where $G = (V, E)$ is an undirected planar graph, $S \subseteq V$ is the set of nodes that are occupied by spiders, and $F \subseteq V$ is the set of nodes that are occupied by flies.

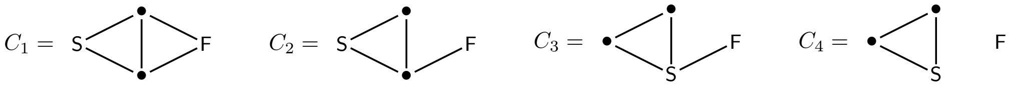


Figure 1 Example of a run $\sigma = (C_1, \dots, C_4)$ that is winning for Player 1.

A *valid move for Player 1* consists of a tuple (C_1, C_2), where $C_1 = (G_1, S_1, F_1)$ and $C_2 = (G_2, S_2, F_2)$ are game situations, such that $S_1 = S_2$, $F_1 = F_2$, and G_2 is obtained from G_1 by removing one edge, that is, $G_1 = (V, E)$ and $G_2 = (V, E \setminus \{e\})$ for some $e \in E$. A *valid move for Player 2* consists of a tuple (C_1, C_2), where $C_1 = (G_1, S_1, F_1)$ and $C_2 = (G_2, S_2, F_2)$ are game situations, for which holds that $G_1 = G_2 = (V, E)$; that $S_2 = \{\, f(s) \mid s \in S_1 \,\}$, where $f\colon S_1 \to V$ is an injective function such that for all $s \in S_1$ it holds that $\{s, f(s)\} \in E$ or $f(s) = s$; and that $F_1 = F_2$.

A game situation (G, S, F) is *winning for Player 1* if for each $s \in S$ and each $f \in F$, there is no path in G from s to f. A game situation (G, S, F) is *winning for Player 2* if $S \cap F \neq \emptyset$. A game situation is *terminal* if it is winning for either of the players, and it is *non-terminal* otherwise.

A *run* σ of the game is a finite sequence $(C_1, \dots, C_n)$ of game situations where (1) for each odd $i \in [n-1]$ it holds that (C_i, C_{i+1}) is a valid move for Player 1, (2) for each even $i \in [n-1]$ it holds that (C_i, C_{i+1}) is a valid move for Player 2, (3) for each $i \in [n-1]$ it holds that C_i is non-terminal, and (4) C_n is terminal. (For each $u, v \in \mathbb{N}$, we use $[u]$ to denote the set $\{1, \dots, u\}$ and $[u, v]$ to denote the set $\{u, \dots, v\}$.) The run σ is winning for either of the players if and only if C_n is.

▶ **Example 1.** See Figure 1 for an example of a run σ of the game *Greedy Spiders* that is winning for Player 1. In this figure (as in all figures in this paper), nodes that are occupied by a spider are marked with S and nodes that are occupied by a fly are marked with F.

We invite the reader to play the game and to verify that there is in fact a winning strategy for Player 1 for the initial game situation C_1 depicted in Figure 1.

A *strategy for Player 1* for an initial game situation C_1 is a finite tree T where each node is labeled with a pair (C, j), where C is a game situation and $j \in [2]$, that satisfies the following conditions:

(a) the root of T is labeled with $(C_1, 1)$;
(b) whenever a node is labeled with $(C, 1)$, for some non-terminal game situation C, it has one single child that is labeled with $(C', 2)$ such that (C, C') is a valid move for Player 1;
(c) whenever a node is labeled with $(C, 2)$, for some non-terminal game situation C, it has m children nodes that are labeled with $(C_1, 1), \dots, (C_m, 1)$, respectively, where $\{(C, C_1), \dots, (C, C_m)\}$ is the set of all valid moves for Player 2 that have C as first component; and
(d) whenever a node is labeled with (C, j), for some terminal game situation C and some $j \in [2]$, it has no children (i.e., it is a leaf).

In the remainder of this paper, we will often slightly abuse notation by identifying a node of a strategy T with the pair (C, j) with which it is labeled. A strategy T for Player 1 is *winning* if all its leaves are labeled with pairs (C, j) where C is winning for Player 1. (In fact, it can easily be verified that this can only be the case if each leaf is labeled with a pair $(C, 2)$ for some game situation C that is winning for Player 1.) Note that any root-to-leaf path in the strategy T corresponds to a run of the game.

Intuitively, a strategy for Player 1 specifies a sequence of valid moves for Player 1 for each possible combination of valid moves that Player 2 makes. A winning strategy for Player 1 specifies what moves Player 1 can make to ensure that she wins the game. (Winning) strategies for Player 2 are defined analogously. Since *Greedy Spiders* is a zero-sum game, there is a winning strategy for Player 1 if and only if there is no winning strategy for Player 2.

Decision Problem

We consider the following decision problems in this paper.

WINNER DETERMINATION FOR PLAYER 1 *Input:* An initial game situation C_1.
Question: Is there a winning strategy for Player 1 for the game situation C_1?

WINNER DETERMINATION FOR PLAYER 2 *Input:* An initial game situation C_1.
Question: Is there a winning strategy for Player 2 for the game situation C_1?

Because the game *Greedy Spiders* never ends in a tie (either Player 1 or Player 2 wins), these problems are complementary. That is, Player 1 has a winning strategy if and only if Player 2 does not have a winning strategy.

4 Preliminaries

We assume the reader to be familiar with basic notions from the theory of computational complexity, such as the complexity classes P and NP, and polynomial-time (many-to-one) reductions. For more details, we refer to textbooks on the topic (e.g., see [7]).

The class PSPACE consists of all decision problems that can be solved by an algorithm that uses a polynomial amount of space (memory). Alternatively, one can characterize the class PSPACE as all decision problems for which there exists a polynomial-time reduction to the problem TQBF, that is defined using quantified Boolean formulas as follows. A quantified Boolean formula (in prenex form) is a formula of the form $Q_1 x_1 Q_2 x_2 \dots Q_n x_n . \psi$, where all x_i are propositional variables, each Q_i is either an existential or a universal quantifier, and ψ is a (quantifier-free) propositional formula over the variables $x_1, \dots, x_n$ (called the *matrix*). Truth for such formulas is defined in the usual way. The problem TQBF consists of deciding whether a given quantified Boolean formula is true. It is well-known that the problem TQBF is PSPACE-complete, and that it remains PSPACE-hard even when restricted to quantified Boolean formulas whose matrix is in 3CNF.

The class PSPACE can also be characterized using alternating Turing machines (ATMs). A problem is in PSPACE if and only if it can be solved in polynomial time by an alternating Turing machine [8]. We refer to textbooks on complexity theory for more details (e.g., see [7]).

One can also restrict the number of quantifier alternations occurring in quantified Boolean formulas, i.e., the number of times where $Q_i \neq Q_{i+1}$. For each constant $k \geq 1$ number of alternations, this leads to a different complexity class. These classes together constitute the Polynomial Hierarchy. We consider the complexity classes Σ^{p}_k, for each $k \geq 1$. The complexity class Σ^{p}_k consists of all decision problems for which there exists a polynomial-time reduction to the problem $\mathrm{TQBF}_{\exists,k}$, that is defined as follows. Instances of the problem are quantified Boolean formulas of the form $\exists x_1 \dots \exists x_{\ell_1} \forall x_{\ell_1+1} \dots \forall x_{\ell_2} \dots Q_k x_{\ell_{k-1}+1} \dots Q_k x_{\ell_k} . \psi$, where $Q_k = \exists$ if k is odd and $Q_k = \forall$ if k is even, where $1 \leq \ell_1 \leq \dots \leq \ell_k$, and where ψ is quantifier-free. The problem is to decide if the quantified Boolean formula is true. For each $k \geq 1$, the dual problem $\mathrm{TQBF}_{\forall,k}$ is defined analogously, where the first quantifier of the formula is universal rather than existential. The complexity class Π^{p}_k consists of all decision problems for which there exists a polynomial-time reduction to the problem $\mathrm{TQBF}_{\forall,k}$. The class NP coincides with Σ^{p}_1, and the class co-NP coincides with Π^{p}_1.

5 Complexity Results for Greedy Spiders

In this section, we show that the problems Winner Determination for Player 1 and Winner Determination for Player 2 for *Greedy Spiders* are PSPACE-complete. Since these problems are complementary, we focus on Winner Determination for Player 1. The result for Winner Determination for Player 2 will then follow immediately, because PSPACE is closed under complement. We begin with showing membership in PSPACE.

▶ **Lemma 2.** Winner Determination for Player 1 *for* Greedy Spiders *is in* PSPACE.

Proof. Let $C_1 = (G, S, F)$ be an initial game situation, where $G = (V, E)$. Since each valid move for Player 1 removes an edge from G, we know that every possible run σ that starts with C_1 is of length at most $2|E|-1$. Therefore, the problem can be solved in polynomial time by an alternating Turing machine. We describe the algorithm that is implemented by such an alternating Turing machine. The algorithm starts with a partial run $\sigma = (C_1)$ that is extended to a complete run. Then, whenever the partial run $\sigma = (C_1, \dots, C_\ell)$ ends with a non-terminal game situation C_ℓ and is of odd length, the algorithm uses existential nondeterminism to guess a game situation $C_{\ell+1}$ such that $(C_\ell, C_{\ell+1})$ is a valid move for Player 1, resulting in the partial run $(C_1, \dots, C_{\ell+1})$. Whenever the partial run $\sigma = (C_1, \dots, C_\ell)$ ends with a non-terminal game situation C_ℓ and is of even length, the algorithm uses universal nondeterminism to guess a game situation $C_{\ell+1}$ such that $(C_\ell, C_{\ell+1})$ is a valid move for Player 2, resulting in the partial run $(C_1, \dots, C_{\ell+1})$. Whenever the partial run $\sigma = (C_1, \dots, C_\ell)$ ends with a terminal game situation C_ℓ, the algorithm accepts if and only if C_ℓ is winning for Player 1. ◀

Next, to show PSPACE-hardness of Winner Determination for Player 1 for *Greedy Spiders*, we will need a technical lemma that states that TQBF is PSPACE-hard even when restricted to instances with a matrix in 3DNF whose incidence graph is planar.

Let $\varphi = Q_1 x_1 \dots Q_n x_n.\psi$ be a quantified Boolean formula, where ψ is a quantifier-free DNF formula. Suppose that $\psi = d_1 \vee \dots \vee d_m$. The *incidence graph* G_φ of φ is a bipartite graph that is defined as follows. The nodes V_φ of G_φ are the literals and the terms of ψ, i.e., $V_\varphi = \{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\} \cup \{d_1, \dots, d_m\}$. A node corresponding to a literal l is connected by an edge to a node corresponding to a term d_j if and only if l occurs in the term d_j. The incidence graph of a formula with a matrix in CNF is defined analogously. (Often a variant of incidence graphs with vertices only for variables, not literals, is used.)

▶ **Lemma 3.** TQBF *is* PSPACE*-hard even when restricted to quantified Boolean formulas (in prenex form) whose incidence graph is planar and whose matrix is a 3DNF formula.*

Proof. It has been shown that TQBF remains PSPACE-hard when restricted to quantified Boolean formulas (in prenex form) whose matrix is a 3CNF formula and whose incidence graph is planar [14, Theorem 1]. This result can easily be adapted to work also for incidence graphs with vertices for literals (by introducing existentially quantified copies of variables and adding clauses to ensure that copies are assigned the same truth value). Then, since PSPACE is closed under complement, and the negation of a quantified Boolean formula whose matrix is in 3CNF is equivalent to a formula whose matrix is in 3DNF, the result follows. ◀

▶ **Theorem 4.** Winner Determination for Player 1 *for* Greedy Spiders *is* PSPACE*-complete.*

Proof. Membership in PSPACE is shown in Lemma 2. We show PSPACE-hardness by means of a polynomial-time reduction from TQBF. Take an arbitrary instance $\varphi = \exists x_1.\forall x_2 \dots \exists x_{n-1}.\forall x_n.\psi$, where $\psi = d_1 \vee \dots \vee d_m$ is a quantifier-free 3DNF formula with n

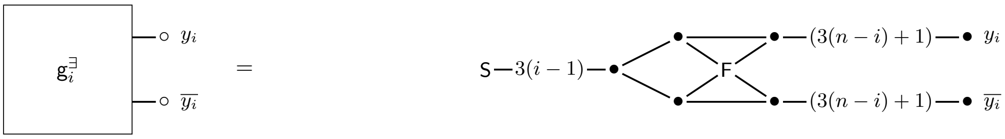


Figure 2 Gadget $g_i^\exists$ for variable x_i, for odd i.

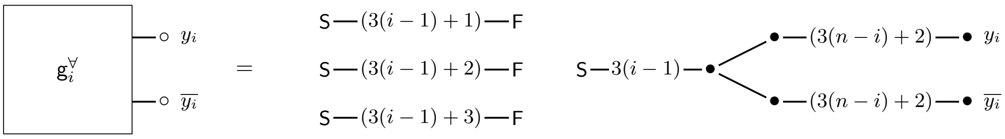


Figure 3 Gadget $g_i^\forall$ for variable x_i, for even i.

variables and m terms – without loss of generality we may assume that the odd-numbered variables x_i are existentially quantified, and that the even-numbered variables x_i are universally quantified. Moreover, by Lemma 3, we may assume that the incidence graph of φ is planar. Also, without loss of generality, we may assume that n is even and that $m \geq 2$.

We construct a game situation $C_1 = (G, S, F)$ as follows. We construct the planar graph $G = (V, E)$, together with the sets $S \subseteq V$ and $F \subseteq V$ by connecting various gadgets for the variables and terms of φ.

The idea of the reduction is as follows. We introduce gadgets $g_i^\exists$ that allow Player 1 to choose a truth assignment for variable x_i, for odd i. Similarly, for even i, we have gadgets $g_i^\forall$ that allow Player 2 to choose a truth assignment for variable x_i. These choices are made one after the other, so that they can depend on the truth assignment of preceding variables. The choices in these first gadgets consist of sending a spider on one of two paths. Then, we have gadgets k_i and k'_i, that serve to let the spiders from gadgets $g_i^\exists$ and $g_i^\forall$ pass onwards, while giving Player 1 time to cut free flies in all but one of the gadgets h_j representing the terms of ψ. If the chosen truth assignment satisfies a term d_j, Player 1 can safely leave the fly in gadget h_j unprotected (and cut free the flies in all other gadgets $h_{j'}$). In order to make this function properly, we additionally have gadgets f_ℓ, forcing Player 1 to cut free a fly in this gadget in one of her first ℓ turns. Figure 7 illustrates this for an example.

For each existentially quantified variable x_i – that is, for every odd $i \in [n]$ – we add the gadget $g_i^\exists$ as depicted in Figure 2. For each universally quantified variable x_i – that is, for every even $i \in [n]$ – we add the gadget $g_i^\forall$ as depicted in Figure 3. In these figures, nodes in S are marked with S and nodes in F are marked with F. Also, each edge that is marked with a number ℓ represents a path containing ℓ edges (where each of the non-depicted nodes are neither in S nor in F). In particular, if $\ell = 0$, the two nodes adjacent to this edge coincide.

Intuitively, the gadgets $g_i^\exists$ and $g_i^\forall$ simulate the quantification over the truth assignments to the variables $x_1, \dots, x_n$. For each $i \in [n]$, in Player 1's $(3(i-1)+1)$-th, $(3(i-1)+2)$-th and $(3(i-1)+3)$-th turn, she is forced to make a move in gadget $g_i^\exists$ or $g_i^\forall$ (depending on the parity of i), in order to prevent the spider in this gadget from capturing a fly in this gadget. Moreover, in gadgets $g_i^\exists$, her choices for these moves determine which of the two paths leading to the nodes labeled y_i and $\overline{y_i}$, respectively, are still available to the spider in this gadget. In the gadgets $g_i^\forall$, Player 2 is free to choose on which of the two paths, leading to the nodes labeled y_i and $\overline{y_i}$, respectively, the spider in this gadget moves. Moving a spider on the path towards y_i corresponds to setting variable x_i to true, and moving a spider on the path towards $\overline{y_i}$ corresponds to setting variable x_i to false. Thus, in this way, Player 1 can choose the truth values for the odd-numbered variables x_i and Player 2 can choose the truth values for the even-numbered variables x_i.

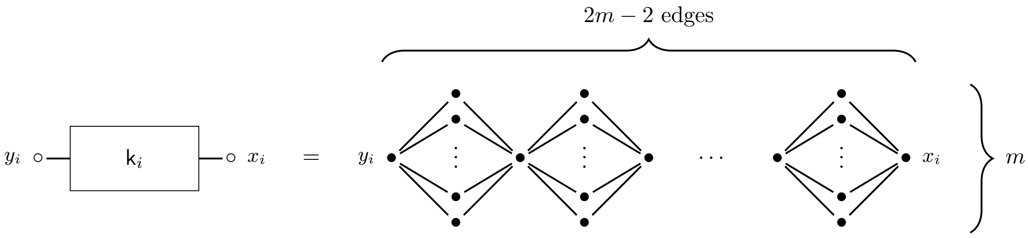


Figure 4 Secondary gadget k_i for literal x_i. The secondary gadget $\mathsf{k}_i^\neg$ for literal $\overline{x_i}$ is entirely similar, replacing y_i by $\overline{y_i}$ and x_i by $\overline{x_i}$.

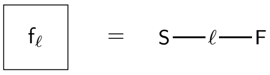


Figure 5 Gadget f_ℓ in which Player 1 is forced to remove an edge in her ℓ-th turn (at the latest).

Then, for each $i \in [n]$, we identify the node labeled with y_i in the gadget $\mathsf{g}_i^\exists$ or $\mathsf{g}_i^\forall$ with the node labeled with y_i in the gadget k_i that is depicted in Figure 4. We similarly identify the node labeled with $\overline{y_i}$ in the gadget $\mathsf{g}_i^\exists$ or $\mathsf{g}_i^\forall$ with the node labeled with $\overline{y_i}$ in the gadget $\mathsf{k}_i^\neg$, which is entirely similar to the gadget depicted in Figure 4 – the only difference is that the node label y_i is replaced by $\overline{y_i}$ and the node label x_i is replaced by $\overline{x_i}$. These gadgets consist of $m-1$ successive pieces, each consisting of m parallel paths of length 2 – here m is the number of terms occurring in the matrix ψ of the quantified Boolean formula φ. Intuitively, the purpose of these gadgets k_i and $\mathsf{k}_i^\neg$ is to ensure that there remains a path of length $2m-2$ from the node labeled with y_i to the node labeled with x_i, even after the next $2m-2$ moves (and similarly for the nodes labeled with $\overline{y_i}$ and $\overline{x_i}$).

For each even $\ell \in [3n+1, 3n+2m-2]$ (so not the odd values), we add the gadget f_ℓ, as depicted in Figure 5. These gadgets force Player 1 to make a move in gadget f_ℓ in her ℓ-th turn (at the latest). As a result, Player 1 has no way of preventing any spider to move from a node labeled with y_i to a node labeled with x_i in her $(3n+1)$-th until her $(3n+2m-2)$-th turn (while also preventing the flies in the gadgets f_ℓ from getting captured by a spider). However, notably, for each odd $\ell \in [3n+1, 3n+2m-2]$, Player 1 is not forced to delete any particular edge in the graph in her ℓ-th turn (in order to avoid losing directly after that turn). This free choice for Player 1 will play a role in the next type of gadget that we will add.

For each term d_j of ψ, we add the gadget h_j, as depicted in Figure 6. The leftmost nodes in this gadget are labeled with x_i or $\overline{x_i}$. We identify these leftmost nodes with the nodes in gadgets k_i and $\mathsf{k}_i^\neg$ that have identical labels. Suppose that $d_j = (l_{j,1} \wedge l_{j,2} \wedge l_{j,3})$, where each $l_{j,u}$, for $u \in [3]$, is either x_i or $\overline{x_i}$ for some $i \in [n]$. Then the leftmost nodes in the gadget h_j coincide with the nodes in gadgets k_i and $\mathsf{k}_i^\neg$ that are labeled with $\overline{l_{j,u}}$, denoting the complementary literal of $l_{j,u}$. For example, if $d_j = (x_1 \wedge \overline{x_2} \wedge x_3)$, then the leftmost nodes in the gadget h_j are identified with the nodes labeled with $\overline{x_1}$, x_2 and $\overline{x_3}$ in gadgets $\mathsf{k}_1^\neg$, k_2 and $\mathsf{k}_3^\neg$.

Intuitively, the gadgets h_j all contain a fly that needs to be protected from the incoming spiders on the paths from x_i and $\overline{x_i}$. Player 1 has time to remove the edges adjacent to the flies in exactly $m-1$ of these gadgets h_j – she has time to do this in her ℓ-th turns, for odd values of $\ell \in [3n+1, 3n+2m-2]$. In other words, Player 1 needs to choose exactly one $j \in [m]$ such that the fly in gadget h_j is out of reach of the spiders, for her next two turns.

Finally, we add the gadgets f_ℓ, as depicted in Figure 5, for both $\ell \in [3n+2m-1, 3n+2m]$ to ensure that after rescuing the flies in all but one of the gadgets h_j, Player 1 has to make a

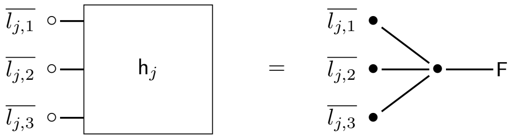


Figure 6 Gadget h_j for the term $d_j = (l_{j,1} \wedge l_{j,2} \wedge l_{j,3})$.

move in these gadgets in her next two turns. In other words, if the fly in the unique gadget h_j whose safety she did not ensure by deleting its adjacent edge is being approached by some spider within distance 2, this spider will then be able to capture the fly. If this is not the case, Player 1 can ensure the safety of this final fly in her $(3n + 2m + 1)$-th turn.

Clearly, this reduction runs in polynomial time. Moreover, since the incidence graph of φ is planar, the graph G that we constructed is also planar.

Verifying the correctness of this reduction is straightforward using the intuitions behind and explanations of the workings of the gadgets $\mathsf{g}_i^{\exists}$, $\mathsf{g}_i^{\forall}$, k_i, $\mathsf{k}_i^{\neg}$, f_ℓ, and h_j – that we gave above – together with the following observations.

The first observation is that for each odd $i \in [n]$, Player 1 can decide which of the two paths, towards the nodes labeled with y_i or $\overline{y_i}$, are left open for the spider in gadget $\mathsf{g}_i^{\exists}$, and she can base this choice on her choices in the gadgets $\mathsf{g}_{i'}^{\exists}$, for odd $i' \in [i]$ and Player 2's choices in the gadgets $\mathsf{g}_{i'}^{\forall}$, for even $i' \in [i]$. Similarly, for each even $i \in [n]$, Player 2 can decide which of the two paths, towards the nodes labeled with y_i or $\overline{y_i}$, are taken by the spider in gadget $\mathsf{g}_i^{\forall}$, and she can base this choice on her choices in gadgets $\mathsf{g}_{i'}^{\forall}$, for even $i' \in [i]$ and Player 1's choices in the gadgets $\mathsf{g}_{i'}^{\exists}$, for odd $i' \in [i]$.

The second observation is that whenever a truth assignment satisfies ψ, it must satisfy some term d_j of ψ. This means that it must satisfy all literals in d_j, and thus must make all their complements false. Therefore, if (and only if) the spiders are on their way towards the nodes labeled with x_i and $\overline{x_i}$ in such a way that the corresponding truth assignment satisfies ψ (and thus satisfies d_j for some $j \in [m]$), Player 1 can safely leave the fly in gadget h_j unprotected during her $(3n + 1)$-th until $(3n + 2m)$-th turn.

This concludes our proof of PSPACE-hardness. ◀

▶ **Example 5.** Consider the quantified Boolean formula $\varphi = \exists x_1.\forall x_2.\exists x_3.\forall x_4.[d_1 \vee d_2]$, where $d_1 = (x_1 \wedge x_2 \wedge x_3)$ and $d_2 = (x_1 \wedge \overline{x_2} \wedge x_3)$. The game situation $C_1 = (G, S, F)$ as constructed in the proof of Theorem 4 is depicted (schematically) in Figure 7. (Note that the last universally quantified variable (x_4) does not occur in the terms d_1 and d_2 – its presence makes n even.)

▶ **Corollary 6.** Winner Determination for Player 2 *for* Greedy Spiders *is* PSPACE-*complete.*

Proof. This follows directly from Theorem 4, since PSPACE is closed under complement and the problems Winner Determination for Player 1 and Winner Determination for Player 2 are complementary. ◀

6 Metatheorems

For our metatheorems, we consider games that are turn-based two-player games modeled on graphs. In the unrestricted version, the players alternate turns and every player has unlimited resources in every turn to calculate her next move. A player is called *strategically*

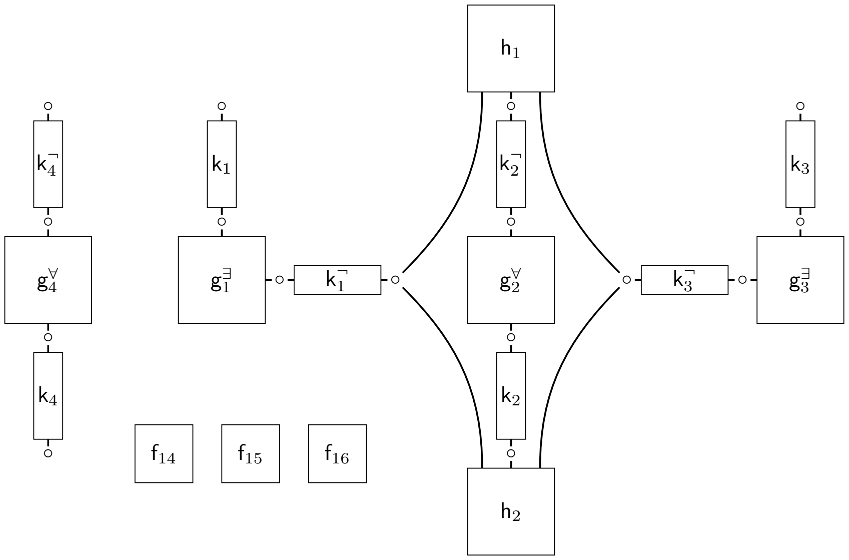


Figure 7 The game situation $C_1 = (G, S, F)$ that is constructed from the quantified Boolean formula $\varphi = \exists x_1.\forall x_2.\exists x_3.\forall x_4.[(x_1 \wedge x_2 \wedge x_3) \vee (x_1 \wedge \overline{x_2} \wedge x_3)]$ in the proof of Theorem 4 – see Example 5. (The nodes that are drawn in this picture between two gadgets are the nodes that belong to both gadgets, and are identified – e.g., the node drawn between the gadgets $g_1^{\exists}$ and $k_1^{\neg}$ is the node labeled with $\overline{y_1}$ that appears in both gadgets.)

restricted if she chooses in her first move a deterministic polynomial-time algorithm with a polynomial-size description, that determines all of her moves on that game instance. The algorithm is then disclosed to the opposing player. This means that the other player can then calculate the reaction of her opponent for any possible situation in polynomial time.

We consider the following game mechanics. A game is said to implement *defense positions*, if there exist positions which must not be reached by attackers of Player 2. Paths that can be eliminated or permanently blocked by Player 1 are called *destroyable paths*. Player 1 has the ability to destroy one destroyable path in each of her turns, while Player 2, in each of her turns, moves all of her attackers (towards the defense positions of Player 1) over one edge each. Player 1 wins the game if there is no path left from any attacker of Player 2 to any defense position of Player 1. Conversely, Player 2 wins if at least one of her attackers has reached a defense position of Player 1.

The decision problems Winner Determination for Player 1 and Winner Determination for Player 2 for games that implement defense positions and destroyable paths are defined analogously as for the game of *Greedy Spiders* (see Section 3).

▶ **Metatheorem 7.** *For a round-based two-player game implementing defense positions and destroyable paths, the problem* Winner Determination for Player 1 *is:*
(1) PSPACE*-hard if neither of the players is strategically restricted;*
(2) Σ_2^p*-hard, if Player 1 is strategically restricted; and*
(3) Π_2^p*-hard, if Player 2 is strategically restricted.*
These hardness results hold even when the game is restricted to planar graphs.

Proof (idea). Statement (1) follows as a corollary from the proof of our hardness result for Theorem 4. The reduction used in this proof is entirely based on the game mechanics of defense positions and destroyable paths. We will prove Statement (2) by modifying the hardness reduction from the proof of Theorem 4 to a reduction from the Σ_2^p-complete problem

$\mathrm{TQBF}_{\exists,2}$ – we will explain this in more detail below. Similarly, we will prove Statement (3) by modifying the same reduction to a reduction from an appropriate Π_2^{p}-complete variant of the problem $\mathrm{TQBF}_{\forall,3}$ – we also work this out in more detail below. All these reductions also work when restricted to planar graphs. ◀

Asking the converse question (i.e., whether Player 2 can win) leads to the following metatheorem.

▶ **Metatheorem 8.** *For a round-based two-player game implementing defense positions and destroyable paths, the problem* Winner Determination for Player 2 *is:*

(1) PSPACE*-hard if neither of the players is strategically restricted;*

(2) Π_2^{p}*-hard, if Player 1 is strategically restricted; and*

(3) Σ_2^{p}*-hard, if Player 2 is strategically restricted.*

These hardness results hold even when the game is restricted to planar graphs.

Proof. Because the problems Winner Determination for Player 1 and Winner Determination for Player 2 are complementary, these statements follow directly from Metatheorem 7. ◀

We now turn to proving Metatheorem 7(2–3).

Proof of Metatheorem 7(2). We describe how the hardness reduction from the proof of Theorem 4 can be used to form a reduction from $\mathrm{TQBF}_{\exists,2}$ to Winner Determination for Player 1 where Player 1 is strategically restricted. Let $\varphi = \exists x_1 \dots \exists x_{n_1} \forall x_{n_1+1} \dots \forall x_n.\psi$ be an instance of $\mathrm{TQBF}_{\exists,2}$. Without loss of generality, we may assume that ψ is in 3DNF and has a planar incidence graph.

We consider the formula $\varphi' = \exists x_1 \forall y_1 \dots \exists x_{n_1-1} \forall y_{n_1-1} \exists x_{n_1} \forall x_{n_1+1} \exists y_{n_1+1} \dots \forall x_{n-1} \exists y_{n-1} \forall x_n.\psi$, where the variables in $Y = \{y_1, \dots, y_{n_1-1}, y_{n_1+1}, \dots, y_{n-1}\}$ are fresh variables that do not occur in ψ. That is, φ' differs from φ only in that variables from Y are added to the quantifier prefix to ensure that existential and universal quantifiers alternate. We know that φ is true if and only if φ' is true. Then, because the quantifiers in φ' alternate between existential and universal quantifiers, we can employ the reduction from the proof of Theorem 4 to construct a game situation C_1 where Player 1 has a winning strategy if and only if φ' is true (which is the case if and only if φ is true).

All that remains to show that whenever Player 1 has a winning strategy for C_1, she can – in her first turn – submit an algorithm (whose description is of polynomial size) that computes the moves of her winning strategy in polynomial time. By construction of the game instance C_1, and because the variables $y_1, \dots, y_{n_1-1}$ do not occur in ψ, we know that any winning strategy for Player 1 does not depend on Player 2's moves in the gadgets $\mathsf{g}_i^{\forall}$ corresponding to the variables $y_1, \dots, y_{n_1-1}$. Moreover, since the variables $y_{n_1+1}, \dots, y_{n-1}$ do not occur in ψ, Player 1's optimal strategy in the gadgets $\mathsf{g}_i^{\exists}$ corresponding to the variables $y_{n_1+1}, \dots, y_{n-1}$ is easy to determine. Player 1's only moves that depend on the choice of Player 2 in the gadgets $\mathsf{g}_i^{\forall}$ are Player 1's moves in the gadgets h_j, and Player 1's optimal moves in these latter gadgets are easy to determine – these moves correspond to evaluating ψ once the truth value of each variable is set. Therefore, the optimal moves for carrying out her winning strategy can be generated by a polynomial-time algorithm that she can submit at the beginning of the game. Thus, this reduction works for the case where Player 1 is strategically restricted. ◀

In order to prove Metatheorem 7(3), we consider a Σ_2^{p}-complete variant of $\mathrm{TQBF}_{\exists,3}$.

▶ **Lemma 9.** *There is a class of quantified Boolean formulas of the form $\varphi = \exists x_1 \dots \exists x_{\ell_1} \forall y_1 \dots \forall y_{\ell_2} \exists z_1 \dots \exists z_{\ell_3}.\psi$ with the following properties:*

(1) $\mathrm{TQBF}_{\exists,3}$ *restricted to this class of quantified Boolean formulas is* Σ_2^{p}*-complete;*

(2) *each quantified Boolean formula φ in this class has a matrix in 3CNF and has a planar incidence graph; and*

(3) *for each quantified Boolean formula $\varphi = \exists x_1 \dots \exists x_{\ell_1} \forall y_1 \dots \forall y_{\ell_2} \exists z_1 \dots \exists z_{\ell_3}.\psi$ in this class, and for each truth assignment $\alpha : \{x_1, \dots, x_{\ell_1}, y_1, \dots, y_{\ell_2}\} \to \{0, 1\}$, it can be decided in polynomial time (given φ and α) if there exists a truth assignment $\beta : \{z_1, \dots, z_{\ell_3}\} \to \{0, 1\}$ such that $\psi[\alpha \cup \beta]$ evaluates to true, and such a truth assignment β can be computed in polynomial time, if it exists.*

Proof. We provide a reduction from $\mathrm{TQBF}_{\exists,2}$ to $\mathrm{TQBF}_{\exists,3}$ and show that the class of quantified Boolean formulas that are produced by this reduction has Properties (1)–(3). Hardness for Σ_2^{p} for the problem $\mathrm{TQBF}_{\exists,3}$ restricted to this class of quantified Boolean formulas follows immediately from this reduction.

Let φ be an instance of $\mathrm{TQBF}_{\exists,2}$. Without loss of generality, we may assume that φ has a matrix ψ in 3DNF. We then transform the matrix ψ to 3CNF using the standard Tseitin transformation [18], by adding additional existentially quantified variables at the end of the quantifier prefix – this will result in an equivalent quantified Boolean formula φ' with an "$\exists\forall\exists$" quantifier prefix. We then transform φ' into an equivalent quantified Boolean formula φ'' with a matrix in 3CNF and a planar incidence graph using the gadgets used in the proof that 3SAT restricted to planar formulas is NP-hard [14, Theorem 1] – this will add additional existentially quantified variables at the end of the quantifier prefix.

The reduction clearly results in quantified Boolean formulas that satisfy Property (2). The resulting formulas also satisfy Property (3). Once the variables from the original quantified Boolean formula φ have been instantiated, only clauses corresponding to the introduced gadgets in the two-step reduction described above (containing only existentially quantified variables) remain – finding satisfying truth assignments for these remaining clauses can be done in polynomial time. This is because both steps in the reduction have the property that given any satisfying truth assignment α for the matrix of the original formula, one can compute in polynomial time a truth assignment β such that $\alpha \cup \beta$ satisfies the matrix of the constructed formula – and that both steps of the reduction are reversible in polynomial time. For the first step of the reduction (where the matrix ψ is transformed to 3CNF) this is the case because the introduced clauses form a renamable Horn formula – thus after instantiating the formula with α, a renamable Horn formula remains, and a satisfying truth assignment for renamable Horn formulas can be found in polynomial time. For the second step of the reduction (where the formula is transformed to an equivalent formula that has a planar incidence graph) this property follows directly from the shape of the gadgets used in the reduction [14, Theorem 1].

As a result of Property (3), we get membership in Σ_2^{p} for the problem $\mathrm{TQBF}_{\exists,3}$ restricted to quantified Boolean formulas produced by the reduction above. Together with Σ_2^{p}-hardness, this gives us Property (1). ◀

The main idea behind the proof of Σ_2^{p}-hardness is to apply Tseitin transformations [18] to inputs of the problem $\mathrm{TQBF}_{\exists,2}$. We denote the problem $\mathrm{TQBF}_{\exists,3}$ restricted to the class of quantified Boolean formulas identified in Lemma 9 by $\mathrm{TQBF}^{\star}_{\exists,3}$. Similarly, we consider the Π_2^{p}-complete dual problem $\mathrm{TQBF}^{\star}_{\forall,3}$, that concerns formulas that are equivalent to the negation of instances of $\mathrm{TQBF}^{\star}_{\exists,3}$.

Proof (sketch) of Metatheorem 7(3). We modify the proof of Theorem 4 to a reduction from the problem $\mathrm{TQBF}^{\star}_{\forall,3}$. These modifications are entirely analogous to the modifications in the proof of Metatheorem 7(2). That is, we introduce new variables (not occurring in the matrix of the quantified Boolean formula) to ensure that existential and universal quantifiers alternate strictly.

In the resulting game, whenever Player 2 has a winning strategy that corresponds to a way of assigning the universally quantified variables that makes the remaining formula false (for any assignment to the existentially quantified variables), the optimal moves for carrying out this strategy can be generated by a polynomial-time algorithm that she can submit at the beginning of the game. This is because her only moves that (non-trivially) depend on the choice of Player 1 in the gadgets $\mathsf{g}_i^{\exists}$ are her moves in the gadgets $\mathsf{g}_i^{\forall}$ corresponding to the variables in the third quantified block and her moves in the gadgets h_j, and Player 2's optimal moves in these latter gadgets are easy to determine – this is due to Lemma 9(3). Thus, this reduction works for the case where Player 2 is strategically restricted. ◀

7 Application of Metatheorems

In this section we describe how to apply our metatheorems to tower-defense games. Games of this genre can be described as two-player games where the defending Player 1 must prevent the attackers of Player 2 from reaching designated locations on the playing field. For this purpose Player 1 can place towers on the field which damage every attacker in their reach. To place the towers, Player 1 usually has to pay some amount of a currency which is steadily credited to Player 1 over time. In most tower-defense games Player 1 is played by the user, while Player 2 is played by the computer. The strategy of Player 2 is fixed per instance, but differs from instance to instance, so we will apply Metatheorem 7(3).

To apply Metatheorem 7(3), we have to show that all elements of the metatheorem can be modeled within the game. Defense positions are naturally a part of tower-defense games, since they all include positions which have to be protected from the attacking enemies. Destroyable paths are implemented in the following way. A path is said to be destroyed if no attacker can cross it (and survive). Therefore we can destroy a path by placing a strong enough tower somewhere on the path to kill every attacker in its reach. The accessible environment of this tower is regarded as the destroyed path. Every spot on the map where a tower can be placed represents therefore a destroyable path. While most tower defense-games are not round-based in a strong sense, we can still model them as round-based. To implement the game elements, we only have to consider one type of attackers and one type of towers. Since Player 1 earns coins of a currency every fixed amount of time we can graduate the time in steps which are as long as it takes Player 1 to earn enough coins to buy one tower instance. The step range of the attackers of Player 2 is therefore as long as the distance they can walk in one time step. Thus we can assume the game to be round-based. Since all criteria of Metatheorem 7(3) can be implemented, this shows that tower-defense games in general are Π_2^{p}-hard.

In concrete terms, the above described implementation works among others for games like *Bloons Tower Defense 5* [2], *Warcraft 3* [1], and *Starcraft* [5].

8 Conclusion

We showed PSPACE-completeness for the problem of deciding whether Player 1 has a winning strategy for the game *Greedy Spiders*, as well as for the problem of deciding whether

Player 2 has a winning strategy. Afterwards we generalized the idea of our proof to give two metatheorems referring to [11, 12, 19], which granulate the computational complexity of the core element of the game by restricting the computational power of the players. In particular, we showed that Winner Determination for Player 1 in a turn-based two-player game containing defense positions and destroyable paths is in general PSPACE-hard, becomes Σ_2^p-hard if Player 1 is strategically restricted, and Π_2^p-hard if Player 2 is strategically restricted. The reverse question of Winner Determination for Player 2 is in general PSPACE-hard, becomes Π_2^p-hard if Player 1 is strategically restricted, and Σ_2^p-hard if Player 2 is strategically restricted. Finally, we discussed the applicability of our metatheorems on tower-defense games and mentioned some specific games to which our metatheorems can be applied.

Finding metatheorems for the computational complexity of computer games has recently become more and more of a focus. With tower-defense games, we grazed with our metatheorems a previously untouched game genre in terms of computational complexity and provided new tools to investigate them. As most metatheorems are discovered in the area of platform- and puzzle-games, they can only be applied to single-player games. Therefore with our metatheorems, we give new impulses in looking for metatheorems, which describe multiplayer (specifically two-player) games. To our knowledge, our results are the first hardness results for the complexity classes Σ_2^p and Π_2^p in the field of computational complexity of computer games.

A possibility for further research in this field is to look at two-player games and restrict the computational power of one of the players. This approach could also be applied to well studied board games like *Chess*, *Checkers*, or *Mill*. In general the field of multiplayer strategy games seems to afford more yet undiscovered metatheorems and should be investigated in the future. Beside tower-defense games, our metatheorem should also be applicable to other strategic games, such as war simulations or any game in which one player has the role of a defender who has to prevent the other player (with the role of an attacker) from reaching certain locations in the game. Over the last few years, more and more complex and modern games have been explored, resulting in metatheorems which are applicable to state of the art games. Since many modern computer games provide scripting languages with whom the players can modify the game, the games themselves are instantly Turing-complete. We think that examining restricted versions of these games is still worth a try and can lead to metatheorems for the essential elements of the games, taking off the focus from the scripting languages.

References

1 Blizzard Entertainment: Warcraft III. `http://eu.blizzard.com/en-gb/games/war3/`. Accessed: 2018-02-17.

2 Bloons Tower Defense 5. `http://bloons.wikia.com/wiki/Bloons_Tower_Defense_5`. Accessed: 2018-02-17.

3 Greedy Spiders. `http://greedyspiders.com/`. Accessed: 2018-02-17.

4 Nintendo Entertainment System (NES). `http://www.pcgames.de/Nintendo-Entertainment-System-NES-Konsolen-255246/`. Accessed: 2018-02-01.

5 StarCraft: Remastered. `https://starcraft.com/en-us/`. Accessed: 2018-02-17.

6 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo Games are (Computationally) Hard. *Theoretical Computer Science*, 586:135–160, 2015.

7 Sanjeev Arora and Boaz Barak. *Computational Complexity – A Modern Approach*. Cambridge University Press, 2009.

8 Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. Alternation. *J. of the ACM*, 28(1):114–133, 1981.

9 Erik D. Demaine. Playing Games with Algorithms: Algorithmic Combinatorial Game Theory. In *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 18–33. Springer, 2001.

10 Erik D. Demaine and Robert A. Hearn. Constraint Logic: A Uniform Framework for Modeling Computation as Games. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity, 2008 (CCC 2008)*, pages 149–162. IEEE, 2008.

11 Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The Computational Complexity of Portal and Other 3D Video Games. *arXiv preprint 1611.10319*, 2016.

12 Michal Forišek. Computational Complexity of Two-Dimensional Platform Games. In *Proceedings of the 5th International Conference on Fun with Algorithms (FUN 2010)*, pages 214–227. Springer, 2010.

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

14 David Lichtenstein. Planar Formulae and Their Uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

15 John Michael Robson. The Complexity of Go. In *IFIP Congress*, pages 413–417, 1983.

16 John Michael Robson. N by N Checkers is Exptime complete. *SIAM J. Comput.*, 13(2):252–267, 1984.

17 Jörg Siekmann and Graham Wrightson, editors. *Automation of reasoning. Classical Papers on Computer Science 1967–1970*, volume 2. 1983.

18 G. S. Tseitin. Complexity of a Derivation in the Propositional Calculus. *Zap. Nauchn. Sem. Leningrad Otd. Mat. Inst. Akad. Nauk SSSR*, 8:23–41, 1968. English transl. repr. in [17].

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

Computational Complexity of Motion Planning of a Robot through Simple Gadgets

Erik D. Demaine
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
edemaine@mit.edu

Isaac Grosof[1]
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
isaacg@alum.mit.edu

Jayson Lynch
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
jaysonl@mit.edu

Mikhail Rudoy[2]
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
mrudoy@gmail.com

Abstract

We initiate a general theory for analyzing the complexity of motion planning of a single robot through a graph of "gadgets", each with their own state, set of locations, and allowed traversals between locations that can depend on and change the state. This type of setup is common to many robot motion planning hardness proofs. We characterize the complexity for a natural simple case: each gadget connects up to four locations in a perfect matching (but each direction can be traversable or not in the current state), has one or two states, every gadget traversal is immediately undoable, and that gadget locations are connected by an always-traversable forest, possibly restricted to avoid crossings in the plane. Specifically, we show that any single nontrivial four-location two-state gadget type is enough for motion planning to become PSPACE-complete, while any set of simpler gadgets (effectively two-location or one-state) has a polynomial-time motion planning algorithm. As a sample application, our results show that motion planning games with "spinners" are PSPACE-complete, establishing a new hard aspect of *Zelda: Oracle of Seasons*.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases PSPACE, hardness, motion planning, puzzles

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.18

Acknowledgements This work grew out of an open problem session from the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.890) from Fall 2014. We particularly thank Jeffrey Bosboom for posing the problem of analyzing 4-spinners from Legend of Zelda: Oracle of Seasons (in 2015), for simplifying the 2-state k-tunnels proof, and for other helpful discussions.

[1] Now at Carnegie Mellon University.
[2] Now at Google Inc.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 18; pp. 18:1–18:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Many hardness proofs are based on *gadgets* — local pieces, each often representing corresponding pieces of the input instance, that combine to form the overall reduction. Garey and Johnson [7] called gadgets “basic units” and the overall technique “local replacement proofs”. The search for a hardness reduction usually starts by experimenting with small candidate gadgets, seeing how they behave, and repeating until amassing a sufficient collection of gadgets to prove hardness.

This approach leads to a natural question: what gadget sets suffice to prove hardness? There are many possible answers to this question, depending on the precise meaning of “gadget” and the style of problem considered. Schaefer [11] characterized the complexity of all “Boolean constraint satisfiability” gadgets, including easy problems (2SAT, Horn SAT, dual-Horn SAT, XOR SAT) and hard problems (3SAT, 1-in-3SAT, NAE 3SAT). Constraint Logic [8] proves sufficiency of small sets of gadgets on directed graphs that always satisfy one local rule (weighted in-degree at least 2), in many game types (1-player, 2-player, 2-team, polynomially bounded, unbounded), although the exact minimal sets of required gadgets remain unknown. Both of these general techniques naturally model “global” moves that can be made anywhere at any time (while satisfying the constraints). Nonetheless, the techniques have been successful at proving hardness for problems where moves must be made local to an agent/robot that traverses the instance.

In this paper, we introduce a general model of gadgets that naturally arises from *single-agent motion planning problems*, where a single agent/robot traverses a given environment from a given start location to a given goal location. Our model is motivated by the plethora of existing hardness proofs for such problems, such as Push-1, Push-$*$, PushPush, and Push-X [3]; Push-2-F [5]; Push-1 Pull-1 [4,9]; as well as several Nintendo video games studied at recent FUN conferences [1,6].

1.1 Gadget model

In general, we model a *gadget* as consisting of one or more *locations* (entrances/exits) and one or more *states*. (In this paper, we will focus on gadgets with at most two states.) Each state s of the gadget defines a labeled directed graph on the locations, where a directed edge (a, b) with label s' means that the robot can enter the gadget at location a and exit at location b, and that such a traversal forcibly changes the state of the gadget to s'. Equivalently, a gadget is specified by its *state space*, a directed graph whose vertices are state/location pairs, where a directed edge from (s, a) to (s', b) represents that the robot can traverse the gadget from a to b if it is in state s, and that such traversal will change the gadget’s state to s'. Gadgets are *local* in the sense that traversing a gadget does not change the state of any other gadgets.

A *system of gadgets* consists of gadgets, their initial states, and *connections* between disjoint pairs of locations (forming a matching). If two locations a, b of two gadgets (or the same gadget) are connected, then the robot can traverse freely between a and b (outside the gadgets). (Equivalently, we can think of locations a and b as being identified.) These are all the ways that the robot can move: exterior to gadgets using connections, and traversing gadgets according to their current states. In a *puzzle*, we are given a system of gadgets, the robot starts at a specified start location, and we want to find a sequence of moves that brings the robot to a specified goal location. The main problem we consider here is the obvious decision problem: is the given puzzle solvable?

One type of gadget we always allow in this paper is the **branching hallway** gadget, which has one state and three locations, and always allows traversal between all pairs of

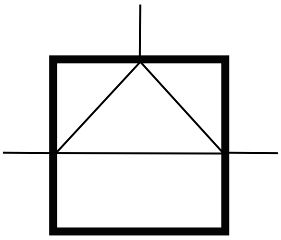


Figure 1 Branching hallway gadget.

locations; see Figure 1. In other words, upon reaching such a gadget, the robot is free to choose and move to any of the three locations. Connecting together multiple branching hallways allows us to effectively connect the other gadgets' locations according to an arbitrary forest (as described in the abstract).

All other gadgets we consider in this paper are "deterministic" and "reversible". A gadget is *deterministic* if its state space has maximum out-degree ≤ 1, i.e., a robot entering the gadget at some location a in some state s (if possible) can exit at only one location b and one new state s'. A gadget is *reversible* if its state space has the reverse of every edge, i.e., it is the bidirectional version of an undirected graph. Thus a robot can immediately undo any gadget traversal.[3] Together, determinism and reversibility are equivalent to requiring that the state space is the bidirectional version of a matching.

Other than the (one-state) branching hallway, we further require that the states of a gadget differ only in their orientations of the possible traversals. More precisely, a *k-tunnel* gadget has $2k$ locations, paired in a perfect matching whose pairs are called *tunnels*, such that each state defines which direction or directions each tunnel can be traversed.

We also consider *planar* systems of gadgets, where the gadgets and connections are drawn in the plane without crossings. Planar gadgets are drawn as small regions (say, disks) with their locations as points in a fixed clockwise order along their boundary. A single gadget type thus corresponds to multiple planar gadget types, depending on the choice of the clockwise order of locations. Connections are drawn as paths connecting the points corresponding to the endpoint locations, without crossing gadget interiors or other connections.

1.2 Our results

We characterize the computational complexity of deciding puzzle solvability when the allowed gadgets consist of the branching hallway and any number of deterministic reversible ≤ 2-state k-tunnel gadgets, for any k. Specifically, if there is at least one gadget type that is not equivalent to a 1-state or 1-tunnel gadget, then the problem is PSPACE-complete; and otherwise, the problem is in P. The same characterization holds for planar systems of gadgets; thus, in applications, we do not have to worry about building a crossover gadget (which is often the most difficult).

In Section 3, we sketch our proof from [4] that motion planning with two-toggle-locks and crossovers is PSPACE-complete. In Section 4, we prove that one particular gadget, the antiparallel two-toggle, can simulate a variety of other gadgets, eventually including a

[3] This notion is different than the sense of "reversible" in reversible computing, which would mean that we could derive which move to undo from the current state.

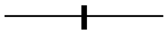
two-toggle-lock and a crossover. As a consequence, motion planning with the antiparallel two-toggle is PSPACE-complete. In Section 5, we show that all nontrivial deterministic reversible 2-state, 2-tunnel gadgets can simulate the antiparallel two-toggle. As a consequence, each corresponding motion planning problem is PSPACE-complete. In Section 7, we extend these results to give a precise hardness characterization for the motion planning problem with each deterministic reversible 2-state k-tunnel gadget.

We also partially characterize the computational complexity of deterministic reversible ≤ 2-state gadgets with three locations. In particular, we study spinners and deterministic forks, as described in Section 6.

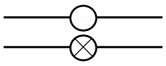
We hope that our approach will be useful for establishing hardness of many real-world motion planning problems and puzzles. As a sample application, our results allow us to establish a new PSPACE-hard aspect of the Nintendo video game *Zelda: Oracle of Seasons* (which features spinners) Section 6.

2 Gadget Basics

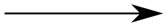
To categorize the possible deterministic reversible 2-state 2-tunnel gadget types, we first categorize the possible tunnel types in such a gadget. A tunnel is *trivial* if it is either never traversable or always traversable. A trivial tunnel can always be split into a separate 1-state 1-tunnel gadget, so we can ignore them. What remain are three possible *nontrivial* tunnel types:



Tripwire A tunnel that can always be traversed in either direction, but traversing it switches the gadget's state.



Lock In the *unlocked* state (shown above), the tunnel can be traversed in either direction; in the *locked* state (shown below), the tunnel cannot be traversed in either direction.



Toggle A tunnel that can always be traversed in a single direction, where the direction differs in the two states of the gadget. The state is switched when the gadget is traversed.

There are six ways to combine these tunnel types into pairs. Two combinations, Lock–Lock and Tripwire–Tripwire, are trivial combinations equivalent to one-state gadgets in which each tunnel is either always traversable in both directions or never traversable. Thus we restrict our attention to the four other combinations, listed below. Because we are interested in planar systems, we consider the multiple planar gadgets for each nontrivial combination. (We do, however, treat a gadget and its reflection as equivalent.) As a result, there are nine different nontrivial two-tunnel two-state gadgets, abbreviated and listed below. The bulk of our paper focuses on the six gadgets shown in Figure 2, which omits most crossing variants.

1. **Tripwire–Lock**: Traversing the tripwire makes the other tunnel flip between being passable and impassable, causing it to 'lock' or 'unlock'. There are crossing and non-crossing varieties, abbreviated **CWL** (crossing wire lock) and **NWL** (non-crossing wire lock).
2. **Toggle–Lock**: Traversing the toggle flips the lock tunnel between being passable and impassable. Crossing the lock tunnel, by definition, does not change the state of the gadget. Notice that one direction of the toggle corresponds to an open lock and the other direction to the closed lock. There are crossing and non-crossing varieties, abbreviated **CTL** (crossing toggle lock) and **NTL** (non-crossing toggle lock).

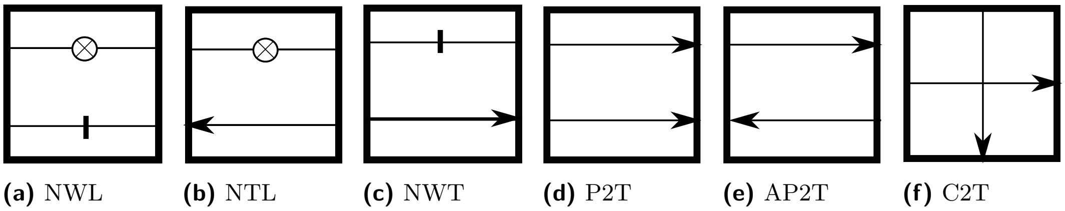


Figure 2 Six of the nine deterministic reversible 2-state gadgets on two tunnels. We leave out the CWL, CTL, and CWT gadgets as they are not heavily used in the paper.

3. **Tripwire–Toggle**: Here traversing either the tripwire or the toggle flips the direction of the toggle. There are crossing and non-crossing varieties, abbreviated **CWT** (crossing wire toggle) and **NWT** (non-crossing wire toggle).
4. **Toggle–Toggle**: Also known as a **2-toggle** [4]. Traversing either toggle flips the direction of both of them. This is the only case where there are two directed tunnels, leading to three possibilities: crossing, parallel, and anti-parallel. They are abbreviated **C2T** (crossing 2-toggle), **P2T** (parallel 2-toggle), and **AP2T** (anti-parallel 2-toggle).

In this paper we will often need to discuss putting gadgets together to create new behavior. We will do so by creating a system of gadgets that is "equivalent" to some target gadget, thereby "simulating" that gadget. Two systems of gadgets are *equivalent* if there is a bijective correspondence between their locations and a correspondence between their states such that the allowed transitions for all (locations, state) pairs are the same under these two correspondences. We will say that a gadget or set of gadgets *simulates* a target gadget if it is possible to combine gadgets from the set (possibly using duplicates) such that the resulting system is equivalent to the target gadget. We will always implicitly allow the use of the branching hallway gadget in these constructions. In all cases, these constructions will be planar.

2.1 Closure Properties

▶ **Lemma 2.1.** *Any system of gadgets composed of two reversible gadgets is reversible.*

Proof. Consider any transition through the system formed by composing two reversible gadgets. This transitions is a walk through the gadgets and connections that form a system. Since both gadgets are reversible, it is possible for the robot to enact the exact reverse of this walk after the walk is done. This will exactly reverse the effect of the walk within each gadget. Thus, it is possible to reverse the entire transition.

Since every transition of the system can be reversed, the system is reversible. ◀

Since all of the gadgets we consider in this paper are reversible, Lemma 2.1 means our systems will all be reversible as well.

▶ **Lemma 2.2.** *Any system of gadgets composed of two deterministic reversible gadgets is deterministic and reversible.*

Proof. The state space of a reversible, deterministic gadget is an undirected matching of some (state, location) pairs to each other. This a necessary and sufficient characterization of reversible, deterministic gadgets.

When we compose two such gadgets, we create paths through the pair of gadgets. However, no (state, location) pair has more than two edges: One connection to the other gadget, and one edge through its original gadget. Moreover, any (state, location) pair that forms an external location has a most one edge, as it does not connect to the other gadget. As a consequence, the path from any external location through the gadget is either a deterministic path to another external location, or a dead end. There is no branching, as branching would require a location with three edges.

Thus, the resultant object is deterministic. By Lemma 2.1 it is reversible as well. ◀

2.2 PSPACE Membership

▶ **Lemma 2.3.** *Deciding puzzle solvability is in PSPACE.*

Proof. The entire state of the system can be described by the current state of the gadgets and the location of the agent. The gadgets have a polynomial number of states and there can only be a polynomial number of gadgets. Since the entire state of the board fits in a polynomial amount of space, we can non-deterministically search for a solution, showing containment in NPSPACE. Savich's Theorem [10] gives PSPACE = NPSPACE. ◀

3 2-toggle-lock and crossover motion planning is PSPACE-complete

In [4] we showed that motion planning with 4-toggles and crossovers is PSPACE-complete. In that construction, the crucial gadget turned out to be a 2-toggle-lock, which is a 3-tunnel, 2-state gadget with two locks and a tunnel. The 4-toggle was not used in any way after the construction of the 2-toggle-lock, showing that 2-toggle-locks and crossovers are PSPACE-hard. For convenience we sketch the proof, with some refinement. One should refer to the prior paper for a more detailed and rigorous proof.

▶ **Definition 3.1.** 3QSAT is the following decision problem. Given a fully quantified boolean formula in prenex normal form and in conjunctive normal form with no more than three variables per clause, decide whether the formula is true.

▶ **Theorem 3.2.** *Motion planning with 2-toggle-locks and crossovers is PSPACE-hard.*

We reduce from 3QSAT to motion-planning with 2-toggle-locks and crossovers. To do so we need to construct clauses, universal variables, and existential variables. Literals will consist of a 2-toggle-lock which will be set from the 2-toggle side and checked by passing through the lock. Clauses are composed of a branching hallway that leads through each of its associated literals.

Existential variables will be a branching hall with a group of toggle-locks in series. Passing through in one direction opens the locks of the gadgets representing true literals of that variable while closing the locks of the false ones. Going through the other way allows this to be undone, as the system is reversible.

To construct universal quantifiers we connect up the 2-toggle sections as in Figure 3, where each universal gadget consists of several antiparallel 2-toggles with locks. Each of these gadgets sends the robot forward in one state or back to the beginning in the other state, and flips the state. Repeatedly entering from the left iterates through all configurations of the states, so the robot must check all of the possible values for the universal variables. The goal state lies at the far end of the eries of universal gadgets.

For both the existentials and the universals, the variables are actually a long series of 2-toggle-locks with one lock for each literal of the variable in the formula.

When putting this all together, as in Figure 3, we need to ensure that the robot cannot sneak back into the variable gadget and change existential settings it shouldn't be allowed

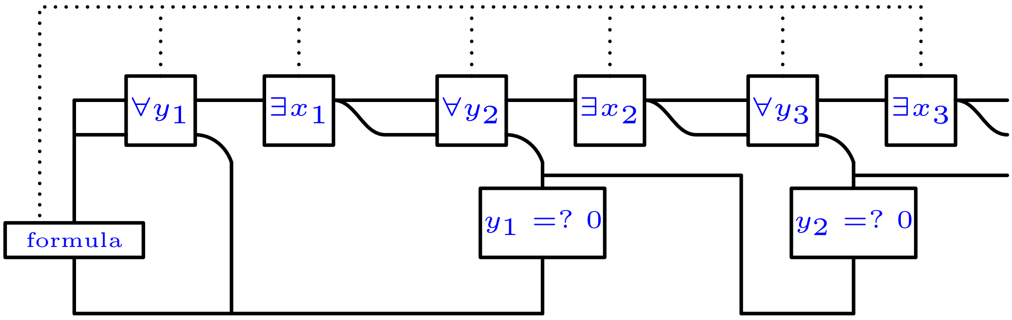


Figure 3 Structure of the QSAT reduction.

to access, namely those existentials beyond the universal it just emerged from. To do this we construct a simple system that puts a lock on the return pathway at the end of each universal variable which only allows passage if the prior variable is set to false. Since the robot will have just exited from a variable which was set to true, this prevents the robot from moving forward in the variable chain. In addition, all earlier variables are false allowing the robot to travel back to the formula, since the universal gadgets take on incrementing binary values with each loop through the gadget. Since those existential variables are ones the robot was allowed to set to any value on the prior passage, going back and changing them now gives no advantage over having set them to that value earlier.

This safeguard is the one difference from the prior construction, which checked the values of all prior universal variables, requiring a quadratic blow-up in number of gadgets. The need for crossovers and a 2D layout will still create a quadratic blowup in problem size overall, but this simplification seemed worth noting and should allow for the 3D result to cause only a linear blowup in problem size.

With this guard in place, the robot can only reach the goal state by demonstrating a solution to the 3QSAT instance, after iterating through all settings of the universal gadget. ◀

4 Antiparallel 2-toggle motion planning is PSPACE-complete

We will show that the question of whether a robot in a system of antiparallel 2-toggle gadgets can reach a specified goal location is PSPACE-complete. To do so, we will simulate various other gadgets using AP2T gadgets, eventually simulating 2-toggle-locks and crossovers. Since motion planning with 2-toggle-locks and crossovers is PSPACE-complete, this implies that AP2T motion planning is PSPACE-complete.

▶ **Theorem 4.1.** *Motion Planning with AP2T gadgets is PSPACE-complete.*

We will simulate the gadgets needed for the PSPACE-completeness proof, and a wide variety of other intermediate gadgets to help us get there. The steps are as follows:

1. Simulate a C2T, using AP2Ts. Lemma 4.2.
2. Simulate a P2T, using C2Ts. Lemma 4.3.
3. Simulate a NTL, using AP2Ts, C2Ts and P2Ts. Lemma 4.4.
4. Simulate various types of 2-toggle locks, with “round” and “stacked” internal connections. The types of internal connections are described in Section 4.1, and the constructions are given in Lemmas 4.6 and 4.7.
5. Simulate a NWL, using the stacked antiparallel 2-toggle lock. Lemma 4.8.
6. Simulate a stacked tripwire-lock-tripwire, using NWLs. Lemma 4.9
7. Simulate a crossover, using stacked tripwire-lock-tripwires. Lemma 4.10

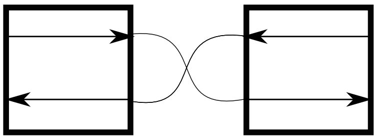


Figure 4 Anti-parallel 2-toggles simulate a crossing 2-toggle.

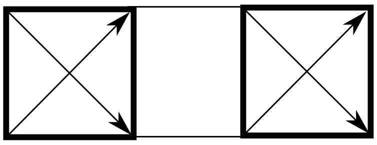


Figure 5 Crossing 2-toggles simulate a parallel 2-toggle.

With a 2-toggle lock and a crossover constructed, we can apply Theorem 3.2 to show that motion planning with AP2Ts is PSPACE-hard. Adding in Lemma 2.3, we find that it is PSPACE-complete.

▶ **Lemma 4.2.** *Antiparallel 2-toggles (AP2Ts) simulate a crossing 2-toggle (C2T).*

Proof. The construction is given in Figure 4. In the state of the construction shown in the figure, there are two possible transitions: the robot can move from the upper left to the bottom right of the construction, or from the upper right to the bottom left. Either of those transitions toggles both AP2Ts, leaving the construction mirrored top to bottom. Thus, the construction has two states. The possible traversals in one state (as shown above) are from the top left to the bottom right and from the top right to the bottom left, while the possible traversals in the other state are (by symmetry) from the bottom left to the top right and from the bottom right to the top left. Following any of these traversals swaps the state of the construction. Notice that this is exactly the behavior of a C2T.

If the robot enters the construction shown from the upper left, upon reaching the center the robot can only proceed to the bottom right, or come back the way it came. Therefore, the upper left to bottom right transition is the only possible transition from that location. By symmetry, the same is true from top left to bottom right. Thus, the one traversal described for each location in each state is the only one possible. ◀

▶ **Lemma 4.3.** *Crossing 2-toggles (C2Ts) simulate a parallel 2-toggle (P2T).*

Proof. The construction is given in Figure 5. In the state of the construction shown in the figure, there are two possible transitions: the robot can move from the top left to the top right of the construction, or from the bottom left to the bottom right. Either of these transitions toggles both C2Ts, leaving the construction mirrored left to right. The allowed traversals in one state (as shown above) are from the top left to the top right and from the bottom left to the bottom right, while the allowed traversals in the other state are (by symmetry) from the top right to the top left and from the bottom right to the bottom left. Following any of these traversals swaps the state of the construction. Notice that this is exactly the behavior of a P2T.

Since the system is composed entirely of C2Ts (without even branching hallways), which are both reversible and deterministic, the result is also both reversible and deterministic, by Lemma 2.2. Thus, the one transition described for each location in each state is the only transition possible. ◀

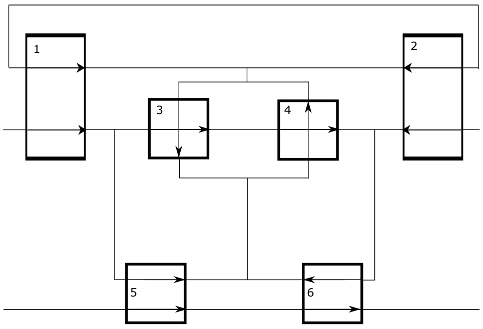


Figure 6 2-toggles simulate 1-toggle-lock.

▶ **Lemma 4.4.** *2-toggles (AP2Ts, P2Ts and C2Ts) simulate a noncrossing toggle lock (NTL).*

Proof. The construction is shown in Figure 6.

In this lemma, we will refer to toggles 1 and 2 in the figure as the "outer toggles", toggles 3 and 4 as the "middle toggles", and toggles 5 and 6 as the "bottom toggles". We will call the pathway through the lower tunnels of the bottom toggles the "bottom tunnel" of the overall gadget, and the rest of the gadget the "middle tunnel" of the overall gadget.

An NTL has two externally observable states: locked, and unlocked. The locked state corresponds to the upper tunnels of the bottom toggles oriented out, and the unlocked state corresponds to the bottom toggles oriented in. The unlocked state is shown in Figure 6.

In this gadget, there are two internal states corresponding to each external state: with the horizontal tunnels of the middle toggles both oriented left, and with both oriented right. The only accessible states of this gadget are the states with the outer toggles oriented in, the middle toggles oriented both left or both right, and upper pathways of the bottom toggles oriented both in or both out. We will show that the gadget allows exactly the traversals of the NTL from these configurations, and cannot be left in any other configuration.

The bottom tunnel traversals are straightforward — the bottom tunnel acts as a toggle, and a traversal flips both bottom toggles, and hence the externally observable state.

Also clearly, the robot cannot move between the bottom tunnel and the middle tunnel.

Now, we wish to establish that in the unlocked state, the robot can always traverse the middle tunnel in either direction. In the state shown, the middle tunnel may be traversed from external location to external location as follows:

- The robot can get across, left to right, by traversing the following toggles in the following order: enter through toggle 1's lower tunnel, down to toggle 5, up to toggle 4's vertical tunnel, through toggle 1's upper tunnel, around the top to toggle 2's top tunnel, back down through toggle 4, back out through toggle 5, across through toggle 3's horizontal tunnel, then through toggle 4's horizontal tunnel, then out through toggle 2's lower tunnel.
- The robot can get across, right to left, by traversing the following toggles in the following order: enter through toggle 2's lower tunnel, down to toggle 6, up to toggle 4's vertical tunnel, through toggle 2's top tunnel, around to toggle 1's top tunnel, down through toggle 3's vertical tunnel, back out through toggle 6, across through toggle 4's horizontal tunnel, then through toggle 3's horizontal tunnel, then out through toggle 1's lower tunnel.
- If the middle toggles are in the opposite orientation, the system is simply mirrored, left to right, and the traversals are still possible.

Next, we wish to establish that the robot cannot cross the middle tunnel in the locked state. After entering from either middle tunnel location, the only traversable toggles are the middle toggles. After traversing those, the robot can go no further. The bottom toggles can't be traversed, so the entire middle region is inaccessible. As a consequence, the opposite outer toggle's upper pathway can't be accessed. Therefore the robot can only leave via its original location.

We also must establish that if the gadget starts in one of the configurations mentioned, the robot must leave it in the proper state, and can't leave it in a configuration that wasn't mentioned. This is straightforward for the bottom tunnel, so we will focus on the middle two locations.

We will show that the accessible configurations of the gadget are exactly as described. To do so, we will make use of the concept of a cut in a gadget.

▶ **Lemma 4.5.** *Let A be a connected region of a planar embedding of a gadget system which does not contain any locations. Then the boundary of A, which we will call a cut, is traversed an even number of times during any traversal of the construction.*

Proof. Whenever the boundary of A is crossed, the robot goes from inside A to outside or vice versa. Since the robot starts a traversal outside A and ends it outside A, it must cross the boundary an even number of times. ◀

The upper pathways of the outer toggles form a cut, and the lower pathways of the outer toggles form a cut. Thus, the upper pathways of the outer toggles are crossed an even number of times, and the lower pathways are passed an even number of times, so the outer toggles must be passed an even number of times in total. Thus, the toggles must either be both oriented in or both out when leaving. However, when leaving the gadget, the outer toggle which the robot exited through must end up oriented in, so both outer toggles must end up oriented in.

The vertical pathways of the middle toggles form a cut. The horizontal pathways form a cut. Thus, upon leaving, the middle toggles must have been traversed an even number of times in total, and hence must end up both left or both right.

The upper pathways of the bottom toggles must be passed an even number of times. So the upper pathways of those toggles must either be both in or both out when leaving the gadget system.

Thus, the gadget system must be left in a state where the outer toggles are oriented in, the middle toggles are oriented either both left or both right, and the upper pathways of the bottom toggles are oriented either both in or both out. Therefore, these are exactly the accessible configurations, as desired.

Finally, we show that the robot leaves the gadget in the same state it was entered in, if it is entered on the middle tunnel. If the robot passes through one of the upper tunnels of the bottom toggles, when it leaves the region bounded by the bottom toggles' upper tunnels, it must leave one of the bottom toggle's upper tunnels oriented in. By the parity constraint, both bottom toggles' upper tunnels will be oriented in, thus leaving the gadget in the unlocked state. If the central tunnels are entered in the unlocked state, they will be left in the unlocked state. In the locked state, the upper tunnels of the bottom toggles cannot be passed, and so the gadget will be left in the locked state.

Thus, the construction correctly simulates a NTL. ◀

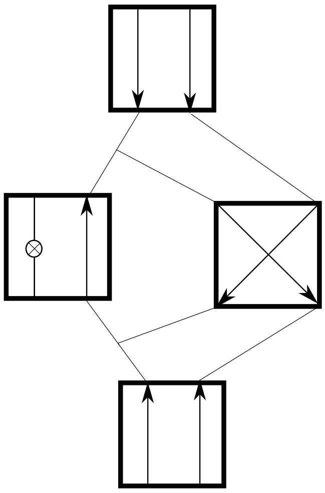


Figure 7 Round antiparallel 2-toggle-lock construction.

4.1 2-toggles and non-crossing toggle locks simulate 2-toggle locks

We introduce some new three tunnel objects. There are several distinct planar topologies of the tunnels in a three tunnel object. We will focus on the two topologies which can be drawn with no internal crossing tunnels: three tunnels around the perimeter, and three tunnels in parallel. We will call the former a "round" topology, and the latter a "stacked" topology. Note that in the stacked topology, the order of the tunnels is relevant. In either topology, if there are multiple toggles, the relative orientation must still be specified.

▶ **Lemma 4.6.** *2-toggles and noncrossing toggle locks simulate a round antiparallel 2-toggle-lock (RAP2TL) and a round parallel 2-toggle-lock (RP2TL).*

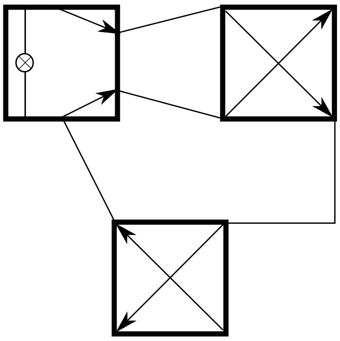
Proof. The construction shown in Figure 7 simulates the behavior of a round antiparallel 2-toggle-lock. It has two externally accessible states: as shown, and with the middle two gadgets flipped. These correspond to the 2-toggle of the RAP2TL being pointed counterclockwise and clockwise respectively.

We will demonstrate that this gadget is equivalent to a RAP2TL by examining all possible traversals. From the two locations that are on the lock tunnel of the NTL, the only possible traversals are to each other, if the lock tunnel is unlocked. This forms the lock tunnel of the RAP2TL.

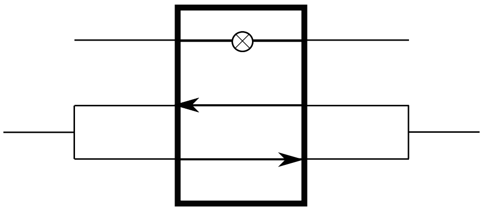
Traversals from the top left location: The robot must go down and to the right, due to the orientation of the toggle of the NTL. Then, the robot can go through the C2T, at which point it is blocked by the orientation of the bottom P2T. Thus, no traversal is possible from this location in this state.

Traversals from the top right location: The robot can go through the C2T, then through the NTL. At this point, the robot cannot go through the C2T again, because the C2T has been toggled. Therefore, its only option is to go through the upper P2T and leave at the top left location. This traversal toggles both of the middle two gadgets, and toggles the upper P2T twice. Thus, the external state of the gadget is flipped. This is the equivalent of traversing the upper toggle of the RAP2TL that we are simulating.

Traversals from the bottom left location: The robot must go up and to the left, due to the orientation of the C2T. Then, the robot can go through the NTL. Due to the orientation of the upper P2T, the robot must now go through the C2T. Now, the robot can leave at the



■ **Figure 8** A round parallel 2-toggle lock is used to construct a stacked antiparallel 2-toggle lock.



■ **Figure 9** A noncrossing tripwire lock constructed from an anti-parallel 2-toggle and lock with the lock on the side.

bottom right location. This traversal toggles both of the middle two gadgets, and toggles the lower P2T twice. Thus, the external state of the gadget is flipped. This is the equivalent of traversing the lower toggle of the RAP2TL that we are simulating.

Traversals from the bottom right location: The robot is blocked by the orientation of the C2T. Thus, no traversal is possible from this location in this state.

The opposite state is equivalent to a top-bottom mirror reversal, except for a change in the state of the lock, which does not affect which traversals are possible. Thus, in every state, this system of gadgets is equivalent to a round antiparallel two-toggle-lock (RAP2TL).

Consider the gadget which is the same as the one in Figure 7, except that the bottom P2T is replaced with a C2T with its toggles allowing traversals from the bottom locations into the gadget. Clearly, the effect of this change is to swap the roles of the bottom two locations. As a result, this new construction is a round parallel two-toggle-lock, a RP2TL. ◀

▶ **Lemma 4.7.** *RP2TLs and 2Ts simulate a stacked antiparallel 2-toggle-lock (SAP2TL).*

Proof. A SAP2TL is a three tunnel gadget where the three tunnels cross the gadget in parallel, with the two antiparallel toggle tunnels next to each other.

Starting with a RP2TL and two C2Ts, we can simulate a SAP2TL as shown in Figure 8. The lock tunnel is straightforward. The two other traversals are from the top left to the bottom left, and from the bottom right to the top right. Both of these traversals pass through every gadget. In the other state, all three gadgets are flipped, and the same traversals are possible in the opposite direction.

Since every state-affecting traversal traverses all gadgets, the states of the three gadgets always switch together, and the behavior is that of an SAP2TL. Equivalently, by Lemma 2.2, the system of gadgets is deterministic and reversible, so the three traversals mentioned are the only ones possible, and the construction simulates a SAP2TL. ◀

4.2 2-toggle locks simulate non-crossing wire locks

▶ **Lemma 4.8.** *AP2TLS simulates a NWL.*

Proof. By connecting the locations of the SAP2TL as shown in Figure 9, we can simulate a NWL.

Each traversal of either connected toggle tunnel flips the state. The connections between these two tunnels ensure that travel in either direction is always possible. As a result, the

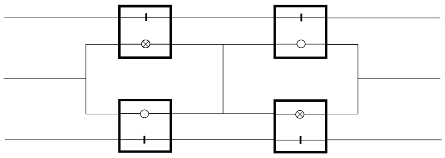


Figure 10 A stacked tripwire-lock-tripwire constructed from non-crossing tripwire locks.

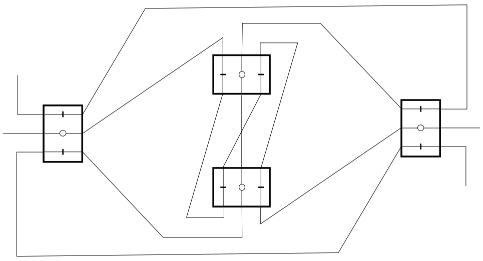


Figure 11 A crossover constructed from stacked tripwire-lock-tripwires.

combination of these connected pathways acts as a tripwire, always allowing the robot to pass in either direction and opening or closing the lock with each traversal. ◀

4.3 Non-crossing wire locks simulate crossovers

On our way to simulating a crossover, we will simulate another three tunnel gadget, a stacked tripwire-lock-tripwire (SWLW). Note that the lock tunnel is specifically the center tunnel.

▶ **Lemma 4.9.** *NWLs simulate a stacked tripwire-lock-tripwire (SWLW).*

Proof. The construction is shown in Figure 10. There are four accessible states of this gadget, which are any of the states where there is one locked and one unlocked NWL among the two top NWLs, and one of each among the two bottom NWLs.

The states can only be changed by traversing the tripwire tunnels, and doing so flips both NWLs on the side traversed, maintaining the invariant.

If both left NWLs are locked, or both right NWLs are locked, the center tunnel is not passable. In the other two accessible states, the center tunnel is passable. The two pairs correspond to the two external states, with the lock locked and unlocked respectively. In any state, traversing either tripwire moves the gadget to a state with the opposite passability of the lock tunnel. Thus, this construction simulates a SWLW. ◀

▶ **Lemma 4.10.** *SWLWs simulate a crossover.*

Proof. The gadget shown in Figure 11 implements a crossover. The robot may always cross from left to right, right to left, top to bottom and bottom to top, but in no other directions. There is a single accessible state, the one with all four SWLWs in the unlocked state.

When the robot enters from any of the four external locations it has only a single option up until the point where it reaches the four-way intersection at the center. Upon reaching this point, the robot has traversed the tripwire tunnels of two of the SWLWs, locking them. In particular, the SWLWs whose lock tunnels are on the two orthogonal pathways are locked. For instance, if the robot entered from the top, the left and right pathway's SWLWs would be locked at this point. As a result, the only way for the robot to continue is to go straight, passing through the other tripwires of the same two SWLWs, and emerging from the other side. The robot has completed a crossover traversal, with no other options.

Because the robot passed through the tripwires of two SWLWs twice, and only the lock tunnels of the other two SWLWs, the object is left in its original state, making the state shown in Figure 11 the only accessible state. This construction correctly simulates a crossover. ◀

For the PSPACE-completeness result, we make use of 2-toggle locks and crossovers. Combining the lemmas in Section 4, we have the result we will make use of:

▶ **Theorem 4.11.** *AP2Ts simulate crossovers and all 2-toggle-locks.*

Proof. By composing the lemmas in Section 4, we see that AP2Ts simulate crossovers and RAP2TLs. By using the crossover to effectively rearrange locations, we can simulate an arbitrary 2-toggle-lock. ◀

5 Everything simulates everything else

The remaining gadgets of interest are each individually (when combined with branching hallways) sufficient to make motion planning problems PSPACE-complete. Moreover, each gadget can be simulated by a constant number of each other gadget. To prove this, we give simple gadgets to show how to construct noncrossing-tripwire-toggles from anti-parallel-2-toggles, and anti-parallel 2-toggles from each of noncrossing-toggle-locks, noncrossing-wire-locks, noncrossing-wire-toggles and parallel-2-toggles. We then show that a crossing version of a gadget can very simply make a non-crossing version of the same gadget.

▶ **Theorem 5.1.** *The 2-toggles, toggle-locks, tripwire-locks and tripwire-toggles, in all orientations, can each simulate each other.*

Proof. We have already established that AP2Ts can simulate P2Ts, C2Ts, NTLs and NWLs and crossovers. We will establish that:

- AP2Ts can simulate NWTs. Lemma 5.3.
- P2Ts, NTLs, NWTs and NWLs can each simulate AP2Ts. Lemmas 5.4, 5.5, 5.6, 5.7, respectively.
- C2Ts can simulate P2Ts by Lemma 4.3, and hence AP2Ts as well.
- CTLs can simulate NTLs, CWLs can simulate NWLs, and CWTs can simulate NWTs. Lemma 5.8.

Thus, every gadget can simulate AP2Ts, and AP2Ts can simulate every non-crossing gadget, as well as crossovers. By combining non-crossing gadgets with crossovers, AP2Ts can simulate every gadget. This gives a simulation of every gadget by every other gadget, via AP2Ts as an intermediate step. ◀

▶ **Corollary 5.2.** *Motion planning with any one of the gadgets in Theorem 5.1 (and branching hallways) is PSPACE-complete.*

Proof. Corollary 5.2 follows from Theorem 5.1, which establishes that each gadget can simulate a AP2T, and Theorem 4.1, which establishes that motion planning with AP2Ts is PSPACE-complete. ◀

▶ **Lemma 5.3.** *AP2Ts simulate an NWT.*

Proof. We will construct a NWT as shown in Figure 12. This requires NWLs, crossovers, and 1-toggles. We already have existing constructions of NWLs and crossovers with AP2Ts. We can also build a 1-toggle with an AP2T simply by ignoring one of the two tunnels. Thus, all that's left is to show that the construction successfully simulates a NWT.

There are four accessible states: As shown in Figure 12, with all of the NWLs flipped, with the toggle flipped, and with everything flipped. The first and last correspond to the external state where the toggle is pointed right, while the other two correspond to the external state where the toggle is pointed right. The horizontal tunnel corresponds to the toggle, while the U-shaped tunnel corresponds to the tripwire in the composed gadget. In the state shown in the figure, the toggle is oriented to the right from the external perspective.

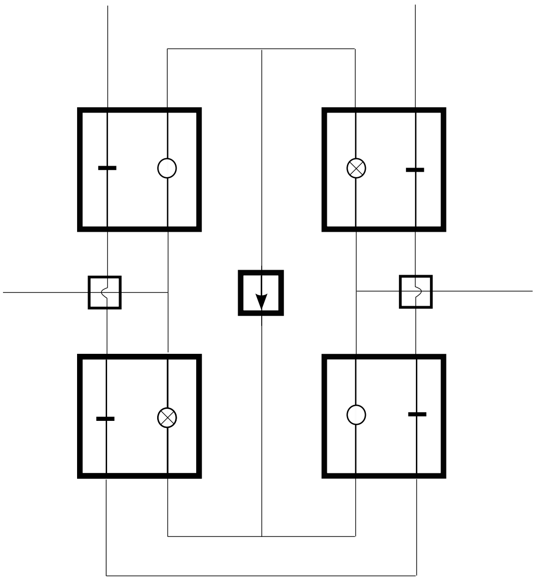


Figure 12 A noncrossing wire toggle constructed from a toggle, four noncrossing tripwire locks, and two crossovers.

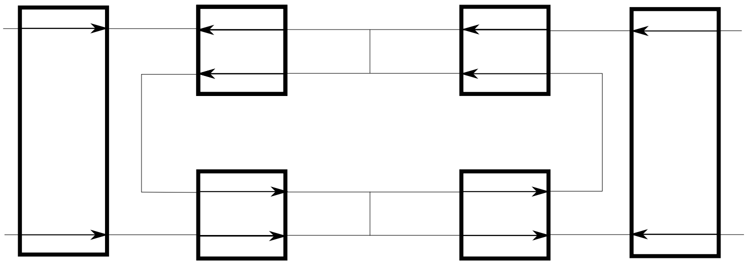


Figure 13 Parallel 2-toggles simulate anti-parallel 2-toggles.

Clearly, traversing the U-shaped tunnel will flip all of the tripwires of the NWL, resulting in a state which corresponds to the opposite external state, as desired.

In the state shown in the figure, the horizontal tunnel may be traversed from left to right along a unique pathway due to the placement of the locks, flipping the toggle along the way. The orientation of the toggle blocks the right to left traversal. Thus, in this state, the upper tunnel may be traversed in one direction resulting in an allowed state which corresponds to the opposite external state, as desired.

Placing the toggle in the opposite state is equivalent to a rotation by π of the upper tunnel, showing this state also correctly simulates an NWT.

Flipping the states of all of the NWLs is equivalent to a vertical reflection of the upper tunnel, showing this state also correctly simulates an NWT. ◀

▶ **Lemma 5.4.** *P2Ts simulate an AP2T.*

Proof. Figure 13 gives a construction of an antiparallel-2-toggle out of parallel-2-toggles.

There are two accessible states: As shown, and with the four inner P2Ts flipped. The former corresponds to the AP2T having a tunnel connecting the left two locations with its toggle oriented upward, and a tunnel connecting the right locations with its toggle oriented downward, while the latter corresponds to the two toggles flipped.

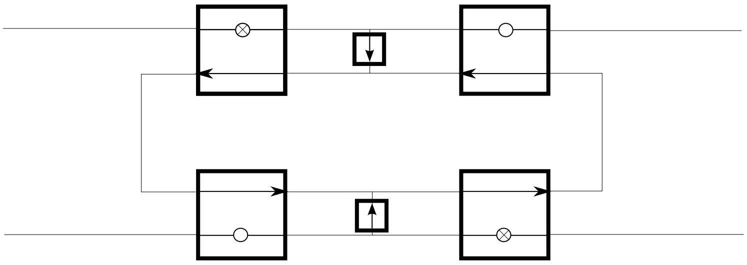


Figure 14 Noncrossing-toggle-lock simulates anti-parallel-2-toggle.

First, let us examine the bottom right location in the state shown in the figure. After passing the rightmost P2T, the robot is blocked. No transitions or state changes are possible. This matches the desired behavior, because the right toggle in the AP2T being simulated is oriented down.

Next, let us examine the top right location in the state shown in Figure 13. After passing the rightmost P2T, then the upper right P2T, the robot may now either proceed along the top tunnel, or down to the central loop. In the former case, the robot may pass through the upper left P2T, but then is blocked. In the later case, the robot may either proceed around the loop to the left or to the right. If the robot goes to the right, it can pass through the lower tunnel of the upper right P2T, but then is stuck. If the robot goes to the left, it can pass through the lower tunnel of the upper left P2T, then the upper tunnel of the lower left P2T.

At this point, the robot may either continue around the loop, or exit the loop downward. If the robot continues around the loop, it can pass through the upper tunnel of the lower right P2T, but then is stuck. If it exits the loop, it can either go left or right on the bottom tunnel. If it goes left, it can pass through the lower tunnel of the lower left P2T, but then is stuck. If it goes right, it can pass through the lower tunnel of the lower right P2T, then the lower tunnel of the rightmost P2T, and exit the gadget.

Overall, we observe that the robot can make exactly one transition, from top right to bottom right. The right toggle is traversed twice, and the inner toggles are all traversed once, leaving the gadget in the other accessible state. No other transition or state change is possible, from that entrance.

Since the gadget is rotationally symmetric about its center, the possible transitions from the right mirror the possible transitions from the left. Since the other state is simply the state shown in the figure mirrored top-to-bottom, the transitions described mirror the transitions in the other state as well. ◀

▶ **Lemma 5.5.** *NTLs simulate an AP2T.*

Proof. The construction is shown in Figure 14. The two accessible states are the state shown in the figure and the state with all of the NTLs flipped, but the one-toggles still oriented inward. These correspond to an AP2T with the top tunnel directed left and bottom tunnel directed right, and the left-right mirror image.

If the robot enters from the top right, after passing the lock of the top right NTL, it must pass the upper one-toggle and proceed into the central loop. Since the lower toggle is directed upward, the robot must eventually leave the central loop via the upper toggle. The robot may now proceed around the loop. The loop may only be traversed counterclockwise, and it may only be traversed once. The robot may of course backtrack at any point, but when it leaves via the upper toggle, it must have either traversed the loop zero or one times.

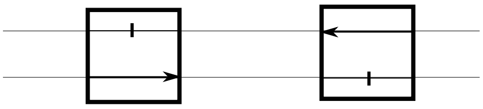


Figure 15 Noncrossing-wire-toggle simulates anti-parallel-2-toggle.

In the former case, the robot must leave via the top right location, leaving the system in its original state. In the latter case, the robot must leave via the top left location, as all of the locks have flipped. Thus, the top tunnel may be traversed via a right to left traversal, flipping the state, and that is the only traversal in that direction.

If the robot enters from the top left, it is immediately blocked by the lock, and no traversal is possible. Thus, the top tunnel works as desired.

Since the gadget possesses rotational symmetry around its center, the bottom tunnel is exactly the same, allowing only a left to right traversal, flipping the state.

The opposite state is the same as the original state except for a left-right right mirror reversal, so it also functions exactly as desired from the AP2T. ◀

▶ **Lemma 5.6.** *NWTs simulate an AP2T.*

Proof. A noncrossing wire toggle can simulate an anti-parallel 2-toggle with the simple construction shown in Figure 15. The direction of each tunnel is dictated by the toggle on the tunnel, and the wire ensures both toggles are synchronized. Thus when either tunnel is traversed, both NWTs flip and the direction each tunnel can be traversed flips. ◀

▶ **Lemma 5.7.** *NWLs simulate an AP2T.*

Proof. The construction of an anti-parallel 2-toggle from non-crossing tripwire locks can be seen in Figure 16. Note that a 1-toggle can be constructed from an NWL by simply connecting one location of the wire to one location of the lock. A closed lock will prevent travel in one direction, but crossing the tripwire in the other direction will open the lock and allow the robot to proceed. An open lock will allow travel in the other direction. In the direction starting from the tripwire, the tripwire will close the lock in front of the robot preventing traversal. In either traversal, the tripwire is crossed, flipping the state.

There are two main parts to this gadget, the top and bottom tunnels, and the inner loop. As with the NTL construction from Lemma 5.5, the 1-toggles ensure that the loop must be exited from the same place it was entered, which ensures all gadgets on the loop are traversed the same number of times. Since all wires are on this loop, in a given traversal of this gadget system, all of the NWLs will change state the same number of times, keeping them in sync. The upper and lower paths each contain a locked and unlocked tunnel. The locked portion prevents entry and interaction with the gadget. From the unlocked side, the robot is able to enter the gadget and flip its state an arbitrary number of times. If the state is flipped an even number of times, the robot's only path out is the way it came. If an odd number of flips have occurred, the robot can now exit through the opposite side of its path, leaving the gadget in the opposite state.

Therefore, the gadget may traversed right to left along the top tunnel, flipping the state, and left to right along the bottom tunnel, flipping the state. We have built an AP2T. ◀

▶ **Lemma 5.8.** *CWTs simulate an NWT, CWLs simulate an NWL, CTLs simulate an NTL.*

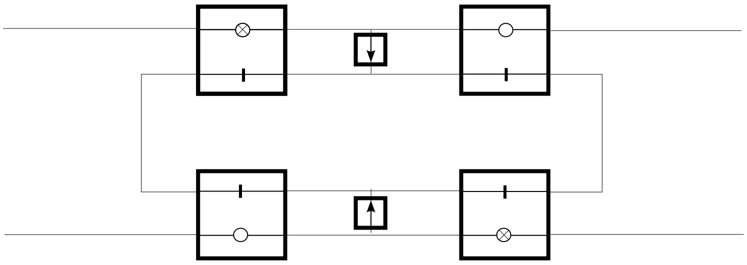


Figure 16 Noncrossing-wire-lock simulates anti-parallel-2-toggle.

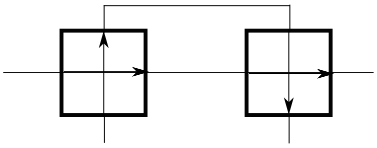


Figure 17 Crossing 2-toggles simulate parallel 2-toggle.

In general, one can very easily simulate a non-crossing version of a 2-tunnel gadget from the crossing version. Figure 17 shows a parallel-2-toggle being constructed from a crossing-2-toggle. The same construction works for uncrossing the other gadgets we have analyzed, namely tripwire-toggles, tripwire-locks and toggle-locks. Going from non-crossing to crossing versions is significantly more complicated (except in the case of anti-parallel-2-toggle to crossing-2-toggle) but we are rescued from the need of such constructions by being able to simulate a general crossover in Lemma 4.10.

6 More reasons Zelda is hard

In this section we use this framework to give an alternate proof that The Legend of Zelda: Oracle of Seasons is PSPACE-complete. Along the way, we will show that motion planning with reversible deterministic gadgets we call 'spinners' is also PSPACE-complete.

A k-spinner is a two state deterministic reversible gadget on k locations. In one state, each location is connected to its neighbor by a directed edge in a clockwise direction. In the other state, all locations are likewise connected in a counterclockwise direction. A 4-spinner is shown in Figure 18. The study of 4-spinners was posed by Jeffrey Bosboom due to their appearance in The Legend of Zelda: Oracle of Seasons. We show that for any $k \geq 4$, path-planning problems with k-spinners and branching hallways is PSPACE-complete.

First, we can take a k spinner and have all but three consecutive locations lead to dead ends. The remaining three locations form a gadget that we call a deterministic fork. A deterministic fork is a reversible, deterministic gadget on three locations. In one state, it allows the robot to go from the center to the right location and return from the left to the center location. In the other state these directions are reversed. Figure 19 shows the construction of a crossing 2-toggle from two 4-spinners or equivalently two deterministic forks.

▶ **Theorem 6.1.** *For any $k \geq 4$, the path-planning problem with k-spinners and branching hallways is PSPACE-complete.*

Proof. We construct a deterministic fork by ignoring $k - 3$ of the edges in the spinner. Two deterministic forks together simulate a crossing 2-toggle as shown in Figure 19. By Corollary 5.2, the motion planning problem with crossing 2-toggles is PSPACE-complete. ◀

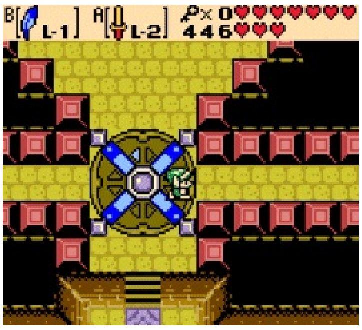


Figure 18 Example of a 4-spinner in The Legend of Zelda: Oracle of Seasons.

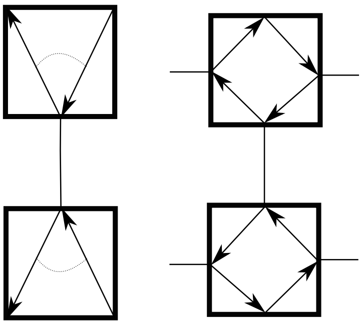


Figure 19 4-spinners simulate deterministic forks which simulate crossing 2-toggles.

▶ **Corollary 6.2.** *Determining if a player can beat a level in generalized The Legend of Zelda: Oracle of Seasons is PSPACE-hard.*

Proof. The Legend of Zelda: Oracle of Seasons contains 4-spinners and requires the player to navigate from one location to a target location in a grid. Since planar graphs can be laid out in a grid with only quadratic blowup [2], we can reduce from motion planning problems with 4-spinners which are PSPACE-complete by Theorem 6.1. ◀

The complexity of motion planning with 3-spinners, as well as the two other reversible, deterministic, 2 state, 3 location gadgets, remains open. Since 2-spinners are the same as an edge in a graph, this would give a tight characterization for the spinner gadget. The authors would also be interested to know what other games and puzzles use spinners.

7 General hardness characterization

Here, we tightly characterize the hardness of the motion planning problem with all deterministic, reversible, 2-state, k-tunnel gadgets.

▶ **Theorem 7.1.** *Motion planning with any deterministic, reversible, 2-state, k-tunnel planar gadget (with branching hallways) is PSPACE-complete if and only if the gadget has two toggle tunnels, a toggle tunnel and a tripwire tunnel, a toggle tunnel and a lock tunnel or a tripwire tunnel and a lock tunnel. Motion planning with all other such gadgets is in P.*

First, we provide upper bounds for some classes of simpler gadgets. This shows that, for their category, our hardness results are minimal in the sense that path planning with simpler gadgets in the same class can be solved in P.

▶ **Theorem 7.2.** *Gadgets with only one state are in NL.*

Proof. One state gadgets cannot change in any way. Thus they must all be comprised of static descriptions of allowed traversals from one location to another. This can be modeled as a mixed graph. Path planning in mixed graphs is in NL [10]. ◀

The only nontrivial gadget on 1 tunnel with two states which is reversible and deterministic is the 1-toggle.

▶ **Theorem 7.3.** *Motion planning with 1-toggles is in NL.*

Proof. We reduce this problem to ST connectivity in mixed graphs. To solve this problem we simply treat every 1-toggle as a directed edge pointed in the direction the 1-toggle is initially oriented and then run the standard algorithm. It is obvious that if a solution here exists then a path in the 1-toggle planning problem also exists. What is less clear is that this is sufficient to find any such path.

Consider a path which traverses at least one toggle more than once. Consider the last toggle on the path which is traversed more than once. After this toggle is traversed, only toggles which are traversed at most once are on the path. Call this toggle t, and let its final traversal be from u to v. Since t was traversed repeatedly, there was some previous point in the path where the robot was at v, before it traversed t the second-to-last time. Let us create a new path where the robot skips the cycle in the original path from v through t to u, then eventually back to u through t to v. This path must successfully reach the end, as every toggle after t is traversed at most once, and so is in the same state regardless of whether the cycle is omitted.

Thus, under the assumption that there is a path which traverses toggle more than once, there is another, shorter path. Thus, the shortest path must not traverse toggles more than once, and so such a path must exist if any path exists. ◀

The remaining two-state two-tunnel deterministic reversible gadgets are also in P. We note that a wire-wire never changes its connectivity and is thus no different then two undirected edges. A lock-lock can never change its state and thus is reducible to a one state gadget, simply zero, one, or two undirected edges. A gadget with a tunnel which does not change and is not changed by the state of the gadget is reducible to two gadgets on one tunnel each, which are in P by Theorem 7.3. This exhausts the 2-state 2-tunnel reversible undirected gadgets.

Proof of Theorem 7.1. Now, we can characterize all two state, deterministic, reversible gadgets on any number of tunnels.

Any gadget with two toggle tunnels, a toggle tunnel and a tripwire tunnel, a toggle tunnel and a lock tunnel or a tripwire and a lock tunnel is sufficient to make motion planning hard, by ignoring all other tunnels and using one of the constructions from this paper.

We can divide all other gadgets into three categories: those with tripwires and trivial tunnels, those with locks and trivial tunnels, and those with a single toggle and trivial tunnels. The passability of a tunnel in a gadget with only tripwires and trivial tunnels never changes, making motion planning equivalent to st-connectivity. A gadget with only locks and trivial tunnels can never have its state change, allowing us to apply Theorem 7.2. A gadget with a single toggle and some number of trivial tunnels can be treated as a one-toggle together with some number of undirected edges. Thus, any system of gadgets of these types is equivalent to a system of 1-toggles and undirected edges. After that, the same argument as in Theorem 7.3 can be used to solve the motion planning problem in that system. ◀

8 Open Problems / Conclusion

This framework for abstract motion planning problems leaves open the question of the computational complexity of motion planning with many other types of gadgets. One can examine gadgets with more states, without the tunnel restriction, or without the deterministic and reversible restrictions. Since this is a vast undertaking with many of the gadgets and their combinations likely to be uninteresting, we suggest some of the following categories to be of particular interest.

- 3 spinners are the only size of spinner for which motion planning remains open.
- Three location, 2-state, deterministic, reversible gadgets seem like the obvious 'simplest' category of gadgets.
- Are there any sets of purely deterministic and reversible gadgets for which motion planning is PSPACE-complete (e.g. without branching hallways, which are non-deterministic)?
- What about reversible but nondeterministic gadgets on two tunnels or three locations?

There is currently significant partial progress on all of the listed topics. Please contact us before spending significant time working on the open problems listed to prevent duplication of effort.

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. Originally at FUN 2014.

2 Hubert de Fraysseix, János Pach, and Richard Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990. doi:10.1007/BF02122694.

3 Erik D. Demaine, Martin L. Demaine, Michael Hoffmann, and Joseph O'Rourke. Pushing blocks is hard. *Computational Geometry: Theory and Applications*, 26(1):21–36, August 2003.

4 Erik D. Demaine, Isaac Grosof, and Jayson Lynch. Push-pull block puzzles are hard. In Dimitris Fotakis, Aris Pagourtzis, and Vangelis Th. Paschos, editors, *Algorithms and Complexity - 10th International Conference, CIAC 2017, Athens, Greece, May 24-26, 2017, Proceedings*, volume 10236 of *Lecture Notes in Computer Science*, pages 177–195, 2017. doi:10.1007/978-3-319-57586-5_16.

5 Erik D. Demaine, Robert A. Hearn, and Michael Hoffmann. Push-2-f is pspace-complete. In *Proceedings of the 14th Canadian Conference on Computational Geometry*, pages 31–35, Lethbridge, Alberta, Canada, August 12–14 2002.

6 Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder/easier than we thought. In *Proceedings of the 8th International Conference on Fun with Algorithms*, pages 13:1–13:14, La Maddalena, Italy, June 8–10 2016.

7 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.

8 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A. K. Peters, Ltd., Natick, MA, USA, 2009.

9 André Grahl Pereira, Marcus Ritt, and Luciana S. Buriol. Pull and pushpull are pspace-complete. *Theor. Comput. Sci.*, 628:50–61, 2016. doi:10.1016/j.tcs.2016.03.012.

10 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

11 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 216–226, San Diego, California, May 1978.

The Computational Complexity of Portal and Other 3D Video Games

Erik D. Demaine
MIT CSAIL, 32 Vassar Street, Cambridge, MA 02139, USA
edemaine@mit.edu

Joshua Lockhart[1]
Department of Computer Science, University College London, London, WC1E 6BT, UK
joshua.lockhart.14@ucl.ac.uk

Jayson Lynch
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
jaysonl@mit.edu

Abstract

We classify the computational complexity of the popular video games Portal and Portal 2. We isolate individual mechanics of the game and prove NP-hardness, PSPACE-completeness, or pseudo-polynomiality depending on the specific game mechanics allowed. One of our proofs generalizes to prove NP-hardness of many other video games such as Half-Life 2, Halo, Doom, Elder Scrolls, Fallout, Grand Theft Auto, Left 4 Dead, Mass Effect, Deus Ex, Metal Gear Solid, and Resident Evil. These results build on the established literature on the complexity of video games [1,3,7,18].

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases video games, hardness, motion planning, NP, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.19

Related Version https://arxiv.org/abs/1611.10319

Acknowledgements All raster figures are screenshots from Valve's Portal or Portal 2, either using Portal 2's Puzzle Maker or by way of the Portal Unofficial Wiki (http://theportalwiki.com/).

1 Introduction

In Valve's critically acclaimed *Portal* franchise, the player guides *Chell* (the game's silent protagonist) through a "test facility" constructed by the mysterious fictional organization Aperture Science. Its unique game mechanic is the Portal Gun, which enables the player to place a pair of portals on certain surfaces within each test chamber. When the player's avatar jumps into one of the portals, she is instantly transported to the other. This mechanic, coupled with the fact that in-game items can be thrown through the portals, has allowed the developers to create a series of unique and challenging puzzles for the player to solve as they guide Chell to freedom. Indeed, the Portal series has proved extremely popular, and is estimated to have sold more than 22 million copies [2,20].

[1] Work started while author was at School of Electronics, Electrical Engineering and Computer Science, Queen's University, Belfast, BT7 1NN, UK

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 19; pp. 19:1–19:22
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Table 1 Summary of new Portal complexity results

Mechanics	Portals	Long fall	Complexity
Emancipation Grills, No Terminal Velocity	Yes	Yes	Weakly NP-comp. (§4)
Turrets	No	Yes	NP-hard (§5)
Timed Door Buttons and Doors	No	No	NP-hard (§6)
HEP Launcher and Catcher	Yes	No	NP-hard (§7)
Cubes, Weighted Buttons, Doors	No	No	PSPACE-comp. (§8)
Lasers, Relays, Moving Platforms	Yes	No	PSPACE-comp. (§9)
Gravity Beams, Cubes, Weighted Buttons, Doors	No	No	PSPACE-comp. (§9)

We analyze the computational complexity of Portal following the recent surge of interest in complexity analysis of video games and puzzles. Examples of previous work in this area includes NP-completeness of Tetris [5], PSPACE-completeness of Lemmings [19] and Super Mario Bros. [6], and hardness of many other classic video games [7, 18]. See also the surveys [4, 9, 11].

In this paper, we explore how different game elements contribute to the computational complexity of Portal 1 and Portal 2 (which we collectively refer to as *Portal*), with an emphasis on identifying gadgets and proof techniques that can be used in hardness results for other video games. We show that a generalized version of Portal with Emancipation Grills is weakly NP-hard (Section 4); Portal with turrets is NP-hard (Section 5); Portal with timed door buttons and doors is NP-hard (Section 6); Portal with High Energy Pellet launchers and catchers is NP-hard (Section 7); Portal with Cubes, Weighted Buttons, and Doors is PSPACE-complete (Section 8); and Portal with lasers, laser relays, and moving platforms is PSPACE-complete (Section 8).

Table 1 summarizes these results. The first column lists the primary game mechanics of Portal we are investigating. The second and third column note whether the long fall or Portal Gun mechanics are needed for the proof. Section 2 provides more details about what these models mean. The turret proof generalizes to many other video games, as described in Section 5.4.

2 Definitions of Game Elements

Portal is a single-player *platform game*: a game with the goal of navigating the avatar from a start location to an end location of a series of stages, called *levels*. The gameplay in Portal involves walking, turning, jumping, crouching, pressing buttons, picking up objects, and creating portals. The locations and movement of the avatar and all in-game objects are discretized. For convenience we make a few assumptions about the game engine, which we feel preserve the essential character of the games under consideration, while abstracting away certain irrelevant implementation details in order to make complexity analysis more amenable:

- Positions and velocities are represented as triples of fixed-point numbers in Cartesian coordinates.[2] Each velocity vector is limited in magnitude by a terminal velocity v_{max}.

[2] The actual game uses floats in many instances. We claim that all our proofs work if we round the numbers involved, and only encode the problems in the significand.

- Time is discretized and represented as a fixed-point number. Parameter δ defines the amount of time advanced during each simulation time step.
- At each time step, there is only a constant number of possible user inputs: button presses and the cursor position. The user is able to apply any of these inputs within a time step.
- The cursor position is represented by two fixed-point numbers in spherical coordinates.
- At each time step, we update all objects' positions and velocities as follows:
  - Update velocities based on acceleration from user commands and from gravity: $\vec{v}_{t+1} = \vec{v}_t + \delta(\vec{a}_{input} + \vec{a}_\gamma)$ where $\vec{a}_\gamma = [0, 0, -\gamma]$ and g is a constant.
  - If a velocity vector $\vec{v}_{t+1}$ has magnitude $> v_{max}$, scale it down to have magnitude v_{max}.
  - Update positions according to these velocities: $\vec{p}_{t+1} = \vec{p}_t + \delta\vec{v}$.
  - Check for collisions by extruding the objects into a fourth temporal dimension by δ and checking for intersection of those objects.[3]
  - For the purposes of this paper, we define a collision model only between single moving objects and non-moving objects, as this is all we need in our proofs possibly involving collisions (Sections 4 and 7). We ignore details of more complex collisions as they are not relevant to our results.
  - For an inelastic collision between a moving object A and a non-moving object B, we calculate the first time $\delta' \leq \delta$ at which the objects would intersect, and move A instead to this position (scaling the velocity vector by δ' instead of δ). Then we project A's velocity vector onto the surface of B at the point of intersection.
  - For an elastic collision, we similarly calculate the first time of intersection and update the position of A, but update the velocity vector instead to its reflection off of the surface at the point of intersection.
  - If an object passes through a portal, its velocity vector is rotated by the rotation that brings the entering portal frame to the exiting portal frame.
- Portals from the portal gun and bullets from turrets are resolved instantaneously in a single time step by line-of-effect rather than any ballistic simulation.[4]

In Portal, a *level* is a description of the polygonal surfaces in 3D defining the geometry of the map, along with a simulation rate and a list of game elements with their locations and, if applicable, connections to each other. In general, we assume that the level can be specified succinctly as a collection of polygons whose coordinates may have polynomial precision, (and thus so can the player coordinates), and thus exponentially large values (ratios). This assumption matches the Valve Map Format (VMF) used to specify levels in Portal, Portal 2, and other Source games [16]. A realistic special case is where we aim for *pseudopolynomial* algorithms, that is, we assume that the coordinates of the polygons and player are assumed to have polynomial values/ratios (logarithmic precision), as when the levels are composed of explicit discrete blocks. This assumption matches the voxel-based P2C format sometimes used for community-created Portal 2 levels [15].

In this work, we consider the following decision problem, which asks whether a given level has a path from the given start location the end location.

▶ **Problem 1.** PORTAL

Parameter: A set of allowed gameplay elements.

[3] This approach is precise, and should reasonably capture the relevant dynamics in the game, but computationally inefficient and likely not how collision detection is performed in practice.

[4] The end of Portal 2 gives a very large lower bound on the speed of effect of the portal gun.

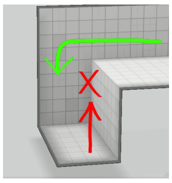
Input: A description of a Portal level using only allowed gameplay elements, and spatial coordinates specifying a start and end location.

Output: Whether there exists a path traversable by a Portal player from the start location to the end location.

3 Game Element Descriptions

The key game mechanic, the *Portal Gun*, creates a portal on the closest surface in a direct line from the player's avatar if the surface is of the appropriate type. We call surfaces that admit portals *portalable*. There are a variety of other gameplay elements which can be a part of a Portal level. Below we give descriptions and images of various game elements used in Portal 1 and 2.

1. A *long fall* is a drop in the level terrain that the avatar can jump down from without dying, but cannot jump up.



It's a long way down.

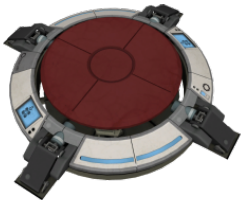
2. A *door* can be open or closed, and can be traversed by the player's avatar if and only if it is open. In Portal, many mechanics can act as doors, such as literal doors, laser fields, and moving platforms. On several occasions we will assume the door being used also blocks other objects in the game, such as High Energy Pellets or lasers, which is not generally true.

A Door in Portal 2

3. A *button* is an element which can be interacted with when the avatar is nearby to change the state of the level, e.g., a button to open or close a door.
4. A *timed button* will revert back to its previous state after a set period of time, reverting its associated change to the level too, e.g., a timed button which opens a door for 10 seconds, before closing it again.

Timed Button

5. A *weighted floor button* is a an element which changes the state of a level when one or more of a set of objects is placed on it. In Portal, the 1500 Megawatt Aperture Science Heavy Duty Super-Colliding Super Button is an example of a weighted floor button which activates when the avatar or a Weighted Storage Cube is placed on top of it. An activated weighted floor button can activate other mechanics such as doors, moving platforms, laser emitters, and gravitational beam emitters.

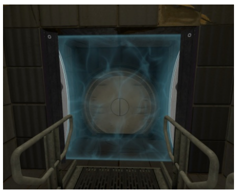


Heavy Duty Super-Colliding Super Button

6. *Blocks* can be picked up and moved by the avatar. The block can be set down and used as a platform, allowing the avatar to reach higher points in the level. While carrying a block, the avatar will not fit through small gaps, rendering some places inaccessible while doing so. In Portal, the Weighted Storage Cube is an example of a block that can be jumped on or used to activate weighted floor buttons. We will refer to Weighted Storage Cubes, Companion Cubes, etc. as simply *cubes*.

Weighted Storage Cube

7. A *Material Emancipation Grid*, also called an *Emancipation Grill* or *fizzler*, destroys some objects which attempt to pass through it, such as cubes and turrets. When the avatar passes through an Emancipation Grid, all previously placed portals are removed from the map. Portals cannot be shot through an emancipation grid.

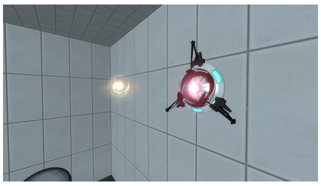


Emancipation Grid

8. The *Portal Gun* allows the player to place portals on portalable surfaces within their line of effect. Portals are orange or blue. If the player jumps into an orange (blue) portal, they are transported to the blue (orange) portal. Only one orange portal and one blue portal may be placed on the level at any given time. Placing a new orange (blue) portal removes the previously placed orange (blue) portal from the level.

Portal Gun

9. A *High Energy Pellet* (HEP) is a spherical object which moves in a straight line until it encounters another object. HEPs move faster than the player avatar. If they collide with the player avatar, then the avatar is killed. If a HEP encounters a wall or another object, it will bounce off it with equal angle of incidence and reflection. In Portal, some HEPs have a finite lifespan, which is reset when the HEP passes through a portal, and others have an unbounded lifespan. These unbounded HEPs are referred to as *Super High Energy Pellets*.



A HEP about to reach a HEP Collector

10. A *HEP Launcher* emits a HEP at an angle normal to the surface upon which it is placed. These are launched when the HEP launcher is activated or when the previously emitted HEP has been destroyed.

HEP Launcher

11. A *HEP Catcher* is a device which is activated if it is ever hit by a HEP. In Portal, this device can act as a button, and is commonly used to open doors or move platforms when activated.

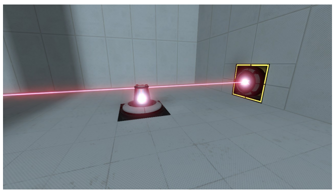
HEP Catcher

12. A *Laser Emitter* emits a *Thermal Discouragement Beam* at an angle normal to the surface upon which it is placed. The beam travels in a straight line until it is stopped by a wall or another object. The beam causes damage to the player avatar and will kill the avatar if they stay close to it for too long. We call the beam and its emitter a *laser*.

A Laser Emitter and Thermal Discouragement Beam.

13. A *Laser Relay* is an object which can activate other objects while a laser passes through it.

14. A *Laser Catcher* is an object which can activate other objects while a contacts it.



An active laser relay and laser catcher.

15. A *Moving Platform* is a solid polygon with an inactive and an active position. It begins in the inactive position and will move in a line at a constant velocity to the active position when activated. If it becomes deactivated it will move back to the inactive position with the opposite velocity.

A horizontal moving platform.

16. A *Turret* is an enemy which cannot move on its own. If the player's avatar is within the field of view of a turret, the turret will fire on the avatar. If the avatar is shot sufficiently many times within a short period of time, the avatar will die.

Turret from Portal 2

A Gravity Beam and Excursion Funnel.

17. An *Excursion Funnel*, also called a *Gravitational Beam Emitter* emits a gravitational beam normal to the surface upon which it is placed. The gravitational beam is directed and will move small objects at a constant velocity in the prescribed direction. Importantly, it will carry Weighted Storage Cubes and the player avatar. Gravitational Beam Emitters can be switched on and off, as well as flipping the direction of the gravitational beam they emit.

There are two main pieces of software for creating levels in Portal 2: the *Puzzle Maker* (also known as the *Puzzle Creator*), and the *Valve Hammer Editor* equipped with the *Portal 2 Authoring Tools*. Both of these tools are publicly available for players to create their own levels. The Puzzle Maker is a more restricted editor than Hammer, with the advantage of providing a more user-friendly editing experience. However, levels created in the Puzzle Maker must be coarsely discretized, with coarsely discretized object locations, and must be made of voxels. In particular, the Puzzle Maker uses the P2C file format while Hammer uses VMF, which restricts it to instances where the size of the level is polynomial in the size of the problem description. Furthermore, no HEP launchers or additional doors can be placed in Puzzle Maker levels. We will often comment on which of our reductions can be constructed with the additional Puzzle Maker restrictions (except, of course, the small level size and item count), but this distinction is not a primary focus of this work.

4 Portal with Emancipation Grills is Weakly NP-complete

In this section, we prove that Portal with portals and Emancipation Grills is weakly NP-hard by reduction from Subset Sum [8], which is defined like so.

▶ **Problem 2.** Subset Sum

Input: A set of integers $A = \{a_1, a_2, \ldots, a_n\}$, and a target value t.
Output: Whether there exists a subset $\{s_1, s_2, \ldots, s_m\} \subseteq A$ such that

$$\sum_{i=1}^{m} s_i = t.$$

The reduction involves representing the integers in A as distances which are translated into the avatar's velocity. More explicitly, the input A will be constructed from long holes the avatar can fall down, and the target will be encoded in a distance the avatar must launch themselves after falling. For the next theorem, it is necessary to allow the terminal velocity v_{max} to be specified as input to the problem (so it can scale with the level size).

▶ **Theorem 3.** Portal *with portals, long fall, Emacipation Grills, and generalized terminal velocity is weakly NP-hard.*

Proof. Refer to Figure 1. The elements of A are represented by a series of wells, each of width c and depth $b \cdot a_i$ as measured from the ceiling directly above it. Here $a_i \in A$ is the number to be encoded, $b = 2 \cdot c \cdot n^2 \cdot t$ is a large number, c is a large constant expansion factor greater than the height of the avatar plus the height she can jump, n is the number of elements in A, and t is the target value of the Subset Sum instance. The bottom of each well is a portalable surface, and the ceiling above each well is also a portalable surface. Each well also has an Emancipation Grill a distance c from the ceiling. This construction allows

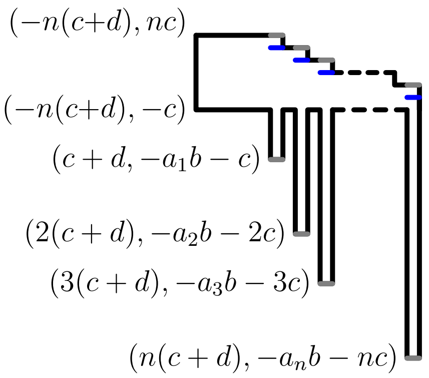


Figure 1 A cross-section of the element selection gadget, where $b = 2 \cdot c \cdot n^2 \cdot t$. Grey lines are portalable surfaces and blue lines are Emancipation Grills.

the avatar to shoot a portal to the bottom of the well they are falling into, and to a ceiling tile of another well, selecting the next number.

If the Subset Sum instance has a solution S, we can fall through the wells of depth $b \cdot a_i$ for each $a_i \in S$ in order, without touching any walls, for a total fall distance of $b \cdot t$. After such a fall, we reach a "target" velocity $v_t = g\sqrt{2bt}$.

We cannot allow the avatar to select the same element more than once. The Emancipation Grills below each portalable ceiling serve to remove the portal from the ceiling of the well into which the avatar is currently falling, and to prevent sending a portal up to that same ceiling tile. The stair-stepped ceiling allow the player to see the ceilings of all of the wells with index greater than the one they are currently at, but prevents them from seeing the portalable surface of the wells with a lower index. This construction ensures that the player can select each element only once using portals. The enforced order of choosing does not matter when solving Subset Sum.

We also need to prevent the avatar from moving horizontally from one well to another while falling. The avatar can move horizontally (via user input) up to a small fixed acceleration α_h. To successfully fall through one well of width c and depth at least b below the ground without hitting its side walls, the avatar's horizontal velocity v_h over vertical velocity v_v must be at most c/b. Also, after falling at least b, we must have vertical velocity $v_v \geq \sqrt{2b}$. The fall through the top part of the next well, of depth less than $(n+1)c$, will thus take $s \leq (n+1)c/v_v$ time. During this fall, the avatar can add at most $\alpha_h s \leq \alpha_h(n+1)c/v_v$ to horizontal velocity. Thus, during this fall, the avatar can travel horizontally by at most

$$\begin{aligned}
v_h s + \frac{1}{2}\alpha_h s^2 &\leq \frac{v_v c}{b}\frac{(n+1)c}{v_v} + \frac{1}{2}\alpha_h\left(\frac{(n+1)c}{v_v}\right)^2 \\
&= (n+1)\frac{c^2}{b} + \frac{\alpha_h(n+1)^2c^2}{2v_v^2} \\
&\leq (n+1)\frac{c^2}{b} + \frac{\alpha_h(n+1)^2c^2}{b} \\
&= \left(n+1+\alpha_h(n+1)^2\right)\frac{c^2}{b} \\
&= \left(n+1+\alpha_h(n+1)^2\right)\frac{c}{2n^2t} \\
&= \frac{\alpha_h}{2t}c + O(1/n).
\end{aligned}$$

Setting d to be at least this value (and at least c), we prevent the player from reaching an adjacent well by horizontal travel.

We must also ensure that the player actually able to target the portable surfaces to select the elements of A. To do so, we set the time step δ to be less than $c/(10v_t)$ where v_t is the target velocity. This ensures that the player will have at least 9 time steps to target while falling c units, in particular while passing between the heights of each target surface for A and its emancipation grid.

The verification gadget (not drawn) involves two main pieces: a single portalable surface on a vertical wall ("launch point") and a $c \times c$ horizontal floor ("target platform") for the player to reach. We place the launch point so it can always be shot from the region above the wells. Relative to the launch point, the target platform is placed $g/2$ units below and at a horizontal distance of v_t in front, so that leaving the portalable surface with the target velocity v_t will cause the player to reach the target platform in 1 unit of time. The size of the target platform is much smaller than the difference ($\geq \sqrt{b} \geq n$) if the target value t differed by 1. If the player enters the final portal with horizontal velocity v_h and vertical velocity v_v, satisfying $v_h/v_v \leq c/b$ as proved above, then the avatar launches with horizontal velocity v_v and vertical velocity $v_h \leq v_v c/b$. This vertical velocity is insufficient to affect the landing position by as much as changing t by 1. Similarly, user input during the 1 unit of time has minimal effect on the horizontal velocity. ◀

All of the game elements needed for this construction can be placed in the Puzzle Maker. However, this reduction would not be constructible because maps in the Puzzle Maker appear to be specified in terms of voxels. Because Subset Sum is only weakly NP-hard [8], we need the values of the elements of A to be exponential in n. Thus we need to describe the map in terms of coordinates specifying the polygons making up the map, whereas the Puzzle Maker specifies each voxel in the map.

▶ **Theorem 4.** Portal *with portals, long fall, emancipation grills, and generalized terminal velocity can be solved in pseudopolynomial time.*

Proof. We construct a state-space graph of the Portal level. Each vertex represents a tuple comprised of the avatar's position vector within the level, the avatar's velocity vector (limited by the terminal velocity v_{max}), the avatar's orientation, the position vector of the blue portal, and the position vector of the orange portal. The vertices are connected with directed edges encoding the state transitions caused by user input. Finally, for each edge that would represent traversal through an emancipation grid, we replace it by an edge that maps to the same state of the avatar but with both portal locations removed. We can then search for a path from the initial game state to any of the winning game states in time polynomial in the size of the graph. ◀

5 Portal with Turrets is NP-hard

In this section we prove Portal with turrets is NP-hard, and show that our method can be generalized to prove that many 3D platform games with enemies are NP-hard. Although enemies in a game can provide interesting and complex interactions, we can pull out a few simple properties that will allow them to be used as gadgets to reduce solving a game from 3-SAT, defined like so.

▶ **Problem 5.** 3-SAT
Input: A 3-CNF boolean formula f.
Output: Whether there exists a satisfying assignment for f.

This proof follows the architecture laid out in [1]:

1. The enemy must be able to prevent the player from traversing a specific region of the map; call this the *blocked region*.
2. The player avatar must be able to enter an area of the map, which is path-disconnected from the blocked region, but from which the player can remove the enemy in the blocked region.
3. The level must contain long falls.

We further assume that the behavior of the enemies is local, meaning an interaction with one enemy will not effect the behavior of another enemy if they are sufficiently far away. In many games one must also be careful about ammo and any damage the player may incur while interacting with the gadget, because these quantities will scale with the number of literals. Here long falls serve only in the construction of one-way gadgets, and can of course be replaced by some equivalent game mechanic. Similarly, a 2D game with these elements and an appropriate crossover gadget should also be NP-hard. The following is a construction proving Portal with Turrets is NP-hard using this technique. Note that these gadgets can be constructed in the Portal 2 Puzzle Maker.

5.1 Literal

Each literal is encoded with a hallway with three turrents placed in a raised section, illustrated in Figure 2. The hallway must be traversed by the player, starting from "Traverse In", ending at "Traverse Out". If the turrets are active, they will kill the avatar before the avatar can cross the hallway or reach the turrets. The literal is true if the turrets are deactivated or removed, and false if they are active. The "Unlock In" and "Unlock Out" pathways allow for the player avatar to destroy the turrets from behind, deactivating them and counting as a true assignment of the literal.

5.2 Variable

The variable gadget consists of a hallway that splits into two separate paths. Each hallway starts and ends with a one-way gadget constructed with a long fall. This construction forces the avatar to commit to one of the two paths. The hallways connect the "Unlock In" and "Unlock Out" paths of the literals corresponding to a particular variable. Furthermore, one path connects all of the true literals, the other connects all of the false literals.

5.3 Clause Gadget

Each clause gadget is implemented with three hallways in parallel. A section of each hallway is the "Traverse In" through the "Traverse Out" corresponding to a literal. The avatar can progress from one end of the clause to the other if any of the literals is true (and thus passable). Furthermore, each of the clause gadgets is connected in series. Figures 3 and 4 illustrate a full clause gadget.

▶ **Theorem 6.** PORTAL *with Turrets and long falls is NP-hard.*

Proof. Given an instance of a 3SAT problem, we can translate it into a Portal with Turrets map using the above gadgets. This map is solvable if and only if the corresponding 3SAT problem is solvable. ◀

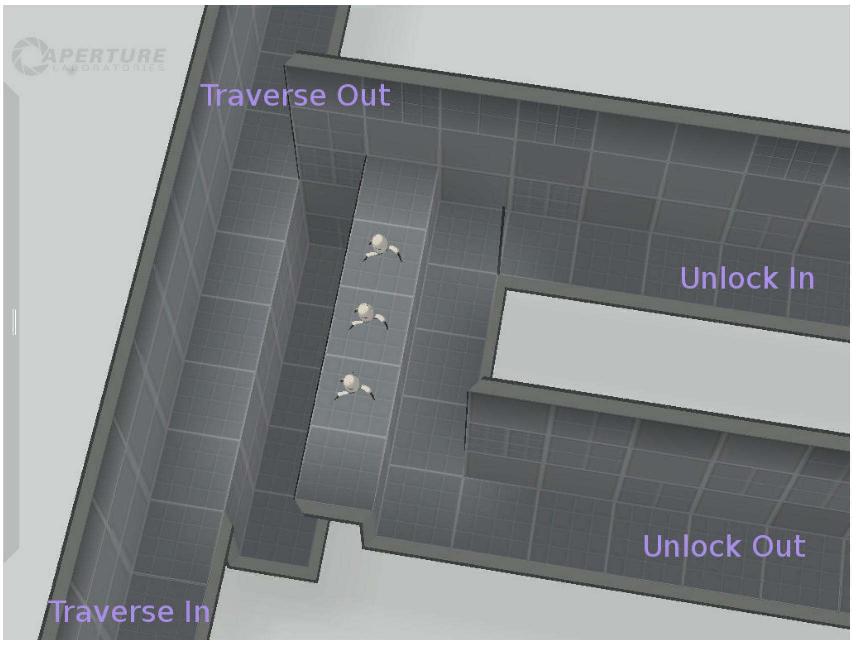


Figure 2 An example of a (currently) false literal constructed with Turrets. Labels added over the screenshot denote

It is tempting to claim NP-completeness because disabling the turrets need only be performed once per turret and thus seems to have a monotonically changing state. However, the turrets themselves are physical objects that can be picked up and moved around. Their relocation add an exponential amount of state to the level. Further, if they can be jumped on top of or used to block the player in a constrained hallway, they may conceivably cause the level to be PSPACE-complete in the same way boxes can add significant complexity to a game.

5.4 Application to Other Games

While the framework we have presented is shown using the gameplay elements of Portal, similar elements to those we have used show up in other video games. Hence, our framework can be generalized to show hardness of other games. In this section we note several common features of games which would allow for an equivalent to the turret “guarding unit” in Portal. We list examples of notable games which fit the criteria. We give ideas how to use our framework to prove hardness results for these games, but it is important to note that game-specific implementation details will need to be taken into account for any hardness proof.

The first examples are games that include player controlled weapons with fixed positions, such as stationary turrets or gun emplacements. The immovable turrets should be placed at the unlock points of the literal gadget, so that they only allow the player to shoot the one desired blocking unit. Examples in contemporary video games include the Emplacement Gun in Half-Life 2, the Type-26 ASG in Half-Life, and the Anti-Infantry Stationary Guns in Halo 1 through 4.

Another set of examples are games which include a pair of ranged weapons, where one is more powerful than the other, but has shorter range. In place of the turrets in the Portal

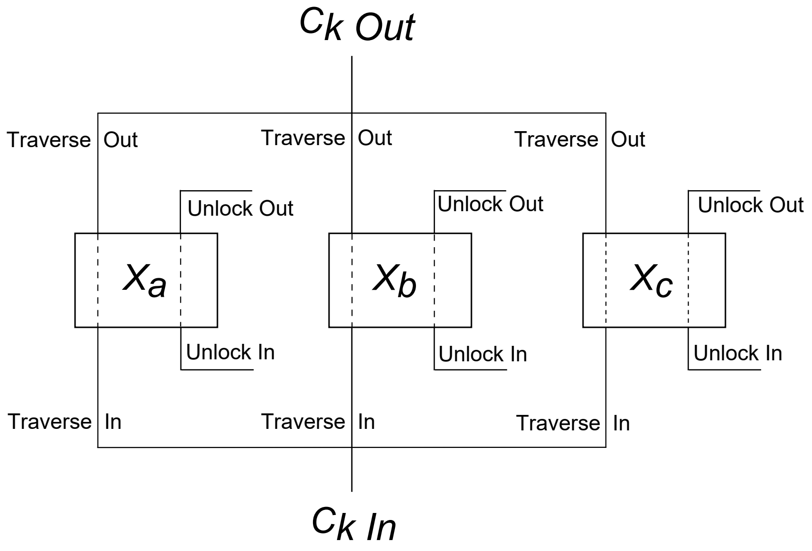


Figure 3 A diagram of clause C_k which contains variables x_a, x_b, and x_c.

literal gadgets, we place an enemy unit equipped with the short range weapon, and give the player avatar the long range weapon. We place the blocked region such that it is in range and line of sight of the player while standing in the unlock region of the literal gadget. Additionally, we place the player such that they are not in range of the enemy's weapon. Thus the player can kill the enemy from the unlock area. Suppose further that the blocked region is built in such a way that the player can only pass through it by moving within range of the enemy. One way of doing this would be to build it with tight turns. The result would be an equivalent implementation of the variable and clause gadgets from our Portal constructions. Note that a special case involves melee enemies. This construction applies to Doom, the Elder Scrolls III–V, Fallout 3 and 4, Grand Theft Auto 3–5, Left 4 Dead 1 and 2, the Mass Effect series, the Deus Ex series, the Metal Gear Solid series, the Resident Evil series, and many others. The complementary case occurs when the player has the short ranged, but more powerful weapon and the enemy has the weaker, long ranged weapon. Here the unlock region provides close proximity to the enemy unit but the locked region involves a significant region within line of sight and range of the enemy but is outside of the player's weapon's range. Although most games where this construction is applicable will also fall into the prior case, examples exist where the player has limited attacks, such as in the Spyro series.

A third case is where the environment impacts the effectiveness of attacks. For example, certain barriers might block projectile weapons but not magic spells. Skills that can shoot above or around barriers like this show up with Thunderstorm in Diablo II, Firestorm in Guild Wars, and Psi-storm in StarCraft. Another common effect is a location based bonus, for example the elevated-ground bonus in XCOM. Unfortunately these games lack a long-fall, and thus require the construction of a one-way gadget if one wishes to prove hardness.

While we have so far only covered NP-hardness, we conjecture that these games are significantly harder. Assuming simple AI and perfect information, many are likely PSPACE-complete; however, when all of the details are taken into consideration, EXPTIME or

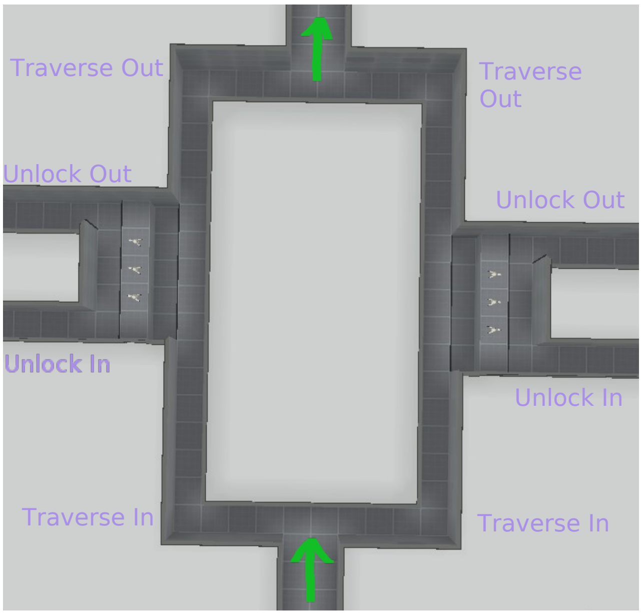


Figure 4 An example of a clause gadget with two literals.

NEXPTIME seem more likely. Proving such results will require development of more sophisticated mathematical machinery.

6 Portal with Timed Door Buttons is NP-hard

We provide a new metatheorem related to Forisek's Metatheorem 2 [7] and Viglietta's Metatheorem 1 [18].

▶ **Metatheorem 7.** *A platform game with doors controlled by timed switches is NP-hard.*

Proof. We will prove hardness by reducing from finding Hamiltonian cycles in grid graphs [10]. Every vertex of the graph will be represented by a room with a timed switch in the middle. These rooms will be laid out in a grid with hallways in-between. The rooms are small in comparison to the hallways. In particular, the time it takes to press a timed button and travel across a room is δ and the time it takes to traverse a hallway is $\alpha > n \cdot \delta$ where n is the number of nodes in the graph. This property ensures the error from turning versus going straight through a room won't matter in comparison to traveling from node to node. All of the timed switches will be connected to a series of closed doors blocking the exit hallway connected to the start node. The timers will be set, such that the doors will close again after $(\alpha + \delta) \cdot (t + 1) + \varepsilon$ where ε is the time it takes to move from the switch at the start node through the open doors to the exit. The exit is thus only reachable if all of the timed switches are simultaneously active. Because we can make α much larger than ε, we can ensure that there is only time to visit every switch exactly once and then pass through before any of the doors revert. ◀

▶ **Corollary 8.** *A Portal level with only timed door buttons is NP-hard.*

A screenshot of an example map for Corollary 8 is given in Figure 5. Because the Portal 2 Workshop does not allow additional doors, the example uses collapsible stairs. We note that anything which will prevent the player from passing unless currently activated by a timed

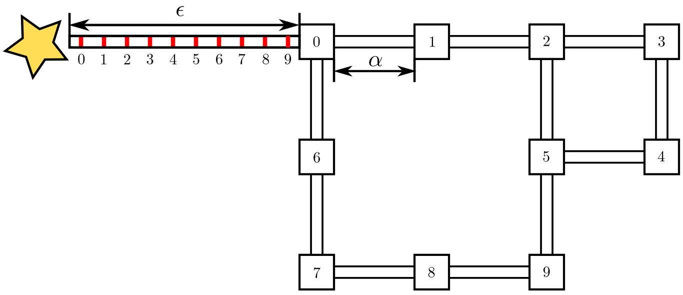


Figure 5 An example of a map forcing the player to find a Hamiltonian cycle in a grid graph.

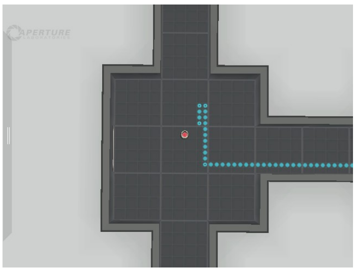


Figure 6 Close-up of a node in the grid graph.

button will suffice. Moving platforms and Laser Fields are other examples. Unfortunately, the Puzzle Maker does not allow the timer length to be specified, which is a needed generalization for the reduction and available in the Hammer editor.

7 Portal with High-Energy Pellets and Portals is NP-hard

In Portal, the High-Energy Pellet, HEP, is an object which moves in a straight line until it encounters another object. HEPs move faster than the player avatar and if they collide with the player avatar, the avatar is killed. If a HEP encounters another wall or object, it will bounce off of that object with equal angle of incidence and reflection. In Portal, some HEPs have a finite lifespan, which is reset when the HEP passes through a portal, and others have an unbounded lifespan. A HEP launcher emits a HEP normal to the surface it is placed upon. These are launched when the HEP launcher is activated or when the previous HEP emitted has been destroyed. A HEP catcher is another device that is activated if it is ever hit by a HEP. When activated this device can activate other objects, such as doors or moving platforms. HEP's are only seen in the first Portal game and are not present in the Portal 2 Puzzle Maker.

▶ **Theorem 9.** PORTAL *with Portals, High-Energy Pellets, HEP launchers, HEP catchers, and doors controlled by HEP catchers is NP-hard.*

Proof. We will reduce from finding Hamiltonian cycles in grid graphs [10]; refer to Figure 7. For this construction, we will need a gadget to ensure the avatar traverses every represented node, as well as a timing element. Each node in the graph will be represented by a room that contains a HEP launcher and a HEP catcher. They are positioned near the ceiling, each facing a portalable surface. The HEP catcher is connected to a closed door preventing the avatar from reaching the exit. The rooms are small in comparison to the hallways. In particular, the time it takes to shoot a portal, wait for it to enter the HEP Catcher, and

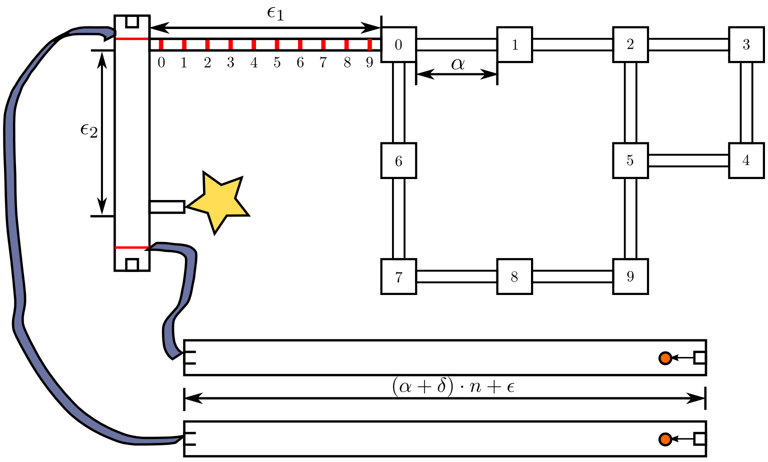


Figure 7 An example level for the HEP reduction. Not drawn to scale.

travel across a room is δ and the time it takes to traverse a hallway is $\alpha > n \cdot \delta$ where n is the number of nodes in the graph. This property ensures the error from turning versus going straight through a room won't matter in comparison to traveling from node to node.

The timer will contain two elements. First, we will arrange for a hallway with two exits and a HEP launcher behind a door on one end. The hallway is long enough so it is impossible for the avatar to traverse the hallway when the door is open. Call this component the *time verifier*. In another area, we have a HEP launcher and a HEP catcher on opposite ends of a hallway that is inaccessible to the avatar. The catcher in this section will open the door in the time verifier. This construction ensures that the player can only pass through the time verifier if they enter it before a certain point after starting. To complete the proof, we set the timer equal to $(\alpha + \delta) \cdot n + \varepsilon_1 + \varepsilon_2$ where ε_1 is the minimum time needed for the avatar to traverse the hallway with doors, ε_2 is the minimum time needed for the avatar to traverse the time verifier, α is the minimum time it takes for the player to move to an adjacent room and change the trajectory of the HEP, and $n + 1$ is the number of HEP catchers in the level. Thus concludes our reduction from the Hamiltonian cycle problem in grid graphs. ◀

The HEP Catchers are only able to be activated once, so one may be tempted to claim this problem is in NP. This is not necessarily the case because navigating around HEP particles with more complicated trajectories might require long paths or wait times. The PSPACE-hardness of motion planning with periodic obstacles [14] suggests the natural class for this problem is actually PSPACE-complete.

8 Portal is PSPACE-complete

In this section we give a new metatheorem for games with doors and switches, in the same vein as the metatheorems in [7], [18], and [17]. We use this metatheorem to give proofs of PSPACE-completeness of Portal with various game elements, included here and in Section 9. All of the gadgets in this section can be created in the Portal 2 Puzzle Maker.

The proofs in this section revolve around constructing game mechanics which implement a switch: the construction can be in one of two states, and the state is controllable by the player. When the avatar is near the switch, it can be freely set to either state. Each state has a set of doors which are open and others which are closed when the switch is in that state. A switch is very similar to a button in that it controls whether doors are open or closed, and the player has the option of interacting with it. The key difference is that buttons can be pressed multiple times to open or close its associated doors, and cannot necessarily be ‘unpressed’ to undo the action. We show that a game with switches and doors is PSPACE-complete, using similar techniques to [17].

In what follows we will use the nondeterministic constraint logic framework [9], wherein the state of a nondeterministic machine is encoded by a graph called a *constraint graph*. The state is updated by changing the orientation of the edges in such a way that constraints stored on the vertices are satisfied.

Formally, an constraint graph is an undirected simple graph $G = (V, E)$ with an assignment of nonnegative integers to the edges $w : E \to \mathbb{Z}^+$, referred to as *weights*, and an assignment of integers to the vertices $c : V \to \mathbb{Z}$, referred to as *constraints*. Each edge has an orientation $p : E \to \{+1, -1\}$. A constraint graph is fully specified by the tuple $\mathcal{G} = (G, w, c, p)$. The edge orientation p induces a directed graph $D_{G,p}$. Let $v \in V$ be a vertex of G. Its *in-neighborhood*

$$N^-(v, p) = \{w \mid (v, w) \in A\}$$

is the set of vertices of $D_{G,p} = (V, A)$ with an arc oriented towards it. The constraint graph $\mathcal{G}$ is *valid* if, for all $y \in V$, $\sum_{x \in N^-(y,p)} w((x, y)) \geq c(x)$. The state of a constraint graph can be changed by selecting an edge and multiplying its orientation by -1, such that the resulting constraint graph is valid. We say that we have *flipped* the edge.

A vertex v in a constraint graph with three incident edges x, y, o can implement an AND gate by setting $c(v) = 2$, $w(x) = w(y) = 1$, and $w(o) = 2$. Clearly, the edge o can only point away from v if both x and y are pointing towards v. In a similar fashion, we can implement an OR gate by setting $w(v) = 2$, $w(x) = w(y) = w(o) = 2$. A constraint graph where all vertices are AND or OR vertices is called an *AND/OR constraint graph*. The following decision problem about constraint graphs is PSPACE-complete.

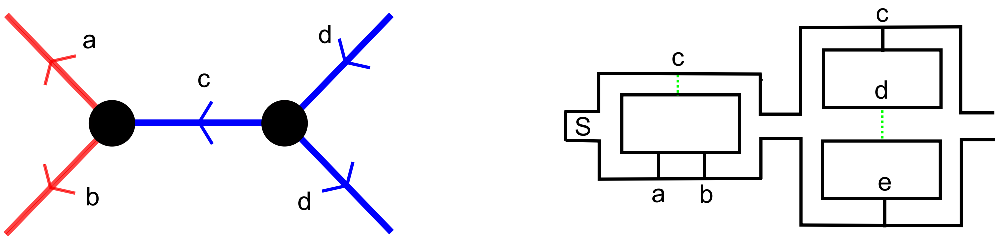
▶ **Problem 10.** Nondeterministic Constraint Logic

Input: An AND/OR constraint logic graph $\mathcal{G} = ((V, E), w, c, p)$, and a target edge $i, j \in E$.

Output: Whether there exists a constraint graph $\mathcal{G}' = ((V, E), w, c, p')$ such that $p'(\{i, j\}) = -p(\{i, j\})$, and which can be obtained from $\mathcal{G}$ by a sequence of valid edge flips.

▶ **Metatheorem 11.** *Games with doors that can be controlled by a single switch and switches that can control at least six doors are PSPACE-complete.*

Proof. We prove this by reduction from Nondeterministic Constraint Logic. The edges of the consistency graph are represented by a single switch whose state represents the edge orientation. Connected to each switch is a *consistency check gadget*. This gadget consists of a series of hallways that checks that the state of the two vertices adjacent to the simulated edge are in a valid configuration and thus that the update made to the graph was valid. Each edge switch is connected to doors in up to six consistency checks, two for itself and four for the adjacent edges. For an AND vertex, the weight-two edge is given by the door with the single hallway, and the weight one edges connect to the two doors in the other hallway. For an OR vertex we have a hallway that splits in three, each with one node. An example is given in Figure 8. Each switch thus connects to five doors. All of the edge gadgets, with their constraint checks, are connected together. This construction allows the player to change the direction of any edge they choose. However, to get back to the main hallway connecting the gadgets, the graph must be left in a valid state. Off the main hallway



(a) Section of a constraint logic graph being simulated. Blue edges are weight 2 and red edges are weight 1.

(b) Gadget simulating edge c in the constraint logic graph. Green dotted lines are open doors.

Figure 8 Example of an edge gadget built from switches and doors.

there is a final exit connected to the target location, but blocked by a door connected to the target edge. If the player is able to flip the edge by visiting the edge gadget, flip the switch which opens the exit door, and return through the graph consistency check, then the avatar can reach the target location. ◀

▶ **Theorem 12.** Portal *with any subset of long falls, portals, Weighted Storage Cubes, doors, Heavy Duty Super Buttons, lasers, laser relays, gravity beams, turrets, timed buttons, and moving platforms is in PSPACE.*

Proof. Portal levels do not increase in size and the walls and floors have a fixed geometry. Assuming all velocities are polynomially bounded, all gameplay elements have a polynomial amount of state which describes them. For example the position and velocity of the avatar or a HEP; whether a door is open or closed; and the time on a button timer. The number of gameplay elements remains bounded while playing. Most gameplay elements cannot be added while playing, and items like the HEP launcher and cube suppliers only produce another copy when the prior one has been destroyed. We only need a polynomial amount of space to describe the state of a game of Portal at any given point in time. Thus one can nondeterministically search the state space for any solutions to the Portal problem, putting it in NPSPACE. Thus by Savitch's Theorem [13] the problem is in PSPACE. ◀

▶ **Theorem 13.** Portal *with Weighted Storage Cubes, doors, and Heavy Duty Super Buttons is PSPACE-complete.*

Proof. We will construct switches and doors out of doors, Weighted Storage Cubes, and Heavy Duty Super Buttons. Then, we invoke Metatheorem 11 to complete the proof. A switch is constructed out of a room with a single cube and two buttons as in Figure 9. Which of the buttons being pressed by the cube dictates the state of the switch. Each button is connected to the corresponding doors which should open when the switch is in that state. To ensure the switch is always in a valid state, we put an additional door in the only entrance to the room. This door is only open if at least one of the two buttons is depressed. Furthermore, this construction prevents the cube from being removed from the room to be used elsewhere. As long as there are no extra cubes in the level, the room must be left in exactly one of the two valid switch states for the avatar to exit the room. We now apply our doors and simulated switches as in Metatheorem 11 completing the hardness proof. Theorem 12 implies inclusion in PSPACE. ◀

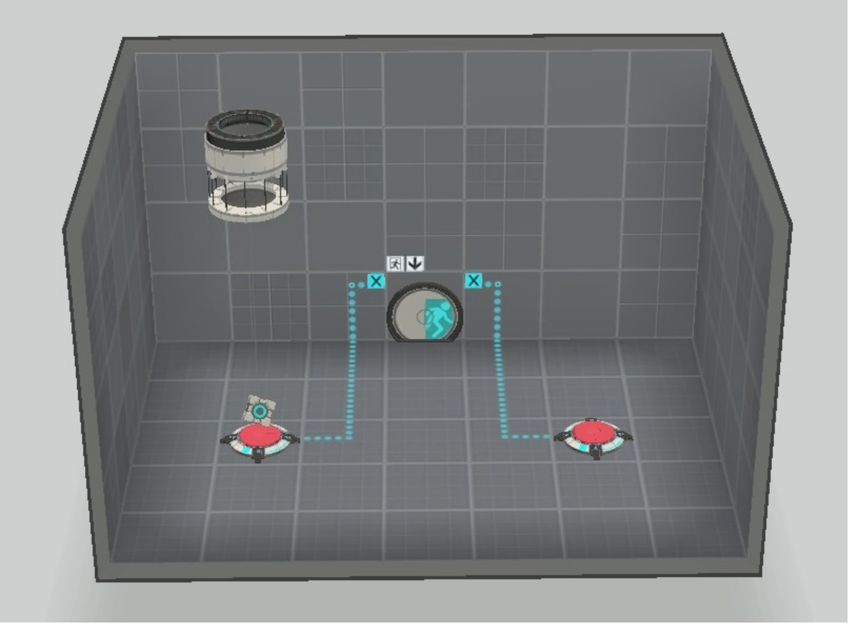


Figure 9 An example of a single switch implemented with cubes, doors, and buttons. The door will only open if at least one of the buttons is pressed.

9 Additional Applications of NCL Construction

In this section we use Theorem 11 to prove additional results about Portal.

▶ **Theorem 14.** PORTAL *with lasers, relays, portals, and moving platforms is PSPACE-complete.*

Proof. We will construct doors and switches out of lasers, relays, and moving platforms allowing us to use Metatheorem 11. In Portal 2, the avatar is not able to cross through an active laser. Because lasers can be blocked by the moving platforms game element, a door can be constructed by placing a moving platform and laser at one end of a small hallway. If the moving platform is in front of the laser, the gadget is in the unlocked state. If the moving platform is to the side, then the player cannot pass through the hallway and it is in the locked state. Moving platforms can be controlled by laser relays and will switch position based on whether the laser relay is active. Lasers can be directed to selectively activate laser relays with portals, so we have a mechanism to lock or unlock the doors.

As it stands, once a new portal is created the previously opened door will revert to its previous state. To prove PSPACE-hardness, we need to make these changes persist. To do so, we introduce a memory latch gadget, shown in Figures 10 and 11. When the relay in this gadget is activated for a sufficiently long period of time, the platform will move out of the way and the laser will keep the relay active. If the relay has been blocked for enough time, the platform moves back and blocks the laser. Thus, the state of the gadget persists.

The last construction is the switch, which we build out of two groups of lasers, moving platforms, and laser relays, as well as a memory latch. The player has the ability to change the state of the memory latch. We interpret the state of the memory latch as the state of the switch. When active, one of the relays in the latch moves a platform out of the way of one of the lasers, activating the corresponding relays and opening the set of doors to which they are connected. Another relay in the latch moves the second moving platform into the path of the second laser, deactivating its corresponding laser relays and the doors they control. Likewise, deactivating the memory latch causes both moving platforms to revert

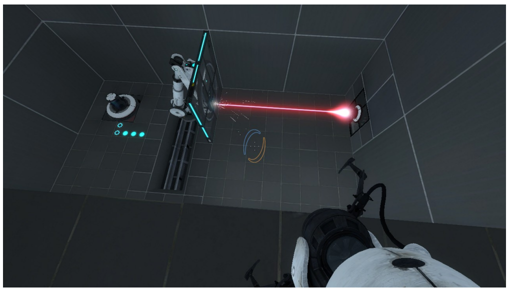


Figure 10 A memory latch in the off state.

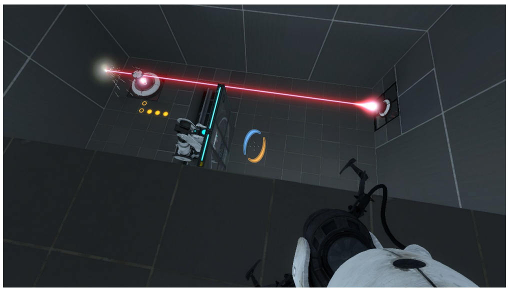


Figure 11 A memory latch in the on state.

to their original positions, blocking the first laser and letting the second through. We have now successfully constructed doors and switches, so by Metatheorem 11 and Theorem 12, PSPACE-completeness follows. ◀

Note that in the proof of the preceding theorem, laser catchers could be used in place of laser relays, although the relays have the convenient property that they each need only be connected to a single moving platform. It is also possible that the proof could be adapted to use a single Reflection Cube instead of portals. Additional care would be required with respect to the construction of the door, and it would need to be the case that lasers from multiple directions blocked the avatar. Emancipation Grills or long falls with the moving platforms would simplify this particular door construction.

The game elements in the following corollary are a superset of those used in Theorem 13, so this result follows trivially. However, we prove it by using a construction similar to that in Theorem 14, as we feel that the gadgets involved are interesting. We also note that the proof only uses Heavy Duty Super Buttons placed on vertical surfaces, whereas Theorem 13 relies on their placement on the floor.

▶ **Corollary 15.** PORTAL *with gravity beams, cubes, Heavy Duty Super Buttons, and long fall is PSPACE-complete.*

Proof. When active, a gravity beam causes objects which fit inside its diameter to be pushed or pulled in line with the gravity beam emitter. Objects in the gravity beam ignore the normal pull of gravity, and thus float along their course. We construct a simple door by placing a gravity beam so that it can carry the player avatar across a pit large enough that the avatar would otherwise be unable to traverse. We hook the gravity beam emitter up to a button allowing it to be turned on and off, unlocking and locking the door.

If we wish to only use buttons placed on vertical surfaces, we are now faced with the problem of making changes to doors persist once the avatar stops holding a cube next to the button. To solve this problem, we construct a memory latch as in Theorem 14. If a weighted cube button is placed in the path of a gravity beam, a weighted cube caught in the beam can depress the button as in Figure 13. A cube on the floor near a gravity beam, as in Figure 12 will be picked up by the beam. Weighted cube buttons can activate and deactivate the same mechanics as laser catchers, including gravity beam emitters. Figures 12 and 13 demonstrate a memory latch in the off and on positions, respectively. We also note that gravity beams are blocked by moving platforms, just like lasers. At this point, we have the properties we need from the laser, laser catcher, and moving platform. We also note that the player can pick up and remove cubes from the beam, meaning that portals are not needed. ◀

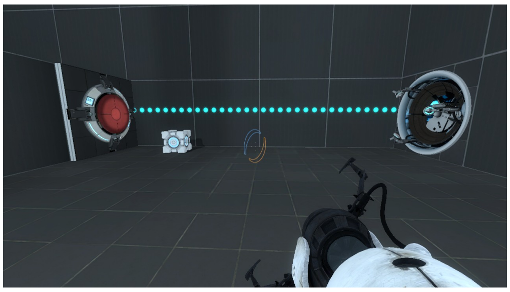


Figure 12 A memory latch in the off state.

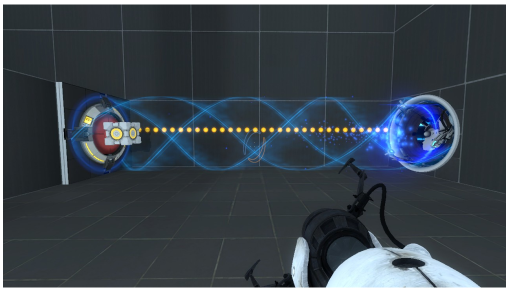


Figure 13 A memory latch in the on state.

10 Conclusion

In this paper we proved a number of hardness results about the video game Portal. In Sections 4 through 7 we have identified several game elements that, when accounted for, give Portal sufficient flexibility so as to encode instances of NP-hard problems. Furthermore, in Section 8 we gave a new metatheorem and use it to prove that certain additional game elements, such as lasers, relays and moving platforms, make the game PSPACE-complete. The unique game mechanics of Portal provided us with a beautiful and unique playground in which to implement the gadgets involved in the hardness proofs. Indeed, our work shows how clause, literal, and variable gadgets inspired by the work of Aloupis et al. [1] can be implemented in a 3D video game. While our results about Portal itself will be of interest to game and puzzle enthusiasts, what we consider most interesting are the techniques we utilized to obtain them. Adding new, simple gadgets to this collection of abstractions gives us powerful new tools with which to attack future problems. In Section 5.4 we identified several other video games that our techniques can be generalized to. We also believe the decomposition of games into individual mechanics will be an important tactic for understanding games of increasing complexity. Metatheorems 7 and 11 are new metatheorems for platform games. We hope that our work is useful as a stepping stone towards more metatheorems of this type. Additionally, we hope the study of motion planning in environments with dynamic topologies leads to new insights in this area.

10.1 Open Questions

This work leads to many open questions to pursue in future research. In Portal, we leave many hardness gaps and a number of mechanics unexplored. We are particularly curious about Portal with only portals, and Portal with only cubes. The removal of Emancipation Fields from our proofs would be very satisfying. The other major introduction in Portal 2 that we have not covered is co-op mode. If the players are free to communicate and have perfect information of the map, this feature should not add to the complexity of the game. However, the game seems designed with limited communication in mind and thus an imperfect-information model seems reasonable. Although perfect-information team games tend to reduce down to one- or two-player games, it has been shown that when the players have imperfect information the problem can become significantly harder. In particular, a cooperative game with imperfect information can be 2EXPTIME-complete [12].

More than the results themselves, one would hope to use these techniques to show hardness for other problems. Many other games use movable blocks, timed door buttons, and stationary turrets and may have hardness results that immediately follow. Some techniques like encoding numbers in velocities might be transferable. It would be good to generalize some of these into metatheorems which cover a larger variety of games.

References

1 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (NP-)hard. In *Proceedings of the 7th International Conference on Fun with Algorithms (FUN 2014)*, Lipari Island, Italy, July 1–3 2014.

2 Eric Caoili. Portal 2 has sold over 4m copies. http://www.gamasutra.com/view/news/169967/Portal_2_has_sold_over_4M_copies.php. Accessed: 2015-08-21.

3 G. Cormode. The hardness of the Lemmings game, or oh no, more NP-completeness proofs. In *Proceedings of Third International Conference on Fun with Algorithms*, pages 65–76, 2004. URL: ../papers/cormodelemmings.pdf.

4 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.

5 Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. In *Proceedings of the 9th International Computing and Combinatorics Conference (COCOON 2003)*, pages 351–363, Big Sky, Montana, July 25–28 2003.

6 Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder-/easier than we thought. In *Proceedings of the 8th International Conference on Fun with Algorithms*, La Maddalena, Italy, June 2016.

7 Michal Forisek. Computational complexity of two-dimensional platform games. In *Proceedings International Conference on Fun with Algorithms (FUN 2010)*, pages 214–227, 2010.

8 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.

9 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* A. K. Peters, Ltd., Natick, MA, USA, 2009.

10 Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982.

11 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008. URL: http://dblp.uni-trier.de/db/journals/icga/icga31.html#KendallPS08.

12 Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7):957–992, 2001.

13 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *J. Comput. Syst. Sci.*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.

14 Klaus Sutner and Wolfgang Maass. Motion planning among time dependent obstacles. *Acta Informatica*, 26(1-2):93–122, 1988.

15 Valve Developer Community. P2C. https://developer.valvesoftware.com/wiki/P2C, 2013.

16 Valve Developer Community. Valve map format. https://developer.valvesoftware.com/wiki/VMF_documentation, 2016.

17 Tom C. van der Zanden and Hans L. Bodlaender. Pspace-completeness of bloxorz and of games with 2-buttons. In Vangelis Th. Paschos and Peter Widmayer, editors, *Algorithms and Complexity - 9th International Conference, CIAC 2015, Paris, France, May 20-22, 2015. Proceedings*, volume 9079 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2015. doi:10.1007/978-3-319-18173-8_30.

18 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

19 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

20 Wesley Yin-Poole. Portal sells nearly four million. http://www.eurogamer.net/articles/2011-04-20-portal-sells-nearly-four-million. Accessed: 2015-08-21.

Faster Evaluation of Subtraction Games

David Eppstein[1]
Computer Science Department, University of California, Irvine
eppstein@uci.edu

Abstract

Subtraction games are played with one or more heaps of tokens, with players taking turns removing from a single heap a number of tokens belonging to a specified *subtraction set*; the last player to move wins. We describe how to compute the set of winning heap sizes in single-heap subtraction games (for an input consisting of the subtraction set and maximum heap size n), in time $\tilde{O}(n)$, where the $\tilde{O}$ elides logarithmic factors. For multi-heap games, the optimal game play is determined by the *nim-value* of each heap; we describe how to compute the nim-values of all heaps of size up to n in time $\tilde{O}(mn)$, where m is the maximum nim-value occurring among these heap sizes. These time bounds improve naive dynamic programming algorithms with time $O(n|S|)$, because $m \leq |S|$ for all such games. We apply these results to the game of subtract-a-square, whose set of winning positions is a maximal square-difference-free set of a type studied in number theory in connection with the Furstenberg–Sárközy theorem. We provide experimental evidence that, for this game, the set of winning positions has a density comparable to that of the densest known square-difference-free sets, and has a modular structure related to the known constructions for these dense sets. Additionally, this game's nim-values are (experimentally) significantly smaller than the size of its subtraction set, implying that our algorithm achieves a polynomial speedup over dynamic programming.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases subtraction games, Sprague–Grundy theory, nim-values

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.20

1 Introduction

Subtraction games were made famous by the French film *L'Année dernière à Marienbad* (1961), which showed repeated scenes of two men playing Nim. A subtraction game is played by two players, with some heaps of game tokens (such as coins, stones, or, in the film, matchsticks) between them. On each turn, a player may take away a number of tokens from a single heap. The tokens removed in each turn are discarded, and play continues until all the tokens are gone. Under the *normal winning convention*, the last player to move is the winner [2]. In Nim, any number of tokens may be removed in a turn. This game has a simple analysis according to which it is a winning move to make the bitwise exclusive-or of the binary representations of the heap sizes become zero. If this bitwise exclusive-or is already zero, the player who just moved already has a winning position [4]. However, other subtraction games require the number of removed tokens to belong to a predetermined set of numbers, the *subtraction set* of the game. Different subtraction sets lead to different games with different strategies.[2]

[1] Supported in part by NSF grants CCF-1618301 and CCF-1616248.

[2] Golomb [7] has considered an even more general class of games, in which the subtraction set specifies combinations of numbers of tokens that may be simultaneously removed from each pile.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 20; pp. 20:1–20:12
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

All subtraction games are *impartial*, meaning that the choice of moves on each turn does not depend on who is making the move. As such, with the normal winning convention, these games can be analyzed by the Sprague–Grundy theory, according to which each heap of tokens in a subtraction game has a *nim-value*, the size of an equivalent heap in the game of Nim [4, 8, 17]. The optimal play in any such game is to move to make the bitwise exclusive-or of the nim-values zero. The winning positions are the ones in which this bitwise exclusive-or is already zero. Unlike Nim itself, positions with a single nonempty heap of tokens may be winning for the player who just moved; this is true when the nim-value of the heap is zero. The heap sizes whose nim-values are zero are called “cold”, while the remaining heap sizes are called “hot”. In a game with a single heap of tokens, it is a winning move to take a number of tokens such that the remaining tokens form a cold position. If the position is already cold, the player who just moved already has a winning position, because the player to move must move to a hot position.

Every finite subtraction set leads to a game with periodic nim-values (depending only on the sizes of the heaps modulo a fixed number). Some natural choices of infinite subtraction set, such as the prime numbers, also do not lead to interesting subtraction games [7]. However, a more complicated subtraction game, “subtract-a-square”, has the square numbers as its subtraction set. That is, on each move, each player may remove any square number of tokens from any single heap of tokens. The game of subtract-a-square was studied in 1966 by Golomb [7], who calls it “remarkably complex”; Golomb credits its invention to Richard A. Epstein.[3] Its sequence of nim-values,

$$0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 0, 1, 0, 1, 2, 3, 2, 3, 4, 5, 3, 2, 3, 4, 0, \dots$$

(sequence A014586 in the Online Encyclopedia of Integer Sequences, OEIS) displays no obvious patterns.

Subtract-a-square has another reason for interest, beyond investigations related to combinatorial game theory. The set C of cold positions in this game,

$$0, 2, 5, 7, 10, 12, 15, 17, 20, 22, 34, 39, 44, 52, 57, 62, 65, 67, 72, 85, 95, \dots$$

(sequence A030193 in the OEIS) has the property that no two elements of C differ by a square number. A sequence with this property is called a square-difference-free set. The cold positions of subtract-a-square are square-difference-free because, whenever c is a cold position, and i is a positive integer, $c + i^2$ must be hot, as one could win by moving from $c + i^2$ to c. The square-difference-free sets have been extensively investigated in number theory, following the work of Furstenberg [6] and Sárközy [14], who showed that they have natural density zero. This means that, for all ϵ, there exists an N such that, for all $n > N$, the fraction of positive integers up to n that belong to the set is at most ϵ.

More strongly, the set C of cold positions in subtract-a-square is a maximal square-difference-free set. Every positive integer that is not in C (a hot position) has a move to a cold position, so it could not be added to C without destroying the square-difference-free property. Every maximal square-difference-free subset of the range $[0, n]$ must have size at least $\Omega(\sqrt{n})$ (otherwise there would not be enough sums or differences of set elements and squares to prevent the addition of another number in this range)[4] and size at most

$$O\left(\frac{n}{(\log n)^{\frac{1}{4}\log\log\log\log n}}\right)$$

[3] No relation.

[4] See Golomb [7], Theorem 4.1.

by quantitative forms of the Furstenberg–Sárközy theorem [12]. In particular, these bounds apply to $|C \cap [0,n]|$, the number of cold positions of subtract-a-square up to n. However, it is not known whether these upper and lower bounds are tight or where the number of cold positions lies with respect to them. In the densest known maximal square-difference-free sets, the number of elements up to n is

$$\Omega\left(n^{(1+\log_{205} 12)/2}\right) \approx n^{0.733412}.$$

The construction for these dense sets involves finding a square-difference-free set modulo some base b, and selecting the numbers whose base-b representation has these values in its even digit positions and arbitrary values in its odd digit positions [13]. The bound given in the formula above comes from applying this method to a square-difference-free set of 12 values modulo 205 [1, 10]. Plausibly, a greater understanding of the nim-values of subtract-a-square could lead to progress in this area of number theory.

Algorithmically, for a subtraction game in which the allowed moves are to take a number of tokens in a given set S, the nim-values can be computed by dynamic programming, using the recurrence

$$\mathrm{nimvalue}(n) = \operatorname*{mex}_{i \in S, i \le n} \mathrm{nimvalue}(n - i).$$

Here, the "mex" operator (short for "minimum excludent" [4]) returns the smallest non-negative integer that cannot be represented by an expression of the given form. No separate base case is needed, because in the base case (when $n = 0$), the set of available moves (numbers in S that are at most n) is empty and the mex of an empty set of choices is zero. Evaluating this recurrence, for all heap sizes up to a given threshold n, takes time $O(n|S|)$. The set C of cold positions can be determined within the same time bound, by applying this recurrence and then returning the set of positions whose nim-value is zero.

However, in the study of algorithms, many naive dynamic programming algorithms turn out to be suboptimal: they can be improved by more sophisticated algorithms for the same problem. Is that the case for this one? We will see that it is. We provide the following two results:

- We show how to compute the set of cold positions in a given subtraction game, for heaps of size up to a given threshold n, in time $\tilde{O}(n)$.
- We show how to compute the nim-values of a given subtraction game, for heaps of size up to a given threshold n, in time $\tilde{O}(mn)$.

In these time bounds, the $\tilde{O}$ notation elides logarithmic factors in the time bound, and the parameter m refers to the maximum nim-value of any position within the given range.

Ignoring the logarithmic factors hidden in the $\tilde{O}$ notation, our time bounds are always at least as good as the $O(n|S|)$ time for naive dynamic programming, because for any subtraction game $m \le |S|$ (if there are only $|S|$ possible moves, the mex of their values can be at most m). But are they actually a significant improvement? To answer this, we need to know how quickly m grows compared to the known growth rate of $|S|$.

To determine whether our algorithms provide a speedup for the game of subtract-a-square, we performed a sequence of computational experiments to determine the density of this game's cold positions and the growth rate of its largest nim-values. We find experimentally that, up to a given n, the largest nim-value appears to grow as $O(n^{0.35})$, significantly more slowly than the $O(n^{1/2})$ growth rate of the subtraction set. The difference in the growth rates for these quantities shows that our algorithms are indeed an asymptotic improvement by a polynomial factor. Additionally, the number of cold positions appears to grow at least as quickly as $n^{0.69}$. That is, the cold positions of this game provide an unexpectedly large square-difference-free set, competitive with the best theoretical constructions for these sets. Examining the modular structure of the set of cold positions, we find that it appears to be similar to the structure of these theoretical constructions, with a square-difference-free set of digit values in even positions and arbitrary values in odd positions.

2 Algorithms

2.1 Subtraction with hotspots

In order to evaluate subtraction games efficiently, it will be convenient to generalize them somewhat, to a class of *subtraction games with hotspots*. Given two sets S and H (of positive and non-negative integers respectively), we define a subtraction game with subtraction set S and hotspot set H as follows. The game starts with a single pile of some number of tokens, and the players alternate in choosing a number from S and removing that number of tokens from the pile, as before. However, if any move leaves a pile whose remaining number of tokens belongs to H, then the player who made that move immediately loses. (This is not quite the same as allowing the other player to remove all the tokens from piles whose size belongs to H, because H might contain the number zero, in which case removing all the tokens could be a losing move instead of a winning move.)

The presence of these hotspots makes defining a nim-value for these games problematic: they are not played by the normal winning convention, so what would happen if we played a game with multiple piles and one player moved to a hotspot? Nevertheless, a recurrence of the usual form suffices to determine the set of hot and cold positions of such a game:

$$\operatorname{hot}(n) = n \in H \vee \bigvee_{i \in S, i \leq n} \neg \operatorname{hot}(n-i).$$

2.2 Finding the hotspots

In a subtraction game (with subtraction set S, with or without hotspots), suppose that some set C of positions has already been determined to be cold. Then all positions H that can reach C in a single move are automatically hot. We can formulate membership in this set of hot positions as a Boolean formula in conjunctive normal form (2-CNF):

$$(i \in H) \iff \bigvee_{j+k=i} (j \in C) \wedge (k \in S).$$

Now suppose that C and S are both represented as bitvectors: arrays of binary values that are 0 for non-members and 1 for members of each set. Then the problem of computing the bitvector representation of H from the above formula is a standard problem known as *Boolean convolution*, studied for its applications in string matching [5,9,11]. It is an instance of a more general class of convolution problems in which we compute

$$C[i] = \bigoplus_{j+k=i} A[j] \otimes B[j]$$

for an "addition" operation $\oplus$ and "multiplication" operation $\otimes$. In Boolean convolution, $\otimes$ is conjunction ($\wedge$), and $\oplus$ is disjunction ($\vee$).

If the input bitvectors have total length n, their Boolean convolution can be computed in $O(n \log n)$ time by replacing their Boolean values with the numbers 0 and 1 and computing a numerical convolution (with addition as $\oplus$ and multiplication as $\otimes$) using the fast Fourier transform algorithm.

2.3 Divide and conquer

We are now ready to describe our algorithm for finding the hot and cold positions of a subtraction game with hotspots. We assume that we are given as input a range $[x, y)$ of integer values to evaluate (following the Python convention for half-open integer ranges where the bottom delimiter is inside the range and the top delimiter is outside it), together with two sets: the subtraction set S and a set H of predetermined hotspots.

As a base case, if the range has zero or one values in it, we can solve the problem directly: each value in the range is hot or cold accordingly as it belongs or does not belong to H, respectively. Otherwise, we perform the following steps:

1. Find the midpoint $m = (x + y)/2$ of the range, and partition the range into the two subranges $[x, m)$ and $[m, y)$.
2. Recursively evaluate the lower subrange $[x, m)$, determining its hot and cold positions (H_x and C_x respectively).
3. Use Boolean convolution to find the positions H_m in the upper subrange $[m, y)$ that are hot because they can be reached in a single step from a cold position C_x in the lower subrange.
4. Recursively evaluate the upper subrange $[m, y)$, with hotspot set $H \cup H_m$, determining its hot and cold positions (H_y and C_y respectively).
5. Return the hot set $H_x \cup H_y$ and cold set $C_x \cup C_y$.

The time for this algorithm can be analyzed using the master method, as is standard for such divide-and-conquer algorithms, giving the following result:

▶ **Theorem 1.** *We can determine which positions are hot and which are cold, in a range of n positions of a subtraction game with hotspots, in time $O(n \log^2 n)$.*

2.4 Nim-values

We can reduce the computation of nim-values in a subtraction game to the computation of hot and cold positions in a subtraction game with hotspots, via the following lemma.

▶ **Lemma 2.** *Let S be a subtraction set, and let H be the set of positions in the subtraction game for S that have nim-value at most t. Then the positions that have nim-value $t + 1$ are exactly the cold positions of the subtraction game with hotspots with subtraction set S and hotspot set H.*

Proof. A position has nim-value $t + 1$ if it does not belong to H (else it would have a smaller nim-value) and does not have a move to a smaller position with nim-value $t + 1$ (else $t + 1$ would not be one of its excluded values). But this is exactly the defining condition for the cold positions of the subtraction game with hotspots. ◀

▶ **Theorem 3.** *In any subtraction game with subtraction set S, we can determine the nim-values of the first n positions in time $O(mn \log^2 n)$, where m is the maximum nim-value of any of these positions.*

Proof. We loop over the range of nim-values from 0 to s, using Theorem 2 to compute the set of positions having each successive nim-value in time $O(n \log^2 n)$ per nim-value. The loop terminates when all of the first n positions have been assigned a nim-value. ◀

The maximum nim-value of a subtraction game is $|S|$, so (except for the logarithmic factors) this time bound compares favorably with a naive $O(n|S|)$ dynamic programming algorithm for computing the nim-values of each position by finding the minimum excluded value among the other positions reachable from it.

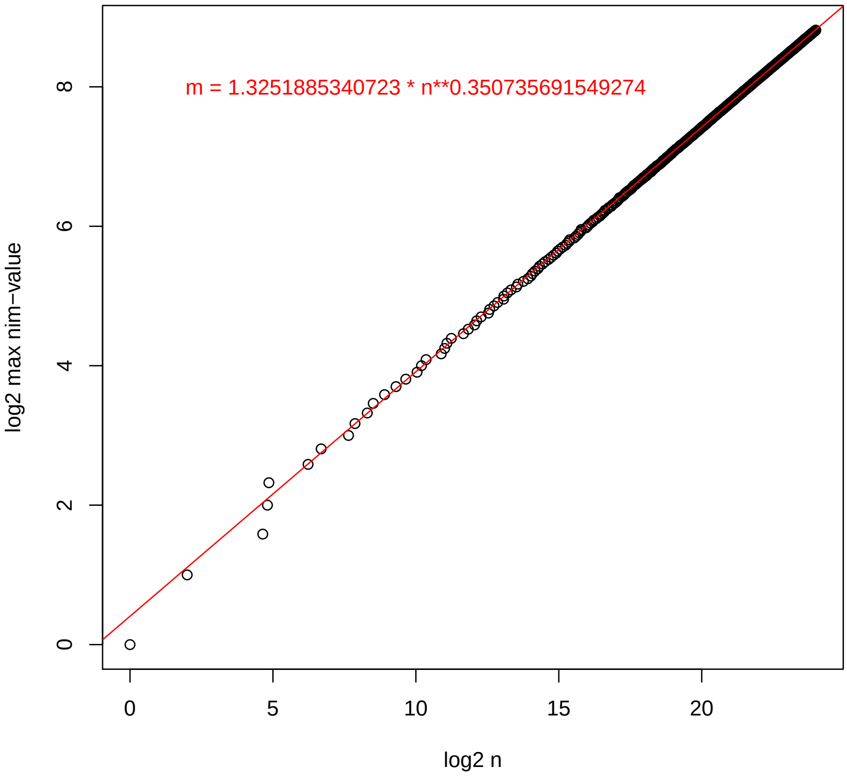


Figure 1 The maximum nim-values m seen among the first n positions in subtract-a-square.

3 Experiments

To compare the performance of our Boolean convolution based evaluation algorithms to naive algorithms for subtract-a-square, we performed some computational experiments, which we describe here.

3.1 Maximum nim-value

Figure 1 plots (on a doubly logarithmic scale) the maximum nim-values m seen among the first n positions in subtract-a-square. Only the positions where a new maximum is attained are included in the plot.

We fitted a function of the form cn^e (a monomial with constant coefficient c and exponent e) to these points, by using Siegel's repeated median estimator [16], a form of robust statistical regression that is insensitive to outliers (as would be expected to occur in the lower left parts of the plot). This estimator fits a line through a sample of points by, for each point, computing the median of the slopes formed by it and the other points, and then choosing the slope of the fit line to be the median of these medians. It similarly chooses the height of the fit line so that it passes above and below an equal number of points. We applied this to the points on our log-log plot, using the mblm library of the R statistical package, which implements this estimator, and then transformed the fit line back to a monomial over the

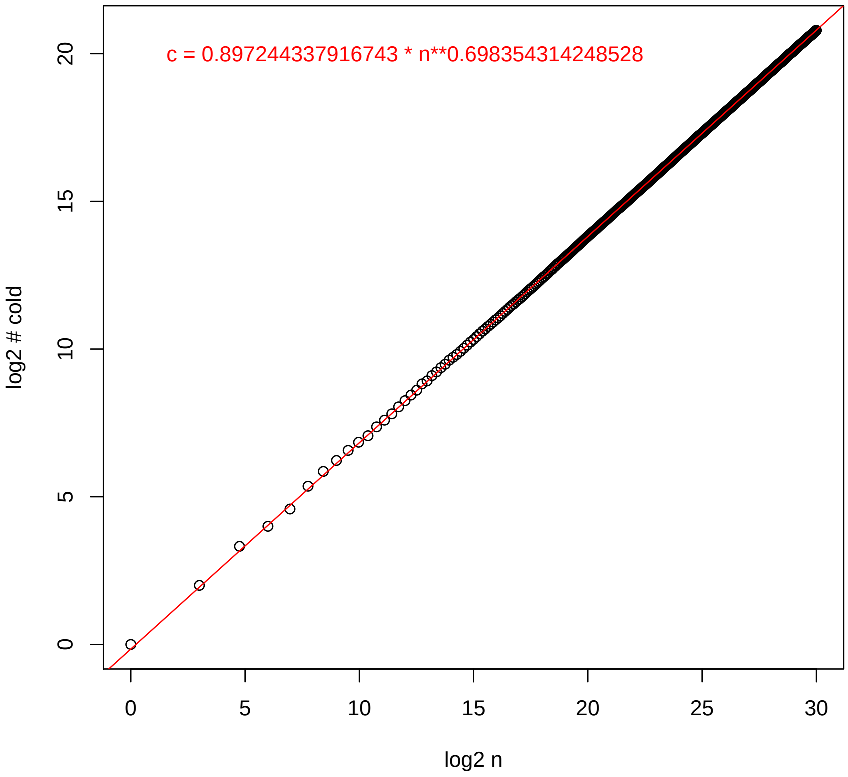


Figure 2 The number of cold positions among the first n positions in subtract-a-square.

original coordinates of the data points. The result is shown in red in the figure.

As the figure shows, the maximum nim-value m among the first n positions of subtract-a-square is accurately estimated by a function of the form $O(n^{0.351})$, well below the $O(n^{0.5})$ size of the subtraction set for this game. Therefore, we would expect our $O(mn\log^2 n)$-time convolution-based algorithm for computing the nim-values of this game to be asymptotically faster than the $O(n^{3/2})$ time for dynamic programming. However, even if we ignore the different constant factors in the running times of these two algorithms, n needs to be approximately 10^{26} in order for $n^{1.35}\log_2^2 n$ to be smaller than $n^{1.5}$, so we would not expect this speedup to be applicable to practically relevant ranges of n. Because of the simplicity and relative efficiency of the dynamic programming algorithm for small n, the values in Figure 1 (for n up to 2^{24}) were computed by dynamic programming rather than convolution.

3.2 Number of cold positions

Our next experiment measures the number of cold positions among the first n positions in subtract-a-square (Figure 2). In order to provide more data points in the lower left part of the log-log plot than would be visible if we used uniform sampling of the range of values of n, the plot of Figure 2 shows the number of cold positions for each value of n that is a perfect cube (that is, for the values $1, 8, 27, 64, \dots$) up to 2^{30}. As in the previous experiment, we fitted a monomial function to these points using Siegel's repeated median estimator.

The number of cold positions does not directly affect the time bound for our convolution-based algorithm. However, it does affect the time for a different algorithm, for computing the set of cold positions (but not their nim-values) directly, in any subtraction game. This algorithm is analogous to the sieve of Eratosthenes, which finds prime numbers iteratively, for each one marking off the numbers that are not prime. To compute the cold positions among the first n positions, it performs the following steps:

1. Initialize a Boolean array H of length n (indicating whether each position is hot) to be false in each cell.
2. For each position i from 0 to n, test whether $H[i]$ is still false. If it is, perform the following steps:
   a. Output i as one of the cold positions.
   b. For each value s in the subtraction set S, mark $i + s$ as hot by setting $H[i + s]$ to be true.

If the set of cold positions up to n is C, and the subtraction set is S, then this sieving algorithm takes time $O(|C| \cdot |S|)$.

In some subtraction games, C could be as small as $n/|S|$, in which case the sieving algorithm would take linear time. However, our experiments show that, for subtract-a-square, C appears to grow more like $n^{0.7}$, giving the sieving algorithm a running time of approximately $n^{1.2}$, compared to the $O(n \log^2 n)$ time bound of the convolution based algorithm. Again ignoring the constant factors in the time bounds, n would need to be approximately 10^{18} for the convolution-based algorithm to be faster than the sieving algorithm. Because it is simple to code and fast for smaller values of n, the results in Figure 2 were calculated using the sieving algorithm.

3.3 Modularity

The high density of cold positions in subtract-a-square is surprising, especially in view of earlier conjectures in the theory of square-difference-free sets that the number of values up to n in such a set could be at most $n^{1/2+o(1)}$ [15]. These conjectures were disproven by finding sets of numbers of a special form: numbers whose radix-b representation, for a carefully chosen base b, use only base-b digits from a square-difference-free set (modulo b) in their even digit positions [1, 10, 13]. Although the cold positions of subtract-a-square have somewhat lower density than these constructions, they arise more naturally, and it is of interest to investigate their modular structure and compare it to the structure of these other known dense square-difference-free sets.

The idea of considering the base-b structure of these positions, for different choices of the base b, also arises from the consideration of a different subtraction game, described by Golomb [7]. This game has as its subtraction set the Moser–de Bruijn sequence

$$0, 1, 4, 5, 16, 17, 20, 21, 64, 65, 68, 69, \ldots$$

of numbers that are sums of distinct powers of four. That is, when written in base 4, the numbers of the subtraction set have only 0 and 1 as their base-4 digits. The nim-value of any position n may be obtained by writing n in base 4, taking each digit modulo 2 (reducing it to 0 or 1), and then reinterpreting the resulting string of 0's and 1's as a binary number. Because of this simple formula for its nim-values, the Moser–de Bruijn subtraction game has both a maximum nim-value and a number of cold positions (among the first n positions) proportional to $\Theta(\sqrt{n})$. It subtraction set size, also $\Theta(\sqrt{n})$, is comparable to that

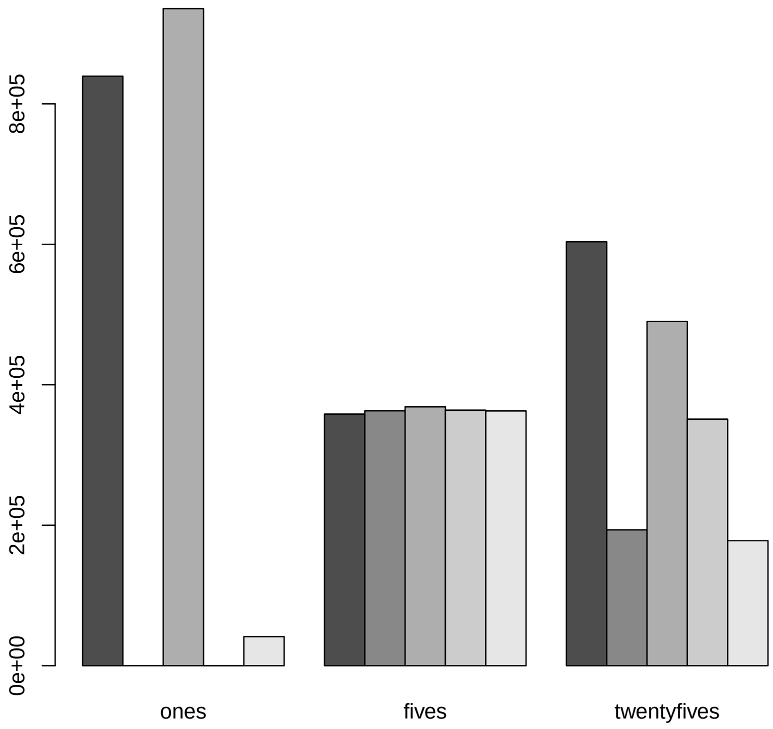


Figure 3 The distribution of digit values among the three low-order base-5 digits of cold positions (for $n < 2^{30}$) in subtract-a-square.

for subtract-a-square. In particular, for this game, convolution is neither asymptotically faster than dynamic programming nor than sieving, although all of these algorithms can be improved by using the formula instead. What makes subtract-a-square so different from the Moser–de Bruijn subtraction game?

To approach these questions, we performed more computational experiments studying the distribution of digit values for the cold positions in subtract-a-square, for various bases. This study follows the earlier work of Golomb [7], who observed that the low-order base-5 digits of the cold positions among the first first 20,000 game positions were highly non-uniformly distributed, and of Bush [3], who extended this study to the first 40,000,000 game positions. Figure 3 shows an extension of this study to the first 2^{30} game positions, and to the three low-order base-5 digits of each cold position. As the figure shows, with a few exceptions, the ones digit of the cold positions lies within the square-difference-free set $\{0, 2\} \pmod 5$. The fives digit shows no significant non-uniformities, but the twentyfives digit is quite non-uniformly distributed, and is possibly heading towards the same square-difference-free set $\{0, 2\} \pmod 5$. In this way, the cold positions of subtract-a-square appear to be emulating the strategy of the known dense square-difference-free sets $[1, 10, 13]$ of having a square-difference-free set of digits in even digit positions and all possible digits in odd positions modulo a base b. In this case $b = 5$, and following this strategy perfectly for $b = 5$ would lead to a set of size $n^{\log_{10} 25} \approx n^{0.71534}$. The slightly slower growth rate of the cold positions in subtract-a-square can be explained by the slow convergence of its higher-order base-5 digits to square-difference-free sets of digits.

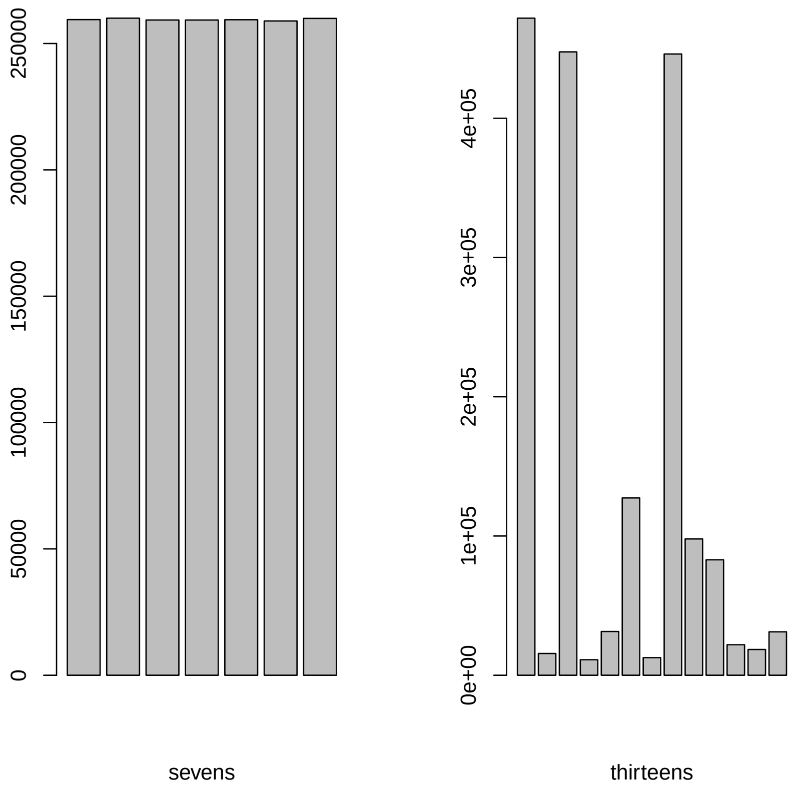


Figure 4 The distribution of digit values among the low-order base-7 and base-13 digits of cold positions (for $n < 2^{30}$) in subtract-a-square.

What about other bases? Figure 4 shows the results of the same experiment (for the low-order digits only) for base 7 and base 13. Because 7 is 3 modulo 4, there are no nontrivial square-difference-free sets modulo 7: every two numbers modulo 7 differ by a square (mod 7). Perhaps because of this, the digit values in base 7 show no significant nonuniformities. However, modulo 13, the squares are 0, ± 1, ± 3, and ± 4. Because 13 is 1 modulo 4, each of the nonzero squares occurs four times among the squares of values mod 13; for instance, ± 3 is the square of 4, 6, 7, and 9 (mod 13). As the figure shows, the low-order digits of the base-13 representations of the cold positions in subtract-a-square appear to be converging towards the square-difference-free set $\{0, 2, 7\}$ (mod 13). Perhaps subtract-a-square implements the modular strategy for finding dense square-difference-free sets in all prime bases (congruent to 1 mod 4) simultaneously?

4 Conclusions

We have developed new convolution-based methods for evaluating arbitrary subtraction games (either to determine the set of cold positions or to evaluate the nim-value of each position). Our experiments on the subtract-a-square game show that its maximum nim-value is lower than the theoretical value for games with subtraction sets of the same size, and its number of cold positions is higher than the theoretical value. These results show that, asymptotically, our new algorithms are faster than alternative dynamic programming or sieving approaches for the same problems on this game. However, the breakeven point for the new algorithms is high enough that our convolution-based approach is not yet practical. It would be of interest to develop improved algorithms that are both asymptotically faster and more practical than existing approaches.

In an attempt to investigate why the cold positions of subtract-a-square produce a dense square-difference-free set, we investigated the base-b representations of the cold positions for several small prime choices of b. Our tests found significant irregularities in the even positions of these base-b representations, when b is congruent to 1 mod 4. We leave the problem of finding a theoretical explanation for these patterns, and for the density of the cold positions in subtract-a-square, as open for future research.

References

1 Richard Beigel and William Gasarch. Square-difference-free sets of size $\Omega(n^{0.7334\cdots})$. Electronic preprint arxiv:0804.4892, 2008.

2 Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. Subtraction games. In *Winning Ways for your Mathematical Plays, Vol. I: Games in General*, pages 83–86. Addison-Wesley, 1982.

3 David Bush. The uniqueness of 11,356. sci.math usenet newsgroup, October 12 1992. URL: https://www.ics.uci.edu/~eppstein/cgt/subsquare.html.

4 J. H. Conway. Chapter 11: Impartial Games and the Game of Nim. In *On Numbers and Games*, pages 122–135. Academic Press, 1976.

5 Michael J. Fischer and Michael S. Paterson. String-matching and other products. In *Complexity of computation (Proc. SIAM-AMS Appl. Math. Sympos., New York, 1973)*, volume 7 of *SIAM-AMS Proceedings*, pages 113–125, Providence, RI, 1974. American Mathematical Society.

6 Harry Furstenberg. Ergodic behavior of diagonal measures and a theorem of Szemerédi on arithmetic progressions. *Journal d'Analyse Mathématique*, 31:204–256, 1977. doi:10.1007/BF02813304.

7 Solomon W. Golomb. A mathematical investigation of games of "take-away". *Journal of Combinatorial Theory*, 1:443–458, 1966. doi:10.1016/S0021-9800(66)80016-9.

8 P. M. Grundy. Mathematics and games. *Eureka*, 2:6–8, 1939.

9 Adam Kalai. Efficient pattern-matching with don't cares. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '02)*, pages 655–656, Philadelphia, PA, 2002. Society for Industrial and Applied Mathematics. URL: https://dl.acm.org/citation.cfm?id=545381.545468.

10 Mark Lewko. An improved lower bound related to the Furstenberg-Sárközy theorem. *Electronic Journal of Combinatorics*, 22(1):P1.32, 2015. URL: https://www.combinatorics.org/ojs/index.php/eljc/article/view/v22i1p32.

11 S. Muthukrishnan and Krishna V. Palem. Non-standard stringology: algorithms and complexity. In Frank Thomson Leighton and Michael T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 770–779. ACM, 1994. doi:10.1145/195058.195457.

12 János Pintz, W. L. Steiger, and Endre Szemerédi. On sets of natural numbers whose difference set contains no squares. *Journal of the London Mathematical Society (2nd Series)*, 37(2):219–231, 1988. doi:10.1112/jlms/s2-37.2.219.

13 I. Z. Ruzsa. Difference sets without squares. *Periodica Mathematica Hungarica*, 15(3):205–209, 1984. doi:10.1007/BF02454169.

14 A. Sárkőzy. On difference sets of sequences of integers. I. *Acta Mathematica Academiae Scientiarum Hungaricae*, 31(1–2):125–149, 1978. doi:10.1007/BF01896079.

15 A. Sárközy. On difference sets of sequences of integers. II. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae*, 21:45–53, 1978.

16 Andrew F. Siegel. Robust regression using repeated medians. *Biometrika*, 69(1):242–244, 1982. doi:10.1093/biomet/69.1.242.

17 R. P. Sprague. Über mathematische Kampfspiele. *Tohoku Math. J*, 41:438–444, 1935. URL: https://www.jstage.jst.go.jp/article/tmj1911/41/0/41_0_438/_article.

Making Change in 2048

David Eppstein[1]
Computer Science Department, University of California, Irvine
eppstein@uci.edu

Abstract

The 2048 game involves tiles labeled with powers of two that can be merged to form bigger powers of two; variants of the same puzzle involve similar merges of other tile values. We analyze the maximum score achievable in these games by proving a min-max theorem equating this maximum score (in an abstract generalized variation of 2048 that allows all the moves of the original game) with the minimum value that causes a greedy change-making algorithm to use a given number of coins. A widely-followed strategy in 2048 maintains tiles that represent the move number in binary notation, and a similar strategy in the Fibonacci number variant of the game (987) maintains the Zeckendorf representation of the move number as a sum of the fewest possible Fibonacci numbers; our analysis shows that the ability to follow these strategies is intimately connected with the fact that greedy change-making is optimal for binary and Fibonacci coinage. For variants of 2048 using tile values for which greedy change-making is suboptimal, it is the greedy strategy, not the optimal representation as sums of tile values, that controls the length of the game. In particular, the game will always terminate whenever the sequence of allowable tile values has arbitrarily large gaps between consecutive values.

2012 ACM Subject Classification Theory of computation → Discrete optimization

Keywords and phrases 2048, change-making problem, greedy algorithm, integer sequences, halting problem

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.21

1 Introduction

The solitaire game 2048 was developed in 2014 by Gabriele Cirulli, based on another game called Threes developed earlier in 2014 by Asher Vollmer [28]. It is played on a 16-cell square grid, each cell of which can either be empty or contain a tile labeled with a power of two. In each turn, a tile of value 2 or 4 is placed by the game software on a randomly chosen empty cell. The player then must tilt the board in one of the four cardinal directions, causing its tiles to slide until reaching the edge of the board or another tile. When two tiles of equal value slide into each other, they merge into a new tile of twice the value. The game stops when the whole board fills with tiles, and the goal is to achieve the highest single tile value possible. Figure 1 shows the state of the game after approximately 4000 moves, when a tile with value 8192 has been reached.

As most players of the game quickly learn, it is not possible to keep playing a single game of 2048 forever. At any step of the game, there must be at least one tile for each nonzero bit in the binary representation of the total tile value. For total tile values just below a large power of two, the number of ones in the binary representation is similarly large, eventually exceeding the number of cells in the board.

[1] Supported in part by NSF grants CCF-1618301 and CCF-1616248.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 21; pp. 21:1–21:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

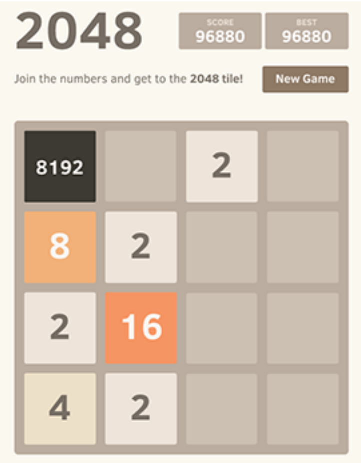


Figure 1 A state in the game 2048 in which a tile of value 8192 has been reached.

But other variants of 2048 use different tile values than powers of two. Threes uses the sequence of numbers that are either powers of two or three times a power of two:

$$1, 2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 64, \ldots$$

(It also restricts tile merges to pairs of tiles whose values are equal or differ by a factor of two.) Fives uses 2, 3, and powers of two times 5, giving the sequence of allowable values [12]

$$1, 2, 3, 5, 10, 20, 40, 80, 160, 320, 640, \ldots$$

Another variant, called 987, uses as its tile values the Fibonacci numbers,

$$1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, \ldots$$

We can find analogous ad-hoc arguments for why these games must terminate, but can we generalize them to arbitrary systems of tile values? If we define a 2048-like game with a set S as its tile values, is the length of the game and the maximum value that can be achieved controlled, as it is for binary numbers, by the lengths of the shortest representations of arbitrary numbers as sums of members of S?

For instance, suppose that we allow any *practical number* as a tile value, and any merge of two tiles that produces another practical number. The practical numbers are defined by the property that, for a practical number n, every integer $m < n$ can be expressed as a sum of distinct divisors of n. Their sequence begins

$$1, 2, 4, 6, 8, 12, 16, 18, 20, 24, 28, 30, 32, \ldots$$

There are many more practical numbers than powers of two, and the practical numbers behave in many ways like the prime numbers. In particular, analogously to Goldbach's conjecture for the prime numbers, every even integer can be expressed as a sum of two practical numbers [17], and therefore every integer can be expressed as a sum of three practical numbers. Because we can express every tile value using a bounded number of practical-number tiles, does the practical-number variant of 2048 go on forever?

Alternatively, suppose we use 3-smooth tile values, the numbers whose only prime factors are two or three:

$$1, 2, 3, 4, 6, 8, 9, 12, 16, 18, 24, 27, 32, 36, \ldots$$

Because the number of distinct 3-smooth numbers in the range from 1 to n is only $O(\log^2 n)$, an information-theoretic argument shows that some numbers in this range will require $\Omega(\log n/\log\log n)$ terms in their shortest representation as a sum of 3-smooth numbers. Therefore, for a game using these tile values to last for n moves, it must use a game board that has at least $\Omega(\log n/\log\log n)$ cells. Is this analysis tight?

1.1 New results

In this paper we show that the answer to these questions is no. 2048-like games are not controlled by the shortest representations of numbers as sums of tile values, but rather by their *greedy representations*, representations generated by a greedy heuristic for the problem of making change using the smallest number of coins from a given coinage system. For the powers of two, the Fibonacci numbers, and the numbers used by Threes and Fives, these greedy representations coincide with the shortest representations, but that is not true for many other natural sets of numbers including the practical numbers and the 3-smooth numbers. The lengths of greedy representations, in turn, are controlled by the lengths of the gaps between consecutive tile values.

As a consequence, we show that whenever a sequence of numbers has arbitrarily large gaps, the 2048-like game based on those numbers must terminate with a finite limit on its number of moves and on its largest achievable tile value. For instance, because the practical numbers have inverse-logarithmic density (analogously to the prime number theorem for the density of the prime numbers) [29], they have arbitrarily large gaps and the game based on them terminates, albeit much more slowly than for the powers of two.

1.2 Related work

2048 has been the subject of much past research. Its past investigations include studies of its computational complexity [1,2,12,16], artificial intelligence based game strategies [10,15,19, 23,27,30], computer science education [18], and computer-human interaction [22].

2 Simplification through abstraction

Several features of 2048 and its variants complicate their analysis, possibly making its game play more interesting but without (it appears) greatly affecting the questions we wish to study, on how long a game can last or which tile values can be achieved.

Board geometry. The cells of the 2048 board are arranged in a square grid, which controls both the sliding movement of the tiles across the board and the pairs of tiles that can become adjacent to each other and merge. Much of the strategy of the game involves linearizing this two-dimensional arrangement of cells by finding a zigzag path that covers all the cells of the grid and playing in such a way that tiles move and merge with each other only along this path.

Restricted tile merges. In some variations of 2048, such as Threes, certain pairs of tiles cannot merge even when their summed value would be an allowable tile value. For instance, in Threes, the merge $3 + 1 = 4$ is not allowed; only pairs of tiles with the same value or with one twice the other can merge. Even in 2048, only pairs of tiles, and not larger combinations of tiles, are allowed to merge.

Unknown or random future events. In 2048, the next tile could either have value 2 or 4. In most cases this causes little change to game play, because a tile of value 4 is not significantly different than two consecutive tiles of value 2 that then became merged, but it can interact with the board geometry to cause tiles to become out of position, making continued play more difficult. And in many of these games, the location of each newly placed tile could be any previously-open cell. These unknowns make the game nondeterministic, and complicate the definition of the longest play or highest achievable tile value: do we mean the worst case (the best that a player could achieve against a malicious adversary), best case (the best one could hope to achieve against repeated play with a random adversary), or some kind of probabilistic analysis that determines the distribution or expected value of scores?

To avoid these complications, we define a class of variants of 2048 in which they are eliminated.

▶ **Definition 1** (abstract generalized games)**.** Given a set A of allowable tile values, an initial element $a \in A$ (usually $a = 1$), and a number n of cells, we define the *abstract generalized 2048 game* for A and a to be a solitaire game in which there are n indistinguishable cells, each of which can either be empty or contain a tile with a value in A. We define a *position* of the game to be an assignment of either a tile with a value in A or no tile to each cell of the game. The *initial position* of the game is a position in which all cells are empty. Starting from the initial position, each step of the game consists of the following actions:

- The player chooses any empty cell, and a tile of value a is placed into that cell.
- The player may choose to merge any sets of non-empty cells whose total value belongs to A into a single tile, which is placed on a single cell from its set. The remaining cells in each chosen set become empty.

The game ends when, after one of these steps, all cells are nonempty. When this happens, there would be nowhere to place the new tile of value a in the next step.

We denote the abstract generalized 2048 game on n cells with tile value set A and initial tile value a by $\mathrm{AGG}(n, A, a)$ or (when $a = 1$) by $\mathrm{AGG}(n, A)$.

▶ **Observation 2** (simulation by abstract games)**.** With the possible exception of the value of each newly placed tile, each action in 2048, Threes, Fives, or 987 can be simulated by a corresponding action in the abstract generalized 2048 game with the same set of tile values and the same number of cells. Therefore, any upper bound on the number of moves or maximum tile value in the abstract generalized 2048 game provides a valid upper bound for the number of moves or maximum tile value in the corresponding sliding-tile game.

3 Optimal strategy in the abstract game

The abstract generalized 2048 game eliminates the complications of board geometry, tile position, and sliding mechanics from the game, making its analysis much simpler. As a consequence, we can characterize the optimal strategies in this game. We begin by describing some helpful move-ordering principles.

▶ **Definition 3** (eager sequences)**.** We say that a sequence of steps in $\mathrm{AGG}(n, A)$ is *eager* if each merge of tiles is performed in the first step at which all of the tiles to be merged have their merged values, rather than delaying the merge until some later step.

▶ **Observation 4** (all sequences can be made eager)**.** If a position in $\mathrm{AGG}(n, A)$ can be reached by a sequence of steps, it can be reached by an eager sequence of steps.

▶ **Lemma 5** (single-tile-first strategy). *Let P be a position in* $\mathrm{AGG}(n, A)$ *that can be reached by a sequence of steps from the initial position. Then there exists a non-empty cell c of value v in P, and a sequence of steps that reaches P from the initial position, with the following structure:*

- *First, perform a sequence of steps that reaches the position P', where P' has a tile of value v in cell c and $n - 1$ empty cells.*
- *Next, perform a sequence of steps in the game* $\mathrm{AGG}(n - 1, A)$*, using only the cells that are empty in position P', to reach the position P'' in that game corresponding to position P in* $\mathrm{AGG}(n, A)$ *(the position formed from P by removing one cell of value m).*

Proof. Let S be an eager sequence of steps that produces position P. By assumption, S exists, and we may assume by Observation 4 that S is eager. By running the sequence of steps in S backwards from P, we may determine, for each position reached during the course of sequence S, which of its nonempty tiles eventually contribute to each tile of P. By the eager property of S, each merge produced in each step involves only tiles that contribute to the same cell as the newly-placed tile in that step.

Let c be the cell of P to which the first newly-placed tile contributes, and let v be the value of the tile in cell c of position P. Because the cells are indistinguishable, we may rearrange the cells of $\mathrm{AGG}(n, A)$ so that the first newly-placed tile is placed into cell c, and so that each subsequent merge step involving this tile places the merged tile back into cell c. After this rearrangement, cell c is always occupied by a tile that contributes to the eventual value in cell c. We may then separate S into two subsequences of steps, the subsequence S_1 of steps whose newly placed tile contributes to c and the subsequence S_2 of steps in which the newly placed tile contributes to some other cell of P.

Then subsequence S_1 may be performed first, before any steps of S_2. This change of order causes positions of the game to be empty in S_1 that are non-empty in S, but those positions do not contribute to c and therefore do not affect what happens in these steps. Performing S_1 reaches state P', as described by the statement of the lemma.

The cells other than c form an instance of $\mathrm{AGG}(n - 1, A)$, and each step of S_2 operates only on these cells because at each of these steps, c is occupied by a tile that is unchanged by that step. Therefore, S_2 may be performed on $\mathrm{AGG}(n-1, A)$ to reach position P''. Because c is the only nonempty cell after S_1 and is unused by S_2, it is valid to perform the concatenation of subsequences S_1S_2, which reaches state P with the desired step ordering. ◀

▶ **Lemma 6** (step-by-step reachability for single tiles). *Let $x > 1$ be a tile value in set A, and let y be the largest value in A that is less than x. Let n be a positive integer, let P_x be the position consisting of one cell containing a tile of value x and $n - 1$ empty tiles, and let P_y be defined in the same way for value y. Then there is a sequence of steps in* $\mathrm{AGG}(n, A)$ *that reaches P_x if and only if the following conditions are both true:*

1. *There is a sequence of steps in* $\mathrm{AGG}(n, A)$ *that reaches P_y.*
2. *There is a sequence of steps in* $\mathrm{AGG}(n-1, A)$ *that reaches a position of total value $x - y$.*

Proof. We prove separately that both conditions imply reachability of P_x, and that reachability of P_x implies both conditions.

(1 & 2) $\Rightarrow P_x$:
Clearly if both conditions are true, then we can use the sequence from the first condition to reach P_y, then concatenate the sequence of steps from the second condition to reach a position that includes both y and some other tiles of total value $x - y$, and finally perform a single merge operation to combine all of these tiles to a single tile of value x.

$P_x \Rightarrow$ (1):
Let S be any sequence of steps that reach P_x. Then the first y steps of S reach a position of total value y, from which P_y can be formed by one more merge operation.

$P_x \Rightarrow$ (2):
P_x is reachable if and only if we can reach a position P' of total value $x-1$, with at least one empty cell, so that the newly placed tile of the next step creates total value x. By the single-tile-first strategy (Lemma 5), P' is reachable if and only if there exists $z \in A$, with $0 < z < x$, such that P_z is reachable in $\mathrm{AGG}(n, A)$ and the remaining cells of P', of total value $x - z - 1$ and with at least one empty cell, are reachable in $\mathrm{AGG}(n-1, A)$. If $y = z$ then one more step in $\mathrm{AGG}(n-1, A)$ places a new tile of value 1 in the empty cell and creates a position of total value $x - y$, meeting condition 2. If, on the other hand, $y > z$, then $x - y \le x - z - 1$ and the first $x - y$ steps in $\mathrm{AGG}(n-1, A)$ already create a position of total value $x - y$, again meeting condition 2. ◀

▶ **Corollary 7** (threshold of single-tile reachability). *For every n and A then there exists a value $\mathrm{Single}(n, A) \in A \cup \{\infty\}$ such that the positions P_x (with one tile of value x and $n-1$ empty cells) are reachable in $\mathrm{AGG}(n, A)$ if and only if $x \in A$ and $x \le \mathrm{Single}(n, A)$. If $\mathrm{Single}(n, A)$ is finite, it is the maximum single tile value achievable in game $\mathrm{AGG}(n, A)$; if not, all tile values are achievable.*

▶ **Observation 8** (monoticity of single-tile thresholds). For all $n > 1$ and A, $\mathrm{Single}(n, A) \ge \mathrm{Single}(n-1, A)$.

Proof. If we can reach any single tile value v in game $\mathrm{AGG}(n-1, A)$, we can also reach it in $\mathrm{AGG}(n, A)$ by ignoring the extra cell. ◀

▶ **Theorem 9** (characterization of reachable positions). *Let n and A be given. Then a position P of $\mathrm{AGG}(n, A)$ is reachable by a sequence of steps from its initial position if and only if the sequence of its tile values $v_1, \dots v_n$ (sorted from smallest to largest, with $v_i = 0$ if there are at least i empty cells in P) satisfies the inequalities $v_i \le \mathrm{Single}(i, A)$ for all i.*

Proof. By applying the single-tile-first strategy (Lemma 5) recursively, we may decompose P into a sequence of tile values u_n (the single tile used by the strategy to reach P), u_{n-1} (the tile used by applying Lemma 5 to the game $\mathrm{AGG}(n-1, A)$ after constructing tile u_n, $\dots$ padding the sequence with zeros if necessary. Then by construction u_n is achievable in game $\mathrm{AGG}(n, A)$, u_{n-1} is achievable in game $\mathrm{AGG}(n-1, A)$, etc., so these values satisfy inequalities $u_i \le \mathrm{Single}(i, A)$ for all i like the ones in the statement of the lemma.

This decomposition need not be sorted. However, because the values of $\mathrm{Single}(i, A)$ are monotonically non-decreasing (Observation 8), swapping any two values of u_i and u_j that are out of order preserves the inequalities between these values and $\mathrm{Single}(i, A)$ and $\mathrm{Single}(j, A)$. Since the sorted sequence of values v_i can be obtained from the sequence u_i by such swaps, it also obeys all the same inequalities. ◀

Using this characterization we can strengthen Lemma 5 to more explicitly describe a game strategy for reaching any given position.

▶ **Corollary 10** (how to play to reach any single position). *Let P be any reachable position in game $\mathrm{AGG}(n, A)$. Then the following strategy for playing the game reaches P:*

- *If P contains more than one tile, first play the strategy recursively to reach a position with one nonempty cell, containing the largest tile value in P. Then continue recursively in the game $\mathrm{AGG}(n-1, A)$ on the remaining cells to construct the remaining tiles of P.*

- *If P contains only one tile, of value x, let y be the largest value in A that is less than v. Play the strategy recursively to reach a position with two nonempty cells, with values y and $x - y$, and then in the final step of the recursive strategy merge these two values.*

The correctness of this strategy follows easily by using Theorem 9 to prove that each recursive goal within this strategy is itself reachable.

Putting the results of this section together, we have the following simple recurrence for computing $\mathrm{Single}(n, A)$ and $\mathrm{Total}(n, A)$:

▶ **Theorem 11** (recurrence for single-tile and total-value reachability). *Beginning with*

$$\mathrm{Single}(0, A) = \mathrm{Total}(0, A) = 0,$$

we may compute $\mathrm{Single}(n, A)$ *as the smallest value in A whose difference from the next larger value in A is larger than* $\mathrm{Total}(n - 1, A)$*, or ∞ if no such value exists. We may compute*

$$\mathrm{Total}(n, A) = \mathrm{Single}(n, A) + \mathrm{Total}(n - 1, A) = \sum_{i=1}^{n} \mathrm{Single}(i, A).$$

Proof. The computation of $\mathrm{Single}(n, A)$ follows from Lemma 6. By that lemma, each tile value up to the given value can be reached from its predecessor in A, and each larger value cannot be reached.

The computation of $\mathrm{Total}(n, A)$ follows from Theorem 9. By that lemma, the tile values of any reachable position are individually dominated by the values in the reachable position that has one tile of each value $\mathrm{Single}(i, A)$ for i ranging from 1 to n. The sum in the formula gives the value of this position, which clearly obeys the stated recurrence. ◀

▶ **Corollary 12** (termination if and only if gaps are unbounded). *For every A, the values of* $\mathrm{Single}(n, A)$ *and* $\mathrm{Total}(n, A)$ *are finite for all n (and the game* $\mathrm{AGG}(n, A)$ *necessarily terminates for all n) if and only if the gaps between consecutive members of A are not bounded in size.*

Proof. As a sum of values of Single, Total is finite if and only if Single is. Additionally, Total is strictly monotonically increasing, because it is always possible to add a single tile of value one to a reachable position in $\mathrm{AGG}(n - 1, A)$ and produce a higher-value reachable position in $\mathrm{AGG}(n, A)$. Therefore, for larger and larger values of n, the formula for $\mathrm{Single}(n, A)$ will require us to find correspondingly larger gaps in the sequence of values in A. This will be possible, leading to finite values of $\mathrm{Single}(n, A)$ for all n, if and only if A has gaps of unbounded size. ◀

4 Making change

The change-making problem involves making change for a given amount of money, using as few coins as possible from a given set of coin denominations. Most countries have coinage that allows the problem to be solved optimally by a greedy algorithm: to make change for a given amount of money x, first select the largest-valued coin whose value y is less than or equal to x, and then (if $x \neq y$) recursively solve the remaining change-making subproblem for the value $x - y$. However, greedy change-making is not always optimal. For instance, consider the situation of a cashier who is trying to make change in US money, for which the most commonly-used coin denominations are 1 cent (the penny), 5 cents (the nickel), 10 cents (the dime), and 25 cents (the quarter). To make change for 30 cents, the optimal choice would be the greedy choice, one quarter and one nickel. But if the change tray is out of nickels, so that the only coin values available are 1, 10, and 25 cents, the optimal choice would be three dimes, while the greedy algorithm would instead choose a quarter and five pennies, twice as many coins.

Optimal change-making is weakly NP-hard but has a pseudopolynomial time dynamic program that is often used as an example or an exercise in undergraduate algorithms classes [5, 7]. However, although there have also been studies on sets of coins that would lead to small solutions [24] or on counting distinct ways of making change [4], much of the research on change-making has focused on a different problem: for which coinage systems is the greedy algorithm optimal [3, 6, 11, 14, 20]? This can be tested in polynomial time [20].

A particularly simple test (the Magazine–Nemhauser–Trotter one-shot test) determines whether a system of coins has a stronger property, that if the coins are sorted by value from smallest to largest, every prefix of this sorted sequence forms a set of coins for which greedy change-making is optimal. For each prefix let x and y be the largest and second-largest coins in the prefix; then the one-shot test rounds x up to an integer multiple ky of y and applies the greedy change-making algorithm to this number ky. If it uses more than k coins, the greedy algorithm is suboptimal, but if every prefix uses this number of coins or fewer, then the greedy algorithm can be proven to be optimal for all prefixes [14]. Following Cowen et al. [6], we call a system of coins that passes this test *totally greedy*. Although the change-making problem is usually considered only for finite sets of coin denominations, the one-shot test and the prefix-greedy definition make sense equally well for infinite sets. For instance, the powers of two are totally greedy (the ith instance of the one-shot test uses one coin to represent the test value $2 \cdot 2^{i-1} = 2^i$) as are the Fibonacci numbers (the ith instance of the one-shot test uses two coins to represent the test value $2F_{i-1} = F_i + F_{i-3}$).

In connection with our analysis of abstract generalized 2048, we are interested in the behavior of the greedy algorithm on arbitrary coinage systems, regardless of whether the greedy algorithm is optimal for the system. The following quantity is of particular interest:

▶ **Definition 13** (hard-to-change inputs to the greedy algorithm)**.** For any integer $n \geq 0$ and set of positive integer coin values A, we define $\mathrm{GreedyCoins}(n, A)$ to be the smallest integer x that causes the greedy change-making algorithm to use at least n coins.

The following result is folklore; it is possible that it was first observed by Pillai in his 1930 study of greedy change-making for prime-number coin values [21] but we have been unable to obtain a copy of his paper to check.

▶ **Lemma 14** (recurrence for hard-to-change inputs)**.** *We may compute* $\mathrm{GreedyCoins}(n, A)$ *using the recurrence*

$$\mathrm{GreedyCoins}(n, A) = \mathrm{GreedyCoins}(n-1, A) + x,$$

where x is the smallest member of A such that the difference between x and the next-larger member of A exceeds $\mathrm{GreedyCoins}(n-1, A)$*, and with the base case* $\mathrm{GreedyCoins}(0, A) = 0$.

Proof. The greedy algorithm will use n or more coins on a given number s if and only if s has the form $t + u$ where t is a member of A, $t + u$ is less than the next larger member of A (so that the greedy algorithm begins by choosing t), and the greedy algorithm uses $n - 1$ or more coins on u (its recursive subproblem). The number $\mathrm{GreedyCoins}(n-1, A) + x$ has this form, with $t = x$ and $u = \mathrm{GreedyCoins}(n-1, A)$. It is the smallest number with this form, because any smaller value of u would not cause the greedy algorithm to use $n - 1$ or more coins on u, and any smaller value of t with the same or larger value of u would cause there to exist another member r of A in the range $t < r \leq t + u$, preventing the greedy algorithm from starting by choosing t. ◀

Table 1 Python code to generate the sequence of values Total(n, A) from a generator for sequence A, using only a constant number of additional integer variables.

```
def Total(A):
    single,total = 1,0
    for tile in A:
        while tile > single + total:
            total += single
            yield total
        single = tile
```

We are now ready to prove our min-max theorem relating 2048 to change-making:

▶ **Theorem 15** (equality of 2048 and greedy change-making)**.** *The maximum total value achieved in an n-cell abstract greedy 2048 game,* Total(n, A)*, equals the minimum value that would cause the greedy change-making algorithm to use n or more coins,* GreedyCoins(n, A)*.*

Proof. By Theorem 11 and Lemma 14, both of these numbers are computed by the same recurrence with the same base case. ◀

5 Specific sets of tile values

The Python code in Table 1 takes as input a generator for a sorted sequence A of tile values in an abstract generalized 2048 game (or of coin values in a greedy change-making problem), and returns a generator for the sequence of total tile values Total(n, A) achievable with $n = 1, 2, 3, \dots$ cells. It does so by computing, for each tile value in A, the gap between that value and the previous value, and when that gap is large enough using it to take a step in the recurrence for Total(n, A). As can be seen from the code, the total space necessary (beyond that for generating A) consists only of a constant number of integer variables. It is not possible to analyze the performance of this algorithm in terms of the variable n without knowing more about the behavior of gaps in the sequence A, but we can at least state that the time to generate all values Total(n, A) that are below some threshold value N is at most proportional to the time to generate all values in A below the same threshold.

For sequences that are totally greedy, the same algorithm will determine more strongly the smallest value that requires n terms to represent as a sum of sequence values (not just as a greedy sum). We ran this code using several different integer sequences A, to determine for each one its corresponding sequence of maximum achievable total game values Total(n, A). We identify each sequence using its code in the Online Encyclopedia of Integer Sequences (oeis.org), a string of the form Axxx where the x's are decimal digits. Although some of these sequences would be problematic for games that combine tile values only in pairs (because their tile values cannot be reached by such pairwise combinations), this is not an issue for our abstract generalized 2048 game, which allows combinations of more than two tiles at once.

A000040

This is the sequence of prime numbers, $2, 3, 5, 7, 11, \dots,$ in which we included also 1 (even though it is not prime) to make a valid set of tile or coin values. It is not totally greedy. When A is this sequence, Total(n, A) is Pillai's sequence [13, 21] A066352 of the numbers $1, 4, 27, 1354, 401429925999155061, \dots$. Because the gaps in the prime numbers grow so slowly, it has been estimated in the OEIS that the next number of this sequence would require hundreds of millions of digits.

A000045

This is the sequence of Fibonacci numbers, $1, 2, 3, 5, 8, 13, \ldots$, used in the 987 game. It is totally greedy. When A is this sequence, $\mathrm{Total}(n, A)$ is the sequence A027941 of numbers $F_{2n+1} - 1 = 1, 4, 12, 33, 88, \ldots$ of every other Fibonacci number, minus one.

A000079

This is the sequence of powers of two, $2^i = 1, 2, 4, 8, \ldots$, used in the 2048 game. It is totally greedy. When A is this sequence, $\mathrm{Total}(n, A)$ is the sequence of Mersenne numbers A000225, $2^{n-1} - 1 = 1, 3, 7, 15, \ldots$.

A000225

This is the sequence of Mersenne numbers, $M_i = 2^{i+1} - 1 = 1, 3, 7, 15, \ldots$. It is also totally greedy, because each prefix of the sequence passes the one-shot test according to the identity $3M_i = M_{i+1} + 2M_{i-1}$. When A is this sequence, $\mathrm{Total}(n, A)$ is the sequence A000325 of numbers $2^n - n = 1, 2, 5, 12, 27, \ldots$ which is not totally greedy ($3 \cdot 12 = 27 + 5 + 2 + 2$ is expanded to four coins, not three, by the greedy algorithm, failing the one-shot test).

A005153

This is the sequence of practical numbers $1, 2, 4, 6, 8, 12, 16, \ldots$ discussed in the introduction. It can be generated by using a variation of the sieve of Eratosthenes to generate the factorizations of each positive integer, and then using an efficient test of Stewart and Sierpinski [25, 26] to determine from each factorization whether each integer is practical. When A is this sequence, $\mathrm{Total}(n, A)$ is a sequence beginning $1, 3, 11, 191$, not in the OEIS. Because the gaps in the sequence of practical numbers are (like the gaps in the primes) slowly growing, the next number in the sequence should be quite large.

A003586

This is the sequence of 3-smooth numbers $1, 2, 3, 4, 6, 8, 9, \ldots$ (the numbers having only 2 or 3 as prime factors), discussed in the introduction. It is not totally greedy. When A is this sequence, $\mathrm{Total}(n, A)$ is the sequence A296840: $1, 5, 23, 185, 1721, 15545, 277689, \ldots$.

A029744

This is the sequence of numbers 2^i or $3 \cdot 2^i = 1, 2, 3, 4, 6, 8, 12, \ldots$ used in the game Threes. It is totally greedy. When A is this sequence, $\mathrm{Total}(n, A)$ is the sequence A002450 of numbers $(4^{n+1} - 1)/3 = 1, 5, 21, 85, 341, \ldots$.

A126684

This is the sequence of numbers $1, 2, 4, 5, 8, 10, 16, 17, 20, 21, 32, \ldots$ whose binary representations have either all even bit positions zero or all odd bit positions zero. It gives perhaps the most extreme example of the distinction between optimal and greedy change-making: for a system of coins with these values, any amount of change can be made with at most two coins, and the $\Theta(n^2)$ growth rate of this sequence is the fastest possible for this two-coin property. However, greedy change-making will typically use more than two coins. For instance, although 13 can be represented as the sum of two sequence members $8 + 5$, its greedy representation is $10 + 2 + 1$. When A is this sequence, $\mathrm{Total}(n, A)$ is the sequence A302757 of numbers $1, 3, 13, 55, 225, 907, 3637, \ldots$, which grows exponentially according to the recurrence $a_n = 4a_{n-1} + 2n - 5$.

The tile values $1, 2, 3, 5, 10, 20, 40, 80, \ldots$ in the game Fives are not listed in the OEIS, but the maximum achievable values $\mathrm{Total}(n, A)$ form the sequence A052549 of numbers $1, 4, 9, 19, 39, 79$. They have the formula $\lfloor 5 \cdot 2^{n-2} - 1 \rfloor$.

A similar analysis could be applied to many other sequences, yielding new sequences not already part of the OEIS. For instance, sequences A296840 and A302757 were added to the OEIS as a result of our investigations, not having been studied before.

6 Discussion

We have described an abstract version of the game 2048 that eliminates the geometry and other complicating factors of the game, allowing us to provide a complete analysis of our abstract game for any set of allowable tile values and any number of cells. We proved a min-max theorem equating the maximum total tile value that can be achieved in this game with the minimum value that would cause a greedy change-making algorithm, using coins of the same value as the tiles, to use the same number of coins as the number of cells in the game. Finally, we showed how to compute the values from this theorem by a streaming algorithm that uses only a constant number of integer variables beyond the requirements of generating the tile value sequence itself, and used our implementation to compute the sequences of maximum game values for several choices of allowable tile value sets.

It would be of interest to understand in more detail for which non-abstract 2048-like games this analysis is tight or nearly tight, and for which it fails to capture the game dynamics and produces a bound on the total game value that is large compared to the actual achievable value. For instance, experience with 2048 and 987 suggests that, in those games, a strategy close to that of the abstract game can usually be followed, leading to total game values similar to what could be achieved in the abstract game. On the other hand, in Threes, the inability to add some pairs of game tiles such as $1 + 3$, even when the sum would be another allowable tile value, may cause this game's maximum achievable value to be closer to 2^n than to the $(4^{n+1} - 1)/3$ formula for the total score achievable on the corresponding abstract game. We leave such questions open for future research.

Additionally, some variants of 2048 are not amenable to our analysis. These include 2048 Circle of Fifths, a game based on the circle of fifths in music theory whose tile values involve modular arithmetic [9], and 2048 Numberwang, in which the tile combinations that are allowed on each move vary randomly [8]. Developing a theoretical analysis of these games could be fun.

References

1 Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. On the complexity of slide-and-merge games. Electronic preprint arxiv:1501.03837, 2015.

2 Ahmed Abdelkader, Aditya Acharya, and Philip Dasler. 2048 without new tiles is still hard. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 1:1–1:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.1.

3 Anna Adamaszek and Michal Adamaszek. Combinatorics of the change-making problem. *Eur. J. Comb.*, 31(1):47–63, 2010. doi:10.1016/j.ejc.2009.05.002.

4 Terry Beyer and D. F. Swinehart. Number of multiply-restricted partitions [A1] (algorithm 448). *Commun. ACM*, 16(6):379, 1973. doi:10.1145/362248.362275.

5 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009. Problem 16-1, p. 446.

6 L. J. Cowen, Robert Cowen, and Arthur Steinberg. Totally greedy coin sets and greedy obstructions. *Electronic Journal of Combinatorics*, 15(1):RP90, 2008. URL: https://www.combinatorics.org/Volume_15/Abstracts/v15i1r90.html.

7 Michael T. Goodrich and Roberto Tamassia. *Algorithm Design and Applications*. Wiley, 2015. Exercise A-12.1, p. 349.

8 Lou Huang. 2048 Numberwang. Web applet. URL: https://louh.github.io/2048-numberwang/.

9 Caleb Hugo. 2048 Circle of Fifths. Web applet. URL: https://calebhugo.com/musical-games-interact-with-sound/2048-circle-of-fifths/.

10 Wojciech Jaśkowski. Mastering 2048 with delayed temporal coherence learning, multi-stage weight promotion, redundant encoding and carousel shaping. *IEEE Transactions on Computational Intelligence and AI in Games*, 2017. doi:10.1109/TCIAIG.2017.2651887.

11 Dexter Kozen and Shmuel Zaks. Optimal bounds for the change-making problem. *Theor. Comput. Sci.*, 123(2):377–388, 1994. doi:10.1016/0304-3975(94)90134-1.

12 Stefan Langerman and Yushi Uno. Threes!, fives, 1024!, and 2048 are hard. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 22:1–22:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.22.

13 Florian Luca and Ravindranathan Thangadurai. On an arithmetic function considered by Pillai. *Journal de Théorie des Nombres de Bordeaux*, 21(3):693–699, 2009. URL: https://jtnb.cedram.org/item?id=JTNB_2009__21_3_693_0.

14 M. J. Magazine, G. L. Nemhauser, and L. E. Trotter, Jr. When the greedy solution solves a class of knapsack problems. *Operations Research*, 23(2):207–217, 1975. URL: https://www.jstor.org/stable/169525.

15 Kiminori Matsuzaki. Developing a 2048 player with backward temporal coherence learning and restart. In Mark H. M. Winands, H. Jaap van den Herik, and Walter A. Kosters, editors, *Advances in Computer Games - 15th International Conferences, ACG 2017, Leiden, The Netherlands, July 3-5, 2017, Revised Selected Papers*, volume 10664 of *Lecture Notes in Computer Science*, pages 176–187. Springer, 2017. doi:10.1007/978-3-319-71649-7_15.

16 Rahul Mehta. 2048 is (PSPACE) hard, but sometimes easy. Electronic preprint arxiv:1408.6315, 2014.

17 Giuseppe Melfi. On two conjectures about practical numbers. *J. Number Theory*, 56(1):205–210, 1996. doi:10.1006/jnth.1996.0012.

18 Todd W. Neller. Pedagogical possibilities for the 2048 puzzle game. *Journal of Computing Sciences in Colleges*, 30(3):38–46, 2015. URL: https://dl.acm.org/citation.cfm?id=2675327.2675335.

19 Kazuto Oka and Kiminori Matsuzaki. Systematic selection of n-tuple networks for 2048. In Aske Plaat, Walter A. Kosters, and H. Jaap van den Herik, editors, *Computers and Games - 9th International Conference, CG 2016, Leiden, The Netherlands, June 29 - July 1, 2016, Revised Selected Papers*, volume 10068 of *Lecture Notes in Computer Science*, pages 81–92. Springer, 2016. doi:10.1007/978-3-319-50935-8_8.

20 David Pearson. A polynomial-time algorithm for the change-making problem. *Oper. Res. Lett.*, 33(3):231–234, 2005. doi:10.1016/j.orl.2004.06.001.

21 S. S. Pillai. An arithmetical function concerning primes. *Annamalai University Journal*, pages 159–167, 1930. As cited by Luca and Thangadurai [13].

22 Rebecca S. Portnoff, Linda N. Lee, Serge Egelman, Pratyush Mishra, Derek Leung, and David A. Wagner. Somebody's watching me?: Assessing the effectiveness of webcam

indicator lights. In Bo Begole, Jinwoo Kim, Kori Inkpen, and Woontack Woo, editors, *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems, CHI 2015, Seoul, Republic of Korea, April 18-23, 2015*, pages 1649–1658. ACM, 2015. doi:10.1145/2702123.2702164.

23 Philip Rodgers and John Levine. An investigation into 2048 AI strategies. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–2. IEEE, 2014. doi:10.1109/CIG.2014.6932920.

24 Jeffrey Shallit. What this country needs is an 18¢ piece. *The Mathematical Intelligencer*, 25(2):20–23, 2003. doi:10.1007/bf02984830.

25 Wacław Sierpiński. Sur une propriété des nombres naturels. *Annali di Matematica Pura ed Applicata*, 39(1):69–74, 1955. doi:10.1007/BF02410762.

26 B. M. Stewart. Sums of distinct divisors. *American Journal of Mathematics*, 76(4):779–785, 1954. doi:10.2307/2372651.

27 Marcin Grzegorz Szubert and Wojciech Jaskowski. Temporal difference learning of n-tuple networks for the game 2048. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26-29, 2014*, pages 1–8. IEEE, 2014. doi:10.1109/CIG.2014.6932907.

28 H. Jaap van den Herik. Five new games. *ICGA Journal*, pages 129–130, September 2014. URL: https://icga.leidenuniv.nl/wp-content/uploads/2015/04/September-2014.pdf.

29 A. Weingartner. Practical numbers and the distribution of divisors. *The Quarterly Journal of Mathematics*, 66(2):743–758, 2015. doi:10.1093/qmath/hav006.

30 Kun-Hao Yeh, I-Chen Wu, Chu-Hsuan Hsueh, Chia-Chuan Chang, Chao-Chin Liang, and Han Chiang. Multistage temporal difference learning for 2048-like games. *IEEE Trans. Comput. Intellig. and AI in Games*, 9(4):369–380, 2017. doi:10.1109/TCIAIG.2016.2593710.

Pick, Pack, & Survive: Charging Robots in a Modern Warehouse based on Online Connected Dominating Sets

Heiko Hamann
Institute of Computer Engineering, University of Lübeck, Germany
https://www.iti.uni-luebeck.de
hamann@iti.uni-luebeck.de

Christine Markarian
Heinz Nixdorf Institute, Paderborn University, Germany
https://www.uni-paderborn.de
christine.markarian@upb.de

Friedhelm Meyer auf der Heide
Heinz Nixdorf Institute, Paderborn University , Germany
https://www.uni-paderborn.de
fmadh@upb.de

Mostafa Wahby
Institute of Computer Engineering, University of Lübeck, Germany
https://www.iti.uni-luebeck.de
mostafa.wahby@uni-luebeck.de

Abstract

The modern warehouse is partially automated by robots. Instead of letting human workers walk into shelfs and pick up the required stock, big groups of autonomous mobile robots transport the inventory to the workers. Typically, these robots have an electric drive and need to recharge frequently during the day. When we scale this approach up, it is essential to place recharging stations strategically and as soon as needed so that all robots can survive. In this work, we represent a warehouse topology by a graph and address this challenge with the *Online Connected Dominating Set* problem (OCDS), an online variant of the classical *Connected Dominating Set* problem [10]. We are given an undirected connected graph $G = (V, E)$ and a sequence of subsets of V arriving over time. The goal is to grow a connected subgraph that dominates all arriving nodes and contains as few nodes as possible. We propose an $\mathcal{O}(\log^2 n)$-competitive randomized algorithm for OCDS in general graphs, where n is the number of nodes in the input graph. This is the best one can achieve due to Korman's randomized lower bound of $\Omega(\log n \log m)$ [14] for the related *Online Set Cover* problem [2], where n is the number of elements and m is the number of subsets. We also run extensive simulations to show that our algorithm performs well in a simulated warehouse, where the topology of a warehouse is modeled as a randomly generated geometric graph.

2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases connected dominating set, online algorithm, competitive analysis, geometric graph, robot warehouse, recharging stations

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.22

Acknowledgements This work was partially supported by the Federal Ministry of Education and Research (BMBF) as part of the project 'Resilience by Spontaneous Volunteers Networks for

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 22; pp. 22:1–22:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Coping with Emergencies and Disaster' (RESIBES) - grant number 13N13955 to 13N13957 and the German Research Foundation (DFG) within the Collaborative Research Center 'On-The-Fly Computing' (SFB 901).

1 Introduction

Ever wondered what happens in a warehouse when Amazon orders are placed? Modern warehouses (so-called pick-pack-and-ship warehouses [20]) are nowadays run by hundreds of mobile robots (cf. swarm robotics [4]). A popular example is the Kiva system [6, 11] that is used by the Amazon.com corporation. When orders are placed, robots bring inventory to warehouse workers while workers stay stationary. Since robots need to act autonomously, the system is considered as a multi-agent system that requires rather complex path planning and resource allocation. In this paper, we focus on the specific issue of recharging robots. The robots have rechargeable lead-acid batteries that need to be recharged frequently throughout the day. In the case of Kiva robots, for example, they recharge every 55 minutes for five minutes. Once a robot detects a low battery level, it drives autonomously to the next charging station. Such warehouses can have several 10,000 square meters and do 200,000 picks per day [11]. When one wants to scale up, it will get important to position the recharging stations strategically, otherwise the system's efficiency will decrease or robots will even run out of energy before they reach a charging station. In addition, extreme differences in the distribution shipping volumes per day are typical for Internet retailers, for example, before Christmas. Hence, it is important to be able to scale up on demand.

In a hypothetical scenario, one can imagine a huge warehouse where certain areas are more busy during the week than others. Hence, it would be useful to have movable charging stations and to position them *on demand* in the respective busy areas. In addition, one can assume that any two charging stations should not be too far apart from each other, because robots may be required to move from any of these busy areas to any other (i.e., there may be a massive task allocation going on, which is however beyond this paper's scope). As robots may not always be fully charged when they start their travel and because we want to be scalable to huge warehouses, robots may even require to recharge on their move. So besides positioning charging stations at busy areas we also may position charging stations in between busy areas as kind of relay stations to guarantee reachability of all busy areas. We represent the warehouse topology as a graph whose nodes represent potential locations for recharging stations and busy areas. Recharging stations are placed sequentially on the nodes depending on current demands. The edges in the graph represent distances that can safely be covered by robots with a battery of average charge. A sequence of requests arrives over time such that a request consists of a subset of nodes representing busy areas that require a charging station nearby. These areas cannot be known in advance and must be provided with a charging station as soon as revealed. Charging stations need to be placed on a small subset of nodes such that all busy areas have a nearby station. Moreover, to guarantee reachability of all busy areas, we require that the subset of nodes induces a connected subgraph (see Fig. 1). The goal is to serve all requests upon their arrival by placing *as few* recharging stations as possible and without knowing future busy areas. At the core of this scenario we have a complex online optimization problem, the *Online Connected Dominating Set* problem (OCDS), defined as follows.

▶ **Definition 1.** (Online Connected Dominating Set problem - OCDS) Given an undirected connected graph $G = (V, E)$ and a sequence of subsets of V arriving over time. In each step, nodes from V are revealed and need to be dominated by a connected subgraph of G - a

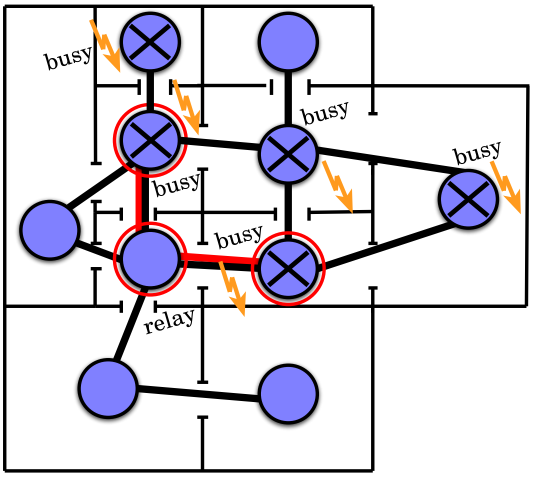


Figure 1 Warehouse topology modeled as a graph. Nodes represent busy areas and potential locations for recharging stations. Three recharging stations are placed (in red) one of which acts as a relay station.

subset is dominated if each node in it is either in the subgraph or adjacent to a node in the subgraph. OCDS asks to grow a connected subgraph containing as few nodes as possible without knowing future nodes in advance.

OCDS is an online variant of the classical *Connected Dominating Set* problem (CDS). Given a graph $G = (V, E)$, CDS asks to construct a minimum subset S of V such that each node in V is dominated and the subgraph induced by S is connected. CDS has been widely studied in both theory and practice, with various applications in wireless networks [21]. It is $\mathcal{NP}$-complete even in planar graphs [9]. For general graphs, it admits an $\mathcal{O}(\ln \Delta)$-approximation, where Δ is the maximum node degree of the input graph [10]. This is the best possible unless $\mathcal{NP} \subset DTIME(n^{\log n \log n})$ [10]. We propose a randomized online algorithm for OCDS and evaluate it using the classical notion of *competitive analysis*. Given an input sequence σ, let $\mathcal{C}_A(\sigma)$ and $\mathcal{C}_{OPT}(\sigma)$ denote the cost incurred by an algorithm A and an optimal offline algorithm OPT, respectively. Algorithm A is said to be c-*competitive* (or has competitive ratio c) if there exists a constant α such that $\mathcal{C}_A(\sigma) \leq c \cdot \mathcal{C}_{OPT}(\sigma) + \alpha$ for all input sequences σ. We show that our proposed algorithm is $\mathcal{O}(\log^2 n)$-competitive against an oblivious adversary, where n is the number of nodes in the input graph. This is the best one can achieve due to Korman's randomized lower bound for the related *Online Set Cover* problem (OSC) [2], an online variant of the classical *Set Cover* problem. OSC is defined as follows. Given a universe $\mathcal{U}$ of elements and a collection $\mathcal{S}$ of subsets of $\mathcal{U}$. In each step, elements from $\mathcal{U}$ are revealed and need to be covered by subsets from $\mathcal{S}$. OSC asks to cover all given elements while minimizing the total number of chosen subsets. Korman [14] has shown a randomized lower bound of $\Omega(\log n \log m)$ for OSC, where n is the number of elements and m is the number of subsets. It has been shown that the *Set Cover* problem can be reduced to CDS [8, 10, 15]. By a similar argumentation, a reduction from OSC to OCDS can be made, implying a randomized lower bound of $\Omega(\log^2 n)$ for OCDS. We also run extensive simulations to show that our algorithm performs well in a simulated warehouse, where the topology of a warehouse is modeled as a randomly generated geometric graph.

2 Related Work

In this section, we give an overview of literature related to online connected dominating sets and robot warehouses, respectively.

2.1 Online Connected Dominating Sets

While there are many works that address CDS and its variants in the offline setting [10, 21], only few consider the online setting. Eidenbenz [7] has studied an online variant of CDS, in which the input graph is restricted to a class (tree, unit disk graph, bounded degree graph) and is not known in advance. It is revealed over time such that in each step a node is either inserted or deleted. The goal is to maintain a connected subgraph that contains as few nodes as possible and dominates all present nodes. Eidenbenz has shown that a simple greedy approach attains a $(1 + \frac{1}{\mathcal{OPT}})$-competitive ratio in trees - where $\mathcal{OPT}$ is the cost of the optimal offline solution, an $(8 + \epsilon)$-competitive ratio in unit disk graphs - for arbitrary small $\epsilon > 0$, and b-competitive ratio in b-bounded degree graphs. Note that in OCDS, the input graph is given in advance (offline) and only the nodes that need to be dominated are revealed over time (online). For the *Online Set Cover* problem defined earlier, Alon *et al.* [2] have proposed an $\mathcal{O}(\log n \log m)$-competitive deterministic algorithm, where m is the number of sets and n is the number of elements. They have also shown a nearly matching deterministic lower bound of $\Omega(\frac{\log n \log m}{\log \log n + \log \log m})$ for interesting values of m and n. Naor *et al.* [17] have studied an online variant of the classical *Steiner Tree* problem with weighted nodes and edges, the *Online Node-Weighted Steiner Tree* problem (ONWST), defined as follows.

▶ **Definition 2.** (Online Node-Weighted Steiner Tree problem - ONWST) Given a graph $G = (V, E)$ with weighted nodes and edges. In each step, nodes from V (called terminals) are revealed and need to be connected to each other. ONWST asks to grow a subgraph S (called a Steiner tree) that connects all terminals while minimizing the total cost of edges and nodes in S.

Naor *et al.* [17] have given an $\mathcal{O}(\log n \log^2 k)$-competitive randomized algorithm for ONWST, where k is the number of terminals and n is the number of nodes in the input graph. A special case of ONWST in which all edges have cost 0 and all nodes have cost 1 can be reduced to OCDS by setting the nodes to be dominated as terminals. This variant will appear in the analysis of our algorithm in Section 4.

2.2 Robot Warehouses

Even though it may seem a rather specialized problem of positioning charging stations for multi-robot and swarm systems, there is a rich literature about this problem. Kannan *et al.* define the *Autonomous Recharging Problem* (ARP) as the problem of planning and coordinating when, where, and how to recharge robots [13]. They consider both static and mobile charging stations and how to find recharging schedules to maximize efficiency of the system. Couture-Beil and Vaughan argue that suboptimal positioning of the charging station may cause spatial interference. Therefore, they study an adaptive mobile charging station [5] where the charging station itself is a mobile robot. They argue that in a dynamic task the correct and adaptive placement of the charging station is even more important. A similar approach is studied by Arvin *et al.* for recharging a robot swarm with a mobile charging station [3]. Kamagaew *et al.* discuss the problem of how to switch from a central approach to a multiplicity of small self-organizing transport units in the warehouse [12]. They study a decentralized control system for autonomous vehicle swarms and how methods of swarm intelligence can help. In summary, the *Autonomous Recharging Problem* and in particular the optimal positioning of charging stations in multi-robot systems is relevant and currently investigated intensively.

3 Online Algorithm

In this section, we propose an online algorithm for OCDS. Given an undirected connected graph $G = (V, E)$. In each step t, a subset $D_t \subseteq V$ of nodes is given. Let S_t denote the set of nodes selected by the algorithm in step t. Let $S := \sum_t S_t$, $\forall t$ be the solution set and $D := \sum_t D_t$, $\forall t$ be the demand set. We fix a cost $c_j = 1$ to each $i \in V$. The algorithm runs in two phases. In the first phase, it dominates each node in D_t with a subset $S'_t \subseteq V$. In the second phase, it connects the nodes in S'_t with an additional subset $S''_t \subseteq V$, forming the set $S_t = S'_t \cup S''_t$. We call the set of nodes that can dominate i the *candidates* of i and denote it as Q_i. The algorithm uses a *randomized rounding* approach commonly used in designing online algorithms [17], in both of its phases. In Phase 1, to dominate a given node i, the algorithm buys fractions of i's candidates until they sum up to 1. These fractions are then rounded using a randomized process in attempt to add at least one candidate into the solution. If the latter does not happen, the algorithm arbitrary adds one of the candidates into the solution. In Phase 2, for each node j selected by the algorithm in Phase 1, the algorithm chooses a representative node k from D that is dominated by j and connects k to the current solution S, as follows. The *minimum cut value* (or *maximum flow*) between a node k and a set S is the smallest total weight of edges which if removed would disconnect k from S. These edges form a *minimum cut*. The algorithm transforms the weights of the graph from the nodes to the edges such that the weight of an edge (u, v) is set to $\min\{w_u, w_v\}$, where w_u and w_v are the weights of u and v, respectively. If either u or v is in S, the weight of edge (u, v) is set to the weight of the one not in S. It then constructs a minimum cut C between k and S by running the algorithm by Schroeder *et al.* for undirected connected edge-weighted graphs [19]. For an edge $(u, v) \in C$ and $w_u < w_v$, u is called a *minimum cut node*. As long as the maximum flow between k and S is less than 1, the algorithm constructs a minimum cut and increases the weights of the corresponding minimum cut nodes. Then, it rounds these weights using a randomized process in attempt to add at least one path connecting k to S into the solution. If the latter does not happen, the algorithm adds a cheapest path that connects k to S into the solution. For **Phase 1**, we maintain a fraction f_i to each node $i \in V$, initially set to zero and non-decreasing throughout the algorithm. We define a random variable μ as $\mu := \min\{X_{(q)}\}$ such that $2\lceil\log(n+1)\rceil$ independent random variables $X_{(q)}$ are distributed uniformly in the interval $[0, 1]$ and $1 \leq q \leq 2\lceil\log(n+1)\rceil$. As for **Phase 2**, we maintain a weight w_i to each $i \in V$, initially set to zero and non-decreasing throughout the algorithm. We define a random variable μ' as $\mu' := \min\{X_{(q)}\}$ such that $2\lceil\log(n+1)\rceil$ independent random variables $X_{(q)}$ are distributed uniformly in the interval $[0, 1]$ and $1 \leq q \leq 2\lceil\log(n+1)\rceil$. The two phases of the algorithm are depicted in **Algorithm 1** below. Fig. **??** shows the result of a two-step run on a randomly generated connected graph of 10 nodes.

4 Competitive Analysis

We dedicate this section to showing that the algorithm above is $\mathcal{O}(\log^2 n)$-competitive for OCDS, where n is the number of nodes in the input graph. Recall that $\Omega(\log^2 n)$ is a lower bound for OCDS [14].

Algorithm 1

```
Phase 1. For each $i \in D_t$ not dominated by some node in $S \cup S'_t$,
    Step 1: while $\sum_{j \in Q_i} f_j < 1$,
                for each $j \in Q_i$: $f_j = f_j \cdot (1 + 1/c_j) + \frac{1}{|Q_i| \cdot c_j}$
    Step 2: add $j \in Q_i$ to $S'_t$ if $f_j > \mu$
    Step 3: if $i$ is not dominated by some node in $S'_t$, add an arbitrary $j \in Q_i$ to $S'_t$
Phase 2. For each $j \in S'_t$, if it is adjacent to some node in $S$, add $j$ to $S$, else,
    Step 1: choose a node $k$ from the set $D_t$ dominated by $j$ and add it to $S''_t$
    Step 2: if $k$ is not connected to $S$,
      While the maximum flow between $k$ and $S$ is not 1,
       - Construct a minimum cut and select the minimum cut nodes $K \subseteq V \setminus S$
       - For each $i \in K$, set $w_i = w_i \cdot (1 + 1/c_i) + \frac{1}{|K| \cdot c_i}$
      Add $v \in V$ to $S''_t$ if $w_v > \mu'$
      If $k$ is still not connected via nodes in $S''_t$, choose a shortest path connecting $k$ to $S$
and add its nodes to $S''_t$
    Step 3: Add $j$ and the nodes in $S''_t$ that connect $j$ to $S$, to $S$
```

▶ **Lemma 3.** *The expected cost C_1 of the algorithm in Phase 1 is at most $\mathcal{O}(\log^2 n) \cdot \mathcal{OPT}$, where $\mathcal{OPT}$ is the cost of the optimal offline solution.*

Proof. Let C_1 be the expected cost of the algorithm in Phase 1 and let $\mathcal{OPT}$ be the cost of the optimal offline solution. For any $j \in V$, the probability that the algorithm adds j into the solution is the probability that $f_j > \mu$, which is at most $2\log(n+1) \cdot f_j$. Adding up over all $j \in V$, the expected cost C_1 of the algorithm will be at most:

$$\sum_{j \in V} 2\log(n+1) \cdot f_j = 2\log(n+1) \cdot \sum_{j \in V} f_j \tag{1}$$

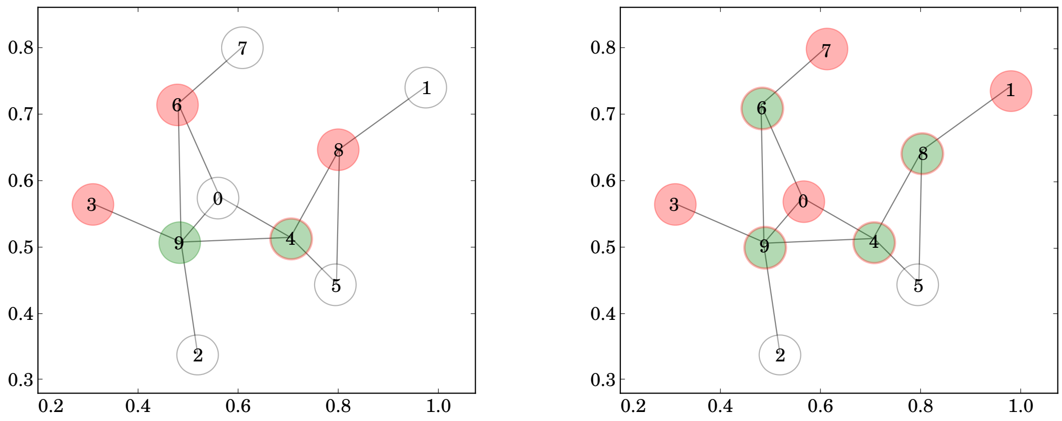
Next we bound $\sum_{j \in V} f_j$. Whenever we want to dominate a node $i \in V$ not yet dominated, we increase the fraction corresponding to each of its $|Q_i|$ candidates. The fraction f_j of each candidate $j \in Q_i$ is increased by $\left(\frac{f_j}{c_j} + \frac{1}{|Q_i| \cdot c_j}\right)$. Summing up over all i's candidates, we get an overall fractional increase of:

$$\sum_{j \in Q_i} \left(\frac{f_j}{c_j} + \frac{1}{|Q_i| \cdot c_j}\right) \leq 2 \tag{2}$$

The above inequality holds since $\sum_{j \in Q_i} f_j \leq 1$ before any fractional increase and $c_j = 1$: $\forall j \in V$. An optimal solution must contain at least one node. Let us fix any such node $p \in Q_i$ ($\mathcal{OPT} \geq c_p$). The fraction f_p corresponding to p becomes at least 1 after at most $\mathcal{OPT} \cdot \log |Q_i|$ fractional increases and hence no further fractional increases can be made. With this observation together with inequality 2, we conclude that:

$$\sum_{j \in V} f_j \leq \mathcal{OPT} \cdot 2 \cdot \log(\Delta + 1) \tag{3}$$

The above inequality holds since $|Q_i| \leq \Delta + 1$, where Δ is the maximum number of nodes adjacent to any node in V. So far we have measured the cost of the algorithm during the first two steps. Equations 1 and 3 yield a cost of $\mathcal{O}(\log^2 n) \cdot \mathcal{OPT}$. It remains to measure the additional cost incurred by Step 3 of Phase 1, which is necessary to guarantee a feasible



First time step.

Second time step.

Figure 2 Two-step run of **Algorithm 1** on randomly generated connected graph with $|V| = 10$. Nodes in red or with red border represent demand nodes and nodes in green represent solution nodes.

solution. For a single $1 \le q \le 2\lceil \log(n+1) \rceil$, the probability that a node i is not covered is at most:

$$\prod_{j \in Q_i} (1 - f_j) \le e^{-\sum_{j \in Q_i} f_j} \le 1/e$$

The last inequality holds because the algorithm guarantees in the first step that $\sum_{j \in Q_i} f_j \ge 1$. Hence, the probability that i is not covered, for all $1 \le q \le 2\lceil \log(n+1) \rceil$, is at most $1/n^2$. The additional expected cost for each of the at most n nodes is then upper bounded by $n \cdot 1/n^2 \cdot \mathcal{OPT}$, since the cost of adding one additional node is clearly less than $\mathcal{OPT}$. Therefore, we conclude that $C_1 \le \mathcal{O}(\log^2 n) \cdot \mathcal{OPT}$. ◀

▶ **Lemma 4.** *The expected cost C_2 of the algorithm in Phase 2 is at most $C_1 + \mathcal{O}(\log^2 n) \cdot \mathcal{OPT}$, where $\mathcal{OPT}$ is the cost of the optimal offline solution and C_1 is the expected cost of the algorithm in Phase 1.*

Proof. in Appendix A. ◀

Adding C_1 and C_2 from Lemma 3 and Lemma 4, respectively, ultimately leads to the theorem below.

▶ **Theorem 5.** *There is an optimal $\mathcal{O}(\log^2 n)$-competitive randomized algorithm for the Online Connected Dominating Set problem (OCDS).*

5 Simulation Study

In this section, we show that our proposed algorithm for OCDS performs well in a simulated warehouse, where the topology of a warehouse is modeled as a randomly generated connected geometric graph.

Since no other algorithm has been proposed for OCDS in the literature, the only algorithm we could compare to is offline, that is, an algorithm for CDS. Since the latter is $\mathcal{NP}$-complete, the comparison is made against an optimal $\mathcal{O}(\ln \Delta)$-approximation algorithm for CDS, based on a greedy approach, following Guha *et al.* [10]. Recall that an input to CDS is a graph $G = (V, E)$ in which all nodes need to be dominated and to which the algorithm reacts once.

In OCDS, we are given a graph $G = (V, E)$ and the nodes to be dominated are given in steps. The algorithm needs to react to each step without knowing about future steps. The comparison is made at the final step, after which the algorithm had accumulated its solution over the steps. Without loss of generality, we assume all nodes at this step have been asked to be dominated in at least one of the steps. We perform our simulation study in five different settings, each with different number of nodes (i.e., potential locations for recharging stations): $|V| \in \{50, 100, 150, 200, 250\}$. For each setting, we perform 100 runs. Since the algorithm is randomized, we run it 10 times for each instance and observe its mean, best, and worst case performance. In each simulation run, we generate a connected geometric graph $G = (V, E)$ whose nodes are placed uniformly at random in a unit square Euclidean plane (see Fig. 2). The connectivity threshold r is set to a small value of 0.17 to provide graphs with minimal number of edges. A value below 0.17 decreases the chance to generate geometric graphs that are connected. The input to the approximation algorithm is $G = (V, E)$ whereas the input to the online algorithm is composed of offline and online parts. The offline part is $G = (V, E)$ and the online part is the sequence of subset of nodes revealed over time. In each step t, a subset D_t of nodes is revealed (i.e., the current demand of recharging stations at step t). The cardinality of D_t is uniformly sampled from the interval $\left[0, \frac{|V| - \sum_{i=1}^{i=t-1} |D_i|}{2}\right]$. The subset D_t excludes previously given nodes and is sampled randomly.

Results. The boxplots in Fig. 3 show the performance of the online algorithm and the offline approximation algorithm for $|V| = 50$, 100, 150, 200, and 250. For each $|V|$, 100 instance graphs are generated.

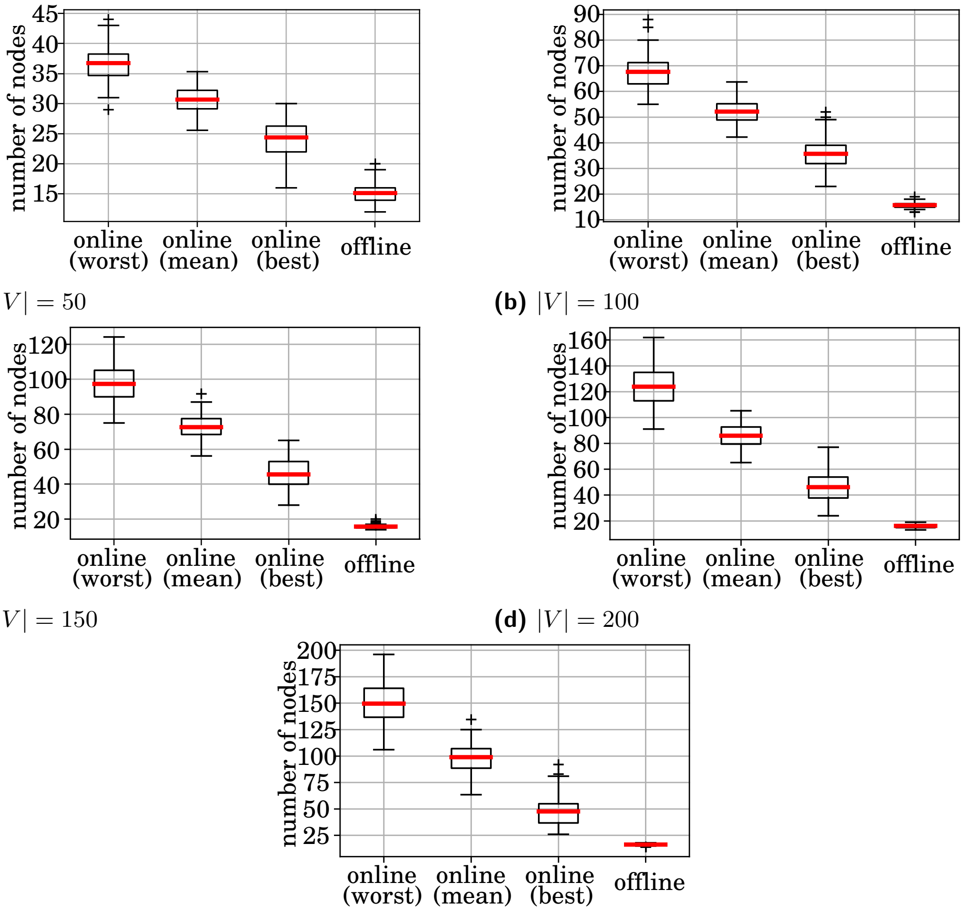
- The online algorithm is run 10 times for each instance graph: its *mean*, *best*, and *worst* case performance for each of the 100 instances are recorded.
- The offline approximation algorithm is run for each instance graph: its performance for each of the 100 instances is recorded.
- The datasets represented by the boxplots shown in Fig. 3 are all pairwise statistically significantly different (i.e., all p-values ≤ 0.05 based on Wilcoxon signed-rank test).

We define two performance measures:

1. the *percentage difference*, which is the number of nodes the online algorithm outputs *more*, in comparison to the offline algorithm, in percentage.
2. the *average competitive ratio*, which is the ratio of the number of nodes outputted by the online algorithm to that by the offline approximation algorithm.

Table 1 shows the two performance measures for each $|V|$ in the mean, best, and worst cases, evaluated by taking the average over all 100 instance graphs. Notice that the percentage difference never exceeds 51.08% in the worst case. For $|V| = 50$, it is as small as 11.10% in the best case. Moreover, as $|V|$ grows, the percentage difference does not increase significantly. Instead, it sometimes gets smaller - for instance, in the mean case as $|V|$ grows from 150 to 250 (from 34.25% to 32.15% to 30.84%). One reasoning for the latter might be the following. Recall that the connectivity threshold r is set to 0.17 for all $|V|$ and the nodes are placed uniformly at random in a bounded region. Thus the generated graphs become denser as $|V|$ grows, resulting in less complex solutions.

In terms of average competitive ratio, notice that the online algorithm's worst output is at most 6.83 times the output of the offline algorithm. Moreover, for $|V| = 50$, the online algorithm is nearly optimal with an average competitive ratio of 1.63 and 1.29 in the mean and best cases, respectively.



(a) $|V| = 50$

(b) $|V| = 100$

(c) $|V| = 150$

(d) $|V| = 200$

(e) $|V| = 250$

Figure 3 Boxplots showing the performance of the online algorithm (mean, best, and worst cases) and the offline algorithm for $|V| = 50$, 100, 150, 200, and 250. All datasets are pairwise statistically significantly different ($p \leq 0.05$, Wilcoxon signed-rank test).

6 Open problems

We have presented a provably optimal online approach to positioning recharging stations in a robot warehouse. Partially automated robot warehouses are state-of-the-art and in the future they will need to scale up. The problem of correctly placing charging stations is highly relevant and intensively studied. Efficient online algorithms for placing charging stations are essential to ensure scalability and efficiency. This work has been a small attempt towards this goal. There is certainly much more to do. As a first next step, it would be interesting to target better competitive ratios for OCDS in restricted graph classes, e.g., by employing properties of geometric graphs. Another important direction is to consider stations that do not serve forever but are renewed whenever needed. A related model is Meyerson's *leasing model* [16], that has been studied in the context of many optimization problems such as the *Online Set Cover* problem [1], in which sets can cover elements for limited duration and costs are incurred accordingly. Furthermore, our simulation results show that, even without knowing the demands of the day, an efficient placement of charging stations can be done using our algorithm. Moreover, our random-geometric-graph based simulated

Table 1 Percentage difference and average competitive ratio for $|V| = 50$, 100, 150, 200, and 250 in the mean, best, and worst cases.

Mean Performance	$\|V\|$	Percentage Difference	Average Competitive Ratio
	50	23.74	1.63
	100	31.37	2.51
	150	34.25	3.43
	200	32.15	3.96
	250	30.84	4.52
Best Performance			
	50	11.10	1.29
	100	14.96	1.72
	150	16.32	2.16
	200	12.21	2.12
	250	10.30	2.18
Worst Performance			
	50	35.78	1.95
	100	46.91	3.26
	150	50.79	4.61
	200	51.09	5.71
	250	51.08	6.83

warehouse arguably generalizes topologies of warehouses and the modeled temporal evolution of incoming demands is based on rather rough assumptions. Still, it would be interesting to extend our simulation study to include data acquired from actual Internet retailers. Therefore, the model would be refined so that it resembles actual evolutions of demands during the day or week in automated warehouses. Also an application to the domain of swarm robotics for any scenario, such as collective transport, collective construction, etc., may be possible and advantageous either with manually placed or autonomous mobile charging stations [3, 4]. We have also skipped the required task allocation for the robots in this study. An interesting extension hence can be a combined analysis of task allocation algorithms together with the challenge of positioning charging stations. Besides central task allocation algorithms, there are decentralized approaches to task allocation that don't require global information and scale up well [18].

References

1 Sebastian Abshoff, Peter Kling, Christine Markarian, Friedhelm Meyer auf der Heide, and Peter Pietrzyk. Towards the price of leasing online. *J. Comb. Optim.*, 32(4):1197–1216, 2016. doi:10.1007/s10878-015-9915-5.

2 Noga Alon, Baruch Awerbuch, Yossi Azar, Niv Buchbinder, and Joseph Naor. The online set cover problem. In Lawrence L. Larmore and Michel X. Goemans, editors, *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA*, pages 100–105. ACM, 2003. doi:10.1145/780542.780558.

3 Farshad Arvin, Khairulmizam Samsudin, and Abdul Rahman Ramli. Swarm robots long term autonomy using moveable charger. In *Future Computer and Communication, 2009. ICFCC 2009. International Conference on Future Computer and Communication*, pages 127–130. IEEE, 2009.

4 Manuele Brambilla, Eliseo Ferrante, Mauro Birattari, and Marco Dorigo. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013. `doi:10.1007/s11721-012-0075-2`.

5 Alex Couture-Beil and Richard T. Vaughan. Adaptive mobile charging stations for multi-robot systems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, pages 1363–1368. IEEE, 2009.

6 Raffaello D'Andrea. Guest editorial: A revolution in the warehouse: A retrospective on Kiva systems and the grand challenges ahead. *IEEE Transactions on Automation Science and Engineering*, 9(4):638–639, 2012.

7 Stephan Eidenbenz. Online Dominating Set and Variations on Restricted Graph Classes. Technical report, Department of Computer Science, ETH Zürich, 2002.

8 Uriel Feige. A threshold of ln n for approximating set cover (preliminary version). In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 314–318. ACM, 1996. `doi:10.1145/237814.237977`.

9 Michael R. Garey and David S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, New York, 1979.

10 Sudipto Guha and Samir Khuller. Approximation algorithms for connected dominating sets. *Algorithmica*, 20(4):374–387, 1998. `doi:10.1007/PL00009201`.

11 Eric Guizzo. Three engineers, hundreds of robots, one warehouse. *IEEE spectrum*, 45(7):26–34, 2008.

12 Andreas Kamagaew, Jonas Stenzel, Andreas Nettsträter, and Michael ten Hompel. Concept of cellular transport systems in facility logistics. In *5th International Conference on Automation, Robotics and Applications (ICARA)*, pages 40–45. IEEE, 2011.

13 Balajee Kannan, Victor Marmol, Jaime Bourne, and M. Bernardine Dias. The autonomous recharging problem: Formulation and a market-based solution. In *IEEE International Conference on Robotics and Automation (ICRA 2013)*, pages 3503–3510. IEEE, 2013.

14 Simon Korman. On the use of randomization in the online set cover problem. In *M.S. thesis, Weizmann Institute of Science*, 2005.

15 Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994. `doi:10.1145/185675.306789`.

16 Adam Meyerson. The parking permit problem. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2005), 23-25 October 2005, Pittsburgh, PA, USA, Proceedings*, pages 274–284. IEEE Computer Society, 2005. `doi:10.1109/SFCS.2005.72`.

17 Joseph Naor, Debmalya Panigrahi, and Mohit Singh. Online node-weighted steiner tree and related problems. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 210–219. IEEE Computer Society, 2011. `doi:10.1109/FOCS.2011.65`.

18 Giovanni Pini, Arne Brutschy, Gianpiero Francesca, Marco Dorigo, and Mauro Birattari. Multi-armed bandit formulation of the task partitioning problem in swarm robotics. In *8th Int. Conf. on Swarm Intelligence (ANTS)*, pages 109–120. Springer, 2012.

19 Jonatan Schroeder, André Guedes, and Elias P. Duarte Jr. Computing the minimum cut and maximum flow of undirected graphs. Technical report, Federal University of Paraná, Department of Informatics, 2004.

20 Peter R. Wurman, Raffaello D'Andrea, and Mick Mountz. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI magazine*, 29(1):9, 2008.

21 Jiguo Yu, Nannan Wang, Guanghui Wang, and Dongxiao Yu. Connected dominating sets in wireless ad hoc and sensor networks - A comprehensive survey. *Computer Communications*, 36(2):121–134, 2013. `doi:10.1016/j.comcom.2012.10.005`.

A Proof of Lemma 4

Proof. Let C_2 be the expected cost of the algorithm in Phase 2 and let $\mathcal{OPT}$ be the cost of the optimal offline solution. Phase 2 connects the nodes in $S't : \forall t$ that were not connected to S in Phase 1. For each such node u, Step 1 chooses a representative node k from D that is dominated by u. These representative nodes are then connected in Step 2 and form the terminals of a special case instance of the *Online Node-weighted Steiner Tree* problem (ONWST) (**Definition 2**), in which all nodes have cost 1 and all edges have cost 0. Let R be the set of these representative nodes. By similar arguments as in Phase 1, we show that Step 2 of Phase 2 admits an $\mathcal{O}(\log^2 n)$-competitive algorithm for this special case of ONWST. Let C_{St} be the expected cost of the Steiner tree constructed in Step 2 of Phase 2 and let $\mathcal{OPT}_{St}$ be the cost of an optimal Steiner tree. For any $i \in V$, the probability that the algorithm adds i into the solution is the probability that $w_i > \mu'$, which is at most $2\log(n+1) \cdot w_i$. Adding up over all $i \in V$, the expected cost C_{St} will be at most:

$$\sum_{i \in V} 2\log(n+1) \cdot w_i = 2\log(n+1) \cdot \sum_{i \in V} w_i \tag{4}$$

Next we bound $\sum_{i \in V} w_i$. To connect a node $k \in V$ not yet connected, the algorithm constructs a minimum cut. Let P_k denote the corresponding minimum cut nodes of such a cut. We increase the weight corresponding to each node in P_k. The weight w_i of each $i \in P_k$ is increased by $\left(\frac{w_i}{c_i} + \frac{1}{|P_k| \cdot c_i}\right)$. Summing up over all the nodes in P_k, we get an overall weight increase of:

$$\sum_{i \in P_k} \left(\frac{w_i}{c_i} + \frac{1}{|P_k| \cdot c_i}\right) \leq 2 \tag{5}$$

The above inequality holds since $\sum_{i \in P_k} w_i \leq 1$ before any weight increase and $c_i = 1$: $\forall i \in V$. An optimal Steiner tree must contain at least one node $p \in P_k$, in order to connect k. The weight w_p corresponding to p becomes at least 1 after at most $\mathcal{OPT}_{St} \cdot \log |P_k|$ weight increases and hence no further weight increases can be made. The weight of w_p becomes 1 and so it cannot belong to any other minimum cut chosen afterwards, since the algorithm constructs a minimum cut only if the maximum flow is less than one. The same argument holds for all minimum cuts chosen afterwards such that each can contain a distinct node in the optimal solution. With this observation together with inequality 5, we conclude that:

$$\sum_{i \in V} w_i \leq \mathcal{OPT}_{St} \cdot 2 \cdot \log n \tag{6}$$

The above inequality holds since $|P_k| \leq n$. Equations 4 and 6 yield a cost of $\mathcal{O}(\log^2 n) \cdot \mathcal{OPT}_{St}$. Let $i \in P_k$ be a node with weight $w_i > \mu'$. Now, we need to measure the additional cost incurred in the last part of Step 2, which is necessary to guarantee a feasible solution. Note that, we have that the weight of each node in a path connecting k to S must be at least the flow going through the path. For a single $1 \leq q \leq 2\lceil \log(n+1) \rceil$, the probability that a node k is not connected is at most:

$$\prod_{i \in P_k} (1 - w_i) \leq e^{-\sum_{i \in P_k} w_i} \leq 1/e$$

The last inequality holds since the algorithm guarantees that $\sum_{i \in P_k} w_i \geq 1$. Hence, the probability that k is not connected, for all $1 \leq q \leq 2\lceil \log(n+1) \rceil$, is at most $1/n^2$.

The additional expected cost for each of the at most n nodes is then upper bounded by $n \cdot 1/n^2 \cdot \mathcal{OPT}_{St}$, since the cost of adding one additional node is clearly less than $\mathcal{OPT}_{St}$. Therefore, we conclude that:

$$C_{St} \leq \mathcal{O}(\log^2 n) \cdot \mathcal{OPT}_{St} \tag{7}$$

Thus, we have $C_2 \leq C_1 + C_{St} \leq C_1 + \mathcal{O}(\log^2 n) \cdot \mathcal{OPT}_{St}$, where C_1 results from adding at most one node for each node in $S't$. Moreover, since an optimal offline solution for OCDS dominates all the given nodes and is connected, it forms a Steiner tree over the demand set D and consequently over the set R of representative nodes. Hence, $\mathcal{OPT}_{St} \leq \mathcal{OPT}$ and therefore:

$$C_2 \leq C_1 + \mathcal{O}(\log^2 n) \cdot \mathcal{OPT} \tag{8}$$

◀

Selection Via the Bogo-Method – More on the Analysis of Perversely Awful Randomized Algorithms

Markus Holzer
Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
holzer@informatik.uni-giessen.de

Jan-Tobias Maurer
Institut für Informatik, Universität Giessen
Arndtstr. 2, 35392 Giessen, Germany
jan.t.maurer@math.uni-giessen.de

Abstract

We continue our research on perversely awful randomized algorithms, which started nearly a decade ago. Based on the bogo-method we design a bogo-selection algorithm and variants thereof and analyse them with elementary methods. Moreover, practical experiments are performed.

2012 ACM Subject Classification Mathematics of computing → Combinatorics, Mathematics of computing → robability and statistics, Theory of computation → Algorithm design techniques

Keywords and phrases selection, bogo-method, combinatorial sums and series, inverse binomial coefficients, experimental result

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.23

Acknowledgements Thanks goes to Gerrit Eichner for his help on the statistical analysis of our experimental data.

1 Introduction

Bogo-sort, also known as Monkey-sort, is according to [6] the archetypical perversely awful randomized algorithm. It is the equivalent to repeatedly throwing a deck of cards in the air, picking them up at random, and then testing whether they are in order. The analysis of bogo-sort carried out in [3] shows that this algorithm, while having best-case expected running time as low as $\mathrm{O}(n)$, achieves an asymptotic expected running time as high as $\Omega(n \cdot n!)$ already in the average case. Although there are other sorting algorithms known such as, e.g., Bogobogo-sort, Evil-sort, etc., with even higher inefficiency, the concept of bogusness was not considered as an algorithm design principle in general. Thus, the question arises, what is the concept of bogusness? In fact the design of Bogo-sort as stated in the beginning nicely generalizes to what we call the bogo-method by repeatedly throwing a deck of cards in the air and picking up sufficient cards at random in order to test whether a certain property is met. At first glance this design principle looks crazy, but it can be nicely applied to other problems than sorting such as, e.g., the selection problem or the two-element sum problem. In the former case the property that has to be met is that the kth largest element of a given array $a[1 \ldots n]$ is exactly on the kth position and that all the elements to the left of $a[k]$ are smaller while all the element to the right of $a[k]$ are larger – we obviously require that $1 \leq k \leq n$ holds. Thus, it is obvious that we have to pick up all cards at random in order to verify the stated property.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 23; pp. 23:1–23:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Here we will focus on the analysis of several selection algorithms following the bogo-method. The pseudo code for bogo-select that ensures the property mentioned above reads as follows:

Listing 1 Algorithm: Bogo-Select (by partitioning)

```
Input array a[1..n] and k with 1<=k<=n
while a[1..n] is not partitioned(k) do
        randomly permute a[1..n]
endwhile
Output a[k]
```

The test whether the array is partitioned such that $a[i] \leq a[k]$, for $1 \leq i < k$, and $a[k] \leq a[i]$, for $k < i \leq n$ as well as the permutation of the array have to be programmed with some care:

```
procedure partitioned (int: k):
// returns true if the array
// is partitioned according
// to a[k] and false otherwise
for i=1 to k-1 do
        if a[i]>a[k] then
                return false
        endif
endfor
for i=k+1 to n do
        if a[i]<a[k] then
                return false
        endif
endfor
return true
```

```
procedure randomly permute:
// permutes the array a[1..n]
for i=1 to n-1 do
        j := rand[i..n]
        swap a[i] and a[j]
endfor
```

The second algorithm was already used for the bogo-sort algorithm and is found, e.g., in [4, p.139]. The random permutation is done quickly by a single loop, where RAND gives a random value in the specified range. And the test for partitioning according to the element $a[k]$ is carried out from left to right.

In this work we present a detailed analysis of the bogo-select algorithm in several variations. It is worth mentioning that we still obtain a randomized selection algorithm if we relax the condition of being partitioned to a condition that only requires that there are exactly $k-1$ elements in $a[1 \ldots k-1, k+1, \ldots n]$ that are smaller or equal to $a[k]$; equivalently one requires that exactly $n-k$ elements in the array $a[1 \ldots k-1, k+1 \ldots n]$ are larger or equal to $a[k]$. This variant of the bogo-select algorithm will be analyzed later on. Our proofs require only a basic knowledge of probability and can be readily understood by non-specialists. This makes the analysis well-suited to be included as motivating example in courses on randomized algorithms. We will analyze the expected running time for bogo-select under the assumption that we are given an array $\overline{x} = x_1 x_2 \ldots x_n$ containing a permutation of the set of numbers $\{1, 2, \ldots, n\}$ with $n \geq 2$. To analyze the running time of the bogo-select algorithm and the variants thereof, we on the one hand count the number of comparisons, and on the other hand the number of swaps. This is similar to the analysis of the bogo-sort algorithm done in [3]. An immediate observation is that the algorithm isn't guaranteed to terminate at all. However, as we will prove that the *expectation* of the running time T is finite as we see by

Markov's inequality

$$\mathbb{P}[T \geq t] \leq \frac{\mathbb{E}[T]}{t}, \quad \text{for } t > 0,$$

that the probability of this event equals 0. There are essentially two different initial configurations: either the list $\overline{x}$ is initially partitioned according to the kth element or it is not. We have to make this distinction as the algorithm is smart enough to detect if the given list is initially well partitioned, and has much better running time in this case. This welcome built-in feature also makes the running time analysis of this case very easy: the number of total comparisons equals $n-1$, and the total number of swaps equals zero, since the while-loop is never entered.

We come to the case where the array is not initially well partitioned. Note that the first shuffle yields a randomly ordered list, so the behavior of the algorithm does no longer depend on the initial order – the number of comparisons before the first shuffle still does.

2 Bogo-Select: Selection By Partitioning

2.1 How Long Does it Take to Check Whether the kth Element is on its Correct Position?

Before the analysis of the procedure PARTITIONED we begin with a detour on inverse binomial coefficients, since they are mostly not too intensively covered by monographs on combinatorics – an exception is, e.g., [1]. Obviously, an inverse binomial coefficient obeys the identity $\binom{n}{k}^{-1} = \frac{k!(n-k)!}{n!}$. The first theorem is based on the elementary identity

$$\binom{n}{k}^{-1} = \binom{n-1}{k-1}^{-1} - \frac{(n-k)}{(n-k+1)}\binom{n}{k-1}^{-1}$$

on inverse binomial coefficients. The proof of the following theorem is literally taken from [7].

▶ **Theorem 1.** *For $n \geq 2$, we have*

$$\sum_{i=0}^{\infty}\binom{n+i}{i}^{-1} = \frac{n}{n-1}.$$

Proof. The proof is by induction on n. For $n = 2$, the sum equals

$$\sum_{i=0}^{\infty}\binom{2+i}{i}^{-1} = \sum_{i=0}^{\infty}\frac{i!\cdot(2+i-i)!}{(2+i)!} = 2\sum_{i=0}^{\infty}\frac{1}{(i+1)(i+2)} = 2\sum_{i=0}^{\infty}\left(\frac{1}{i+1} - \frac{1}{i+2}\right) = 2$$

for the terms pairwise cancel. For $n > 2$, we observe that

$$\sum_{i=0}^{\infty}\binom{n+i}{i}^{-1} = \binom{n+0}{0}^{-1} + \sum_{i=1}^{\infty}\binom{n+i}{i}^{-1} = 1 + \sum_{i=0}^{\infty}\binom{n+(i+1)}{i+1}^{-1}.$$

Applying the elementary equation on inverse binomial coefficients mentioned before the theorem to the rightmost sum, we have

$$\sum_{i=0}^{\infty}\binom{n+i}{i}^{-1} = 1 + \sum_{i=0}^{\infty}\left[\binom{n+i}{i}^{-1} - \frac{n}{n+1}\binom{n+(i+1)}{i}^{-1}\right].$$

Assuming $\sum_{i=0}^{\infty} \binom{n+i}{i}^{-1} = \frac{n}{n-1}$ and hence is finite, we obtain

$$\frac{n}{n+1} \sum_{i=0}^{\infty} \binom{(n+1)+i}{i}^{-1} = 1,$$

completing the proof. ◀

In the forthcoming we will come across more involved combinatorial sums and series containing inverse binomial coefficients, which will be evaluated with the help of a more general approach based on Euler's Gamma function. This will detailed in Subsection 3.1.

Now we are ready to analyse the procedure PARTITIONED. Recall that we demand n to be at least 2 as a prerequisite. Thus, we will not state $n \geq 2$ explicitly in all theorems and lemmata to come.

▶ **Theorem 2.** *Assume $\overline{x}$ is a random permutation of $\{1, 2, \dots, n\}$, and let C denote the random variable counting the number of comparisons carried out in the test whether $\overline{x}$ is partitioned according to the kth element. Then*

$$\mathbb{E}[C] = \sum_{i=1}^{k-1} \frac{1}{i} + \frac{1}{k} \sum_{i=k}^{n-1} \binom{i}{k}^{-1}.$$

It holds $\mathbb{E}[C] = H_{n-1}$, if $k = 1$ or $k = n$, and $\mathbb{E}[C] \sim H_{k-1} + \frac{1}{k-1}$, otherwise.

Proof. Let k with $1 \leq k \leq n$ be fixed. In order to calculate $\mathbb{E}[C] = \sum_{i \geq 1} \mathbb{P}[C \geq i]$ we determine the probability for every valid i, that is $1 \leq i \leq n-1$ since we can compare $a[k]$ with at most $n-1$ other elements. Observe that the ith comparison is reached if and only if the algorithm did not drop out at the $i-1$ comparisons before. Furthermore observe that the first comparison within the second loop equals the kth total comparison. Therefore we consider two cases:

1. Let $1 \leq i < k$. If the routine makes a minimum of i comparisons, the $i-1$ first comparisons needed to be successful in order for the ith one to be carried out, yielding us $x_1, x_2, \dots, x_{i-1} \leq x_k$. Thus, the probability $\mathbb{P}[C \geq i]$ computes as

$$\mathbb{P}[C \geq i] = \frac{\binom{n}{i} \cdot (i-1)! \cdot (n-i)!}{n!}.$$

   The numerator is the product of the number of possibilities to choose i elements to arrange the $(i-1)$ smaller ones at the beginning of the array and the number of possibilities to arrange them, place the maximum from the i chosen elements on position k, and the number of possibilities to arrange the remaining $n-i$ elements at the uncharted part of the array, and the denominator is just the total number of arrays of length n. Reducing this fraction, we obtain $\mathbb{P}[C \geq i] = \frac{1}{i}$.
2. For $k \leq i < n$ we argue in similar veins as above and again find that $\mathbb{P}[C \geq i]$ can be estimated as

$$\mathbb{P}[C \geq i] = \frac{\binom{n}{i} \cdot (k-1)! \cdot (i-k)! \cdot (n-i)!}{n!}.$$

   The numerator is the product of the number of possibilities to choose i elements to arrange the $k-1$ smallest elements at the beginning of the array and the number of possibilities to arrange them, place the kth largest element on position k, and the remaining $i-k$ elements just after the kth position and the number of possibilities to arrange them, and finally the number of possibilities to arrange the remaining $n-i$ elements at the end part of the array with the denominator again being the total number of arrays of length n. Simplifying this fraction, results in $\mathbb{P}[C \geq i] = \frac{1}{k}\binom{i}{k}^{-1}$.

As the range of C is non-negative, we combine those two cases for $\mathbb{P}[C \geq i]$ for the expected value of C as follows:

$$\mathbb{E}[C] = \sum_{i \geq 1} \mathbb{P}[C \geq i] = \sum_{i=1}^{k-1} \frac{1}{i} + \sum_{i=k}^{n-1} \frac{1}{k}\binom{i}{k}^{-1}.$$

The former sum can easily be identified as the harmonic number $H_{k-1} = \sum_{i=1}^{k-1} \frac{1}{i}$. For the second term of the expected value above we have to do a little bit more calculation and the help of Theorem 1 we get:[1]

$$\sum_{i=k}^{n-1} \frac{1}{k}\binom{i}{k}^{-1} = \frac{1}{k}\sum_{i=0}^{n-1-k}\binom{k+i}{k}^{-1} \leq \frac{1}{k}\sum_{i=0}^{\infty}\binom{k+i}{i}^{-1} = \frac{1}{k}\cdot\frac{k}{k-1} = \frac{1}{k-1},$$

if $k \geq 2$. In case $k = 1$ we find

$$\mathbb{E}[C] = \sum_{i=1}^{1-1} \frac{1}{i} + \sum_{i=1}^{n-1} \frac{1}{1}\binom{i}{1}^{-1} = \sum_{i=1}^{n-1} \frac{1}{i},$$

which equals the harmonic number H_{n-1} similar to above – observe, that for $k = 1$ and $k = n$ the values for $\mathbb{E}[C]$ are the same further stressing the highly symmetrical nature of this very algorithm. Putting things together we obtain $\mathbb{E}[C] = H_{n-1}$, if $k = 1$ or $k = n$, and $\mathbb{E}[C] \sim H_{k-1} + \frac{1}{k-1}$, otherwise, as n tends to infinity. This proves the stated claim on the asymptotics. ◀

Theorem 2 tells us that we can expect comparisons in the range of H_n to check if a large array is well partitioned, and for n large enough, this number is about $\ln n$, because H_n satisfies the asymptotic

$$H_n \sim \ln n + \gamma,$$

where $\gamma \approx 0.5772156649$ is the Euler-Mascheroni constant. Compare to the worst case, where we have to compare $n-1$ times. This is a noticeable departure from the check for being sorted from left to right as analysed in [3], where it was shown that on average one only needs a constant number of comparisons, namely $e - 1 \approx 1.718281828$ for large enough arrays.

2.2 The Expected Number of Swaps in Bogo-Select

When computing the expected number of iterations in bogo-select, we concentrate on the case where the input $\overline{x}$ is not partitioned according to the kth element; for the other case it

[1] With a little bit more effort one can show that

$$\sum_{i=0}^{n}\binom{k+i}{i}^{-1} = \frac{k}{k-1}\left[1 - \binom{n+k}{n+1}^{-1}\right],$$

for $k \geq 2$ and $n \geq 0$; e.g., see the alternative proof of Theorem 1 given on page 33:9 or [2, page 16, Equation (2.2)]. This allows us to refine the expected value of C to the precise expression $\mathbb{E}[C] = H_{k-1} + \frac{1}{k-1}\left[1 - \binom{n-1}{k-1}^{-1}\right]$, for $k \geq 2$.

equals 0, because of the intelligent design of the algorithm. In each iteration, the array is permuted uniformly at random, and we iterate until we hit a sequence that is well partitioned for the first time. As a well partitioned ordered sequence w.r.t. the kth element is hit with probability $\frac{(k-1)!(n-k)!}{n!}$ in each trial, the number of iterations I is a random variable with a probability as follows:

$$\mathbb{P}[I=i] = \left(1 - \frac{(k-1)!(n-k)!}{n!}\right)^{i-1} \cdot \frac{(k-1)!(n-k)!}{n!}$$

That is, I is a geometrically distributed random variable with hitting probability $p = \frac{(k-1)!(n-k)!}{n!}$. Simple calculations show that $p = \frac{1}{k}\binom{n}{k}^{-1}$ and that the expected value of I is equal to $\mathbb{E}[I] = p^{-1} = k \cdot \binom{n}{k}$

In each iteration, the array is shuffled and a shuffle costs $n-1$ swaps. As the algorithm operates kind of economically with respect to the number of swaps, these are *the only* swaps carried out while running the algorithm. If S denotes the random variable counting the number of swaps, when asking for the kth element, we have $S = (n-1) \cdot I$. By linearity of expectation, we derive:

▶ **Theorem 3.** *If S denotes the total number of swaps carried out for an input $\overline{x}$ of length n, when asking for the kth element, for $1 \leq k \leq n$, we have*

$$\mathbb{E}[S] = \begin{cases} 0 & \text{if } \overline{x} \text{ is well partitioned according to the } k\text{th element} \\ k(n-1)\binom{n}{k} & \text{otherwise.} \end{cases}$$

Thus we immediately obtain:

▶ **Corollary 4.** *Let S denote the number of swaps carried out by bogo-select on a given input $\overline{x}$ of length n and an integer k with $1 \leq k \leq n$. Then*

$$\mathbb{E}[S] = \begin{cases} 0 & \text{in the best case} \\ k(n-1)\binom{n}{k} & \text{in the worst and average case.} \end{cases}$$

If $k = 1$, that is, we are interested in obtaining the smallest element in the array, then we have $\mathbb{E}[S] = (n-1) \cdot n = n^2 - n$. This is not to bad for a bogo-algorithm. In case we are interested in the last element we come to the same polynomial of $\mathbb{E}[S] = n^2(n-1)$. On the other hand, since the binomial coefficients are uni-modal[2] the elements in the middle are the largest ones. Using the asymptotic estimate of $\binom{n}{k} \sim \frac{2^{n/2}}{\sqrt{\pi n}}$, if $k = \frac{n}{2} + \mathrm{O}(1)$ (taken from from [8]), we estimate that $\mathbb{E}[S] \sim \frac{n(n-1)\cdot 2^{n/2-1}}{\sqrt{\pi n}}$, if $k = \frac{n}{2} + \mathrm{O}(1)$. Therefore, computing the median induces an exponential number of swaps on average.

3 Two Variations on Bogo-Select

In this section we discuss two natural variants of the bogo-select algorithm by switching out the procedure PARTITIONED. This will influence the expected number of comparisons that are needed in order to verify that the kth element is in its correct position and also the expected number of swaps. We use the generic bogo-select algorithm and replace the probe algorithm in the second line by another appropriate verifier like, e.g., Z-PARTITIONED and P-COUNTED, which will be detailed in the forthcoming.

[2] A finite sequence of numbers is *uni-modal* if the sequence first increases and then decreases.

3.1 Bogo-Select by Zig-Zag-Partition

Checking for being well partitioned w.r.t. the kth element is done from left to right, which corresponds to draw cards from the top of the deck. When using bottom dealing or base dealing, which is by the way considered as cheating in poker, we would end up in a similar situation as with drawing cards from the top (aka. top dealing). Thus, it is left to consider when top and bottom dealing is alternately used for checking well partitioning, i.e., $a[1]$ is checked against $a[k]$, then $a[n]$ against $a[k]$, followed by $a[2]$ against $a[k]$, etc. If alternation between top and bottom dealing is not possible anymore, then one continues with top or bottom dealing only. This results in the bogo-select (by zig-zag-partition), where the implementation of the checker Z-PARTITIONED reads as follows – the pseudo code if k belongs to the second half of the array is straight forward and therefore left to the interested reader:

```
procedure z-partitioned (int: k):
// returns true if the array is
// zig-zag partitioned according
// to a[k] and false otherwise
if k<=(n+1)/2 then
        // k in the first half
        ...
else
        // k in the second half
        ...
endif
return true
```

```
// k in the first half
for i=1 to k-1 do
        if a[i]>a[k] then
                return false
        endif
        if a[n+1-i]<a[k] then
                return false
        endif
endfor
for i=k+1 to n-k+1 do
        if a[i]<a[k] then
                return false
        endif
endfor
```

The analysis of this algorithm is quite similar to before. Again, we encounter a combinatorial sum and series with inverse binomial coefficients. A more general approach to evaluate these sums and series is based on Euler's well known Beta functions defined by

$$B(m,n) = \int_0^1 t^{m-1}(1-t)^{n-1}\,\mathrm{d}t,$$

for all complex numbers m and n with a positive real part. Since

$$B(m,n) = \frac{\Gamma(m)\Gamma(n)}{\Gamma(m+n)} = \frac{(m-1)!(n-1)!}{(m+n-1)!},$$

where Γ refers to the Gamma function satisfying $\Gamma(n+1) = n!$, we get

$$\binom{n}{k}^{-1} = (n+1)\int_0^1 t^k(1-t)^{n-k}\,\mathrm{d}t$$

for all non-negative integers n and k with $n \geq k$. Although this is an explicit formula for the inverse binomial coefficient, it is somehow more convenient to replace inverse binomial coefficients by factorials and in turn by the Gamma and Beta function and its integral representation when dealing with combinatorial sums and series. This allows to simplify the inner terms of a combinatorial sum taking extra factors into account. We demonstrate this strategy in the following theorem, which is known as Lehmer's identity [5], and uses basics on integrals and (inverse) trigonometric functions; the proof is literally taken from [10] to keep this presentation self contained.

▶ **Theorem 5** (Lehmer's Identity). *If* $|x| < 1$, *then*

$$\sum_{i=1}^{\infty} \frac{(2x)^{2i}}{i} \binom{2i}{i}^{-1} = \frac{2x}{\sqrt{1-x^2}} \arcsin(x).$$

Proof. We replace the inverse binomial coefficient and the factor $1/i$ in the infinite sum by the Gamma function and in turn by the Beta function and its integral definition and obtain

$$\begin{aligned}\sum_{i=1}^{\infty} \frac{(2x)^{2i}}{i} \binom{2i}{i}^{-1} &= \sum_{i=1}^{\infty} (2x)^{2i} \cdot \frac{i!(i-1)!}{(2i)!} \\ &= \sum_{i=1}^{\infty} (2x)^{2i} \cdot \frac{\Gamma(i+1)\Gamma(i)}{\Gamma(2i+1)} \\ &= \sum_{i=1}^{\infty} (2x)^{2i} B(i+1,i) = \sum_{i=1}^{\infty} (2x)^{2i} \int_0^1 t^i (1-t)^{i-1} \,\mathrm{d}t.\end{aligned}$$

After exchanging the sum and the integral and evaluating the geometric sum we find

$$\begin{aligned}\sum_{i=1}^{\infty} (2x)^{2i} \int_0^1 t^i (1-t)^{i-1} \,\mathrm{d}t &= \int_0^1 \sum_{i=1}^{\infty} (2x)^{2i} t^i (1-t)^{i-1} \,\mathrm{d}t \\ &= \int_0^1 \frac{1}{1-t} \sum_{i=1}^{\infty} (4x^2 t(1-t))^i \,\mathrm{d}t \\ &= \int_0^1 \frac{4x^2 t}{1-4x^2 t(1-t)} \,\mathrm{d}t.\end{aligned}$$

By considering the derivation of denominator we decide to use the substitution $s = x(2t-1)$ and hence $\frac{\mathrm{d}s}{\mathrm{d}t} = 2x$. Thus the above integral is further equal to

$$\int_{-x}^{x} \frac{s}{s^2 + (1-x^2)} \,\mathrm{d}s + x \int_{-x}^{x} \frac{1}{s^2 + (1-x^2)} \,\mathrm{d}s. \tag{1}$$

Since

$$\int \frac{1}{x} \,\mathrm{d}x = \ln|x|, \quad \text{and} \quad \int \frac{1}{x^2 + a^2} \,\mathrm{d}x = \frac{1}{a} \arctan\left(\frac{x}{a}\right)$$

as found in the theoretical computer science cheat sheet[3] one observes that the former term

[3] The theoretical computer science cheat sheet can be downloaded from, for instance, `https://tug.org/texshowcase/cheat.pdf`.

of Equation (1) evaluates to zero,[4] while the latter term of Equation 1 is equal to

$$x\int\limits_{-x}^{x}\frac{1}{s^2+(1-x^2)}\,\mathrm{d}s = x\left(\frac{1}{\sqrt{1-x^2}}\arctan\left(\frac{s}{\sqrt{1-x^2}}\right)\Big|_{-x}^{x}\right)$$
$$= \frac{2x}{\sqrt{1-x^2}}\arctan\left(\frac{x}{\sqrt{1-x^2}}\right),$$

because $\arctan(-x) = \arctan(x)$, for $|x| < 1$. Thus, we get

$$\sum_{i=1}^{\infty}\frac{(2x)^{2i}}{i}\binom{2i}{i}^{-1} = \frac{2x}{\sqrt{1-x^2}}\arctan\left(\frac{x}{\sqrt{1-x^2}}\right) = \frac{2x}{\sqrt{1-x^2}}\arcsin(x),$$

since the inverse trigonometry functions arcsin and arctan obey

$$\arcsin(x) = \arctan\left(\frac{x}{\sqrt{1-x^2}}\right) \quad \text{for } |x| < 1$$

as mentioned in the theoretical computer science cheat sheet. This proves the stated claim. ◀

It is worth mentioning that Theorem 1 can be shown in similar way as above by using the Beta function and its integral definition.

Alternative Proof of Theorem 1. We show a slightly stronger statement

$$\sum_{i=0}^{m}\binom{n+i}{i}^{-1} = \frac{n}{n-1}\left[1-\binom{m+n}{n+1}^{-1}\right],$$

for $n \geq 2$ and $m \geq 0$. If m tends to infinity, we obtain the statement of Theorem 1, because the inverse binomial coefficient approaches 0.

We proceed as in the proof of Theorem 5. Thus, we replace the inverse binomial coefficient by its definition, then the factorials by the Gamma function and in turn by the Beta function and its integral definition. This results in the following calculation:

$$\sum_{i=0}^{m}\binom{n+i}{i}^{-1} = \sum_{i=0}^{m}\frac{i!n!}{(n+i)!}$$
$$= n\sum_{i=0}^{m}\frac{\Gamma(i+1)\Gamma(n)}{\Gamma(n+i+1)}$$
$$= n\sum_{i=0}^{m}B(i+1,n) = n\sum_{i=0}^{m}\int\limits_{0}^{1}t^i(1-t)^{n-1}\,\mathrm{d}t.$$

[4] A simpler argument that

$$\int\limits_{-x}^{x}\frac{s}{s^2+(1-x^2)}\,\mathrm{d}s = 0$$

is that the function $f(s) = s/(s^2+(1-x^2))$ is symmetric to the origin, and thus the integral from $-x$ to x evaluates to zero.

Next we exchange the sum and the integral and evaluate the geometric sum. This leads us to

$$\begin{aligned} n\sum_{i=0}^{m}\int_{0}^{1} t^i(1-t)^{n-1}\,\mathrm{d}t &= n\int_{0}^{1}\sum_{i=0}^{m} t^i(1-t)^{n-1}\,\mathrm{d}t \\ &= n\int_{0}^{1}(1-t)^{n-1}\sum_{i=0}^{m} t^i\,\mathrm{d}t = n\int_{0}^{1}(1-t)^{n-1}\frac{1-t^{m+1}}{1-t}\,\mathrm{d}t. \end{aligned}$$

To keep the calculation simple we rewrite the above integral by its difference and obtain

$$\begin{aligned} n\int_{0}^{1}(1-t)^{n-1}\frac{1-t^{m+1}}{1-t}\,\mathrm{d}t &= n\int_{0}^{1}(1-t)^{n-2}(1-t^{m+1})\,\mathrm{d}t \\ &= n\int_{0}^{1}(1-t)^{n-2}\,\mathrm{d}t - n\int_{0}^{1}t^{m+1}(1-t)^{n-2}\,\mathrm{d}t. \end{aligned}$$

Since $n \geq 2$ we find that the former term of the above given difference is equal to

$$\begin{aligned} n\int_{0}^{1}(1-t)^{n-2}\,\mathrm{d}t &= n\left(\frac{-1}{n-1}(1-t)^{n-1}\Big|_{0}^{1}\right) \\ &= n\left(\frac{-1}{n-1}(1-1)^{n-1} - \frac{-1}{n-1}(1-0)^{n-1}\right) = \frac{n}{n-1}, \end{aligned}$$

while the latter term can be rewritten by the Beta function and in turn by the Gamma function or by factorials, which reads as

$$\begin{aligned} n\int_{0}^{1}t^{m+1}(1-t)^{n-2}\,\mathrm{d}t &= n\cdot B(m+2,n-1) \\ &= n\cdot\frac{\Gamma(m+2)\Gamma(n-1)}{\Gamma(m+n+1)} \\ &= n\cdot\frac{(m+1)!(n-2)!}{(m+n)!} = \frac{n}{n-1}\binom{m+n}{m+1}^{-1} \end{aligned}$$

which gives the desired result by putting both terms together and factor $n/(n-1)$ out. ◀

We are interested in special values on inverse central binomial coefficients (with some particular factors). Both values are well known, e.g., see [9]. The value for the first combinatorial sum is obtained by evaluating Lehmer's identify for $x = 1/2$ taking into account that $\arcsin(1/2) = \pi/6$. In order to obtain the value for second combinatorial sum we differentiate Lehmer's identity, multiply it with x, integrate it, and evaluate it again at $x = 1/2$.

▶ **Theorem 6.** *It holds*

$$\sum_{i=1}^{\infty}\frac{1}{i}\binom{2i}{i}^{-1} = \frac{\pi\sqrt{3}}{9} \quad \textit{and} \quad \sum_{i=0}^{\infty}\frac{1}{2i+1}\binom{2i}{i}^{-1} = \frac{2\pi\sqrt{3}}{9}.$$

Proof. The value for the first combinatorial sum is obtained by evaluating Lehmer's identify for $x = 1/2$ taking into account that $\arcsin(1/2) = \pi/6$. Thus we have

$$\sum_{i=1}^{\infty} \frac{1}{i}\binom{2i}{i} = \frac{2}{\sqrt{3}}\frac{\pi}{6} = \frac{\pi\sqrt{3}}{9}.$$

In order to obtain the value for other combinatorial sum we start with the differentiation of Lehmer's identity. Differentiation of the polynomial on the left hand-side of Lehmer's identity is straight forward and gives

$$\frac{\mathrm{d}}{\mathrm{d}x}\sum_{i=1}^{\infty} \frac{(2x)^{2i}}{i}\binom{2i}{i}^{-1} = \sum_{i=1}^{\infty} 4(2x)^{2i-1}\binom{2i}{i}^{-1}. \tag{2}$$

On the other hand, the differentiation of the right hand-side involves the product of a quotient with an arcsin. We find

$$\frac{\mathrm{d}}{\mathrm{d}x}\frac{2x}{\sqrt{1-x^2}}\arcsin(x) = \frac{2\arcsin(x)}{\sqrt{1-x^2}} + \frac{2x^2\arcsin(x)}{(1-x^2)^{3/2}} + \frac{2x}{1-x^2}, \tag{3}$$

where we used

$$\frac{\mathrm{d}}{\mathrm{d}x}\arcsin(x) = \frac{1}{\sqrt{1-x^2}}, \quad \text{for } |x| < 1.$$

Equation (3) can be easily verified with a symbolic computation software at hand. Hence, plugging together (2) and (3) we have

$$\sum_{i=1}^{\infty} 4(2x)^{2i-1}\binom{2i}{i}^{-1} = \frac{2\arcsin(x)}{\sqrt{1-x^2}} + \frac{2x^2\arcsin(x)}{(1-x^2)^{3/2}} + \frac{2x}{1-x^2}, \tag{4}$$

Then value for the second combinatorial sum $\sum_{i=1}^{\infty}\frac{1}{2i+1}\binom{2i}{i}^{-1}$ is obtained by multiplying Equation (4) by x and integrating it. For the left hand-side of Equation (4) we thus get

$$\begin{aligned}\int \sum_{i=1}^{\infty} 2(2x)^{2i}\binom{2i}{i}^{-1}\,\mathrm{d}x &= \sum_{i=1}^{\infty}\int 2(2x)^{2i}\binom{2i}{i}^{-1}\,\mathrm{d}x \\ &= \sum_{i=1}^{\infty}\frac{2^{2i+1}x^{2i+1}}{2i+1}\binom{2i}{i}^{-1} = \sum_{i=1}^{\infty}\frac{(2x)^{2i+1}}{2i+1}\binom{2i}{i}^{-1}.\end{aligned} \tag{5}$$

The integration of the right hand-side of Equation (4) is quite tedious for it is done by parts. Hence we use a symbolic manipulation software, in order to minimize our work effort. We obtain

$$\begin{aligned}&\int x\left(\frac{2\arcsin(x)}{\sqrt{1-x^2}} + \frac{2x^2\arcsin(x)}{(1-x^2)^{3/2}} + \frac{2x}{1-x^2}\right)\mathrm{d}x \\ &\quad = \frac{2\arcsin(x)}{\sqrt{1-x^2}} - 2\ln\left(\left|\frac{1}{\sqrt{1-x^2}} + \frac{x}{\sqrt{1-x^2}}\right|\right) - 2x - \ln(|x-1|) + \ln(|x+1|).\end{aligned} \tag{6}$$

Thus, by (5) and (6) we have

$$\begin{aligned}&\sum_{i=1}^{\infty}\frac{(2x)^{2i+1}}{2i+1}\binom{2i}{i}^{-1} \\ &\quad = \frac{2\arcsin(x)}{\sqrt{1-x^2}} - 2\ln\left(\left|\frac{1}{\sqrt{1-x^2}} + \frac{x}{\sqrt{1-x^2}}\right|\right) - 2x - \ln(|x-1|) + \ln(|x+1|),\end{aligned}$$

which evaluated at $x = 1/2$ results in

$$\begin{aligned}\sum_{i=1}^{\infty} \frac{1}{2i+1}\binom{2i}{i}^{-1} &= \frac{4\pi}{6\sqrt{3}} - 2\ln\left(\frac{3}{\sqrt{3}}\right) - 1 - \ln\left(\frac{1}{2}\right) + \ln\left(\frac{3}{2}\right)\\ &= \frac{2\pi\sqrt{3}}{9} - \ln(3) + \ln(2) + \ln\left(\frac{3}{2}\right) - 1 = \frac{2\pi\sqrt{3}}{9} - 1.\end{aligned}$$

At last we shift the sum down to $i = 0$ as we need this form later on:

$$\sum_{i=0}^{\infty} \frac{1}{2i+1}\binom{2i}{i}^{-1} = \frac{1}{2\cdot 0+1}\binom{2\cdot 0}{0}^{-1} + \sum_{i=1}^{\infty} \frac{1}{2i+1}\binom{2i}{i}^{-1} = 1 + \frac{2\pi\sqrt{3}}{9} - 1 = \frac{2\pi\sqrt{3}}{9}$$

This completes the proof. ◀

Now we are ready to determine the expected number of comparisons within the procedure Z-PARTITIONED.

▶ **Theorem 7.** *Assume $\overline{x}$ is a random permutation of $\{1, 2, \ldots, n\}$, and let C denote the random variable counting the number of comparisons carried out by the procedure* Z-PARTITIONED *in the test whether $\overline{x}$ is partitioned according to the kth element, for $1 \leq k \leq (n+1)/2$, then:*

$$\mathbb{E}[C] = \sum_{j=1}^{k-1}\left[\frac{1}{2j-1}\binom{2(j-1)}{j-1}^{-1} + \frac{1}{j}\binom{2j}{j}^{-1}\right] + \frac{1}{k}\sum_{i=2k-1}^{n-1}\binom{i}{k}^{-1}$$

It holds $\mathbb{E}[C] = H_{n-1}$, if $k = 1$, and $\mathbb{E}[C] \sim \frac{\pi\sqrt{3}}{3} + \frac{1}{k-1}\binom{2(k-1)}{k-1}^{-1}$, otherwise. The statement remains valid in case k satisfies $(n+1)/2 < k \leq n$ by replacing k by $n-k+1$.

Proof. Let k with $1 \leq k \leq n$ be fixed. We analyse the procedure Z-PARTITIONED only for the case that k lies in the first half of the array, i.e., $k \leq (n+1)/2$. It is easy to see that if k is in the second half of the array the case is symmetric to the above one by replacing k by $n-k+1$ in the formulas.

Recall that the ith comparison is reached if and only if the algorithm did not drop out at the $i-1$ comparisons before. We therefore consider two cases regarding the number of successful comparisons counting i – the probabilities are modelled similar to the proof of Theorem 2.

1. Let $1 \leq i \leq 2k-2$, i.e., the algorithm has yet to leave the zig-zag-comparison. Then we distinguish two subcases, namely whether i is odd or even. In the former case we have completed $j := (i-1)/2$ pairs of comparisons with $0 \leq j \leq k-2$, which yields the probability

   $$\mathbb{P}[C \geq i] = \frac{\binom{n}{i} j! j! (n-i)!}{n!} = \frac{1}{2j+1}\binom{2j}{j}^{-1}.$$

   In the latter case the $(i-1)$th comparison is carried out at the beginning of the array, and we find

   $$\mathbb{P}[C \geq i] = \frac{\binom{n}{i} j! (j-1)! (n-i)!}{n!} = \frac{1}{j}\binom{2j}{j}^{-1},$$

   where $j := i/2$ with $1 \leq j \leq k-1$.

2. Let $2k-1 \leq i < n-1$, i.e., the algorithm is in the second for-loop and therefore outside the zig-zag-comparison. Then we have
$$\mathbb{P}[C \geq i] = \frac{\binom{n}{i}(k-1)!(i-1-(k-1))!(n-i)!}{n!} = \frac{1}{k}\binom{i}{k}^{-1}.$$

We obtain the expected value of C by summing up the probabilities in every case, since the range of C is non-negative, and we get

$$\begin{aligned}\mathbb{E}[C] = \sum_{i\geq 1}\mathbb{P}[C \geq i] &= \sum_{j=0}^{k-2}\frac{1}{2j+1}\binom{2j}{j}^{-1} + \sum_{j=1}^{k-1}\frac{1}{j}\binom{2j}{j}^{-1} + \sum_{i=2k-1}^{n-1}\frac{1}{k}\binom{i}{k}^{-1} \qquad (7)\\ &= \sum_{j=1}^{k-1}\left[\frac{1}{2j-1}\binom{2(j-1)}{j-1}^{-1} + \frac{1}{j}\binom{2j}{j}^{-1}\right] + \frac{1}{k}\sum_{i=2k-1}^{n-1}\binom{i}{k}^{-1}.\end{aligned}$$

Each of the three sums in Equation (7) can be bounded from above. For the first two sums we use the estimates from Theorem 6, while for the last one we need a more sophisticated estimate than the one encountered within the proof of Theorem 2, because the sum is truncated at the beginning. Consider

$$\sum_{i=2k-1}^{n-1}\binom{i}{k}^{-1} = \sum_{i=k-1}^{n-k-1}\binom{i+k}{k}^{-1} = \sum_{i=0}^{n-k-1}\binom{i+k}{k}^{-1} - \sum_{i=0}^{k-2}\binom{i+k}{k}^{-1},$$

which by the equation given in the footnote on page 5 is equal to

$$\frac{k}{k-1}\left[1-\binom{n-1}{n-k}^{-1}\right] - \frac{k}{k-1}\left[1-\binom{2k-2}{k-1}^{-1}\right] = \frac{k}{k-1}\left[\binom{2(k-1)}{k-1}^{-1} - \binom{n-1}{n-k}^{-1}\right]$$

and in the limit is $\frac{k}{k-1}\binom{2(k-1)}{k-1}^{-1}$ as n tends to infinity. Therefore $\mathbb{E}[C]$ is at most

$$\frac{2\pi\sqrt{3}}{9} + \frac{\pi\sqrt{3}}{9} + \frac{1}{k}\cdot\frac{k}{k-1}\binom{2(k-1)}{k-1}^{-1} = \frac{\pi\sqrt{3}}{3} + \frac{1}{k-1}\binom{2(k-1)}{k-1}^{-1},$$

for $k \geq 2$. In case $k=1$ the former two combinatorial sums are zero and the remaining terms sum up to $\sum_{i=1}^{n-1}\frac{1}{i}$, which again equals the harmonic number of H_{n-1}. This completes our proof. ◀

Wasn't that awesome? Theorem 7 tells us that we need only a constant number of comparisons on the average to check if a large array is well partitioned w.r.t. k, if $k \geq 2$, and for n large enough, and this number is about

$$\frac{\pi\sqrt{3}}{3} \approx 1.81379936423.$$

Compare this to Theorem 2 and to to the worst case, where we have to compare $n-1$ times.

When considering the expected number of swaps, we find an identical result as in the case of the original bogo-select (by partition) algorithm, which we state without proof.

▶ **Theorem 8.** *If S denotes the total number of swaps carried out by the bogo-select (by zig-zag partition) for an input $\overline{x}$ of length n, when asking for the kth element, for $1 \leq k \leq n$, we have*

$$\mathbb{E}[S] = \begin{cases} 0 & \textit{if } \overline{x} \textit{ is well partitioned according to the } k\textit{th element}\\ k(n-1)\binom{n}{k} & \textit{otherwise.}\end{cases}$$

3.2 Bogo-Selection by Counting

As already mentioned in the introduction we also consider a variant of bogo-select relaxing the condition of being partitioned to a condition that only requires that there are exactly $k-1$ elements in $a[1 \ldots k-1, k+1, \ldots n]$ smaller or equal to $a[k]$; equivalently one requires that exactly $n-k$ elements in the array $a[1 \ldots k-1, k+1 \ldots n]$ are larger or equal to $a[k]$. This slight change of the condition alters the running time of the algorithm significantly. For instance, in case $n = 7$ and $k = 3$ the original bogo-select and its newly introduced variant would terminate on input $\overline{x} = 1\,2\,3\,4\,5\,6\,7$: the values 1 and 2 are smaller or equal than 3 while the remaining values are strictly larger and so the input is well partitioned according to the 3rd element of $\overline{x}$. On the other hand, on input $\overline{x} = 5\,6\,3\,1\,2\,4\,7$ the original bogo-select algorithm starts with randomly permuting the input, because in this case $\overline{x}$ is not well partitioned w.r.t. the 3rd element. For the newly introduced variant this second input $\overline{x} = 5\,6\,3\,1\,2\,4\,7$ leads to immediate termination in contrast. The new variant of bogo-select (now by counting) gives rise to a new procedure named P-COUNTED that requires an additional counter c initially set to 0 in order to collect the number of elements that are smaller or equal to $a[k]$. Observe, that the value of c is only compared to k in the second for-loop (and once after the for-loop is finished). The procedure P-COUNTED reads as follows:

```
procedure p-counted (int: k):
// returns true if exactly k-1 elements
// of a[1..k-1,k+1..n] are smaller or
// equal than a[k] and false otherwise
c := 0
for i=1 to k-1 do
        if a[i]<=a[k] then
                c := c+1
        endif
endfor
for i=k+1 to n do
        if a[i]<=a[k] then
                c := c+1
                if c>=k then
                        return false
                endif
        endif
endfor
if c<k-1 then
        return false
endif
return true
```

Initially, enumerating the number of comparisons of the bogo-select (by counting) algorithm looks more involved, since one has also to take the counter values into consideration. In fact, the analysis is surprisingly easy.

▶ **Theorem 9.** *Assume $\overline{x}$ is a random permutation of $\{1, 2, \ldots, n\}$, and let C denote the random variable counting the number of comparisons of array elements carried out by the procedure* P-COUNTED *in the test whether $\overline{x}$ is partitioned according to the kth element. Then*

$$\mathbb{E}[C] = \begin{cases} H_{n-1} & \text{if } k = 1, \\ (k-1) + k(H_{n-1} - H_{k-1}) & \text{otherwise.} \end{cases}$$

Proof. Let k with $1 \leq k \leq n$ be fixed. We proceed as in the previous two proofs. Therefore we again consider two cases:

1. Let $1 \leq i < k$. Then by the careful design of the algorithm we have

   $$\mathbb{P}[C \geq i] = 1,$$

   because in the first for-loop the procedure P-COUNTED in no case terminates. Furthermore the first comparison within the second loop is always carried out totalling to a minimum of k comparisons.
2. Next assume $k \leq i < n$. Up until the ith comparison the value of the counter variable c ranges from 0 to at most $k-1$, since otherwise the procedure P-COUNTED would have already returned *false*. Thus, in order to determine the value of $\mathbb{P}[C \geq i]$ we first consider the following example: let $n = 10$ and $k = 4$. Assume we are about to reach $i = 7$ comparisons with counter c set to 3. Then we have hit i elements out of n, that is, for instance, $\{1, 3, 4, 7, 8, 9, 10\}$. Now c indicates that exactly three elements from that sublist are already smaller than our given element x_k, thus yielding us 7 to be x_k – in general x_k then equals the $(c+1)$st element. The remaining $i-1$ elements from the selected set can be freely permuted as well as the rest with a size of $n-i$. A last ith comparison is then carried out that is allowed to also fail now thus setting no further restrictions on the input. From this consideration we can derive in general

   $$\mathbb{P}[C \geq i] = k \cdot \frac{\binom{n}{i}(i-1)!(n-i)!}{n!} = \frac{k}{i}.$$

Summing up gives

$$\mathbb{E}[C] = \sum_{i>0} \mathbb{P}[C \geq i] = \sum_{i=1}^{k-1} 1 + \sum_{i=k}^{n-1} \frac{k}{i} = (k-1) + k \sum_{i=k}^{n-1} \frac{1}{i}.$$

The remaining sum is then equal to $H_{n-1} - H_{k-1}$, yielding

$$\mathbb{E}[C] = (k-1) + k(H_{n-1} - H_{k-1}),$$

if $k \geq 2$. In case $k = 1$ the former sum disappears and we end up with $\sum_{i=1}^{n-1} \frac{1}{i}$, which is the $(n-1)$th harmonic number H_{n-1}. This proves the claimed bounds. ◀

A more detailed analysis of the procedure P-COUNTED also taking into account the comparisons of the counter c at lines 14 and 19 is given next. While the upper theorem offers a complete analysis of P-COUNTED regarding the comparison of array elements it ignores that counter c is also monitored *via* a comparison (in line 14). As it turns out this does not factor into the asymptotics in a meaningful way. Going back to the procedure P-COUNTED we can see that the only repeated comparison not covered by the analysis above happens in line 14 where the counter c is checked for overflow. This very comparison is only carried out when c itself has been incremented one line earlier, i.e., the comparison of array elements within the second loop was successful. We can therefore deduce that the number of comparisons within the second loop is at maximum double the comparisons we counted in Theorem 9. Furthermore a last comparison can happen afterwards summing up to an upper bound. Thus, we have

$$\mathbb{E}[C] \leq \mathbb{E}[C_{\text{tot}}] \leq \mathbb{E}[C] + \begin{cases} H_{n-1} + 1 & \text{if } k = 1 \\ k(H_{n-1} - H_{k-1}) + 1 & \text{otherwise,} \end{cases}$$

where C_{tot} denotes the random variable counting the *total* number of comparisons (lines 12, 14, and 19) carried out by the procedure P-COUNTED in the test whether $\overline{x}$ is partitioned according to the kth element. The next theorem gives a more precise estimate on $\mathbb{E}[C_{\text{tot}}]$, by a more detailed analysis of the P-COUNTED procedure.

▶ **Theorem 10.** *Assume $\overline{x}$ is a random permutation of $\{1, 2, \ldots, n\}$, and let C_{tot} denote the random variable counting the* total *number of comparisons (lines 12, 14, and 19) carried out by the procedure* P-COUNTED *in the test whether $\overline{x}$ is partitioned according to the kth element. Then*

$$\mathbb{E}[C_{tot}] = \mathbb{E}[C] + \frac{k+1}{2} - \frac{k(k-1)}{2n}.$$

It holds $\mathbb{E}[C_{tot}] \sim \mathbb{E}[C] + \frac{k+1}{2}$.

Proof. Let R denote the random variable counting the raises of counter c within the procedure P-COUNTED. We calculate the expected value using the ansatz $\mathbb{E}[R] = \sum_{i \geq 0} i \cdot \mathbb{P}[R = i]$. Since we drop out for $c = k$ we can deduce $0 \leq i \leq k$. We consider two cases:

1. When c is raised at most $0 \leq i \leq k-1$ times there are a total of i smaller elements within the array and therefore x_k has to be the $(i+1)$st element which we hit with a probability of

$$\mathbb{P}[R = i] = \frac{1}{n}.$$

2. Otherwise $c = k$, which can be modeled using the complementary probability to the case above:

$$\mathbb{P}[R = k] = 1 - \sum_{i=0}^{k-1} \frac{1}{n} = 1 - \frac{k}{n}$$

Summing it up we get

$$\begin{aligned}\mathbb{E}[R] &= \sum_{i \geq 0} i \cdot \mathbb{P}[R = i] = \sum_{i=0}^{k-1} i \frac{1}{n} + k\left(1 - \frac{k}{n}\right) = \frac{1}{n} \sum_{i=1}^{k-1} i + k - \frac{k^2}{n} \\ &= \frac{1}{n}\left(\frac{(k-1)k}{2}\right) + k - \frac{k^2}{n} = \frac{k^2 - k + 2kn - 2k^2}{2n} = \frac{2kn - k - k^2}{2n}.\end{aligned}$$

Up next let the random variable R_1 count the raises of counter c only within the first loop. Going over the first k elements we can at most find $k-1$ smaller elements. Each possible counter value fixes us a specific element again, allowing us to calculate the expected value as follows:

$$\mathbb{E}[R_1] = \sum_{i \geq 0} i \cdot \mathbb{P}[R = i] = \sum_{i=0}^{k-1} i \frac{1}{k} = \frac{1}{k} \sum_{i=1}^{k-1} i = \frac{1}{k}\left(\frac{(k-1)k}{2}\right) = \frac{k-1}{2}$$

Lastly let random variable F indicate whether the final comparison in line 19 is carried out. We reach this comparison when we have not dropped out before, that is if and only if there were a maximum of $k-1$ smaller elements to x_k. In other words x_k is one of the k smallest elements within the array yielding us an expected value of

$$\mathbb{E}[F] = \sum_{i \geq 0} i \cdot \mathbb{P}[F = i] = 0 \cdot \mathbb{P}[F = 0] + 1 \cdot \mathbb{P}[F = 1] = \frac{k}{n}.$$

Now observe that every comparison within procedure P-COUNTED that was not covered by random variable C of the theorem 9 happens either when the counter is raised within the second loop or in line 19 after the complete array has been covered. So the random variable C_{tot} counting the total number of comparison within procedure P-COUNTED is given by

$$C_{\text{tot}} = C + (R - R_1) + F.$$

By linearity of the expected value we can calculate

$$\mathbb{E}[C_{\text{tot}}] = \mathbb{E}[C] + \mathbb{E}[R] - \mathbb{E}[R_1] + \mathbb{E}[F],$$

where

$$\mathbb{E}[R] - \mathbb{E}[R_1] + \mathbb{E}[F] = \frac{2kn - k - k^2}{2n} - \frac{k-1}{2} + \frac{k}{n} = \frac{kn + k - k^2 + n}{2n}$$

and therefore

$$\mathbb{E}[C_{\text{tot}}] = \mathbb{E}[C] + \frac{kn + k - k^2 + n}{2n} \qquad = \mathbb{E}[C] + \frac{k+1}{2} - \frac{k(k-1)}{2n},$$

yielding the stated result. Thus, asymptotically $\mathbb{E}[C_{\text{tot}}] \sim \mathbb{E}[C] + \frac{k+1}{2}$, as n tends to infinity. ◀

This completes the more detail analysis of the procedure P-COUNTED.

Finally, for the total number of swaps of the bogo-select (by counting) algorithm we obtain the following result. A permutation (or array) $\overline{x}$ satisfying

$$|\{\, i \mid x_i \leq x_k, \text{ for } 1 \leq i \leq n \text{ and } i \neq k \,\}| = k - 1$$

is said to be *p-counted w.r.t. the kth element.*

▶ **Theorem 11.** *If S denotes the total number of swaps carried out by bogo-select by counting for an input $\overline{x}$ of length n, when asking for the kth element, for $1 \leq k \leq n$, we have*

$$\mathbb{E}[S] = \begin{cases} 0 & \textit{if } \overline{x} \textit{ is p-counted w.r.t the kth element} \\ n(n-1) & \textit{otherwise.} \end{cases}$$

Proof. We argue as in the case of the bogo-select algorithm. The hitting probability for a permutation to be p-counted by the kth element is $\frac{(n-1)!}{n!}$ in each trial; the kth element is fixed and the remaining $n-1$ elements can be arbitrarily arranged. The number of iterations I is a geometrically distributed variable with hitting complexity $p = \frac{1}{n}$ and $\mathbb{E}[I] = p^{-1} = n$. Since each iteration costs $n-1$ swaps the expected number of swaps counted by the random variable S is equal to $\mathbb{E}[S] = n(n-1)$, if the input is not p-counted w.r.t. the kth element of the array, and $\mathbb{E}[S] = 0$, otherwise. ◀

4 Experimental Results

We have implemented the considered algorithms in LUA[5] and have performed some experiments. The source code as well as the test scripts are available on request by email to one of

[5] LUA is a powerful, efficient, lightweight, embeddable scripting language that can be downloaded form `https://www.lua.org` as a small package that builds out-of-the-box on all platforms that have a standard C compiler.

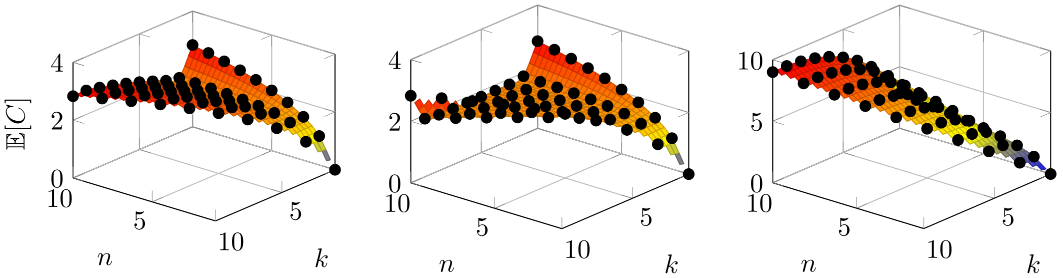


Figure 1 The expected number of comparisons carried out by the three considered probe procedures (i) PARTITIONED, (ii) Z-PARTITIONED, and (iii) P-COUNTED – diagrams are from left to right. Observe, that the vertical axis is scaled differently on the rightmost diagram.

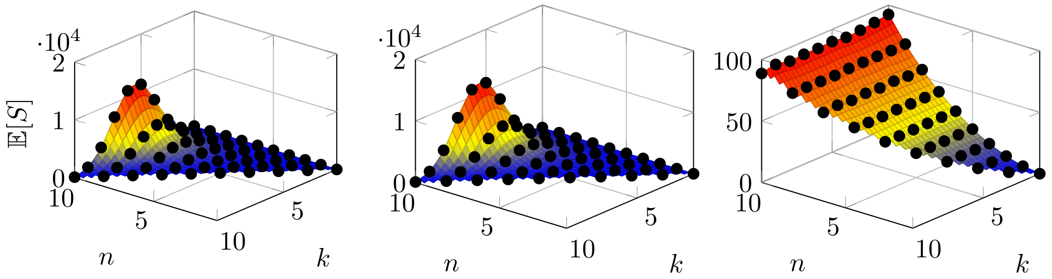


Figure 2 The expected number of swaps carried out by the three considered randomized selection algorithms bogo-select (i) by partition, (ii) by zig-zag-partition, and (iii) by counting – diagrams are from left to right. Observe, that the vertical axis is differently scaled on the rightmost diagram.

the authors. The experiments were conducted on an iMac (mid 2011 version) with an Intel Core i5 processor (2.7 GHz) and 4GB main memory. It took quite some time to collect our results, but this was no problem, since we started in early January with our experiments. We first implemented every investigated variant of the bogo-select as-is in order to observe the performance on the given machine depending on different input sizes.

We ran all three probe procedures on every possible permutation of input array $a[1\dots n]$ up to length $n = 10$ in order to track the arithmetic mean of the number of comparisons. As expected our experimental values for every valid combination of n, k exactly match the expected value calculated earlier. The results on the number of comparisons are shown in Figure 1. Since the number of possible inputs grows superexponentially, exact values for higher input sizes become quite costly and would necessitate further experiments.

It is important to note that our calculations for the number of swaps are based on the premise that the input does not comply to the property at hand initially. Furthermore neither the number of swaps nor the state of the input from iteration 2 onward in any way depend on the original input – as we already stated in the introduction. We therefore take an array that meets the given property – i.e., a fully ordered one – and swap entries x_k, x_{k+1} yielding us an input as required. *Via* the insertion of a counter variable we tracked the swaps carried out in 10.000 runs for every valid combination of n, k up to $n = 10$. The experimental results depicted in Figure 2 nicely line up with the theoretical values. A more detailed statistical analysis for the number of swaps is given next. When we consider the swaps the analysis is not that simple for we counted them over $r = 10000$ randomized runs of each algorithm. First up we use their arithmetic mean $\overline{S}$ as an unbiased estimate for the expected value due to the number of runs r being sufficiently large – as stated earlier the values lined

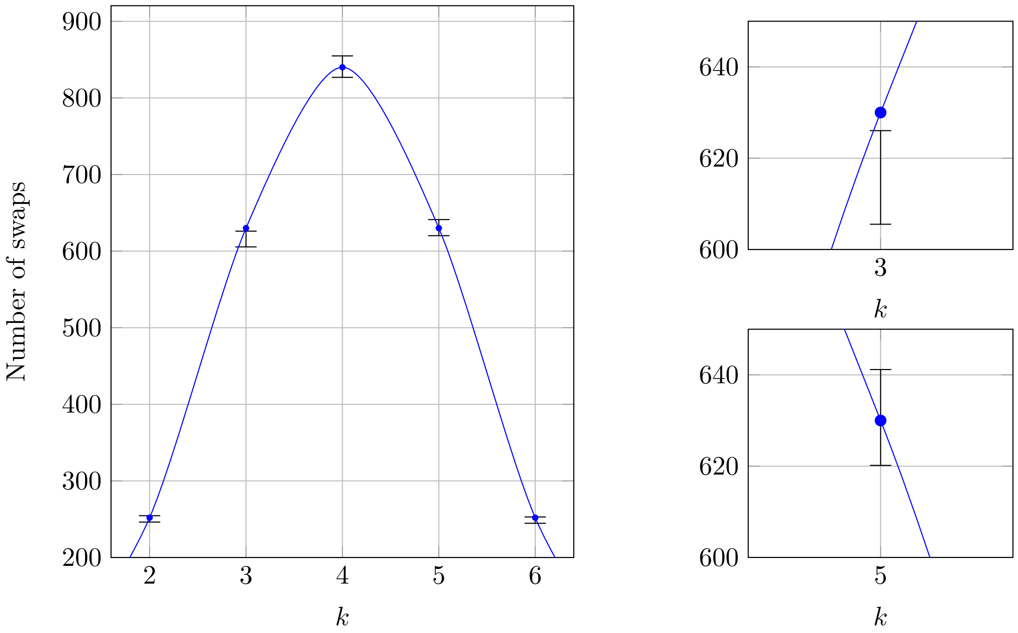


Figure 3 Confidence intervals for the number of swaps performed in bogo-select (by zig-zag-partition) with $n = 7$ and $k \in \{2, 3, 4, 5, 6\}$. The expected number of swaps is drawn blue.

up reasonably well with the theoretical results. We also calculated the sample standard deviation via $\sigma = \sqrt{\frac{1}{r-1} \sum_{i=1}^{r} (S_i - \overline{S})^2}$ to quantify how far the experimental results are dispersed in relation to the arithmetic mean. Having this information we now perform some statistical analysis upon the experimentally produced data sets – we do not factor in our theoretical results to reach a second (and independent) conclusion.

The generally known hypothesis test does not offer further insight here since it is only significant when it comes to rejecting an hypothesis while our objective is to verify the theoretical results as calculated earlier. Based on our experimental data it is good practise to calculate a confidence interval for the expected value of the number of swaps $\mathbb{E}[S]$ instead. It is a statistical technique that gives an interval covering the underlying expected value of a given probe with a fixed probability α. In our case we make no preconception about the distribution at hand and have a probe size r sufficiently large. Therefore we can calculate the confidence interval as $[\overline{S} \pm t_{\infty,1-\alpha/2} \frac{\sigma}{\sqrt{r}}]$ with $t_{\infty,1-\alpha/2}$ being the $1 - \alpha/2$ quantile of the t-distribution with unlimited degrees of freedom.

With this method we are now able to check whether our theoretically calculated expected value falls into the range that covers the expected value within the experiments (with probability $\alpha = 95\%$). E.g. looking into the results for bogo-select (by zig-zag-partition), we can see that for $n = 7$ and $k = 5$ we have an expected value of $\mathbb{E}[S] = 630$ that comfortably sits in the confidence interval of $I \approx 630.68 \pm 10.5$ we have generated experimentally. On the other hand for $n = 7$ and $k = 3$ the expected value of $\mathbb{E}[S] = 630$ misses the confidence interval $I \approx 615.79 \pm 10.253$ by a noticeable margin. Still the overall fit for $n = 7$ is quite good as we can see in Figure 3. Deviations like the above are to be expected since the confidence interval itself depends upon the randomness inherent to the experiments. In order to further quantify how often the individual confidence intervals match with our theoretical results we

Table 1 We cumulated the ratio whether the confidence intervals cover the theoretical values (Conf) for all three bogo-select variants as well as the maximum deviation between arithmetic mean and theoretical value relative to the latter ($\mathrm{maxDev} = \max\left(\frac{|\mathbb{E}[S]-\overline{S}|}{\mathbb{E}[S]}\right)$).

Bogo-select by…	Conf	maxDev
partitioning	$81.\overline{81}\%$	2.507619%
z-partitioning	$89.\overline{09}\%$	2.255524%
p-counting	$94.\overline{54}\%$	2.73%

analyse the generated data sets with the powerful statistical programming language R.[6] First we calculated the percentage for said matches regarding every bogo-select variant that was analysed within this paper. As you can see (in the first column of the Table 1) the coverage is quite high. Please note that this ratio is not directly related to the significance of the confidence intervals. The given level only ensures that a specific interval under identical conditions covers the real value in 95% of all cases. In our experiments on the other hand we only calculated a single confidence interval for every valid pair of n and k. Additionally we are interested how much the experimental value differs from the theoretical results in general. For this reason we also included the biggest distance between the two (relatively to the theoretical expected value) in Table 1.

As one can see we stay well below a 3% discrepancy for every algorithm. In summary it is safe to say that our experiments also support our theoretical analysis regarding the number of swaps.

5 Conclusions

We continued our research on the still unexplored research field of pessimal algorithm design with a theoretical and experimental study of bogo-select and variants thereof, which are archetypical perversely awful algorithms. Remarkably, the expected running time of these algorithms in terms of the number of swaps and comparisons can be determined exactly using only elementary methods in probability and combinatorics. Though optimizing the running time seems somewhat out of place in the field of *pessimal* algorithm design, it can be quite revealing for beginners in both fields of optimal and pessimal algorithm design to see how a single optimization step can yield a dramatic speed-up. The very first obvious optimization step in all aforementioned algorithms is to swap two elements only if this makes sense. That is, before swapping a pair, we check if it is an inversion: a pair of positions (i, j) in the array $a[1 \dots n]$ is an *inversion* if $i < j$ and $a[i] > a[j]$. This leads to an optimized variant of bogo-select (by partitioning or zig-zag partition), which we refer to as bogo-select$_{opt}$. As there are at most $\binom{n}{2}$ inversions, this number gives an immediate upper bound on the number of swaps for the optimized bogo-select variants. Thus a single optimization step yields *polynomial* running time. This can be shown with a similar proof based on the coupon collectors' problem as given in [3] for an appropriate optimized inversion based bogo-sort variant.

[6] The statistical programming language R is a industry standard to analyse huge amounts of data with a vast library of statistical functions available out of the box.

References

1 L. Comtet. *Advanced Combinatorics—The Art of Finite and Infinite Expansions*. Reidel Publishing, 1974.

2 H. W. Gould. *Combinatorial Identities*. Morgantown Printing and Binding, 1972.

3 Hermann Gruber, Markus Holzer, and Oliver Ruepp. Sorting the slow way: An analysis of perversely awful randomized sorting algorithms. In Pierluigi Crescenzi, Giuseppe Prencipe, and Geppino Pucci, editors, *Fun with Algorithms, 4th International Conference, FUN 2007, Castiglioncello, Italy, June 3-5, 2007, Proceedings*, volume 4475 of *Lecture Notes in Computer Science*, pages 183–197. Springer, 2007. doi:10.1007/978-3-540-72914-3_17.

4 D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1969.

5 D. H. Lehmer. Interesting series involving the central binomial coefficient. *Amer. Math. Mon.*, 92(7):449–457, – 1985. doi:10.2307/2322496.

6 E. S. Raymond. *The New Hacker's Dictionary*. MIT Press, 1996.

7 A. M. Rockett. Sums of inverses of binomial coefficients. *Fibonacci Quart.*, 19(5):433–437, 1981.

8 R. Sedgewick and P. Flajolet. *An Introduction to the Analysis of Algorithms*. Pearson Education, 2013.

9 R. Sprugnoli. Sum of reciprocals of the central binomial coefficients. *Integers*, 6:A27, 2006.

10 B. Sury, T. Wang, and F.-Z. Zhao. Identities involving reciprocals of binomial coefficients. *Integer Seq.*, 7:04.2.8, 2004.

Correction of Theorem 7

In Theorem 7 the expected value of the random variable C that counts the number of comparisons carried out by the procedure Z-PARTITIONED was determined to be

$$\mathbb{E}[C] = \sum_{j=1}^{k-1} \left[\frac{1}{2j-1} \binom{2(j-1)}{j-1}^{-1} + \frac{1}{j} \binom{2j}{j}^{-1} \right] + \frac{1}{k} \sum_{i=2k-1}^{n-1} \binom{i}{k}^{-1}. \tag{8}$$

Then the asymptotic value of $\mathbb{E}[C]$ that comes from the first sum was overestimated under the constraints that k is fixed and that n, the number of elements, tends to infinite. Under these prerequisites Theorem 7 must read as follows, where (8) is also simplified:

▶ **Theorem 7*.** *Assume $\overline{x}$ is a random permutation of $\{1, 2, \ldots, n\}$, and let C denote the random variable counting the number of comparisons carried out by the procedure* Z-PARTITIONED *in the test whether $\overline{x}$ is partitioned according to the kth element, for $1 \leq k \leq (n+1)/2$, then:*

$$\mathbb{E}[C] = 3 \sum_{j=1}^{k-1} \frac{1}{j} \binom{2j}{j}^{-1} + \frac{1}{k} \sum_{i=2k-1}^{n-1} \binom{i}{k}^{-1}$$

It holds $\mathbb{E}[C] = H_{n-1}$, if $k = 1$, and

$$\frac{\pi\sqrt{3}}{3} - \frac{1}{k-1} \binom{2(k-1)}{k-1}^{-1} \leq \mathbb{E}[C] \leq \frac{\pi\sqrt{3}}{3} + \frac{1}{k-1} \binom{2(k-1)}{k-1}^{-1},$$

otherwise. The statement remains valid in case k satisfies $(n+1)/2 < k \leq n$ by replacing k by $n - k + 1$.

Observe, that the left term of the first sum in (8) can be rewritten as

$$\frac{1}{2j-1} \binom{2(j-1)}{j-1}^{-1} = \frac{1}{2j-1} \cdot \frac{(j-1)!(j-1)!}{(2j-2)!} = \frac{2j}{j \cdot j} \cdot \frac{j!j!}{(2j)!} = \frac{2}{j} \binom{2j}{j}^{-1}$$

and therefore the former sum simplifies to

$$\sum_{j=1}^{k-1} \left[\frac{1}{2j-1} \binom{2(j-1)}{j-1}^{-1} + \frac{1}{j} \binom{2j}{j}^{-1} \right] = 3 \sum_{j=1}^{k-1} \frac{1}{j} \binom{2j}{j}^{-1}.$$

Next, in order to determine the bounds on $\mathbb{E}[C]$ we have to estimate the error in terms of k that is induced by partial summation. To this end we use the following upper and lower bound result from [1] on approximation of finite sums: let $S = \sum_{i=1}^{\infty} a_i$ and let the kth partial sum be $S_k = \sum_{i=1}^{k} a_i$. Suppose that the sequence $a_1, a_2, \ldots$ is a positive decreasing sequence and $\lim_{i \to \infty} \frac{a_{i+1}}{a_i} = q < 1$. Then we distinguish two cases:

1. If $\frac{a_{i+1}}{a_i}$ decreases to the limit q, then

$$S - \frac{a_{k+1}}{1 - \frac{a_{k+1}}{a_k}} \leq S_k \leq S - a_k \frac{q}{1-q}.$$

2. If $\frac{a_{i+1}}{a_i}$ increases to the limit q, then

$$S - a_k \frac{q}{1-q} \leq S_k \leq S - \frac{a_{k+1}}{1 - \frac{a_{k+1}}{a_k}}.$$

In other words, the kth partial sum S_k can be bounded from above and below in terms of the last element from the partial sum and its successor element, that is already outside the partial sum.

Consider the first sum

$$3\sum_{j=1}^{k-1}\frac{1}{j}\binom{2j}{j}^{-1} = \sum_{j=1}^{k-1} a_i,$$

in Theorem 7*, where $a_i = \frac{3}{i}\binom{2i}{i}^{-1}$. Obviously, the infinite sequence $a_1, a_2, \ldots$ is positive and decreasing and

$$\lim_{i\to\infty}\frac{a_{i+1}}{a_i} = \lim_{i\to\infty}\frac{3}{(i+1)}\cdot\frac{(i+1)!(i+1)!}{(2i+2)!}\cdot\frac{i}{3}\cdot\frac{(2i)!}{i!i!} = \lim_{i\to\infty}\frac{i(i+1)}{(2i+1)(2i+2)} = \frac{1}{4},$$

By inspection, the sequence $\frac{a_{i+1}}{a_i}$ increases to the limit $q := \frac{1}{4}$ leading to the second case for bounding the partial summation. Since $\frac{q}{1-q} = \frac{1}{3}$, we therefore bound the sum of the $(k-1)$st partial sum of the a_i's by

$$\frac{\pi\sqrt{3}}{9} - a_{k-1}\cdot\frac{1}{3} \leq \sum_{j=1}^{k-1} a_j \leq \sum_{j=1}^{\infty} a_j = \frac{\pi\sqrt{3}}{9},$$

where the last equality follows from Lehmer's identity. Since $a_{k-1} = \frac{3}{k-1}\binom{2(k-1)}{k-1}^{-1}$, together with the already correctly stated upper bound $\frac{1}{k-1}\binom{2(k-1)}{k-1}^{-1}$ of the second sum in Theorem 7* in the original proof, one observes that $\mathbb{E}[C]$ lies in between

$$\frac{\pi\sqrt{3}}{3} - \frac{1}{k-1}\binom{2(k-1)}{k-1}^{-1} \leq \mathbb{E}[C] \leq \frac{\pi\sqrt{3}}{3} + \frac{1}{k-1}\binom{2(k-1)}{k-1}^{-1}.$$

This proves the stated result.

References

1 B. Braden. Calculating sums of infinite series. *American Mathematical Monthly*, 99(7):649–655, August–September 1992.

Revision Notice

This is a revised version of the eponymous paper appeared in the proceedings of FUN 2018 (LIPIcs, volume 100, `http://www.dagstuhl.de/dagpub/978-3-95977-067-5`, published in May, 2018), in which Theorem 7 was revised to correct an overestimation of the asymptotic bound of the expected number of comparisons for one of the methods.

Dagstuhl Publishing – October 15, 2018.

Herugolf and Makaro are NP-complete

Chuzo Iwamoto[1]
Hiroshima University, Graduate School of Engineering, Higashi-Hiroshima 739-8527, Japan
chuzo@hiroshima-u.ac.jp

Masato Haruishi
Hiroshima University, Graduate School of Engineering, Higashi-Hiroshima 739-8527, Japan

Tatsuaki Ibusuki
Hiroshima University, School of Integrated Arts and Sciences, Higashi-Hiroshima 739-8521, Japan

Abstract

Herugolf and Makaro are Nikoli's pencil puzzles. We study the computational complexity of Herugolf and Makaro puzzles. It is shown that deciding whether a given instance of each puzzle has a solution is NP-complete.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases Herugolf, Makaro, pencil puzzle, NP-complete

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.24

1 Introduction

The Herugolf puzzle is played on a rectangular grid of cells (see Fig. 1(a)). Initially, there are circles (balls) and holes (H) on the grid, where an integer is in each circle. The purpose of the puzzle is to move (hit) all balls one or more times, and bring them to a cell with an H in the following rules [1]: (1) One ball must be brought to every hole H. (2) The movement of a ball is shown by an arrow, with the tip of the arrow in the cell where it stops. The arrows cannot cross other balls, holes, or lines of other arrows. (3) A ball moves across as many cells as the number in it in the first move, vertically or horizontally. The next move becomes one shorter; it decreases one by one. (4) The direction of movement may change after a move. When the next movement becomes 0, or the ball stops at an H, the ball cannot move any further. (5) A ball cannot leave the grid (OB), and cannot stop in a grey area (water hazard).

Figure 1(a) is the initial configuration of a Herugolf puzzle. In this figure, there are eight balls and eight holes in the 6 × 6 cells. From Figs. 1(b)–(f), the reader can understand the basic technique for finding a solution. (b) The bottom right ball ③ must be moved 3 cells to the left, since there is a hole H in the blue cell. Then the ball is moved 2 cells to the upper direction, since there is a water hazard in the bottom left cell. (c) There is exactly one ball which can be brought to the hole H in the red cell. On the other hand, there is exactly one hole which the ball in the blue cell can reach. (d) Balls ③ and ① are moved to holes H in the red and blue cells, respectively. (e) If the ball ② in the blue cell is brought to the hole H in the red cell, then one of the two balls ② in the green cells cannot reach any hole. (f) is a solution.

[1] This work was partially supported by JSPS KAKENHI Grant Number 16K00020.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 24; pp. 24:1–24:11
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

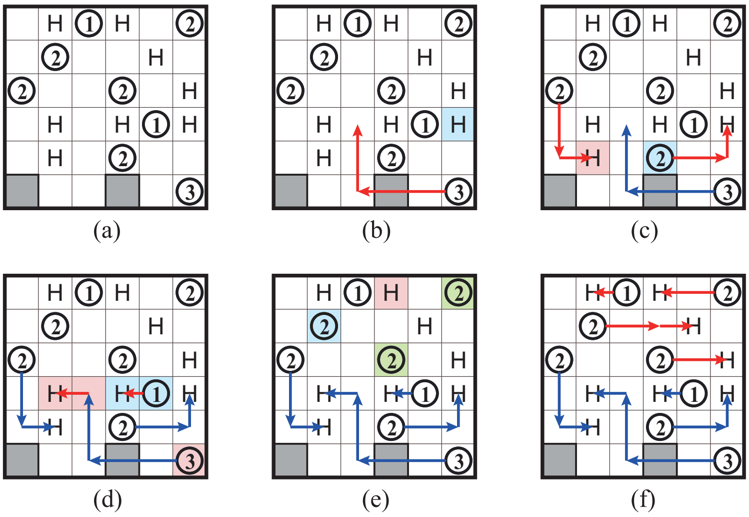


Figure 1 (a) Initial configuration of a Herugolf puzzle. (b)–(f) are the progress from the initial configuration to a solution.

The Makaro puzzle is also played on a rectangular grid of cells (see Fig. 2(a)). Initially, some of the cells are colored black and contain an arrow, and the remaining cells are divided into *rooms* surrounded by bold lines. The purpose of the puzzle is to fill in all white cells with numbers under the following rules [2]: (1) Each room contains all the natural numbers up to the number of cells in it, starting from 1 (see Fig. 2(h)). (2) Every arrow in black cells must point at the biggest number among the numbers in the adjacent cells. (3) A number must not be next to the same number in another room.

Figure 2(a) is the initial configuration of a Makaro puzzle. (b) Since an arrow exists between two yellow 2-cell rooms, numbers 2 and 1 are placed so that the arrow is between them and points at the number 2, which is bigger than the other number 1. (c) Four numbers are placed in the blue cells. (d) Since number 2 in a green cell is pointed by an arrow, another green cell must contain 1. (e) Since the yellow cell contains number 2, one of the two red cells must contain 2; therefore the green and blue cells must contain 3 and 4, respectively. (f) Since the two red cells can contain numbers less than 3, the number 3 is in the blue cell. (g) Two and seven numbers are placed in the yellow and red cells, respectively. (h) is a solution.

In this paper, we study the computational complexity of the decision version of the Herugolf and Makaro puzzles. The instance of the *Herugolf puzzle problem* is defined as a rectangular grid of cells, on which there are circled integers in {①, ②, ③, . . .} and holes H. The instance of the *Makaro puzzle problem* is a rectangular grid of cells, where some of the cells are colored black and contain an arrow, and the remaining cells are divided into rooms. The problem is to decide whether there is a solution to the instance. It is shown that the Herugolf and Makaro puzzle problems are NP-complete. It is clear that both problems belong to NP.

There has been a huge amount of literature on the computational complexities of games and puzzles. In 2009, a survey of games, puzzles, and their complexities was reported by Hearn and Demaine [9]. After the publication of this book, the following Nikoli's pencil puzzles were shown to be NP-complete: Fillmat [16], Hashiwokakero [4], Hebi, Satogaeri, and Suraromu [12], Kurodoko [13], LITS and Norinori [5], Numberlink [3], Pipe link [17], Shakashaka [7], Shikaku and Ripple Effect [15], Yajilin and Country Road [10], and Yosenabe [11].

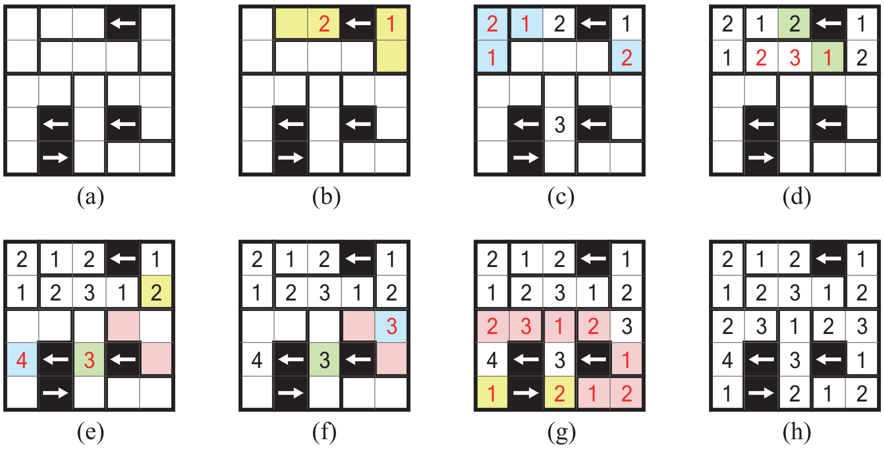


Figure 2 (a) Initial configuration of a Makaro puzzle. (b)–(h) are the progress from the initial configuration to a solution.

2 NP-completeness of Herugolf and Makaro

2.1 3SAT Problem

The definition of 3SAT is mostly from [8]. Let $U = \{x_1, x_2, \ldots, x_n\}$ be a set of Boolean *variables*. Boolean variables take on values 0 (false) and 1 (true). If x is a variable in U, then x and $\overline{x}$ are *literals* over U. The value of $\overline{x}$ is 1 (true) if and only if x is 0 (false). A *clause* over U is a set of literals over U, such as $\{\overline{x_1}, x_3, x_4\}$. It represents the disjunction of those literals and is *satisfied* by a truth assignment if and only if at least one of its members is true under that assignment.

An instance of PLANAR 3SAT is a collection $C = \{c_1, c_2, \ldots, c_m\}$ of clauses over U such that (i) $|c_j| \leq 3$ for each $c_j \in C$ and (ii) the bipartite graph $G = (V, E)$, where $V = U \cup C$ and E contains exactly those pairs $\{x, c\}$ such that either literal x or $\overline{x}$ belongs to the clause c, is planar.

The PLANAR 3SAT problem asks whether there exists some truth assignment for U that simultaneously satisfies all the clauses in C. This problem is known to be NP-complete. For example, $U = \{x_1, x_2, x_3, x_4\}$, $C = \{c_1, c_2, c_3, c_4\}$, and $c_1 = \{x_1, x_2, x_3\}$, $c_2 = \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}$, $c_3 = \{\overline{x_1}, \overline{x_3}, x_4\}$, $c_4 = \{\overline{x_2}, \overline{x_3}, \overline{x_4}\}$ provide an instance of PLANAR 3SAT. For this instance, the answer is "yes," since there is a truth assignment $(x_1, x_2, x_3, x_4) = (0, 1, 0, 0)$ satisfying all clauses. It is known that PLANAR 3SAT is NP-complete even if each variable occurs exactly once positively and exactly twice negatively in C [6].

2.2 Transformation from an Instance of 3SAT to a Herugolf Puzzle

We present a polynomial-time transformation from an arbitrary instance C of 3SAT to a Herugolf puzzle such that C is satisfiable if and only if the puzzle has a solution.

Each variable $x_i \in \{x_1, x_2, \ldots, x_n\}$ is transformed into the variable gadget as illustrated in Fig. 3(a), which is composed of six holes and eight balls. (Each gadget constructed in this section uses no water hazards, although some of the cells in Fig. 3(a) are colored grey.) Note that the instances of 3SAT considered in this section have the restriction explained at the end of Sect. 2.1.

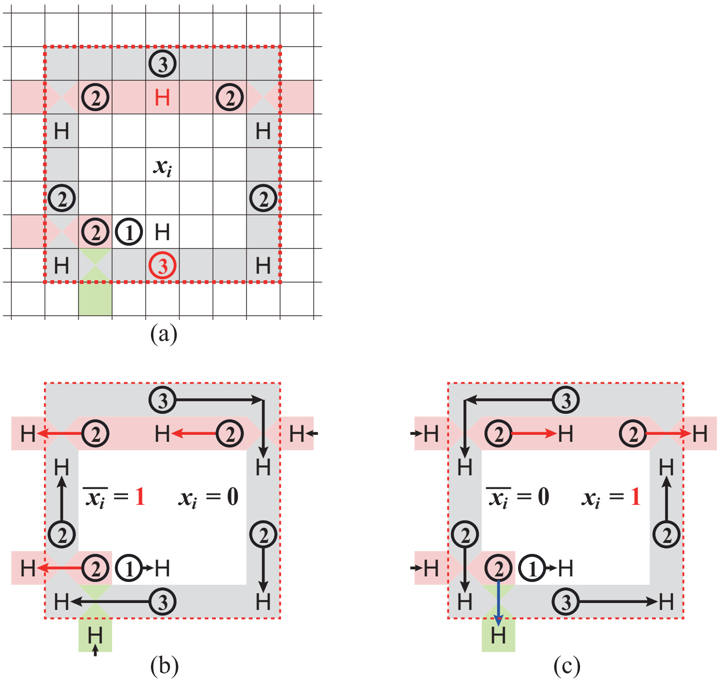


Figure 3 (a) Variable gadget of Herugolf transformed from x_i. (b) Assignment $\overline{x_i} = 1$. (c) Assignment $x_i = 1$.

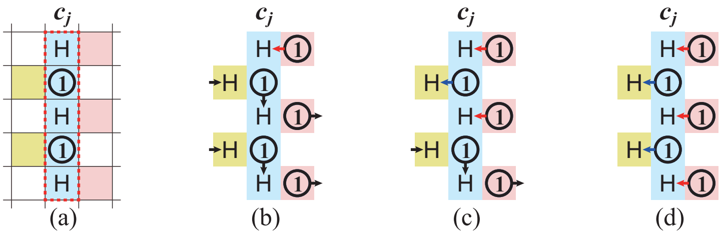


Figure 4 (a) Clause gadget of Herugolf transformed from c_j. (b)–(d) If at least one of the three balls ① is moved from a red cell to a blue cell, then the three holes H in the blue cells receive three balls.

Consider the four balls and four holes in the grey area of Fig. 3(b). There are two possible solutions to those balls. Suppose that the top ball ③ is moved 3 cells to the right and then 2 cells to the downward direction. Then the left ball ② (resp. right ball ②) in the grey area must be moved upward (resp. downward), and the bottom ball ③ must be moved 3 cells to the left. In this case, three balls ② in the red cells can be moved 2 cells to the left. This configuration corresponds to $\overline{x_i} = 1$. (Note that four holes H outside the red dotted square belong to the connection gadget, which will be explained later.)

On the other hand, if the top ball ③ is moved 3 cells to the left and then 2 cells to the downward direction (see Fig. 3(c)), then two of the three balls ② in the red cells can be moved 2 cells to the right, and the remaining one ball ② can be moved downward. This corresponds to $x_i = 1$. In Fig. 3(a), the red ball ③ cannot reach the red hole H, since there is a hole H just above the red ball ③.

Clause $c_j \in \{c_1, c_2, \ldots, c_m\}$ is transformed into the clause gadget as illustrated in the blue cells of Fig. 4(a), which is composed of three holes H and two balls ①. If either literal x_i

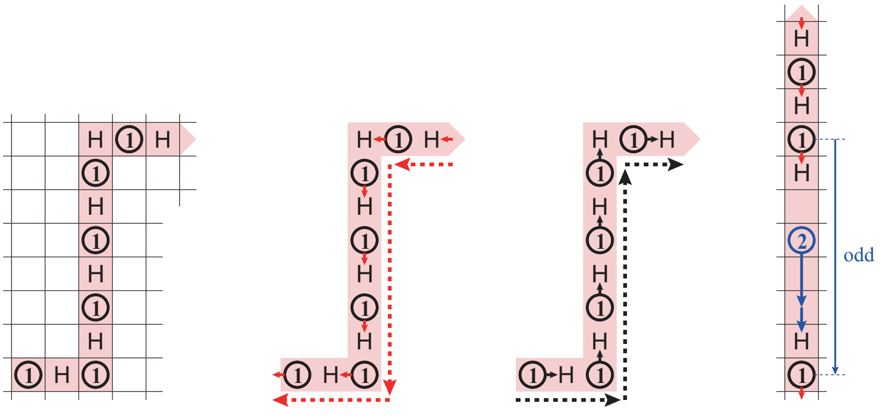


Figure 5 (a) Connection gadget of Herugolf. (b) and (c) are two possible movements. (d) The distance between two balls ① is odd.

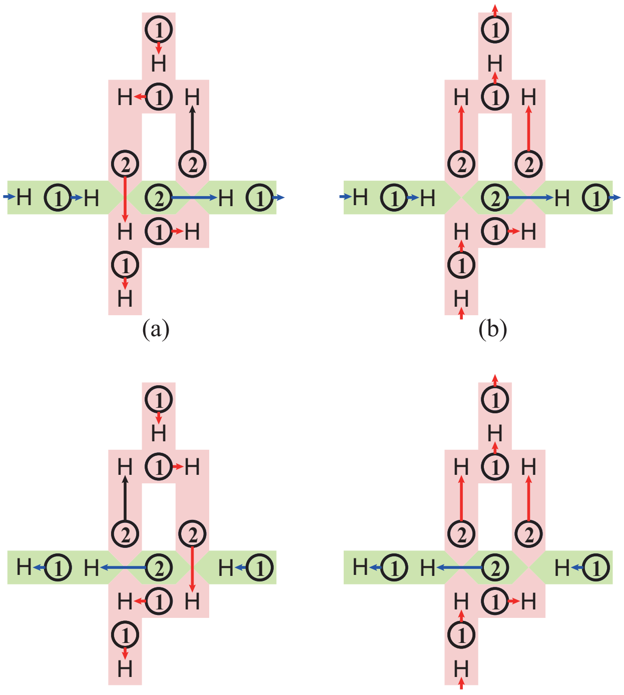
or $\overline{x_i}$ belongs to the clause c_j, then clause gadget c_j is connected to a variable gadget x_i by using the connection gadget as illustrated in Fig. 5(a) (see also Fig. 8). The three holes of a clause gadget can each receive balls if and only if at least one of the three balls are moved into one of these holes from a neighboring red cell. The cases are illustrated in Figs. 4(b)–(d). If clause c_j contains only two literals, Fig. 4(a) is replaced with 3 × 1 blue cells containing "H,①,H." (The clauses of two literals are essential, since it is known that 3SAT with exactly three occurrences per variable is polynomial-time solvable if every clause has three literals [14].)

Figure 5(a) is a connection gadget connecting between variable and clause gadgets. In Figs. 5(b) and 5(c), a "signal" is transmitted from the top right hole H to the bottom left ball ① and vice versa. Namely, if the top right hole of Fig. 5(b) receives a ball from the right side, then the bottom left ball can be moved to the left. If you want the distance between two balls ① to be odd (see Fig. 5(d)), then ball ② is used in a connection gadget. Figure 6 is a crossover gadget.

In each variable gadget (see Fig. 3(a)), the number of balls is two larger than the number of holes. In each clause gadget (see Fig. 4(a)), the number of balls is one smaller than the number of holes. Therefore, the number of balls is $2n - m$ larger than the number of holes in total, where n and m are the numbers of variables and clauses, respectively. Finally, we add a terminator gadget as illustrated in the red dotted rectangle of size $(2n-m) \times (2n+4m-1)$ of Fig. 7. The top row of the terminator gadget is an alternating sequence of "H①H①···①H" of length $2n + 4m - 1$. The second row is an alternating sequence of length $2n + 4m - 3$, and so on.

Every pair of "H,①" in green cells of Fig. 7 is connected to a hole H in the green cell of Fig. 3(a) or a hole H in the yellow cells of Fig. 4(a) by a connection gadget (see Fig. 8). In the terminator gadget, the number of holes is $2n - m$ larger than the number of balls, so $2n - m$ signals are terminated in the terminator gadget. (In Fig. 8, $2n - m$ $(= 4)$ signals are terminated at the leftmost $2n - m$ holes in the red dotted rectangle.)

Figure 8 is a Herugolf puzzle transformed from $C = \{c_1, c_2, c_3, c_4\}$ and $U = \{x_1, x_2, x_3, x_4\}$, where $c_1 = \{x_1, x_2, x_3\}$, $c_2 = \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}$, $c_3 = \{\overline{x_1}, \overline{x_3}, x_4\}$, $c_4 = \{\overline{x_2}, \overline{x_3}, \overline{x_4}\}$. In this figure, several pairs of a black ball and a hole are placed in the white areas so that balls in grey, red, green, and yellow areas do not move to unintended directions. (If variable and clause gadgets


(a) (b)

(c) (d)

Figure 6 Crossover gadget of Herugolf.

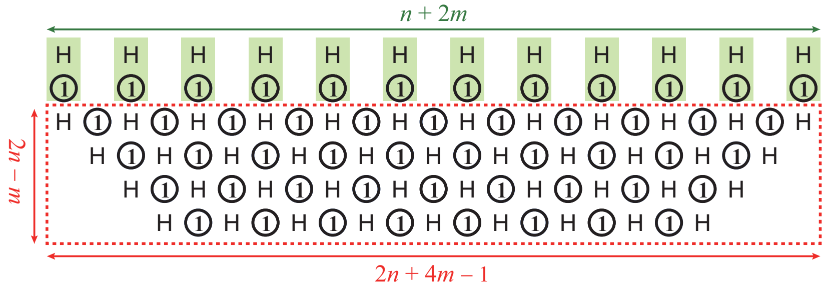


Figure 7 Terminator gadget of Herugolf.

are embedded on a sufficiently large space, no such pair is required.) From this construction, the instance C of 3SAT is satisfiable if and only if the corresponding Herugolf puzzle has a solution.

2.3 Transformation from an Instance of 3SAT to a Makaro Puzzle

We present a polynomial-time transformation from an arbitrary instance C of PLANAR 3SAT to a Makaro puzzle such that C is satisfiable if and only if the puzzle has a solution.

Each variable $x_i \in \{x_1, x_2, \ldots, x_n\}$ is transformed into the variable gadget as illustrated in Fig. 9(a), which is composed of three grey 2-cell rooms. Figures 9(b) and 9(c) correspond to $\overline{x_i} = 1$ and $x_i = 1$, respectively. Note that the instances of PLANAR 3SAT considered in this section have the restriction explained at the end of Sect. 2.1.

Clause $c_j \in \{c_1, c_2, \ldots, c_m\}$ is transformed into the clause gadget as illustrated in Fig. 10(a), which is composed of one blue 4-cell room, three black cells, two grey 4-cell rooms, and four 1-cell rooms. If all of the three black cells are adjacent to number 2 in red 2-cell rooms (see Fig. 10(b)), then there is no solution to the blue 4-cell room. On the other hand, if at least one of the three black cells is adjacent to number 1 in a red 2-cell room (see Figs. 10(c) and 10(d), and Fig. 12), then there is a solution to the blue 4-cell room. If clause c_j contains only two literals, the gadget is composed of one blue 3-cell room, two black cells, and four grey 1-cell rooms (see Figs. 10(e)–(g)).

Fig. 11(a) is a connection gadget connecting between variable and clause gadgets. In Figs. 11(b) and 11(c), a “signal” is transmitted from the top right room to the bottom left room and vice versa. If you want the distance between two 2-cell rooms to be odd, you can use a gadget of Fig. 11(d).

Figure 12 is a Makaro puzzle transformed from $C = \{c_1, c_2, c_3, c_4\}$ and $U = \{x_1, x_2, x_3, x_4\}$, where $c_1 = \{x_1, x_2, x_3\}$, $c_2 = \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}$, $c_3 = \{\overline{x_1}, \overline{x_3}, x_4\}$, $c_4 = \{\overline{x_2}, \overline{x_3}, \overline{x_4}\}$. In this figure, there are six large white rooms separated by connection, variable, and clause gadgets. Those white rooms can easily be filled with numbers $1, 2, 3, \cdots$. From this construction, the instance C of PLANAR 3SAT is satisfiable if and only if the corresponding Makaro puzzle has a solution.

References

1 `http://nikoli.co.jp/en/puzzles/herugolf.html`.

2 `http://nikoli.co.jp/en/puzzles/makaro.html`.

3 Aaron B. Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O’Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-zag numberlink is NP-complete. *J. Inf. Process.*, 23(3):239–245, 2015. doi:10.2197/ipsjjip.23.239.

4 Daniel Andersson. Hashiwokakero is NP-complete. *Inf. Process. Lett.*, 109:1145–1146, 2009. doi:10.1016/j.ipl.2009.07.017.

5 Michael Biro and Christiane Schmidt. Computational complexity and bounds for Norinori and LITS. In *33rd European Workshop on Computational Geometry, Malmö, Sweden, April 5–7, 2017*, pages 29–32, 2017.

6 M.R. Cerioli, L. Faria, T.O. Ferreira, C.A.J. Martinhon, F. Protti, and B. Reed. Partition into cliques for cubic graphs: planar case, complexity and approximation. *Discrete Appl. Math.*, 156(12):2270–2278, 2008. doi:10.1016/j.dam.2007.10.015.

7 Erik D. Demaine, Yoshio Okamoto, Ryuhei Uehara, and Yushi Uno. Computational complexity and an integer programming model of Shakashaka. In *25th Canadian Conference on Computational Geometry, Waterloo, Ontario, Canada, August 8–10, 2013*, (online) `http://cccg.ca/`.

8 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

9 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A K Peters Ltd., 2009.

10 Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. NP-completeness of two pencil puzzles: Yajilin and Country Road. *Utilitas Mathematica*, 88:237–246, 2012.

11 Chuzo Iwamoto. Yosenabe is NP-complete. *J. Inf. Process.*, 22(1):40–43, 2014. doi:10.2197/ipsjjip.22.40.

12 Shohei Kanehiro and Yasuhiko Takenaga. Satogaeri, Hebi and Suraromu are NP-complete. In *3rd Intl. Conf. on Applied Computing and Information Technology, Okayama, Japan, July 12–16, 2015*, pages 47–52.

13 Jonas Kölker. Kurodoko is NP-complete. *J. Inf. Process.*, 20(3):694–706, 2012. doi:10.2197/ipsjjip.20.694.

14 Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

15 Yasuhiko Takenaga, Shintaro Aoyagi, Shigeki Iwata, and Takumi Kasai. Shikaku and Ripple Effect are NP-complete. *Congressus Numerantium*, 216:119–127, 2013.

16 Akihiro Uejima and Hiroaki Suzuki. Fillmat is NP-complete and ASP-complete. *J. Inf. Process.*, 23(3):310–316, 2015. doi.org/10.2197/ipsjjip.23.310.

17 Akihiro Uejima, Hiroaki Suzuki, and Atsuki Okada. The complexity of generalized pipe link puzzles. *J. Inf. Process.*, 25:724–729, 2017. doi:10.2197/ipsjjip.25.724.

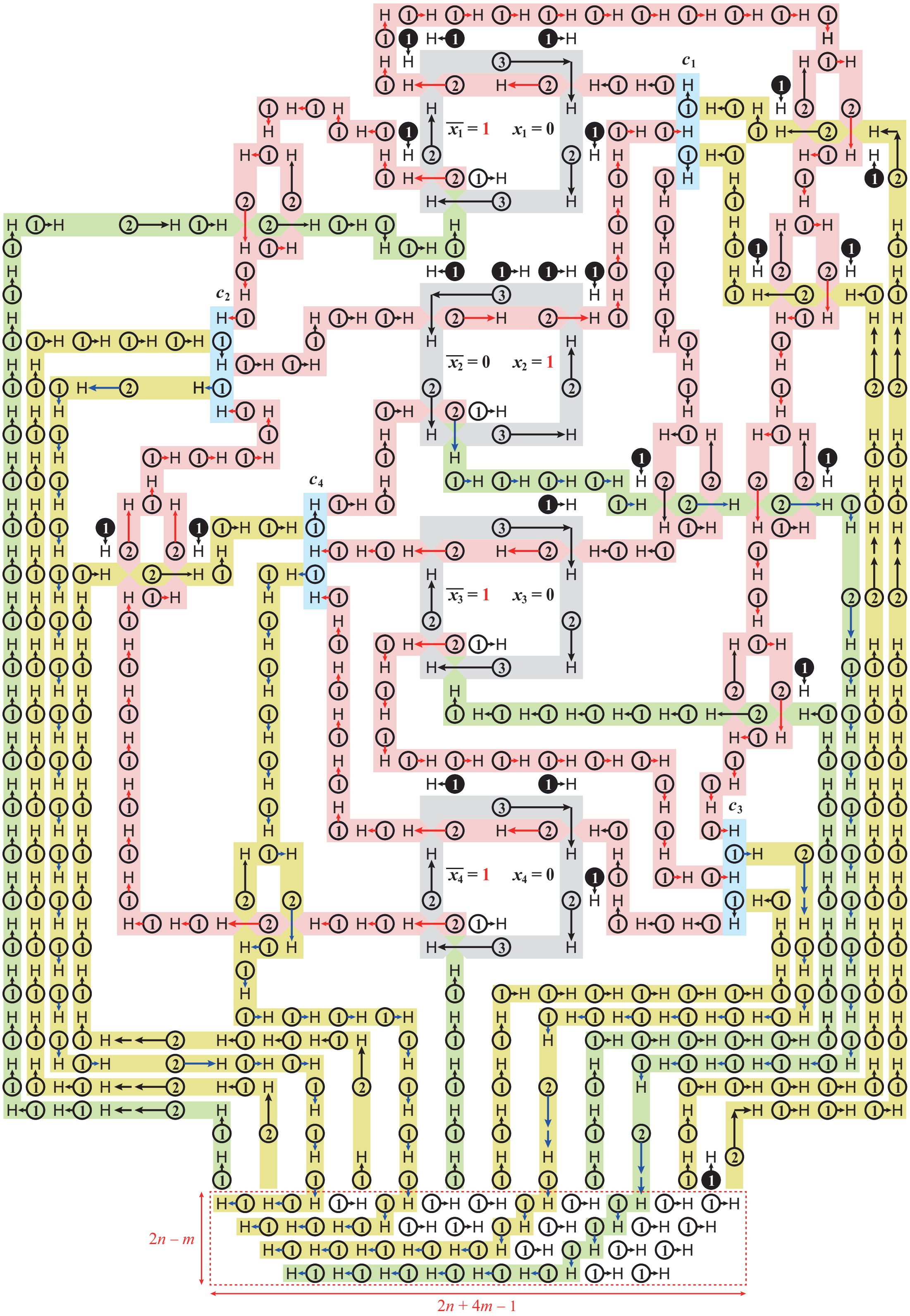


Figure 8 A Herugolf puzzle transformed from $C = \{c_1, c_2, c_3, c_4\}$, where $c_1 = \{x_1, x_2, x_3\}$, $c_2 = \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}$, $c_3 = \{\overline{x_1}, \overline{x_3}, x_4\}$, $c_4 = \{\overline{x_2}, \overline{x_3}, \overline{x_4}\}$. From the solution of the puzzle, one can see that the assignment $(x_1, x_2, x_3, x_4) = (0, 1, 0, 0)$ satisfies all clauses of C.

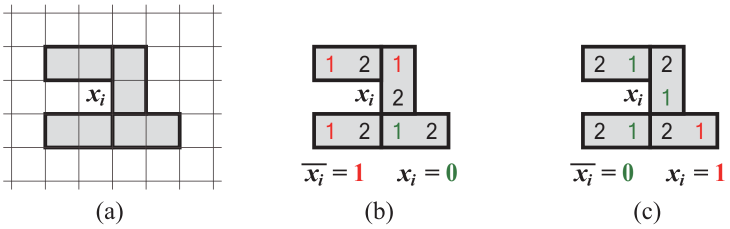

x_i
1 2 1
x_i 2
1 2 1 2
$\overline{x_i} = 1$ $x_i = 0$
2 1 2
x_i 1
2 1 2 1
$\overline{x_i} = 0$ $x_i = 1$
(a)
(b)
(c)

Figure 9 (a) Variable gadget of Makaro transformed from x_i. (b) Assignment $\overline{x_i} = 1$. (c) Assignment $x_i = 1$.

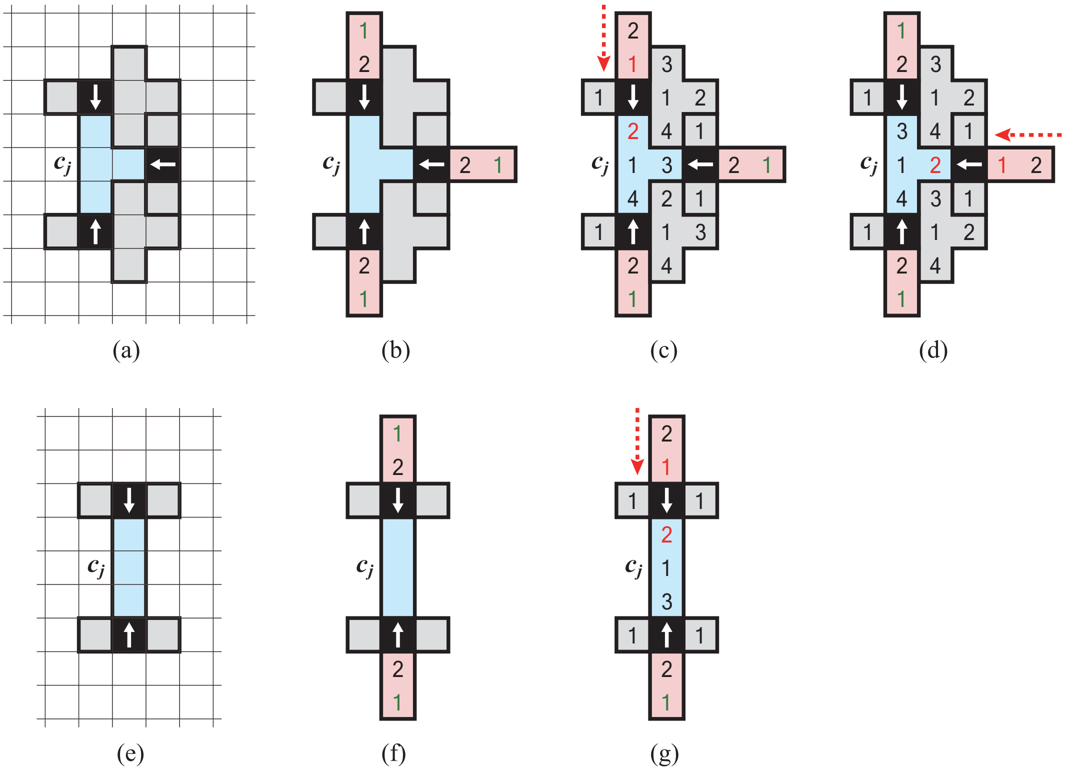


Figure 10 (a) Clause gadget of Makaro transformed from c_j. (b) If all of the three black cells are adjacent to number 2 in red 2-cell rooms, then there is no solution to the blue 4-cell room. (c),(d) If at least one of the three black cells is adjacent to number 1 in a red 2-cell room, then there is a solution to the blue room. (e)–(g) If clause c_j contains only two literals, the gadget is composed of one blue 3-cell room, two black cells, and four grey 1-cell rooms.

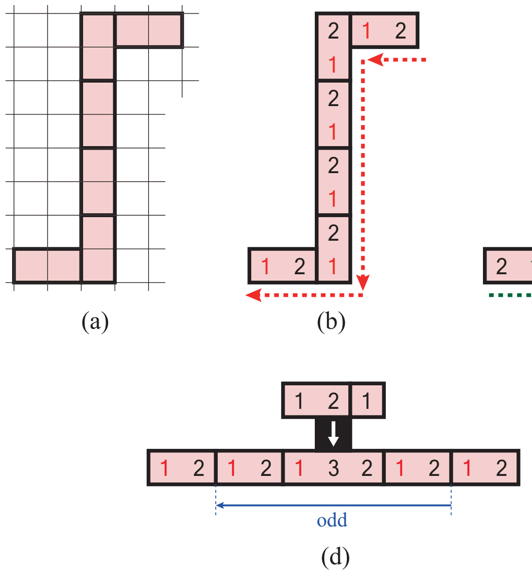


Figure 11 (a) Connection gadget of Makaro. (b) and (c) are two possible solutions. (d) The interval between two 2-cell rooms is odd.

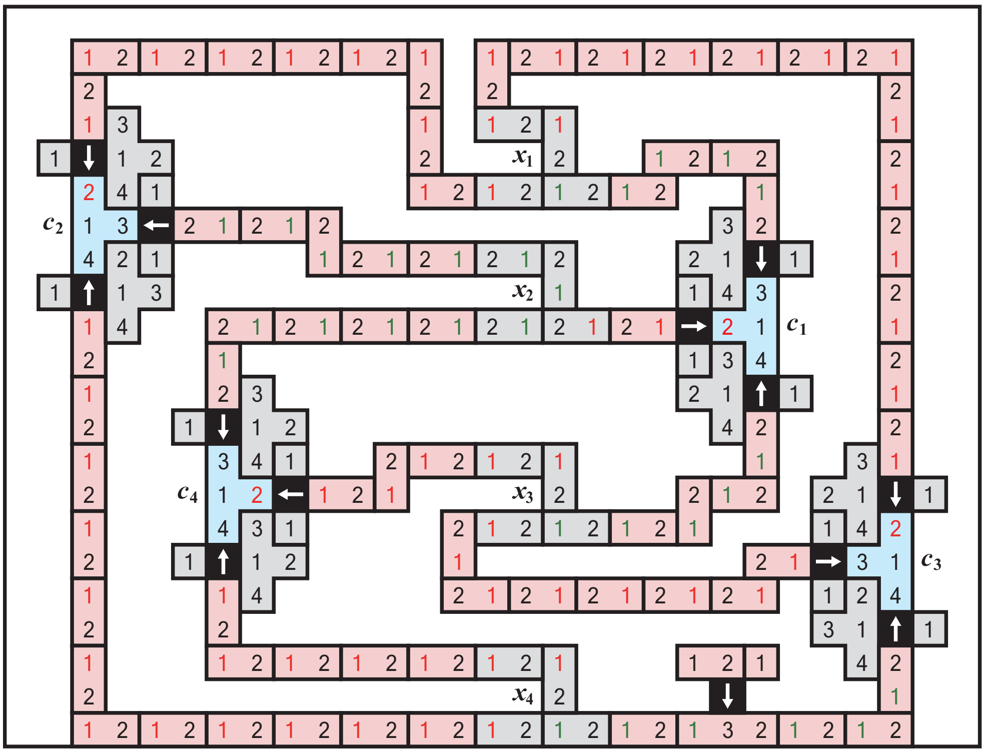


Figure 12 A Makaro puzzle transformed from $C = \{c_1, c_2, c_3, c_4\}$, where $c_1 = \{x_1, x_2, x_3\}$, $c_2 = \{\overline{x_1}, \overline{x_2}, \overline{x_4}\}$, $c_3 = \{\overline{x_1}, \overline{x_3}, x_4\}$, $c_4 = \{\overline{x_2}, \overline{x_3}, \overline{x_4}\}$. From the solution of the puzzle, one can see that the assignment $(x_1, x_2, x_3, x_4) = (0, 1, 0, 0)$ satisfies all clauses of C.

The Fewest Clues Problem of Picross 3D

Kei Kimura[1]
Department of Computer Science and Engineering, Toyohashi University of Technology, 1-1 Hibarigaoka, Tempaku, Toyohashi, Aichi, Japan
kimura@cs.tut.ac.jp

Takuya Kamehashi
Department of Computer Science and Engineering, Toyohashi University of Technology, 1-1 Hibarigaoka, Tempaku, Toyohashi, Aichi, Japan
kamehashi@algo.cs.tut.ac.jp

Toshihiro Fujito[2]
Department of Computer Science and Engineering, Toyohashi University of Technology, 1-1 Hibarigaoka, Tempaku, Toyohashi, Aichi, Japan
fujito@cs.tut.ac.jp

Abstract

Picross 3D is a popular single-player puzzle video game for the Nintendo DS. It is a 3D variant of Nonogram, which is a popular pencil-and-paper puzzle. While Nonogram provides a rectangular grid of squares that must be filled in to create a picture, Picross 3D presents a rectangular parallelepiped (i.e., rectangular box) made of unit cubes, some of which must be removed to construct an image in three dimensions. Each row or column has at most one integer on it, and the integer indicates how many cubes in the corresponding 1D slice remain when the image is complete. It is shown by Kusano et al. that Picross 3D is NP-complete. We in this paper show that the fewest clues problem of Picross 3D is Σ_2^P-complete and that the counting version and the another solution problem of Picross 3D are #P-complete and NP-complete, respectively.

2012 ACM Subject Classification Theory of computation → Theory and algorithms for application domains

Keywords and phrases Puzzle, computational complexity, fewest clues problem

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.25

Acknowledgements The authors are grateful to the reviewers for their valuable comments and suggestions. Especially, one of them pointed out the connection between Picross 3D and discrete tomography.

1 Introduction

Many pencil-and-paper puzzles have been shown to be NP-complete [7]. For example, Akari (also known as Light-ups) [11], Number Place (also known as Sudoku) [13], Shakashaka [5] are all known to be NP-complete. Different from this line of research, Demaine et al. [4] recently introduced the fewest clues problem (FCP) framework for analyzing computational complexity of designing "good" puzzles. The FCP is, given an instance to a puzzle, to decide the minimum number of clues we must add in order to make the instance uniquely solvable. It is of great interest for puzzle makers to know hardness of such a version since it

[1] The first author is supported by JSPS KAKENHI Grant Number JP15H06286.
[2] The third author is supported by JSPS KAKENHI Grant Number JP17K00013.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 25; pp. 25:1–25:13
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is usually the case that they want to ensure a puzzle instance to have a unique solution. In [4], along with the FCP versions of classical NP-complete problems such as 3-SAT, those of the three common Nikoli puzzles (Akari, Number Place, and Shakashaka) are shown to be Σ_2^{P}-complete. Here, Σ_2^{P} is the complexity class that lies on the second level of the polynomial hierarchy and includes the class NP. Hence, Σ_2^{P}-complete problems are at least as hard as NP-complete problems. See, e.g., [1] for more details.

We in this paper investigate computational complexity of the FCP of Picross 3D and show that it is Σ_2^{P}-complete. Picross 3D is a video-game puzzle developed by HAL Laboratory, published by Nintendo, and was first released in 2009. While 2-dimensional Picross (also known as Nonogram) provides a rectangular grid of squares that must be filled in to create a picture, Picross 3D presents a rectangular parallelepiped (i.e., rectangular box) made of unit cubes, some of which must be removed to construct an image in three dimensions. Each row or column has at most one integer on it, and the integer indicates how many cubes in the corresponding 1D slice remain when the image is complete. If the integer is not circled nor boxed, then the remaining cubes in the 1D slice must form a section (i.e., the cubes must be consecutive). If the integer is circled, then the remaining cubes in the 1D slice must be split up into two sections. If the integer is boxed, then the cubes must be split up into three or more sections. If there are no numbers on a row or column, then there are no rules concerning the number of cubes (or sections) to remain. An instance of Picross 3D is shown in Figure 1(a), and its solution is given in 1(b).

As many other puzzles, Picross 3D is shown to be NP-complete via a reduction from 3-SAT [10]. To show the Σ_2^{P}-completeness of the FCP of Picross 3D, we reduce to it the FCP of positive 1-in-3 SAT, which is known to be Σ_2^{P}-complete [4][3]. We note that those Nikoli puzzles were chosen in [4] because their NP-hardness reductions mostly preserve clue structure and their FCP versions were shown Σ_2^{P}-complete by using the same reductions or slightly modified ones. On the other hand, we cannot do the same for the FCP of Picross 3D using the NP-hardness reduction of [10]; we instead modify it to devise a parsimonious reduction from positive 1-in-3 SAT to Picross 3D. Here, a reduction is called *parsimonious* if, for each instance, there exists a one-to-one correspondence between the solution sets of the original instance and the reduced one. Intuitively, since a parsimonious reduction preserves the number of solutions, it helps to provide a reduction that preserves the number of clues. Moreover, it follows from the above parsimonious reduction that (i) the counting version of Picross 3D is #P-complete since so is the counting version of positive 1-in-3 SAT [2, 3][4], and (ii) the another solution problem (ASP) of Picross 3D is NP-complete[5] since so is ASP positive 1-in-3 SAT [12, 13]. Here, ASP Picross 3D is, given an instance of Picross 3D and a solution to it, to determine if there exists another solution to the instance.

We now discuss related work. Picross 3D can be seen as a variant of problems that have been studied in the field of *(3D) discrete tomography.* Discrete tomography deals with problems of determining shape of a discrete object from a set of projections. These problems have applications in, e.g., physical chemistry, medicine, and data coding, and have strong connections with combinatorics and geometry; see [8] for details. Especially, Picross 3D without the consecutiveness conditions on solutions is a basic problem in discrete tomography and intensively studied from an algorithmic point of view; the problem is known to be NP-,

[3] More precisely, the FCP of 1-in-3 SAT is shown Σ_2^{P}-complete in [4]. However, the proof in [4] also shows the Σ_2^{P}-completeness of the FCP of *positive* 1-in-3 SAT.

[4] More precisely, the counting version of monotone 1-in-3 SAT (i.e., each clause has only positive literals or only negative literals) is shown #P-complete in [2]. However, it is not difficult to modify the proof in [2] to show the #P-completeness of the counting version of positive 1-in-3 SAT.

[5] This fact was pointed out by a reviewer.

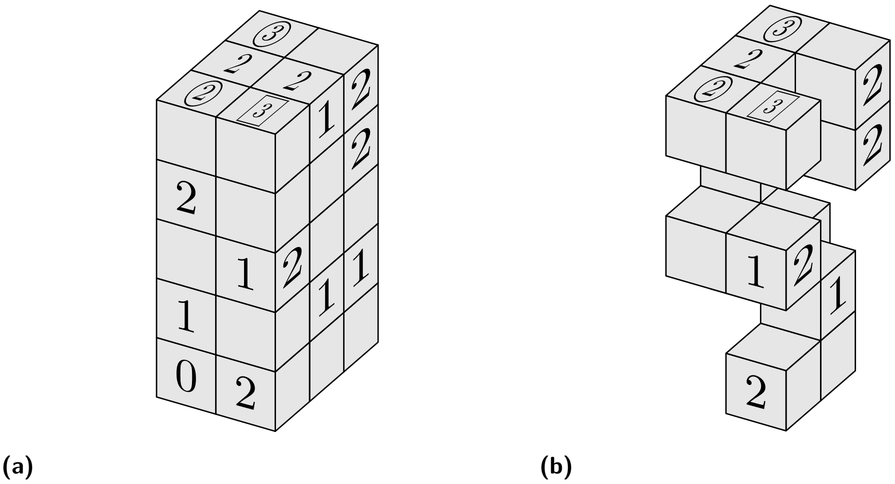


Figure 1 An instance of Picross 3D and its solution

ASP-, and #P-complete [9, 6]. Moreover, the 2D version of the problem, which corresponds to Nonogram, can be solved in polynomial time [8]. We note that the reductions in [9, 6] cannot be used to show our results.

2 Preliminaries

In this section, we first introduce a formal definition of the puzzle Picross 3D and then introduce the fewest clues problems (FCPs) of Picross 3D and positive 1-in-3 SAT.

2.1 Picross 3D

In Picross 3D, we are given a rectangular parallelepiped of height h, width w, and depth d. Each unit square in the front, side and top faces have at most one nonnegative integer that indicates how many cubes the row or column should contain when the image is complete. These integers are conveniently represented by three matrices: an $h \times w$ matrix $F = (f_{i,j})$ called the *front constraint matrix*, an $h \times d$ matrix $S = (s_{i,k})$ called the *side constraint matrix*, and a $d \times w$ matrix $T = (t_{j,k})$ called the *top constraint matrix*. Each element of these matrices is either an integer, a circled integer (e.g., ①), a boxed integer (e.g., $\boxed{1}$), or ε. Here, ε indicates that there is no constraint concerning the remaining cubes in the corresponding row or column. We denote by $I = (h, w, d, F, S, T)$ an instance of Picross 3D.

For the sake of clarity, we index these matrices as follows.

$$F = \begin{pmatrix} f_{1,1} & f_{1,2} & \cdots & f_{1,w} \\ f_{2,1} & f_{2,2} & \cdots & f_{2,w} \\ \vdots & \vdots & \ddots & \vdots \\ f_{h,1} & f_{h,2} & \cdots & f_{h,w} \end{pmatrix}, S = \begin{pmatrix} s_{1,1} & s_{1,2} & \cdots & s_{1,d} \\ s_{2,1} & s_{2,2} & \cdots & s_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ s_{h,1} & s_{h,2} & \cdots & s_{h,d} \end{pmatrix},$$

$$T = \begin{pmatrix} t_{1,d} & t_{2,d} & \cdots & t_{w,d} \\ t_{1,d-1} & t_{2,d-1} & \cdots & t_{w,d-1} \\ \vdots & \vdots & \ddots & \vdots \\ t_{1,1} & t_{2,1} & \cdots & t_{w,1} \end{pmatrix}.$$

Note that j is the index of the columns and k is the index (from bottom to top) of the rows of $T = (t_{j,k})$.

A solution to an instance $I = (h, w, d, F, S, T)$ of Picross 3D is a three dimensional matrix $P = (p_{i,j,k}) \in \{0,1\}^{h \times w \times d}$ that satisfies the following conditions: each integer (even if circled or boxed) indicates the number of 1's in the column or row where the integer is written. Namely, we must have $\sum_{k=1}^{d} p_{i,j,k} = f_{i,j}$ for each i and j, $\sum_{j=1}^{w} p_{i,j,k} = s_{i,k}$ for each i and k, and $\sum_{i=1}^{h} p_{i,j,k} = t_{j,k}$ for each j and k. Moreover, (i) if the integer is not circled nor boxed, then all the 1's in the row or column must be consecutive, (ii) if the integer is circled, then the row or column must contain exactly two sections that consecutively consist of only 1's, and (iii) if the integer is boxed, then the row or column must contain more than two sections that consecutively consist of only 1's. We describe a solution to Picross 3D as a sequence of matrices as follows.

$$P = \left(\begin{pmatrix} p_{1,1,1} & p_{1,2,1} & \cdots & p_{1,w,1} \\ p_{2,1,1} & p_{2,2,1} & \cdots & p_{2,w,1} \\ \vdots & \vdots & \ddots & \vdots \\ p_{h,1,1} & p_{h,2,1} & \cdots & p_{h,w,1} \end{pmatrix}, \begin{pmatrix} p_{1,1,2} & p_{1,2,2} & \cdots & p_{1,w,2} \\ p_{2,1,2} & p_{2,2,2} & \cdots & p_{2,w,2} \\ \vdots & \vdots & \ddots & \vdots \\ p_{h,1,2} & p_{h,2,2} & \cdots & p_{h,w,2} \end{pmatrix}, \dots, \begin{pmatrix} p_{1,1,d} & p_{1,2,d} & \cdots & p_{1,w,d} \\ p_{2,1,d} & p_{2,2,d} & \cdots & p_{2,w,d} \\ \vdots & \vdots & \ddots & \vdots \\ p_{h,1,d} & p_{h,2,d} & \cdots & p_{h,w,d} \end{pmatrix} \right).$$

▶ **Example 1.** Let $h = 5, w = 2$, and $d = 3$, and define the constraint matrices as follows.

$$F = \begin{pmatrix} \varepsilon & \varepsilon \\ 2 & \varepsilon \\ \varepsilon & 1 \\ 1 & \varepsilon \\ 0 & 2 \end{pmatrix}, S = \begin{pmatrix} \varepsilon & 1 & 2 \\ \varepsilon & \varepsilon & 2 \\ 2 & \varepsilon & \varepsilon \\ \varepsilon & 1 & 1 \\ \varepsilon & \varepsilon & \varepsilon \end{pmatrix}, T = \begin{pmatrix} \textcircled{3} & \varepsilon \\ 2 & 2 \\ \textcircled{2} & \boxed{3} \end{pmatrix}.$$

Then, $I = (h, w, d, F, S, T)$ represents the instance given in Figure 1(a).

Let

$$P = \left(\begin{pmatrix} 1 & 1 \\ 0 & 0 \\ 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{pmatrix} \right).$$

Then P is the solution to I that is depicted in Figure 1(b).

2.2 Fewest Clues Problem

We here define the FCP of Picross 3D and positive 1-in-3 SAT. A *positive CNF* is a CNF where every literal occurring in it is positive.

FCP Picross 3D Given an instance I of Picross 3D and an integer ℓ, does there exists a partial assignment of at most ℓ variables such that there exists a unique solution to I extending the partial assignment?

FCP positive 1-in-3 SAT Given a positive 3-CNF φ and an integer ℓ, does there exists a partial assignment of at most ℓ variables such that there exists a unique solution to φ, where every clause has exactly one true literal, extending the partial assignment?

3 Parsimonious Reduction from positive 1-in-3 SAT to Picross 3D

In this section, we provide a parsimonious reduction from positive 1-in-3 SAT to Picross 3D. Recall that a reduction is parsimonious if, for each instance, there exists a one-to-one correspondence between the solution sets of the original instance and the reduced one. The reduction will be used to show the Σ_2^P-completeness of FCP Picross 3D in the next section. We note that from the reduction it follows that the counting version and the another solution problem (ASP) of Picross 3D are respectively #P-complete and NP-complete, since so are these variants of positive 1-in-3 SAT [2, 3, 12, 13]. We also note that our reduction is similar to the reduction from 3-SAT to Picross 3D in [10]; indeed, the variables of a given 3-CNF are represented in the same way. However, the reduction in [10] is not parsimonious and seems hard to be used for showing the Σ_2^P-completeness of FCP Picross 3D.

▶ **Proposition 2.** *There exists a parsimonious reduction from positive 1-in-3 SAT to Picross 3D.*

Proof. Let φ be an instance of positive 1-in-3 SAT, where $\varphi = \bigwedge_{j=1}^{m} C_j$ is a positive 3-CNF with n variables and m clauses, and $C_j = (x_{j_1} \vee x_{j_2} \vee x_{j_3})$ for $j = 1, \ldots, m$. Here, $1 \leq j_1, j_2, j_3 \leq n$ and j_ℓ's are distinct for $j = 1, \ldots, m$. We construct an instance $I_\varphi = (h, w, d, F, S, T)$ of Picross 3D as follows.

We set $h = 4$, $w = 2(m + n - 1) + 1$, and $d = 3n$. Let a function div be defined as

$$\mathrm{div}(i) = \begin{cases} i & \text{if } i = 0, 1 \\ \textcircled{i} & \text{if } i = 2 \\ \boxed{i} & \text{if } i \geq 3. \end{cases}$$

The front constraint matrix F, which is an $h \times w (= 4 \times (2m + 2n - 1))$ matrix, is defined as

$$\text{(i)} \begin{cases} f_{1,j} = f_{4,j} = f_{1,m+n+j} = f_{4,m+n+j} = \mathrm{div}(n), \\ f_{2,j} = 1, \\ f_{3,j} = \mathrm{div}(2), \end{cases}$$

for $1 \leq j \leq m$, (ii) $f_{1,j} = f_{4,j} = f_{1,m+n+j} = f_{4,m+n+j} = \mathrm{div}(m + n - j)$ for $m + 1 \leq j \leq m + n - 1$, (iii) $f_{2,j} = f_{3,j} = 0$ for $m + 1 \leq j \leq w$, and (iv) $f_{1,n+m} = f_{4,n+m} = 0$. Namely,

$$F = \left(\begin{array}{ccccccc} \multicolumn{3}{c}{\overbrace{\hphantom{\mathrm{div}(n) \quad \ldots \quad \mathrm{div}(n)}}^{m}} & \multicolumn{3}{c}{\overbrace{\hphantom{\mathrm{div}(n-1) \quad \ldots \quad \mathrm{div}(1)}}^{n-1}} & \\ \mathrm{div}(n) & \ldots & \mathrm{div}(n) & \mathrm{div}(n-1) & \ldots & \mathrm{div}(1) & 0 \\ 1 & \ldots & 1 & 0 & \ldots & 0 & 0 \\ \mathrm{div}(2) & \ldots & \mathrm{div}(2) & 0 & \ldots & 0 & 0 \\ \mathrm{div}(n) & \ldots & \mathrm{div}(n) & \mathrm{div}(n-1) & \ldots & \mathrm{div}(1) & 0 \end{array} \right.$$

$$\left. \begin{array}{cccccc} \multicolumn{3}{c}{\overbrace{\hphantom{\mathrm{div}(n) \quad \ldots \quad \mathrm{div}(n)}}^{m}} & \multicolumn{3}{c}{\overbrace{\hphantom{\mathrm{div}(n-1) \quad \ldots \quad \mathrm{div}(1)}}^{n-1}} \\ \mathrm{div}(n) & \ldots & \mathrm{div}(n) & \mathrm{div}(n-1) & \ldots & \mathrm{div}(1) \\ 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & \ldots & 0 & 0 & \ldots & 0 \\ \mathrm{div}(n) & \ldots & \mathrm{div}(n) & \mathrm{div}(n-1) & \ldots & \mathrm{div}(1) \end{array} \right).$$

The side constraint matrix S, which is an $h \times d (= 4 \times 3n)$ matrix, is defined as

$$\begin{cases} s_{1,3k-2} = s_{4,3k-1} = m + n - k, \\ s_{2,3k-2} = s_{3,3k-1} = \varepsilon, \\ s_{1,3k} = s_{2,3k} = s_{3,3k} = s_{4,3k} = 0, \end{cases}$$

for $1 \leq k \leq n$. Namely,

$$S = \begin{array}{c} \overbrace{\hphantom{m+n-1 \quad m+n-1 \quad 0 \quad m+n-2 \quad m+n-2 \quad 0 \quad \dots \quad m \quad m \quad 0}}^{3n} \\ \begin{pmatrix} m+n-1 & m+n-1 & 0 & m+n-2 & m+n-2 & 0 & \dots & m & m & 0 \\ \varepsilon & \varepsilon & 0 & \varepsilon & \varepsilon & 0 & \dots & \varepsilon & \varepsilon & 0 \\ \varepsilon & \varepsilon & 0 & \varepsilon & \varepsilon & 0 & \dots & \varepsilon & \varepsilon & 0 \\ m+n-1 & m+n-1 & 0 & m+n-2 & m+n-2 & 0 & \dots & m & m & 0 \end{pmatrix} \end{array}.$$

The top constraint matrix T, which is $d \times w(= 3n \times (2m+2n-1))$ matrix, is defined as follows.

$$\text{(i) } t_{j,3k-2} = \begin{cases} 2 & \text{if } x_k \in \{x_{j_1}, x_{j_2}, x_{j_3}\}, \\ 1 & \text{otherwise,} \end{cases}$$

for $1 \leq j \leq m$ and $1 \leq k \leq n$, (ii) $t_{j,3k-2} = t_{j,3k-1} = 1$ for $m+1 \leq j \leq m+n-k$ and $1 \leq k \leq n-1$, (iii) $t_{j,3k-2} = t_{j,3k-1} = 1$ for $m+n+1 \leq j \leq 2m+2n-k$ and $1 \leq k \leq n-1$, and (iv) $t_{jk} = 0$ for the remaining entries. Hence,

$$T = \left.\left(\begin{array}{ccccccccccccccc} 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 1 & \dots & 1 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 1 & 0 & 0 & \dots & 0 \\ 1 \text{ or } 2 & \dots & 1 \text{ or } 2 & 0 & 0 & \dots & 0 & 0 & 1 & \dots & 1 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 & 0 & 0 & \dots & 0 \\ 1 & \dots & 1 & 1 & 0 & \dots & 0 & 0 & 1 & \dots & 1 & 1 & 0 & \dots & 0 \\ 1 \text{ or } 2 & \dots & 1 \text{ or } 2 & 1 & 0 & \dots & 0 & 0 & 1 & \dots & 1 & 1 & 0 & \dots & 0 \\ & \vdots & & & & \vdots & & 0 & & \vdots & & & & \vdots & \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 1 & \dots & 1 & 1 & \dots & 1 & 0 & 0 & 1 & \dots & 1 & 1 & \dots & 1 & 0 \\ 1 \text{ or } 2 & \dots & 1 \text{ or } 2 & 1 & \dots & 1 & 0 & 0 & 1 & \dots & 1 & 1 & \dots & 1 & 0 \\ 0 & \dots & 0 & 0 & \dots & 0 & 0 & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 0 \\ 1 & \dots & 1 & 1 & \dots & 1 & 1 & 0 & 1 & \dots & 1 & 1 & \dots & 1 & 1 \\ 1 \text{ or } 2 & \dots & 1 \text{ or } 2 & 1 & \dots & 1 & 1 & 0 & 1 & \dots & 1 & 1 & \dots & 1 & 1 \\ \multicolumn{3}{c}{\underbrace{\hphantom{1 \text{ or } 2 \quad \dots \quad 1 \text{ or } 2}}_{m}} & \multicolumn{4}{c}{\underbrace{\hphantom{0 \quad 0 \quad \dots \quad 0}}_{n-1}} & & \multicolumn{3}{c}{\underbrace{\hphantom{1 \quad \dots \quad 1}}_{m}} & \multicolumn{4}{c}{\underbrace{\hphantom{0 \quad 0 \quad \dots \quad 0}}_{n-1}} \end{array}\right)\right\} 3n.$$

▶ **Example 3.** For $\varphi = (x_1 \vee x_3 \vee x_4)(x_2 \vee x_3 \vee x_4)$, I_φ is depicted as Figure 2(a), and its solution is given in 2(b).

We now show that the above reduction is parsimonious. We first show several auxiliary claims.

▶ **Claim 3.1.** *Let $P = (p_{i,j,k})$ be a solution to I_φ. Then, for $1 \leq k \leq n$, we have either*

$$\text{(i)} \begin{cases} p_{1,1,3k-2} = \cdots = p_{1,m+n-k,3k-2} = 1 \\ p_{1,1,3k-1} = \cdots = p_{1,m+n-k,3k-1} = 0 \\ p_{4,1,3k-2} = \cdots = p_{4,m+n-k,3k-2} = 0 \\ p_{4,1,3k-1} = \cdots = p_{4,m+n-k,3k-1} = 1 \\ p_{1,m+n+1,3k-2} = \cdots = p_{1,2m+2n-k,3k-2} = 0 \\ p_{1,m+n+1,3k-1} = \cdots = p_{1,2m+2n-k,3k-1} = 1 \\ p_{4,m+n+1,3k-2} = \cdots = p_{4,2m+2n-k,3k-2} = 1 \\ p_{4,m+n+1,3k-1} = \cdots = p_{4,2m+2n-k,3k-1} = 0, \end{cases}$$

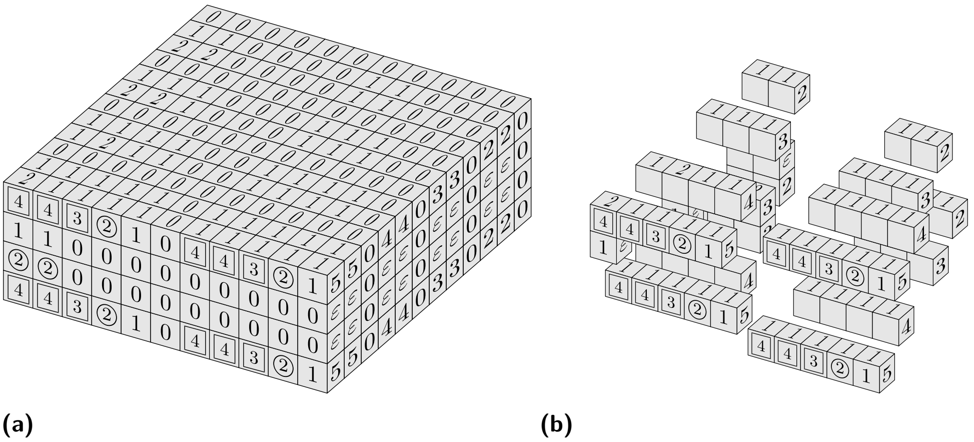


Figure 2 The instance I_φ of Picross 3D and its solution for $\varphi = (x_1 \vee x_3 \vee x_4)(x_2 \vee x_3 \vee x_4)$

or

$$
\text{(ii)} \begin{cases}
p_{1,1,3k-2} = \cdots = p_{1,m+n-k,3k-2} = 0 \\
p_{1,1,3k-1} = \cdots = p_{1,m+n-k,3k-1} = 1 \\
p_{4,1,3k-2} = \cdots = p_{4,m+n-k,3k-2} = 1 \\
p_{4,1,3k-1} = \cdots = p_{4,m+n-k,3k-1} = 0 \\
p_{1,m+n+1,3k-2} = \cdots = p_{1,2m+2n-k,3k-2} = 1 \\
p_{1,m+n+1,3k-1} = \cdots = p_{1,2m+2n-k,3k-1} = 0 \\
p_{4,m+n+1,3k-2} = \cdots = p_{4,2m+2n-k,3k-2} = 0 \\
p_{4,m+n+1,3k-1} = \cdots = p_{4,2m+2n-k,3k-1} = 1.
\end{cases}
$$

Proof. We show the claim by induction on k. Assume that $k = 1$. We first show that either $(p_{i,1,\ell} = \cdots = p_{i,m+n-1,\ell} = 1$ and $p_{i,m+n+1,\ell} = \cdots = p_{i,2m+2n-1,\ell} = 0)$ or $(p_{i,1,\ell} = \cdots = p_{i,m+n-1,\ell} = 0$ and $p_{i,m+n+1,\ell} = \cdots = p_{i,2m+2n-1,\ell} = 1)$ hold for $(i, \ell) \in \{1, 4\} \times \{1, 2\}$. For $(i, \ell) = (1, 1)$, since $s_{1,1} = m+n-1$, we have to consecutively set $p_{1,a,1} = \cdots = p_{1,a+m+n-2,1} = 1$ for some $a \geq 1$. On the other hand, since $f_{1,m+n} = 0$, we have $p_{1,m+n,1} = 0$. Therefore, we must have either $(p_{1,1,1} = \cdots = p_{1,m+n-1,1} = 1$ and $p_{1,m+n+1,1} = \cdots = p_{1,2m+2n-1,1} = 0)$ or $(p_{1,1,1} = \cdots = p_{1,m+n-1,1} = 0$ and $p_{1,m+n+1,1} = \cdots = p_{1,2m+2n-1,1} = 1)$. Similarly, for other (i, ℓ), from $s_{i,\ell} = m+n-1$ and $f_{1,m+n} = 0$, we have either $(p_{i,1,\ell} = \cdots = p_{i,m+n-1,\ell} = 1$ and $p_{i,m+n+1,\ell} = \cdots = p_{i,2m+2n-1,\ell} = 0)$ or $(p_{i,1,\ell} = \cdots = p_{i,m+n-1,\ell} = 0$ and $p_{i,m+n+1,\ell} = \cdots = p_{i,2m+2n-1,\ell} = 1)$. We next show that $p_{1,m+n-1,1} + p_{1,m+n-1,2} = p_{4,m+n-1,1} + p_{4,m+n-1,2} = p_{1,m+n-1,1} + p_{4,m+n-1,1} = 1$ holds. From $f_{1,m+n-1} = \mathrm{div}(1) = 1$, we have $\sum_{\ell=1}^{3n} p_{1,m+n-1,\ell} = 1$. On the other hand, $t_{m+n-1,\ell} = 0$ implies that $p_{1,m+n-1,\ell} = 0$ for $\ell = 3, 4, \ldots, 3n$. Therefore, we have $p_{1,m+n-1,1} + p_{1,m+n-1,2} = 1$. Similarly, from $f_{4,m+n-1} = \mathrm{div}(1) = 1$ and $t_{m+n-1,\ell} = 0$ for $\ell = 3, 4, \ldots, 3n$, we obtain $p_{4,m+n-1,1} + p_{4,m+n-1,2} = 1$. Moreover, from $t_{m+n-1,1} = 1$, we have $\sum_{i=1}^{4} p_{i,m+n-1,1} = 1$. On the other hand, $f_{i,m+n-1} = 0$ implies that $p_{i,m+n-1,1} = 0$ for $i = 1, 2$. Therefore, we have $p_{1,m+n-1,1} + p_{4,m+n-1,1} = 1$. Combining the above equations, we obtain the claim for $k = 1$.

For $k \geq 2$, assume that the claim holds for $1, \ldots, k-1$. The proof is similar to the one for $k = 1$. We first show that either $(p_{i,1,3k-3+\ell} = \cdots = p_{i,m+n-k,3k-3+\ell} = 1$ and $p_{i,m+n+1,3k-3+\ell} = \cdots = p_{i,2m+2n-k,3k-3+\ell} = 0)$ or $(p_{i,1,3k-3+\ell} = \cdots = p_{i,m+n-k,3k-3+\ell} = 0$ and $p_{i,m+n+1,3k-3+\ell} = \cdots = p_{i,2m+2n-k,3k-3+\ell} = 1)$ hold for $(i, \ell) \in \{1, 4\} \times \{1, 2\}$. For $(i, \ell) = (1, 1)$, since $s_{1,3k-2} = m + n - k$, we have to consecutively set $p_{1,a,1} = \cdots = p_{1,a+m+n-k-1,1} = 1$ for some $a \geq 1$. On the other hand, since $f_{1,m+n} = 0$, we have $p_{1,m+n,1} =$

0. Moreover, $f_{1,m+n-a} = \mathrm{div}(a)$ implies that $\sum_{b=1}^{3n} p_{1,m+n-a,b} = a$ for $a = 1, \ldots, k-1$. On the other hand, by the inductive hypothesis, we have $p_{1,m+n-a,3b-2} + p_{1,m+n-a,3b-1} = 1$ for $a = 1, \ldots, k-1$ and $b = 1, \ldots, a$. Therefore, we obtain that $p_{1,m+n-a,3k-2} = 0$ for $a = 1, \ldots, k-1$. These imply that either $(p_{1,1,3k-2} = \cdots = p_{1,m+n-k,3k-2} = 1$ and $p_{1,m+n+1,3k-2} = \cdots = p_{1,2m+2n-k,3k-2} = 0)$ or $(p_{1,1,3k-2} = \cdots = p_{1,m+n-k,3k-2} = 0$ and $p_{1,m+n+1,3k-2} = \cdots = p_{1,2m+2n-k,3k-2} = 1)$ hold. Similarly, for other (i, ℓ), from $s_{i,\ell} = m+n-k$ and $f_{1,m+n-a} = \mathrm{div}(a)$ for $a = 0, \ldots, k-1$, we have either $(p_{i,1,3k-3+\ell} = \cdots = p_{i,m+n-k,3k-3+\ell} = 1$ and $p_{i,m+n+1,3k-3+\ell} = \cdots = p_{i,2m+2n-k,3k-3+\ell} = 0)$ or $(p_{i,1,3k-3+\ell} = \cdots = p_{i,m+n-k,\ell} = 0$ and $p_{i,m+n+1,3k-3+\ell} = \cdots = p_{i,2m+2n-k,3k-3+\ell} = 1)$. We next show that $p_{1,m+n-k,3k-2} + p_{1,m+n-k,3k-1} = p_{4,m+n-k,3k-2} + p_{4,m+n-k,3k-1} = p_{1,m+n-k,3k-2} + p_{4,m+n-k,3k-2} = 1$ holds. From $f_{1,m+n-k} = \mathrm{div}(k)$, we have $\sum_{\ell=1}^{3n} p_{1,m+n-k,\ell} = k$. By the inductive hypothesis, we have $p_{1,m+n-k,3\ell-2} + p_{1,m+n-k,3\ell-1} = 1$ for $\ell = 1, \ldots, k-1$. We also have $p_{1,m+n-k,3\ell} = 0$ for $\ell = 1, \ldots, k-1$, since $t_{m+n-k,3\ell} = 0$ for $\ell = 1, \ldots, k-1$. Hence, we have $\sum_{\ell=3k-2}^{3n} p_{1,m+n-k,\ell} = 1$. On the other hand, $t_{m+n-k,\ell} = 0$ implies that $p_{1,m+n-1,\ell} = 0$ for $\ell = 3k, 3k+1, \ldots, 3n$. Therefore, we have $p_{1,m+n-k,3k-2} + p_{1,m+n-k,3k-1} = 1$. Similarly, from $f_{4,m+n-k} = \mathrm{div}(k)$ and $t_{m+n-k,\ell} = 0$ for $\ell = 3, 6, \ldots, 3k-3$ and $\ell = 3k, 3k+1, \ldots, 3n$, we obtain $p_{4,m+n-k,3k-2} + p_{4,m+n-k,3k-1} = 1$. Moreover, from $t_{m+n-1,k} = 1$, we have $\sum_{i=1}^{4} p_{i,m+n-k,1} = 1$. On the other hand, $f_{i,m+n-k} = 0$ implies that $p_{i,m+n-k,3k-2} = 0$ for $i = 1, 2$. Therefore, we have $p_{1,m+n-k,3k-2} + p_{4,m+n-k,3k-2} = 1$. Combining the above equations, we obtain the claim for k. This completes the proof. ◀

Intuitively, for $1 \leq k \leq n$, $x_k = 1$ if and only if (i) in Claim 3.1 holds. We also need the following claim.

▶ **Claim 3.2.** *Let $P = (p_{i,j,k})$ be a solution to I_φ. Then we have $p_{i,j,3k} = 0$ for $i = 1, 2, 3, 4$, $j = 1, \ldots, 2m+2n-1$, and $k = 1, \ldots, n$. Moreover, we have $p_{i,j,k} = 0$ for $i = 2, 3$, $j = m+1, \ldots, 2m+2n-1$, and $k = 1, 2, \ldots, 3n$. Furthermore, we have $p_{i,j,3k-2} = p_{i,j,3k-1} = 0$ for $i = 2, 3$, $k = 1, \ldots, n$, and $j = m+n-k+1, \ldots, m+n-1, 2m+2n-k+1, \ldots, 2m+2n-1$.*

Proof. For $j = 1, \ldots, 2m+2n+1$ and $k = 1, \ldots, n$, we have $t_{j,3k} = 0$, implying that $p_{i,j,3k} = 0$ holds for $i = 1, 2, 3, 4$.

For $i = 2, 3$ and $j = m+1, \ldots, 2m+2n+1$, we have $f_{i,j} = 0$, implying that $p_{i,j,k} = 0$ holds for $k = 1, 2 \ldots, 3n$.

For $k = 1, \ldots, n$ and $j = m+n-k+1, \ldots, m+n-1, 2m+2n-k+1, \ldots, 2m+2n-1$, we have $t_{j,3k-2} = t_{j,3k-1} = 1$ and $p_{1,j,3k-2} + p_{4,j,3k-2} = p_{1,j,3k-1} + p_{4,j,3k-1} = 1$ by Claim 3.1. Therefore, we have $p_{i,j,3k-2} = p_{i,j,3k-1} = 0$ for $i = 2, 3$. ◀

The following claim indicates which variable is true in each clause.

▶ **Claim 3.3.** *Let $P = (p_{i,j,k})$ be a solution to I_φ. Then, for $j = 1, \ldots, m$ and $k = 1, \ldots, n$, we have (i) in Claim 3.1 and $t_{j,3k-2} = 2$ if and only if $p_{2,j,3k-2} = 1$ holds.*

Proof. Fix $j \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, n\}$. Assume that (i) in Claim 3.1 and $t_{j,3k-2} = 2$ hold. From Claim 3.1, we have $p_{1,j,3k-2} = 1$. Moreover, since $t_{j,3k-2} = 2$ implies that we have to consecutively set $p_{i,j,3k-2} = p_{i+1,j,3k-2}$ for some $i \geq 1$, we have $p_{2,j,3k-2} = 1$.

Conversely, assume that $p_{2,j,3k-2} = 1$ holds. By Claim 3.1, we have either $p_{1,j,3k-2} = 1$ or $p_{4,j,3k-2} = 1$. Hence, we have $\sum_{i=1}^{4} p_{i,j,3k-2} \geq 2$. Since $t_{j,3k-2} = 1$ or 2 by definition, it follows that $t_{j,3k-2} = 2$. This implies that we have to consecutively set $p_{i,j,3k-2} = p_{i+1,j,3k-2}$ for some $i \geq 1$. Together with $p_{2,j,3k-2} = 1$ and either $p_{1,j,3k-2} = 1$ or $p_{4,j,3k-2} = 1$, we have $p_{1,j,3k-2} = 1$. Hence, from Claim 3.1, we have (i) in Claim 3.1. This completes the proof. ◀

Intuitively, for $j = 1, \dots, m$ and $k = 1, \dots, n$, clause C_j contains x_k and x_k is the unique variable that is true in C_j if and only if $p_{2,j,3k-2} = 1$ holds.

We now construct a bijection between the solution sets of φ and I_φ. We first construct a mapping from the solution set of φ to that of I_φ. Let x be a solution to φ. Then define an assignment P to I_φ as follows. For $1 \le k \le n$, if $x_k = 1$ then set

$$\begin{cases} p_{1,1,3k-2} = \cdots = p_{1,m+n-k,3k-2} = 1 \\ p_{1,1,3k-1} = \cdots = p_{1,m+n-k,3k-1} = 0 \\ p_{4,1,3k-2} = \cdots = p_{4,m+n-k,3k-2} = 0 \\ p_{4,1,3k-1} = \cdots = p_{4,m+n-k,3k-1} = 1 \\ p_{1,m+n+1,3k-2} = \cdots = p_{1,2m+2n-k,3k-2} = 0 \\ p_{1,m+n+1,3k-1} = \cdots = p_{1,2m+2n-k,3k-1} = 1 \\ p_{4,m+n+1,3k-2} = \cdots = p_{4,2m+2n-k,3k-2} = 1 \\ p_{4,m+n+1,3k-1} = \cdots = p_{4,2m+2n-k,3k-1} = 0, \end{cases} \tag{1}$$

and if $x_k = 0$ then set

$$\begin{cases} p_{1,1,3k-2} = \cdots = p_{1,m+n-k,3k-2} = 0 \\ p_{1,1,3k-1} = \cdots = p_{1,m+n-k,3k-1} = 1 \\ p_{4,1,3k-2} = \cdots = p_{4,m+n-k,3k-2} = 1 \\ p_{4,1,3k-1} = \cdots = p_{4,m+n-k,3k-1} = 0 \\ p_{1,m+n+1,3k-2} = \cdots = p_{1,2m+2n-k,3k-2} = 1 \\ p_{1,m+n+1,3k-1} = \cdots = p_{1,2m+2n-k,3k-1} = 0 \\ p_{4,m+n+1,3k-2} = \cdots = p_{4,2m+2n-k,3k-2} = 0 \\ p_{4,m+n+1,3k-1} = \cdots = p_{4,2m+2n-k,3k-1} = 1. \end{cases} \tag{2}$$

For $j = 1, \dots, m$, if clause C_j contains x_k and $x_k = 1$, then set $p_{2,j,3k-2} = 1$ and $p_{3,j,3k-2} = 0$. For $j = 1, \dots, m$, if clause C_j contains x_k and $x_k = 0$, then set $p_{2,j,3k-2} = 0$ and $p_{3,j,3k-2} = 1$. For all the remaining $p_{i,j,k}$, set $p_{i,j,k} = 0$. We show that the assignment P constructed from x as above is a solution to I_φ. We show this by showing that each constraint is satisfied by P.

We first examine the constraints for matrix F. For $j = 1, \dots, m$, we have $\sum_{k=1}^{3n} p_{1,j,k} = n$ from (1) and (2). Therefore, the constraint $f_{1,j} = \mathrm{div}(n)$ is satisfied for $j = 1, \dots, m$. Similarly, the constraint $f_{4,j} = \mathrm{div}(n)$ is satisfied for $j = 1, \dots, m$, since $\sum_{k=1}^{3n} p_{4,j,k} = n$ holds. For $j = 1, \dots, m$, we have $\sum_{k=1}^{3n} p_{2,j,k} = 1$ if and only if there exists exactly one x_k in C_j such that $x_k = 1$, since $x_k = 1$ implies that $p_{2,j,3k-2} = 1$. Because x is a solution to φ, we have $\sum_{k=1}^{3n} p_{2,j,k} = 1$ and the constraint $f_{2,j} = 1$ is satisfied for $j = 1, \dots, m$. Similarly, the constraint $f_{3,j} = 2$ is satisfied for $j = 1, \dots, m$, since $\sum_{k=1}^{3n} p_{3,j,k} = 2$ holds if and only if there exist exactly two $x_k{}'s$ in C_j such that $x_k = 0$.

We then focus on the constraints for matrix S. For $i = 1, 4$ and $k = 1, \dots, n$, we have $\sum_{j=1}^{2m+2n-1} p_{i,j,3k-2} = m+n-k$ from (1) and (2). Therefore the constraint $s_{i,3k-2} = m+n-k$ is satisfied for $i = 1, 4$ and $k = 1, \dots, n$. Similarly, the constraint $s_{i,3k-1} = m+n-k$ is satisfied for $i = 1, 4$ and $k = 1, \dots, n$. Furthermore, since $p_{i,j,3k} = 0$ for $i = 1, 2, 3, 4$, $j = 1, \dots, 2m+2n-1$, and $k = 1, \dots, n$, the constraint $s_{i,3k} = 0$ is satisfied for $i = 1, 2, 3, 4$ and $k = 1, \dots, n$.

Finally, we examine the constraints for matrix T. Firstly, for $j = 1, \dots, m$ and $k = 1, \dots, n$, we have $t_{j,3k-2} = 1$ or 2 by definition. Assume first that $t_{j,3k-2} = 1$ holds. Then, by definition, clause C_j does not contain variable x_k. Hence, we have $p_{2,j,3k-2} = p_{3,j,3k-2} = 0$. Moreover, from assignment (1) and (2), we have $p_{1,j,3k-2} + p_{4,j,3k-2} = 1$. Therefore, the constraint $t_{j,3k-2} = 1$ is satisfied. Assume next that $t_{j,3k-2} = 2$ holds. Then, by definition, clause

C_j contains variable x_k. Hence, we have $p_{2,j,3k-2} + p_{3,j,3k-2} = 1$. Furthermore, we have $p_{1,j,3k-2} + p_{4,j,3k-2} = 1$ from (1) and (2). Therefore, $\sum_{i=1}^{4} p_{i,j,3k-2} = 2$ holds. Moreover, from Claim 3.3, we have $p_{1,j,3k-2} = 1$ if and only if $p_{2,j,3k-2} = 1$ holds. Hence, we must have either ($p_{1,j,3k-2} = 1$ and $p_{2,j,3k-2} = 1$) or ($p_{3,j,3k-2} = 1$ and $p_{4,j,3k-2} = 1$). In either case, we have two consecutive 1's. Therefore, the constraint $t_{j,3k-2} = 2$ is satisfied. Secondly, for $j = 1, \dots, m$ and $k = 1, \dots, n$, we have $t_{j,3k-1} = 1$ by definition. Since $p_{1,j,3k-1} + p_{4,j,3k-1} = 1$ by (1) and (2), and $p_{1,j,3k-1} = p_{4,j,3k-1} = 0$, the constraint $t_{j,3k-1} = 1$ is satisfied. Thirdly, for $j = m+1, \dots, m+n-1$ and $k = 1, \dots, n$, we have $t_{j,3k-2} = 1$ if $j \leq m+n-k$ and $t_{j,3k-2} = 0$ if $j \geq m+n-k+1$ by definition. If $j \leq m+n-k$, then we have $p_{1,j,3k-2} + p_{1,j,3k-2} = 1$ from (1) and (2), and $p_{2,j,3k-1} = p_{3,j,3k-1} = 0$. Thus, we have $\sum_{i=1}^{4} p_{i,j,3k-2} = 1$ and the constraint $t_{j,3k-2} = 1$ is satisfied. If $j \geq m+n-k+1$, then we have $p_{i,j,3k-2} = 0$ for $i = 1,2,3,4$. Thus, we have $\sum_{i=1}^{4} p_{i,j,3k-2} = 0$ and the constraint $t_{j,3k-2} = 0$ is satisfied. Similarly, for $j = m+1, \dots, m+n-1$ and $k = 1, \dots, n$, the constraint $t_{j,3k-1} = 0$ is satisfied. Fourthly, for $j = 1, \dots, 2m+2n-1$ and $k = 1, \dots, n$, we have $t_{j,3k} = 0$ by definition. Since $p_{i,j,3k} = 0$ holds for $i = 1,2,3,4$, the constraint $t_{j,3k} = 0$ is satisfied. Finally, for $j = m+n$ and $k = 1, 2 \dots, 3n$, we have $t_{j,k} = 0$ by definition. Since $p_{i,j,k} = 0$ holds for $i = 1,2,3,4$, the constraint $t_{j,k} = 0$ is satisfied.

We next show that if I_φ has a solution, then φ has a solution. To show this, we construct a solution x to φ from a solution P to I_φ as follows. Note that, for each $k = 1, \dots, n$, we have either (i) or (ii) in Claim 3.1, since P is a solution to I_φ. For each k, set $x_k = 1$ if (i) holds and $x_k = 0$ if (ii) holds. We show that x defined as above is a solution to φ. It suffices to show that for each $j = 1, \dots, m$, clause C_j contains exactly one x_k that is set to 1. Fix $j \in \{1, \dots, m\}$. From $f_{2,j} = 1$, we have $\sum_{k=1}^{3n} p_{2,j,k} = 1$. Moreover, from $t_{j,3k} = 0$, we have $p_{3,j,3k-3} = 0$ for $k = 1, \dots, n$. Furthermore, from $t_{j,3k-1} = 1$ and $p_{1,j,3k-1} + p_{4,j,3k-1} = 1$ by Claim 3.1, we have $p_{2,j,3k-1} = 0$ for $k = 1, \dots, n$. Therefore, we have $\sum_{k=1}^{n} p_{2,j,3k-2} = 1$. From Claim 3.3, $p_{2,j,3k-2} = 1$ holds if and only if $p_{1,1,3k-2} = 1$ and $t_{j,3k-2} = 2$ holds. Therefore, together with $\sum_{k=1}^{n} p_{2,j,3k-2} = 1$, there exists exactly one k such that $p_{1,1,3k-2} = 1$ and $t_{j,3k-2} = 2$ holds. By definition, we have $t_{j,3k-2} = 2$ if and only if C_j contains x_k, and $p_{1,1,3k-2} = 1$ if and only if $x_k = 1$. Therefore, C_j contains exactly one x_k that is set to 1. Hence, x is a solution to φ.

We finally show that the above reduction is parsimonious. To show this, we show that the mappings between the solution sets of φ and I_φ defined above are inverse to each other. Let x be a solution to φ and let P be the solution of I_φ corresponding to x. Moreover, let x' be the solution constructed from P. We show that $x = x'$ holds. Observe first that $x_k = 1$ if and only if $p_{1,1,3k-2} = 1$ from (1) and (2) for $k = 1, \dots, n$. Furthermore, $p_{1,1,3k-2} = 1$ if and only if $x'_k = 1$ from Claim 3.1 for $k = 1, \dots, n$. Hence, $x_k = x'_k$ for $k = 1, \dots, n$ and thus $x = x'$.

Conversely, let P be a solution to I_φ and let x be the solution to φ constructed from P. Moreover, let P' be the solution constructed from x. We show that $P = P'$ holds. Firstly, for $k = 1, \dots, n$, P satisfies (i) in Claim 3.1 if and only if $x_k = 1$ holds. Furthermore, $x_k = 1$ holds if and only if P' satisfies (i) in Claim 3.1 for $k = 1, \dots, n$. Hence, P and P' coincide in the indices appearing in Claim 3.1 for $k = 1, \dots, n$. Secondly, from $t_{j,3k-1} = 1$ and $p_{1,k,3k-1} + p_{4,k,3k-1} = p'_{1,k,3k-1} + p'_{4,k,3k-1} = 1$ for $j = 1, \dots, m$ and $k = 1, \dots, n$, we have $p_{2,k,3k-1} = p_{3,k,3k-1} = p'_{2,k,3k-1} = p'_{3,k,3k-1} = 0$ for $j = 1, \dots, m$ and $k = 1, \dots, n$. Thirdly, from the proof of Claim 3.1, we have $p_{i,j,3k-2} = p_{i,j,3k-1} = p'_{i,j,3k-2} = p'_{i,j,3k-1} = 0$ for $i = 1, 4$, $k = 2, \dots, n$, and $j = m+n-k+1, \dots, 2m+2n+1$. Fourthly, from Claim 3.2 we have $p_{i,j,3k} = p'_{i,j,3k} = 0$ for $i = 1,2,3,4$, $j = 1, \dots, 2m+2n+1$, and $k = 1, \dots, n$, and $p_{i,j,k} = p'_{i,j,k} = 0$ for $i = 2, 3$, $j = m+1, \dots, 2m+2n+1$, and $k = 1, 2, \dots, 3n$. Finally, from Claim 3.3, we have $p_{1,1,3k-2} = 1$ and $t_{j,3k-2} = 2$ if and only if $p_{2,j,3k-2} = 1$ holds for $j = 1, \ldots, m$ and $k = 1, \ldots, n$. Since $p_{1,1,3k-2} = p'_{1,1,3k-2}$ holds from the above argument, we have $p_{2,j,3k-2} = p'_{2,j,3k-2}$ for $j = 1, \ldots, m$ and $k = 1, \ldots, n$. Hence, $P = P'$ holds. This completes the proof. ◀

▶ **Corollary 4.** *The counting version of Picross 3D is #P-complete and ASP Picross 3D is NP-complete.*

Proof. The former follows from the #P-completeness of the counting version of positive 1-in-3 SAT [2, 3] and Proposition 2. The latter follows from the NP-completeness of ASP positive 1-in-3 SAT [12, 13] and Proposition 2. ◀

4 Σ_2^P-completeness of FCP Picross 3D

In this section, we show the following theorem using the reduction in the previous section.

▶ **Theorem 5.** *FCP Picross 3D is Σ_2^P-complete.*

Proof. Since Picross 3D is in NP, FCP Picross 3D is in Σ_2^P [4]. We hence show that FCP Picross 3D is Σ_2^P-hard in the following.

Let (φ, ℓ) be an instance of FCP positive 1-in-3 SAT. We show that (φ, ℓ) is a yes instance if and only if (I_φ, ℓ) is a yes instance, where I_φ is defined in the proof of Proposition 2.

We first show that if (φ, ℓ) is a yes instance, then (I_φ, ℓ) is a yes instance. For simplicity, we identify a partial assignment with a set of single-variable assignments corresponding to it in the following. Let $\{x_k = \varepsilon_k \mid k \in K\}$ be a clue that makes φ uniquely solvable, where $K \subseteq \{1, \ldots, n\}$, $|K| \leq \ell$, and ε_k is either 0 or 1 for $k \in K$. We claim that $\{p_{1,1,3k-2} = \varepsilon_k \mid k \in K\}$ is a clue that makes I_φ uniquely solvable. In fact, since $\{x_k = \varepsilon_k \mid k \in K\}$ can be extended to a solution of φ, $\{p_{1,1,3k-2} = \varepsilon_k \mid k \in K\}$ can also be extended to a solution to I_φ. Moreover, if there exist two solutions extending $\{p_{1,1,3k-2} = \varepsilon_k \mid k \in K\}$ in I_φ, then there must be two solutions to φ corresponding to these solutions since the reduction is parsimonious. These two solutions to φ coincide in the indices in K from the argument in the proof of Proposition 2. This contradicts that $\{x_k = \varepsilon_k \mid k \in K\}$ is a clue that makes φ uniquely solvable. Therefore, $\{p_{1,1,3k-2} = \varepsilon_k \mid k \in K\}$ is a clue that makes I_φ uniquely solvable. Since $|\{p_{1,1,3k-2} = \varepsilon_k \mid k \in K\}| \leq \ell$, we have that (I_φ, ℓ) is a yes instance.

We next show that if (I_φ, ℓ) is a yes instance, then (φ, ℓ) is a yes instance. Let $c_{\text{pic}} = \{p_{i_v,j_v,k_v} = \varepsilon_v \mid v \in V\}$ be a clue that makes I_φ uniquely solvable, where $|V| \leq \ell$, $(i_v, j_v, k_v) \in \{1, \ldots, h\} \times \{1, \ldots, w\} \times \{1, \ldots, d\}$ for $v \in V$, and ε_v is either 0 or 1 for $v \in V$. We construct a clue c_{sat} of φ as follows. Set $c_{\text{sat}} = \emptyset$. For $k = 1, \ldots, n$, add $x_k = 1$ to c_{sat} if c_{pic} contains at least one of the following assignments:

$$\begin{aligned}
&p_{1,1,3k-2} = 1, \ldots, p_{1,m+n-k,3k-2} = 1\\
&p_{1,1,3k-1} = 0, \ldots, p_{1,m+n-k,3k-1} = 0\\
&p_{4,1,3k-2} = 0, \ldots, p_{4,m+n-k,3k-2} = 0\\
&p_{4,1,3k-1} = 1, \ldots, p_{4,m+n-k,3k-1} = 1\\
&p_{1,m+n+1,3k-2} = 0, \ldots, p_{1,2m+2n-k,3k-2} = 0\\
&p_{1,m+n+1,3k-1} = 1, \ldots, p_{1,2m+2n-k,3k-1} = 1\\
&p_{4,m+n+1,3k-2} = 1, \ldots, p_{4,2m+2n-k,3k-2} = 1\\
&p_{4,m+n+1,3k-1} = 0, \ldots, p_{4,2m+2n-k,3k-1} = 0.
\end{aligned} \tag{3}$$

Moreover, for $k = 1, \dots, n$, add $x_k = 0$ to c_{sat} if c_{pic} contains at least one of the following assignments:

$$\begin{aligned}
&p_{1,1,3k-2} = 0, \dots, p_{1,m+n-k,3k-2} = 0\\
&p_{1,1,3k-1} = 1, \dots, p_{1,m+n-k,3k-1} = 1\\
&p_{4,1,3k-2} = 1, \dots, p_{4,m+n-k,3k-2} = 1\\
&p_{4,1,3k-1} = 0, \dots, p_{4,m+n-k,3k-1} = 0\\
&p_{1,m+n+1,3k-2} = 1, \dots, p_{1,2m+2n-k,3k-2} = 1\\
&p_{1,m+n+1,3k-1} = 0, \dots, p_{1,2m+2n-k,3k-1} = 0\\
&p_{4,m+n+1,3k-2} = 0, \dots, p_{4,2m+2n-k,3k-2} = 0\\
&p_{4,m+n+1,3k-1} = 1, \dots, p_{4,2m+2n-k,3k-1} = 1.
\end{aligned} \tag{4}$$

Furthermore, for $k = 1, \dots, n$, add $x_k = 1$ to c_{sat} if $t_{j,3k-2} = 2$ and c_{pic} contains at least one of the following assignments:

$$\begin{aligned}
&p_{2,1,3k-2} = 1, p_{2,2,3k-2} = 1, \dots, p_{2,m,3k-2} = 1,\\
&p_{3,1,3k-2} = 0, p_{3,2,3k-2} = 0, \dots, p_{3,m,3k-2} = 0.
\end{aligned} \tag{5}$$

Finally, for $k = 1, \dots, n$, add $x_k = 0$ to c_{sat} if $t_{j,3k-2} = 2$ and c_{pic} contains at least one of the following assignments:

$$\begin{aligned}
&p_{2,1,3k-2} = 0, p_{2,2,3k-2} = 0, \dots, p_{2,m,3k-2} = 0,\\
&p_{3,1,3k-2} = 1, p_{3,2,3k-2} = 1, \dots, p_{3,m,3k-2} = 1.
\end{aligned} \tag{6}$$

Then clearly $|c_{\mathrm{sat}}| \leq \ell$ holds. We show that c_{sat} is a clue that makes φ uniquely solvable. To show this, we construct a clue c'_{pic} to I_φ from c_{sat} as follows. Set $c'_{\mathrm{pic}} = \emptyset$. Firstly, for $k = 1, \dots, n$, if $x_k = 1$ is in c_{sat}, then add to c'_{pic} all the assignments in (3). Secondly, for $k = 1, \dots, n$, if $x_k = 0$ is in c_{sat}, then add to c'_{pic} all the assignments in (4). Thirdly, for $k = 1, \dots, n$, if $x_k = 1$ is in c_{sat} and $t_{j,3k-2} = 2$, then add to c'_{pic} all the assignments in (5). Fourthly, for $k = 1, \dots, n$, if $x_k = 0$ is in c_{sat} and $t_{j,3k-2} = 2$, then add to c'_{pic} all the assignments in (6). Finally, as in Claim 3.2, add $p_{i,j,k} = 0$ to c'_{pic} if $p_{i,j,k} = 0$ holds for any solution to I_φ. Then clearly $c_{\mathrm{pic}} \subseteq c'_{\mathrm{pic}}$ holds. Since c_{pic} determines the solution uniquely, so does c'_{pic}. Moreover, for any solution x to φ extending c_{sat}, the solution P' to I_φ corresponding to x contains c'_{pic}, i.e., $c'_{\mathrm{pic}} \subseteq P'$ holds. Hence, P' is uniquely determined, and so is x from Proposition 2. Therefore, c_{sat} is a clue that makes φ uniquely solvable. This completes the proof. ◀

5 Conclusion

We in this paper show that FCP Picross 3D is Σ_2^{P}-complete. To show the result, we provide a parsimonious reduction from positive 1-in-3 SAT, where the FCP of it is known to be Σ_2^{P}-complete [4]. From the reduction, we also show that the counting version of Picross 3D is #P-complete and ASP Picross 3D is NP-complete.

References

1 Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.

2 Nadia Creignou and Miki Hermann. On #P-completeness of some counting problems. Research report 2144, Institut de Recherche en Informatique et en Automatique, 1993.

3 Nadia Creignou and Miki Hermann. Complexity of generalized satisfiability counting problems. *Information and Computation*, 125:1–12, 1996.

4 Erik D. Demaine, Fermi Ma, Ariel Schvartzman, Erik Waingarten, and Scott Aaronson. The fewest clues problem. In *Proceedings of the 8th International Conference on Fun with Algorithms (FUN 2016)*, volume 49 of *LIPIcs*, pages 12:1–12:12, 2016.

5 Erik D. Demaine, Yoshio Okamoto, Ryuhei Uehara, and Yushi Uno. Computational complexity and an integer programming model of shakashaka. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 97:1213–1219, 2014.

6 R. J. Gardner, P. Gritzmann, and D. Prangenberg. On the computational complexity of reconstructing lattice sets from their X-rays. *Discrete Mathematics*, 202:45–71, 1999.

7 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* A K Peters Ltd., 2009.

8 Gabor T. Herman and Attila Kuba, editors. *Discrete Tomography: Foundations, Algorithms, and Applications.* Birkhäuser Basel, Pennsylvania, USA, 1999.

9 Robert W. Irving and Mark R. Jerrum. Three-dimensional statistical data security problems. *SIAM Journal on Computing*, 23(1):170–184, 1994.

10 Kazuhiko Kusano, Kazuyuki Narisawa, and Ayumi Shinohara. Picross 3D is NP-complete. In *Proceedings of the 15th Game Programming Workshop 2010*, pages 108–113, 2010 (in Japanese).

11 Brandon McPhail. Light up is NP-complete. Unpublished manuscript, 2005. URL: `http://www.mountainvistasoft.com/docs/lightup-is-np-complete.pdf`.

12 Takahiro Seta. The complexity of CROSS SUM. Sig technical reports, Information Processing Society of Japan, 2002 (in Japanese).

13 Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 86:1052–1060, 2003.

Uniform Distribution On Pachinko[*]

Naoki Kitamura
Nagoya Institute of Technology, Syowa-ku, Gokiso-cho, Nagoya, Aichi, 466-8555, Japan
29414045@stn.nitech.ac.jp

Yuya Kawabata
Nagoya Institute of Technology, Syowa-ku, Gokiso-cho, Nagoya, Aichi, 466-8555, Japan
29414043@stn.nitech.ac.jp

Taisuke Izumi
Nagoya Institute of Technology, Syowa-ku, Gokiso-cho, Nagoya, Aichi, 466-8555, Japan
t-izumi@nitech.ac.jp

Abstract

Pachinko is a japanese mechanical gambling game similar to pinball. Recently, Akitaya et al. proposed several mathematical models of Pachinko. A number of pins are spiked in a field. A ball drops from the top-side end of the playfield, and falls down. In the 50-50 model, if the ball hits a pin, it moves to the left or right of the pin with equal probability. An arrangement of pins generates a distribution of the drop probability over all columns. We consider the problem of generating uniform distributions. Akitaya et al. show that $(1/2^a)$-uniform distribution is possible for $a \in \{0, 1, 2, 3, 4\}$ and conjectured that it is possible for any positive integer a. In this paper, we show that the conjecture is true by a constructive way.

2012 ACM Subject Classification Mathematics of computing → Combinatoric problems

Keywords and phrases Pachinko, discrete mathematics

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.26

1 Introduction

1.1 Background

Pachinko is a japanese mechanical gambling game similar to pinball (Figure 1). The machine stands up vertically, and the player shoots a metal ball into the playfield. Many pins are spiked in the playfield, and the ball drops from the top of the field. If it goes into a pocket in the field, then the player gets some reward. Recently, Pachinko is analyzed in the context of discrete mathematics. The origin of mathematical Pachinko is the book written by Akiyama in 2008 [3], and recently, Akitaya et al. study an idealized geometry of a simple form of Pachinko [2]. In this paper, we consider one of the mathematical models, called 50-50 model, posed there.

The 50-50 model consists of three factors, field, pins, and a ball. The field is a half-plane triangle lattice with the top-side end. We can put a pin at any lattice point. A *row* is a horizontal line where lattice points exist, and a *column* is a vertical line where lattice points exist. Since we consider the triangle lattice, intersection points of rows and columns do not necessarily have a lattice point (see Figure 2). The ball drops from the center of the top end and falls down vertically. If the ball hits a pin, then it moves to the left or right of the pin with equal probability, and the ball continues to fall down vertically. Once we fix a pin

[*] A preliminary version of this paper is presented in 19th Japan Conference on Discrete and Computational Geometry, Graphs, and Games (JCDCG^3 2016).

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 26; pp. 26:1–26:14
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

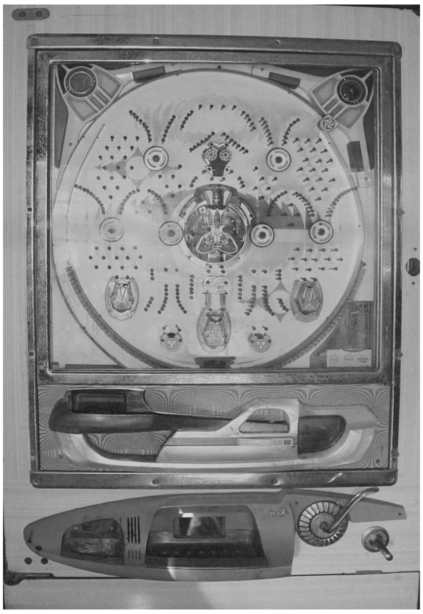


Figure 1 A classical Pachinko (from [1]).

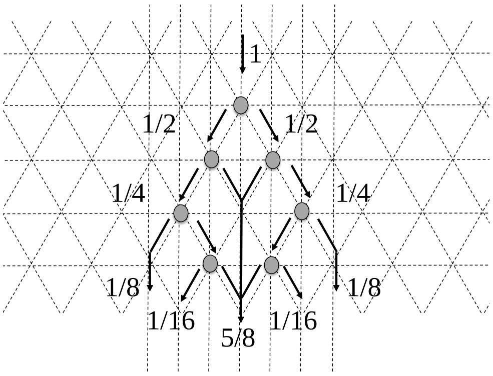


Figure 2 An example of 50-50 model. Each value means the drop probability of each column.

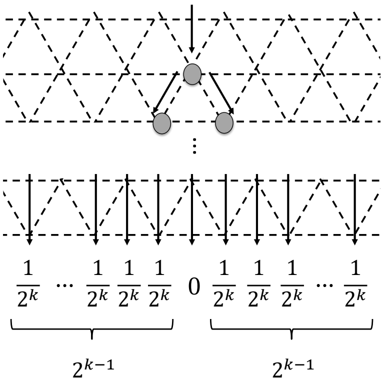

$\frac{1}{2^k}$ ⋯ $\frac{1}{2^k}$ $\frac{1}{2^k}$ $\frac{1}{2^k}$ 0 $\frac{1}{2^k}$ $\frac{1}{2^k}$ $\frac{1}{2^k}$ ⋯ $\frac{1}{2^k}$
2^{k-1}
2^{k-1}

Figure 3 An example of uniform distributions. The center column has probability 0.

arrangement under the 50-50 model, we can calculate the probability that the ball drops to each column. Then we can define its inverse problem of "deciding whether there exists a pin arrangement generating a given distribution or not".

1.2 Problem and Our Result

In [2], it is shown that any probability distribution $\langle p_1, p_2, ..., p_n \rangle$ in the 50-50 model can be constructed within an arbitrarily small additive error, and thus the main interest is the exact generation of a given distribution. The $(1/2^a)$-*uniform distribution* in the 50-50 model is the probability distribution, where the probability that the ball drops at the center is 0 and the probability at the 2^a closest coordinates from the center is $\frac{1}{2^a}$ (see Figure 3). Akitaya et al. show that the $(1/2^a)$-uniform distribution for $a \in \{0, 1, 2, 3, 4\}$ can be constructed, and they conjecture that 2^a uniform distribution for any positive integer a can be constructed in [2]. The contribution of this paper is to show that this conjecture is true. That is, for any $a \geq 1$, the $(1/2^a)$-uniform distribution can be constructed. Moreover, the construction can be done using only a polynomial of 2^a number of pins. To show the result, we introduce a new language-theoretic formulation, which is simple but substantially useful for the analysis of the 50-50 model.

2 Preliminaries

2.1 Configuration and Rewriting Rule

We formulate the problem in the 50-50 model using the notion of formal grammar. A Pachinko machine is represented by a triangle lattice on a half plane with infinite horizontal length and infinite downward vertical length. Each horizontal line containing lattice points is called a *row*. From the top-side end, we assign each row with a y-coordinate $1, 2, \ldots$. Since the field is a triangle lattice, the lattice points on an odd row are half-shifted from those on an even row. To fit them into the standard orthogonal coordinate system, we assign even x-coordinates to the lattice points in even rows, and odd x-coordinates to those in odd rows (see Figure 4). Any coordinate $(i, j) \in \mathbb{N} \times \mathbb{N}^+$ for i and j with different parity is not a lattice point, which is the space for the ball to drop down to lower rows. Those coordinates are called *passages*. Initially, the ball is dropped from the horizontal center of the top-side. Hence, the probability that the ball passes through $(0, 0)$ is one. A pin can be placed at any lattice point. In the 50-50 model, if the dropping ball hits a pin at point (i, j) (i.e., passes through $(i, j-1)$), it moves to either $(i-1, j)$ or $(i+1, j)$ with probability $1/2$. If no pin is spiked at (i, j), the drop probability of (i, j) is equal to that of $(i, j-1)$.

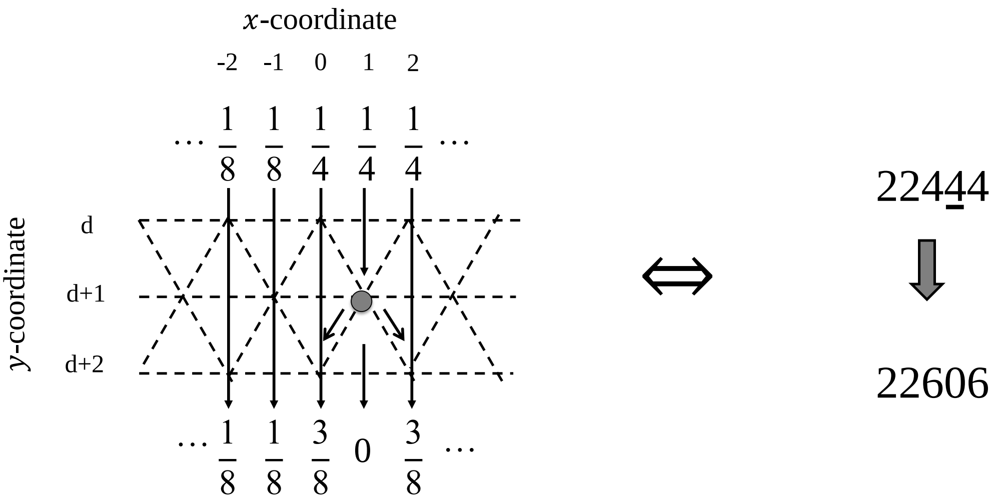
A *pin arrangement* is a set of lattice points where a pin is spiked. Given a pin arrangement P and any $i \geq 1$, P generates the drop probability distribution over all coordinates in the i-th row, which is called the i-th *configuration* of P (or simply say a configuration). Formally, a configuration is a finite odd-length sequence of rational values whose sum is equal to one, where the center of the sequence corresponds to the drop probability at x-coordinate zero and two infinite sequences of zeros spanning x-coordinates $\pm\infty$ are cut off. Throughout this paper, we assume the minimum granularity $1/2^g$ of each probability for some $g \geq 1$. Then, by multiplying each value by 2^g, we can treat any configuration as a sequence of non-negative integer values.

The change of configurations (i.e., the change of the corresponding probability distribution) by placing a pin at a lattice point is expressed as an application of rewriting rules in formal grammar. While we can put two or more pins in the same row, such a pin placement is equivalently translated into the placement in a number of rows where each row contains exactly one pin. Thus, without loss of generality, we assume that each row contains one pin. We regard each configuration as a *word* over the symbol set $[0, 2^g]$. If a pin is put at a lattice point with x-coordinate $(i, j+1)$, the probability mass of coordinate (i, j) is evenly split into $(i-1, j+1)$ and $(i+1, j+1)$, which is expressed by the rewriting rule as follows:

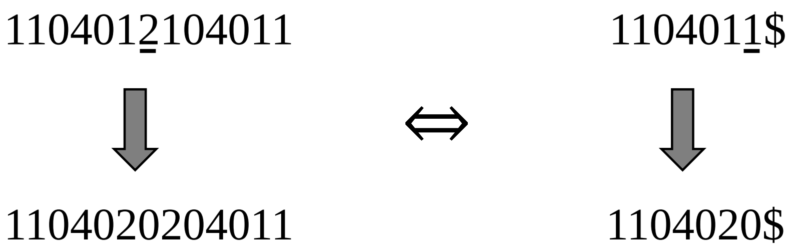
▶ **Definition 1.**

$$abc \to \left[a + \frac{b}{2}\right] 0 \left[c + \frac{b}{2}\right] \qquad \text{(Rule R1)}.$$

where bracket [] represents the single symbol corresponding to the value inside. The symbol a or c may be an implicit zero value omitted in the representation of configurations. An example of rewriting is illustrated in Figure 4.



■ **Figure 4** An example of configurations ($g = 3$) and rewriting.



■ **Figure 5** An example of rewriting symmetric configurations.

2.2 Symmetric Configuration

Throughout this paper, we only consider symmetric configurations, that is, the configurations mirror-symmetric about the center. To express symmetric configurations, the right side from the center is not necessary. More precisely, we denote a symmetric configuration $w[v]w^R$ as $w[v/2]\$$, where w^R is the inversion of w, and \$ is the special symbol representing the right side from the center (say the *boundary*). Except for the center, any rewriting is applied symmetrically. That is, in the transformation of symmetric configurations, putting a pin at (i, j) implies putting another pin at $(-i, j)$. The exceptional case is the rewriting at the center, which is handled by the special rule below (note that the drop probability at the center is expressed by its half).

▶ **Definition 2.**

$$ab\$ \to [a+b]\,0\$ \qquad \text{(Rule R2)}.$$

The symbol \$ corresponds to the center. Figure 5 is an example of how symmetric configurations are rewritten by Rule R2. If we can generate a configuration $u\$$ from $w\$$ by finite-time applications of rewriting rules, then we say $u\$$ is transformed from $w\$$, and write $w\$ \rightsquigarrow u\$$. We also extend this notion of transformability into substring cases. Let $xyz\$$ and $xy'z\$$ be the two words such that y and y' have the same length. If $xyz\$ \rightsquigarrow xy'z\$$ holds for any x and z, we say y' is transformed from y, and write $y \rightsquigarrow y'$.

2.3 Formulation of the Problem

In this section we formalize the problem of generating uniform distributions. The goal of the problem is to generate the probability distribution of $1/2^a$, $1/2^a$, $\cdots$,$1/2^a$, 0, $1/2^a$, $1/2^a$, $\cdots$, $1/2^a$. We call it the $(1/2^a)$*-uniform distribution.* In our construction, the minimum granularity $1/2^{a+1}$ of drop probability suffices to generate the $(1/2^a)$-uniform distribution, and thus the problem is reduced to the transformability of $[2^{a+1}]\$ \rightsquigarrow 2^{(2^a)}\$$ (upper subscripts mean repetition of symbols). The problem actually we solve is a "recursive" version of this transformability, which is stated by the following Theorem.

▶ **Theorem 3.** $4^k0\$ \rightsquigarrow 2^{2k}0\$$ *holds for* $a \geq 4$ *and* $k = 2^a$*.*

In [2], it has been proved that $(1/2^a)$-uniform distribution can be generated for $a \leq 4$. By applying Theorem 3 iteratively, we can show that the $(1/2^a)$-uniform distribution can be generated for any $a \geq 1$.

3 Generating Uniform Distribution

The whole of Section 3 is devoted to the proof of Theorem 3. The proof consists of the following three parts.

1. $4^k0\$ \rightsquigarrow (440)^{\frac{k}{2}}\$$.
2. $(440)^{\frac{k}{2}}\$ \rightsquigarrow 42^{k-3}02^{k-1}4\$$.
3. $42^{k-3}02^{k+1}4\$ \rightsquigarrow 2^{2k}0\$$.

Clearly, the combination of these transformations results in Theorem 3. In the following subsections, we look at the details of each part.

3.1 Part 1: From $4^k0\$$ to $(440)^{\frac{k}{2}}\$$

First, we explain a preliminary lemma.

▶ **Lemma 4.** *Let* $x, y,$ *and* z *be any symbols, and* j *be a positive integer greater than or equal to 3. Then the following transformations are possible.*

$$xy^jz \rightsquigarrow [x+y]0y^{j-2}0[z+y]. \tag{1}$$

$$xy^j0\$ \rightsquigarrow [x+y]0y^{j-1}0\$. \tag{2}$$

Proof. We first consider the transformation (1). The proof is based on the induction on j.
(Basis) In the case of $j = 3$, we have the following transformation (each underline represents the position of rewriting):

$$\begin{aligned}
&x\underline{y}y\underline{y}z \\
\rightsquigarrow &\left[x+\frac{y}{2}\right]0\underline{[2y]}0\left[z+\frac{y}{2}\right] \\
\rightsquigarrow &\left[x+\frac{y}{2}\right]\underline{y}0\underline{y}\left[z+\frac{y}{2}\right] \\
\rightsquigarrow &[x+y]0y0[z+y].
\end{aligned}$$

(Inductive step) Suppose as the induction hypothesis that the transformation (1) is possible for $j = k$. The case of $j = k + 1$ is obtained as follows:

$$\begin{aligned}
&xy^{k+1}z \\
=&\underline{xy^k}yz && \text{(Induction hypothesis)} \\
\rightsquigarrow&[x+y]0y^{k-2}0\underline{[2y]}z \\
\rightsquigarrow&[x+y]0y^{k-2}y0[y+z] \\
=&[x+y]0y^{k-1}0[y+z].
\end{aligned}$$

Thus the transformation (1) is possible. The proof of the transformation (2) follows the rewriting process below:

$$\begin{aligned}
&\underline{xy^j}0\$ && \text{(Transformation (1))} \\
\rightsquigarrow&[x+y]0y^{j-2}0\underline{y}\$ \\
\rightsquigarrow&[x+y]0y^{j-2}y0\$ \\
=&[x+y]0y^{j-1}0\$.
\end{aligned}$$

The lemma is proved. ◀

For simplicity of arguments, we pad an appropriate number of zeros to the left side of w such that the number of zeros in w becomes exactly $k/2 + 1$. The i-th *run* of w ($1 \leq i \leq k/2$) is the substring between i-th zero and $(i + 1)$-th zero (indexed from the left end of w). The length of the i-th run in w is denoted by $l_w(i)$. Now we define the notion of Normal Forms (NFs), which is the class of configurations we have to treat in the proof of Part 1.

▶ **Definition 5.** A word w is a *normal form*(NF) with respect to k if and only if every run in w consists of only symbol 4, the number of runs (of 4) is at most $k/2$, and the symbol neighboring to the boundary is 0.

Let $l_w(j)$ be the length of j-th run in NF w. The *run-length vector* $v(w)$ of w is the $k/2$-dimensional vector whose j-th element corresponds to $l_w(j)$. Let $vol_w(h) = \sum_{j\in[1,h]} l_w(j)$. Then SNFs are defined as follows:

▶ **Definition 6.** A NF w (with respect to k) is a *strongly-normal form*(SNF) (with respect to k) if and only if it satisfies $vol_w(h) \leq 2h$ for any $h \in [1, k/2]$.

Note that $4^k0\$$ and $(440)^{k/2}\$$ are both SNFs. For any two SNFs w_1 and w_2, we define $c(w_1, w_2)$ to be the minimum index such that $l_{w_1}(c(w_1, w_2)) \neq l_{w_2}(c(w_1, w_2))$ holds, and define $\mathcal{N}_k$ as the set of all SNFs with respect to k. Then we define a total order $\prec$ over $\mathcal{N}_k$ by the lexicographic order of corresponding run-length vectors. That is, we define

$$w_1 \prec w_2 \Leftrightarrow l_{w_1}(c(w_1, w_2)) \leq l_{w_2}(c(w_1, w_2)).$$

For any SNF w, let $t(w)$ be the position of the leftmost run with length more than two, that is, $t(w) = \min_{j\in[1,k/2],l_w(j)\geq 3} j$. If no run has a length more than two, we define $t(w) = k/2 + 1$. The rewriting process of $4^k0\$ \rightsquigarrow (440)^{k/2}\$$ is to iterate the application of Lemma 4 (1) (if $t(w) < k/2$) or (2) (if $t(w) = k/2$) to the $t(w)$-th run, until the transformation reaches the word w' with $t(w) = k/2 + 1$. In the remaining part of this section we show that this process correctly creates $(440)^{k/2}$.

▶ **Lemma 7.** *Let x be any SNF, and x' be the word after the application of Lemma 4 to the $t(x)$-th run in x. Then, x' is also an SNF and $x \prec x'$.*

Proof. It is easy to check that any run of x' consists of only 4s and symbol 0 is the neighbor of \$ in x'. By the definition of SNFs for $h = 1$, for any SNF w, $l_w(1) \leq 2$ holds and thus $t(w) > 1$ necessarily holds. Since we have to apply Lemma 4 to the first run for increasing the number of runs to more than $k/2$, the number of runs in x' is at most $k/2$. Consequently x' is a NF. Since the application of Lemma 4 at the i-th run of a word w increases $l_{i-1}(w)$ and $l_{i+1}(w)$ by one, and decreases $l_i(w)$ by two, the transformation from x to x' can increase only the value of $vol_x(t(x)-1)$. For showing that x' is an SNF, it suffices to prove $vol_{x'}(t(x)-1) \leq 2(t(x)-1)$. By the fact of $l_x(t(x)) \geq 3$, we have $vol_x(t(x)-1)+3 \leq vol_x(t(x)) \leq 2t(x)$ and thus $vol_x(t(x)-1) \leq 2(t(x)-1)-1$ holds. Since the length of $t(x)$-th run increases at most by one after the application of Lemma 4. We obtain $vol_{x'}(t(x)-1) \leq vol_x(t(x)-1)+1 \leq 2(t(x)-1)$. Thus x' is a SNF. By the definition, $c(w,w') = t(x)-1$ holds and thus we obtain $l_{c(w,w')}(w') > l_{c(w,w')}(w)$, that is $x' \prec x$. The lemma is proved ◀

▶ **Lemma 8.** *The word $(440)^{k/2}$\$ is the maximum element with respect to $\prec$.*

Proof. Let $w = (440)^{k/2}$\$. Suppose for contradiction that a SNF w' satisfies $w \neq w'$ and $w \prec w'$. Then, $vol_w(c(w,w')) < vol_{w'}(c(w,w'))$ holds. However, since $vol_w(c(w,w')) = 2c(w,w')$ holds, we have $vol_{w'}(c(w,w')) > 2c(w,w')$. It contradicts the fact that w' is an SNF. ◀

The two lemmas above imply that our rewriting process eventually leads the maximum element of SNFs, and thus the following corollary holds.

▶ **Corollary 9.** *Let $k \in \mathbb{N}$ be any even positive integer. Then the following transformation is possible.*

$$4^k0\$ \rightsquigarrow (440)^{\frac{k}{2}}\$.$$

3.2 Part 2: From $(440)^{\frac{k}{2}}\$$ to $42^{k-3}02^{k-1}4\$$

In this section, we first introduce a magical string $B_i = 42^i02^{i+1}4\$$, as well as its nice properties. Before showing the properties of B_i, we present further preliminary lemmas.

▶ **Lemma 10.** *Let x,y, and z be any symbols, and j be any positive integer. Then the following transformations are possible.*

$$x[2y]y^jz \rightsquigarrow [x+y]y^{j-1}0[2y]z. \tag{3}$$

$$x[2y]y^jz \rightsquigarrow [x+y]y^j0[z+y]. \tag{4}$$

Proof. We first consider the transformation (3). The proof is based on the induction on j. (Basis) In the case of $j = 1$, we can have the following transformation:

$$\begin{aligned} & x[\underline{2y}]yz \\ \rightsquigarrow & [x+y]0[2y]z. \end{aligned}$$

(Inductive step) Suppose as the induction hypothesis that the transformation (3) is possible for $j = k$. The case of $j = k + 1$ is obtained as follows:

$$\begin{aligned}
&x[\underline{2y}]y^{k+1}z \\
\rightsquigarrow&[x+y]\underline{0[2y]y^{k}z} && \text{(Induction hypothesis)} \\
\rightsquigarrow&[x+y]y^{k}0[2y]z.
\end{aligned}$$

The proof for the transformation (4) follows the rewriting process presented below:

$$\begin{aligned}
&\underline{x[2y]y^{j}z} && \text{(Transformation(3))} \\
\rightsquigarrow&[x+y]y^{j-1}0\underline{[2y]}z \\
\rightsquigarrow&[x+y]y^{j}0[z+y].
\end{aligned}$$

The lemma is proved. ◀

▶ **Corollary 11.** *Let x,y, and z be any symbols, and j be any positive integer. Then the following transformations are possible.*

$$xy^{j}[2y]z \rightsquigarrow x[2y]0y^{j-1}[z+y]. \tag{5}$$

$$xy^{j}[2y]z \rightsquigarrow [x+y]0y^{j}[z+y]. \tag{6}$$

▶ **Lemma 12.** *Let j be a positive integer greater than or equal to 4. Then $02^{j}4\$ \rightsquigarrow 2^{2}02^{j-2}4\$$ holds.*

Proof. We can rewrite $02^{j}4\$$ as follows.

$$\begin{aligned}
&02^{j}4\$ \\
=&\underline{02222}2^{j-4}4\$ && \text{(Lemma 4 (1), } x=0,\ y=2,\ z=2) \\
\rightsquigarrow&202\underline{042^{j-4}}4\$ && \text{(Lemma 10 (4), } x=0,\ y=2,\ z=2) \\
\rightsquigarrow&202^{j-3}04\underline{4}\$ \\
\rightsquigarrow&202^{j-3}0\underline{8}0\$ \\
\rightsquigarrow&2\underline{02^{j-3}40}4\$ && \text{(Corollary 11 (6), } x=0,\ y=2,\ z=0) \\
\rightsquigarrow&2202^{j-3}24\$ \\
=&2^{2}02^{j-2}4\$.
\end{aligned}$$

The lemma is proved. ◀

The goal of Part 2 is to obtain B_{k-3} from $(440)^{\frac{k}{2}}$. We introduce two important properties of $B_i = 42^{i}02^{i+2}4\$$, which is the primary reason why we claim that B_i is "magical".

▶ **Lemma 13.** *Let i be any positive integer. Then the following transformations are possible.*

$$04B_i \rightsquigarrow 40B_i. \tag{7}$$

$$0440B_i \rightsquigarrow B_{i+2}. \tag{8}$$

Proof. We first consider the transformation (7), which is obtained as follows.

$$\begin{aligned}
&04B_i \\
=&04\underline{4}2^i02^{i+2}4\$ \\
\rightsquigarrow&06\underline{042^{i-1}0}2^{i+2}4\$ && \text{(Lemma 10 (4), } x=0,\ y=2,\ z=0) \\
\rightsquigarrow&062^i\underline{02^{i+3}4}\$ && \text{(Lemma 12)} \\
\rightsquigarrow&0\underline{62^{i+2}0}2^{i+1}4\$ && \text{(Lemma 4 (1), } x=6,\ y=2,\ z=0) \\
\rightsquigarrow&0\underline{8}02^i02^{i+2}4\$ \\
\rightsquigarrow&4042^i02^{i+2}4\$ \\
=&40B_i.
\end{aligned}$$

The proof for the transformation (8) follows the rewriting process below:

$$\begin{aligned}
&0440B_i \\
=&044\underline{042^i0}2^{i+2}4\$ && \text{(Lemma 10 (4), } x=0,\ y=2,\ z=0) \\
\rightsquigarrow&0442^{i+1}02^{i+3}4\$ \\
=&\underline{04B_{i+1}} && \text{(Transformation (7))} \\
\rightsquigarrow&40B_{i+1} \\
=&4\underline{042^{i+1}0}2^{i+3}4\$ && \text{(Lemma 10 (4), } x=0,\ y=2,\ z=0) \\
\rightsquigarrow&42^{i+2}02^{i+4}4\$ \\
=&B_{i+2}.
\end{aligned}$$

The lemma is proved. ◀

Why these properties are so important? The intuitive understanding of the reason for the first property is that we can treat B_i as \$. In Part 1, we only use the application of rule R2 for $b = 4$. Then the behaviors of 4\$ and $4B_i$ are the same, and thus any transformation in Section 3.1 applicable to $w'\$$ is also applicable to $w'B_i$. This fact yields the corollary below.

▶ **Corollary 14.** *Let $k' \in \mathbb{N}$ be an even positive integer, and w be a SNF with respect to k'. Letting w' be the word obtained from w by deleting \$, $wB_i \rightsquigarrow (440)^{k'/2}B_i$ holds.*

Combining this corollary with the second property of Lemma 13, we can show that B_i can recursively "absorb" substring 440 to make itself grow up. The following lemma corresponds to the base case of this rewriting process.

▶ **Lemma 15.**

$$(440)^4\$ \rightsquigarrow 44440B_1.$$

Proof. Deferred to the appendix. ◀

The following two lemmas are the main body of Part 2, which shows the rewriting process of absorbing substring 440.

▶ **Lemma 16.** *Let i and j be any positive integer. Then $0^i(440)^iB_j \rightsquigarrow B_{j+2i}$ holds.*

Proof. The proof is based on the induction on i. (Basis) In the case of $i = 1$, we have the following transformation:

$$\begin{aligned}
&0440B_j && \text{(Lemma 13 (8))} \\
\rightsquigarrow&B_{j+2},
\end{aligned}$$

and in the case of $i = 2$, we also have the following transformation:

$$\begin{aligned}
&0^2(440)^2B_j \\
=&0^244\underline{0440B_j} && \text{(Lemma 13 (8))} \\
\rightsquigarrow&0^24\underline{4B_{j+2}} && \text{(Lemma 13 (7))} \\
\rightsquigarrow&0^2\underline{8}0B_{j+2} \\
\rightsquigarrow&040\underline{4B_{j+2}} && \text{(Lemma 13 (7))} \\
\rightsquigarrow&0440B_{j+2} && \text{(Lemma 13 (8))} \\
\rightsquigarrow&B_{j+4}.
\end{aligned}$$

(Inductive step) Suppose as the induction hypothesis that Lemma 16 holds for $i = k$ ($k \geq 2$). The case of $i = k + 1$ is proved by:

$$\begin{aligned}
&0^{k+1}(440)^{k+1}B_j && \text{(Because of } k \geq 2\text{)} \\
=&0^{k+1}(440)^{k-2}44044\underline{0440B_j} && \text{(Lemma 13 (8))} \\
\rightsquigarrow&0^{k+1}(440)^{k-2}4404\underline{4B_{j+2}} && \text{(Lemma 13 (7))} \\
\rightsquigarrow&0^{k+1}(440)^{k-2}440\underline{8}0B_{j+2} \\
\rightsquigarrow&0^{k+1}(440)^{k-2}4440\underline{4B_{j+2}} && \text{(Lemma 13 (7))} \\
\rightsquigarrow&\underline{0^{k+1}(440)^{k-2}44440B_{j+2}} && \text{(Corollary 14)} \\
\rightsquigarrow&\underline{0^k(440)^kB_{j+2}} && \text{(Induction hypothesis)} \\
\rightsquigarrow&B_{j+2(k+1)}.
\end{aligned}$$

The Lemma is proved. ◀

▶ **Lemma 17.** *Let k be a positive integer greater than or equal to 8. Then $(440)^{\frac{k}{2}}\$ \rightsquigarrow B_{k-3}$ holds.*

Proof.

$$\begin{aligned}
&(440)^{\frac{k}{2}}\$ \\
=&(440)^{\frac{k}{2}-4}\underline{(440)^4}\$ && \text{(Lemma 15)} \\
\rightsquigarrow&(440)^{\frac{k}{2}-4}44440B_1.
\end{aligned}$$

$(440)^{\frac{k}{2}-4}44440$ is an SNF (with respect to $k - 2$). Thus, we can rewrite it as follows.

$$\begin{aligned}
&\underline{(440)^{\frac{k}{2}-4}44440B_1} && \text{(Corollary 14)} \\
\rightsquigarrow&(440)^{\frac{k}{2}-2}B_1 && \text{(Lemma 16)} \\
\rightsquigarrow&B_{k-3}.
\end{aligned}$$

The lemma is proved. ◀

3.3 Part 3: From $42^{k-3}02^{k-1}4\$$ to $2^{2k}0\$$

Finally, we prove that $42^{k-3}02^{k-1}4\$$ can be transformed into $2^{2k}0\$$. We explain four preliminary lemmas used in this section.

▶ **Lemma 18.** *Let x, z be any positive integers, and y be a positive integer greater than or equal to 2. Then the following transformation is possible.*

$$02^x02^{2y}02^z\$ \rightsquigarrow 02^{x+y-1}02^202^{z+y-1}\$.$$

Proof. The proof is based on the induction on y. (Basis) In the case of $y = 2$, we can have the following transformation:

$$\begin{array}{ll} 02^x\underline{02^40}2^z\$ & \text{(Lemma 4 (1), } x = 0,\ y = 2,\ z = 0) \\ \rightsquigarrow 02^{x+1}02^202^{z+1}\$. & \end{array}$$

(Inductive step) Suppose as the induction hypothesis that the Lemma 18 holds for $y = k$. The case of $y = k + 1$ is proved by:

$$\begin{array}{ll} 02^x\underline{02^{2(k+1)}0}2^z\$ & \text{(Lemma 4 (1), } x = 0,\ y = 2,\ z = 0) \\ \rightsquigarrow 02^{x+1}\underline{02^{2k}0}2^{z+1}\$ & \text{(Induction hypothesis)} \\ \rightsquigarrow 02^{x+k}02^202^{z+k}\$. & \end{array}$$

The Lemma is proved. ◀

▶ **Lemma 19.** *Let i be a positive integer greater than or equal to 5. Then $02202^i4\$ \rightsquigarrow 2^{i-4}022024^4\$$ holds.*

Proof. The proof is based on the induction i. (Basis) In the case of $i = 5$, we can have the following transformation:

$$\begin{array}{ll} 022\underline{02^54}\$ & \text{(Lemma 12)} \\ \rightsquigarrow \underline{022220}2^34\$ & \text{(Lemma 4 (1), } x = 0,\ y = 2,\ z = 0) \\ \rightsquigarrow 202202^44\$. & \end{array}$$

(Inductive step) Suppose as the induction hypothesis that Lemma 19 holds for $i = k$. The case of $i = k + 1$ is proved by:

$$\begin{array}{ll} 022\underline{02^{k+1}4}\$ & \text{(Lemma 12)} \\ \rightsquigarrow \underline{022220}2^{k-1}4\$ & \text{(Lemma 4 (1), } x = 0,\ y = 2,\ z = 0) \\ \rightsquigarrow 2\underline{02202^k4}\$ & \text{(Induction hypothesis)} \\ \rightsquigarrow 2^{k-3}02202^44\$. & \end{array}$$

The lemma is proved. ◀

▶ **Lemma 20.** *Let i be any positive integer. Then $x2^i\$ \rightsquigarrow [x+2]2^{i-1}0\$$ holds.*

Proof. The proof is based on the induction on i. (Basis) In the case of $i = 1$, we have the following transformation:

$$\begin{array}{l} x\underline{2}\$ \\ \rightsquigarrow [x+2]0\$. \end{array}$$

(Inductive step) Suppose as the induction hypothesis that Lemma 20 holds for $i = k$. For the case of $i = k + 1$, we have the transformation as follows:

$$\begin{aligned}
&x2^{k+1}\$ \\
=&x\underline{22^k}\$ && \text{(Induction hypothesis)} \\
\rightsquigarrow&\underline{x42^{i-2}0}\$ && \text{(Lemma 10 (4), } x = x,\ y = 2,\ z = 0) \\
\rightsquigarrow&[x + 2]2^{i-2}0\underline{2}\$ \\
\rightsquigarrow&[x + 2]2^{i-1}0\$.
\end{aligned}$$

The case of $i = k + 1$ is proved, and thus the lemma holds. ◀

▶ **Lemma 21.**

$$022022224\$ \rightsquigarrow 222222220\$.$$

Proof. Deferred to the appendix. ◀

The combination of the four lemmas straightforwardly deduces the main lemma of Part 3.

▶ **Lemma 22.** *Let k be a positive integer greater than or equal to 8. The following transformation is possible.*

$$0042^{k-3}02^{k-1}4\$ \rightsquigarrow 2^{2k}0\$.$$

Proof. We can have the following transformation:

$$\begin{aligned}
&0\underline{042^{k-3}0}2^{k-1}4\$ && \text{(Lemma 10 (4), } x = 0,\ y = 2,\ z = 0) \\
\rightsquigarrow&\underline{02^{k-2}0}2^{k}4\$ && \text{(Lemma 18)} \\
\rightsquigarrow&2^{\frac{k}{2}-2}\underline{02^{2}02^{\frac{3k}{2}-2}4}\$ && \text{(Lemma 19)} \\
\rightsquigarrow&2^{\frac{k}{2}-2}2^{\frac{3k}{2}-6}02^{2}02^{4}4\$ \\
=&2^{2k-8}\underline{02^{2}02^{4}4}\$ && \text{(Lemma 21)} \\
\rightsquigarrow&2^{2k-8}222222220\$ \\
=&2^{2k}0\$.
\end{aligned}$$

◀

4 Conclusions and discussion

In this paper, we proved that $(1/2^a)$-uniform distributions in the 50-50 model can be generated for any $a \geq 1$. This is the complete positive answer for the open problem posed by [2]. In this article we do not consider the complexity of the generation process — the number of pins, or the number of rows. While it is not difficult to bound the number of pins used in our construction by a polynomial of 2^a, its fine-grained analysis is not proposed yet (following a rough estimation it is bounded by $O(2^{4a})$, but the tight analysis is probably $O(2^{3a})$ pins). The complexity on the number of rows is much complicated. In our construction, the restriction of one pin at one row made the analysis so simple, but when we want to optimize the number of rows, that restriction cannot be used. It is also an interesting to reveal the computational complexity on the problem of generating given distributions. In the context of formal language theory, our rewriting rule is not a context-free grammar, and thus it is not clear if the decision problem on the generability of a given distribution is in class P or not.

References

1 Pachinko - wikipedia. URL: `https://en.wikipedia.org/wiki/Pachinko`.

2 Hugo A Akitaya, Erik D Demaine, Martin L Demaine, Adam Hesterberg, Ferran Hurtado, Jason S Ku, and Jayson Lynch. Pachinko. *Computational Geometry: Theory and Applications*, 68, 2018.

3 Jin Akiyama and Mari-Jo P. Ruiz. *Pachinko math.In A Day's Adventure in Math Wonderland*. World Scientific, 2008.

Omitted Proofs

▶ **Lemma 15.**

$$(440)^4 \rightsquigarrow 44440B_1.$$

Proof. The lemma is proved by the following transformation:

$$\begin{aligned}
&(440)^4\$ \\
=&(440)^2 4\underline{4}04\underline{4}0\$ \\
\rightsquigarrow&(440)^2 60260\underline{2}\$ \\
\rightsquigarrow&(440)^2 60\underline{2}6\underline{2}0\$ \\
\rightsquigarrow&(440)^2 610\underline{8}01\$ \\
\rightsquigarrow&(440)^2 61\underline{4}0\underline{4}1\$ \\
\rightsquigarrow&(440)^2 630\underline{4}03\$ \\
\rightsquigarrow&(440)^2 63\underline{2}0\underline{2}3\$ \\
\rightsquigarrow&(440)^2 6\underline{4}0204\$ \\
\rightsquigarrow&(440)^2 802204\$ \\
=&440440\underline{8}02204\$ \\
\rightsquigarrow&440444\underline{04220}4\$ && \text{(Lemma 10 (4), } x=0,\ y=2,\ z=0) \\
\rightsquigarrow&44044\underline{42220}24\$ && \text{(Lemma 4 (1), } x=4,\ y=2,\ z=0) \\
\rightsquigarrow&4404\underline{4}6020224\$ \\
\rightsquigarrow&44060\underline{8}020224\$ \\
\rightsquigarrow&4406\underline{4}0\underline{4}20224\$ \\
\rightsquigarrow&440\underline{8}040\underline{4}0224\$ \\
\rightsquigarrow&444044202224\$ \\
=&44404B_1 && \text{(Lemma 13 (7))} \\
\rightsquigarrow&44440B_1.
\end{aligned}$$

◀

▶ **Lemma 21.**

$$022022224\$ \rightsquigarrow 222222220\$.$$

Proof. The lemma is proved by the following transformation:

$$
\begin{aligned}
&022\underline{02222}4\$ && \text{(Lemma 4 (1), } x=0,\ y=2,\ z=2\text{)}\\
\rightsquigarrow&0222020 4\underline{4}\$ &&\\
\rightsquigarrow&0222020\underline{8}0\$ &&\\
\rightsquigarrow&02220240\underline{4}\$ &&\\
\rightsquigarrow&0222024\underline{4}0\$ &&\\
\rightsquigarrow&02220260\underline{2}\$ &&\\
\rightsquigarrow&02220\underline{2}6\underline{2}0\$ &&\\
\rightsquigarrow&022210\underline{8}01\$ &&\\
\rightsquigarrow&02221\underline{4}0\underline{4}1\$ &&\\
\rightsquigarrow&022230\underline{4}03\$ &&\\
\rightsquigarrow&02223\underline{2}0\underline{2}3\$ &&\\
\rightsquigarrow&\underline{022240}204\$ && \text{(Corollary 11 (6), } x=0,\ y=2,\ z=0\text{)}\\
\rightsquigarrow&20222220\underline{4}\$ &&\\
\rightsquigarrow&2\underline{02222240}\$ && \text{(Corollary 11 (6), } x=0,\ y=2,\ z=0\text{)}\\
\rightsquigarrow&22\underline{0222222}\$ && \text{(Lemma 20)}\\
\rightsquigarrow&222222220\$.
\end{aligned}
$$

◀

The complexity of speedrunning video games

Manuel Lafond[1]
Department of Mathematics and Statistics, University of Ottawa, Canada
mlafond2@uOttawa.ca

Abstract

Speedrunning is a popular activity in which the goal is to finish a video game as fast as possible. Players around the world spend hours each day on live stream, perfecting their skills to achieve a world record in well-known games such as Super Mario Bros, Castlevania or Mega Man. But human execution is not the only factor in a successful speed run. Some common techniques such as *damage boosting* or *routing* require careful planning to optimize time gains. In this paper, we show that optimizing these mechanics is in fact a profound algorithmic problem, as they lead to novel generalizations of the well-known NP-hard knapsack and feedback arc set problems.

We show that the problem of finding the optimal damage boosting locations in a game admits an FPTAS and is FPT in $k + r$, the number k of enemy types in the game and r the number of health refill locations. However, if the player is allowed to lose a life to regain health, the problem becomes hard to approximate within a factor 1/2 but admits a $(1/2 - \epsilon)$-approximation with two lives. Damage boosting can also be solved in pseudo-polynomial time. As for routing, we show various hardness results, including $W[2]$-hardness in the time lost in a game, even on bounded treewidth stage graphs. On the positive side, we exhibit an FPT algorithm for stage graphs of bounded treewidth and bounded in-degree.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Theory of computation → Approximation algorithms analysis, Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Approximation algorithms, parameterized complexity, video games, knapsack, feedback arc set

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.27

1 Introduction

The study of the complexity of video games has been a relatively popular area of research in the recent years. This line of work first started in the early 2000s with puzzle-oriented video games such as *Minesweeper*, *Tetris* or *Lemmings* [22, 11, 25][2]. More recently, platforming games were subjected to complexity analysis [17], and it is now known that for a wide variety of such games (including *Super Mario Bros*, *Donkey Kong Country* or *Zelda*), it is NP-hard [5] or sometimes PSPACE-hard [13] to decide whether a given instance of the game can be finished. Notably, Viglietta proposed in [24] a series of *meta-theorems* that describe common video game mechanics under which a game is NP-hard or PSPACE-hard.

Of course, few games are (computationally) hard to finish, as there is little incentive for publishers to release an unfinishable game. Here, we take a different perspective on the complexity of video games, and rather ask *how fast can a game be finished*? This question is of special interest to the adepts of *speedrunning*, in which the goal is to finish a video game

[1] The author acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for the financial support of this project.

[2] All products, company names, brand names, trademarks, and sprites are properties of their respective owners. Video game screen-shots and sprites are used here under Fair Use for educational purposes.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 27; pp. 27:1–27:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

as fast as possible. This has been a relatively obscure activity until the last decade, during which speedrunning has seen a significant increase in popularity. This is especially owing to video game streaming websites, where professional speedrunners can spend hours each day on camera trying to earn a world record whilst receiving enough donations from viewers to make a living. Games Done Quick, one of the most popular events in this discipline, is a speedrunning marathon in which professional gamers take turn on live stream to go through a wide range of games as fast as possible [2]. Performances are broadcast 24 hours a day for a whole week, and viewers are invited to provide donations which are then given to charitable organizations. The event went from raising $10,000 during its first event in 2010 to amassing over $2 million in its January 2018 event.

Owing to this popularity, speedrunning is now an extremely competitive area, and having near-perfect execution is mandatory to obtain reasonable times. A single misplaced jump, or an attack that comes a split-second late, can cost a player a world record. There is, however, a category of speedrunning that can circumvent these harsh execution requirements: Tool Assisted Speedruns (TAS). In a TAS, the player is allowed to use any tool provided by emulators, which include slowing down the game, rewinding the game, saving multiple states and reloading them, etc. In the end, the final speedrun is presented in a continuous segment, as if played by a human. In a TAS, execution is therefore not the main challenge, as the player can retry any portion of the game hundreds of times if necessary. But speedrunning remains a challenging task, as difficult optimization problems arise.

In this paper, we are interested in the algorithmic challenges underlying some common mechanics that are unique to speedrunning. We first formulate the problem of speedrunning by modeling a game as a series of punctual *time-saving events*, which can be taken or not. This is in contrast with the natural formulation "given a video game X, can X be finished in time t", as it was done for *Mario Kart* in [7]. This allows our results to be applicable to any game that can be described by time-saving events, and also enables us to avoid dealing with unfinishable games.

We then study the approximation and parameterized complexity aspects of the techniques of *damage boosting* and *routing stages.* Damage boosting consists in taking damage intentionally to go through some obstacles quickly. The amount of damage that can be taken in a game is limited, and it is possible to regain health using items, or by losing a life (this is called *death abusing*). This can be seen as a generalization of the *knapsack* problem in which the items come in a specific order and some of them have a negative weight. We show that if no life can be lost, optimizing damage boosts in a game admits the same FPTAS as knapsack, and is fixed-parameter tractable (FPT) in the number of possible damage sources and healing locations. If lives can be lost to regain health, we show that damage boosting cannot be approximated within a factor 1/2 or better, but can be approximated within a factor $1/2 - \epsilon$ with two lives and can be solved in pseudo-polynomial time.

Routing applies to games in which the player is free to choose in which order a set of stages is to be completed. This includes the *Mega Man* games, for example. Each completed stage yields a new ability to the player, which can then be used in later stages to gain time on certain events, such as defeating a boss more quickly. The time saved in an event depends on the best ability currently available. As we shall see, this makes Routing a generalization of the well-known *feedback arc set* (FAS) problem, as the time-gain dependencies can be represented as a directed graph D. Unlike FAS though, we show that Routing is $W[2]$-hard in the time lost in a game, even if D has treewidth 1, and that it is also hard to approximate within a $\mathcal{O}(\log n)$ factor. We then show that Routing is FPT in the maximum in-degree of D plus its treewidth.

The paper is structured as follows. In Section 2, we provide a non-technical summary of the speedrunning mechanics that are discussed in this work and present our general model of speedrunning. In Section 3, we formally define the problem of optimizing damage boosting and present our algorithmic results. Then in Section 4, we define our routing optimization problems and provide the underlying algorithmic results.

2 Models, speedrunning mechanics, and problems

In this section, we first motivate our model of speedrunning, and how we depart from the traditional formulation of deciding whether a stage can be finished. We then describe the two speedrunning mechanics that we study in more detail.

As mentioned before, perhaps the most natural formulation of speedrunning is the following: given a set of stages, we are asked whether they can be completed in time at most t [7]. However for many games, it is NP-hard to decide whether a given set of stages can be completed at all (e.g. [24]). It follows that for these games, speedrunning is NP-hard even for $t = \infty$. But in reality, video games that are played by speedrunners are always known to be completable. We will therefore assume that an initial way of finishing the game is known, which yields an upper bound t on the time required to complete the game. This time t is usually the time taken to finish the game "normally", as intended by the developers. The problem of speedrunning now becomes: given an initial way of completing the game in time t, can this be improved to time $t' < t$?

To simplify further, stages are often linear and time saves usually consist of punctual events that allow the player to save a few seconds over the developer-intended path. For example, the player may exploit a glitch to go through a wall, or use a certain item to defeat an enemy faster than usual. These punctual events are assumed to occur one after another, and therefore, we will model a stage as a sequence $S = (e_1, \dots, e_n)$ of *time-saving events*. For each such event e_i, the player has a choice of taking the time save from e_i or not. If the event only has positive consequences, then of course the player must take it and we will assume that all events in S offer some sort of trade-off. A notable advantage of this formulation is that it does not depend on a specific video game. For instance, if the events of S model damage boosting, then our hardness results apply to any game that allows damage boosting as a mechanic. We now describe this latter notion.

Damage boosting

The idea of damage boosting is to take damage to save time. This is a common technique that is useful in one of the following ways. In many games, the player is given some invulnerability time after taking damage. This invincibility period allows unintended behavior such as walking on deadly spikes or going through a horde of enemies quickly. Also, when taking damage, the player often loses control and gets knocked back, regardless of the current location and status of the character. If damage is taken at the apex of a jump, say, then this back-knocking can extend the jump higher and farther than normal, allowing the player to access unintended locations. An example of this is illustrated in Figure 1.

Damage-boosting is not without cost. In a game, the player has a limited number of *hit points*, or HP for short. In the top-left of Figure 1, one can see that the player has a maximum of 16 HP, but has 14 remaining after hitting a bat. Each time damage is taken, the player's HP decreases by a certain amount and a life is lost when it reaches 0. Suppose that each time-saving event is described by a pair (d, t), where d is the damage taken and the time gained t. Then it is easy to see that this is exactly the knapsack problem. Indeed,

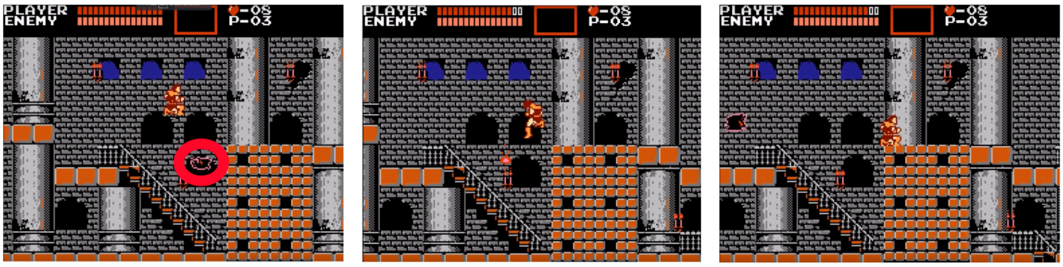


Figure 1 A well-known example of damage boosting in the NES game Castlevania. In this portion of Stage 1, the developer-intended path is to go downstairs, go through an underground section, go up and reappear on the right side of the screen. Here, Simon Belmont can skip the underground section by passing over the wall on the right side of the screen. To do this, the player times a precise jump while facing right, switches direction in mid-air to face left, and lands on a bat passing by at the right moment, damaging the player. When Simon Belmont takes damage, he says "Ow" and gets knocked back (middle figure). This back-knocking allows him to extend his jump farther right and reach the ledge of the wall. This saves 30-40 seconds over taking the normal path.

if *hp* is the starting HP of the player, speedrunning with damage boosting asks for the set of time-saving events that can be taken such that a maximum total time gain is achieved, and such that the total damage of these events does not exceed *hp*.

The problem can be made more interesting by considering the possibility of regaining health during a stage. For instance in Castlevania, there is chicken hidden inside walls and candles across the castle. Fetching these chickens is usually time-consuming, and the player must decide whether the additional damage boosts that this allows is worth it. In Figure 2, Richter Belmont from Castlevania X takes a detour to a dead-end to grab a chicken and regain health. Another way to regain HP is to lose a life. When the player runs out of HP, a life is lost and the game restarts at the last checkpoint with full health. This can be beneficial if the checkpoint is not too far away and new damage boosts are to be taken. It is worth mentioning that the idea of losing lives is used in [13] to establish the hardness of completing a Super Mario Bros game (although losing lives is not used as a health refill mechanism).

Routing

In many games, the player has the freedom to choose the order in which to clear a set of stages or to visit a set of locations. For example, in many Metroid games, a certain set of items scattered across a map must be obtained before reaching the end of the game, and the goal of *routing* is to obtain these items in the optimal order[3].

As another example, in the Mega Man games, the order in which stages are visited is fundamental from a speedrunning perspective. When a stage is cleared after defeating its robot master, Mega Man gains a new weapon/ability that can be used in the latter stages. This has an impact on how fast a stage S can be completed, as previously obtained weapons can be used to gain time – notably during the boss fights. For instance, in Mega Man 2, the boss Crash Man takes 2 damage from the regular weapon, but 20 damage from the weapon left by Air Man. Thus Crash Man can be defeated 10 times faster if the Air Man stage is

[3] The astute reader will observe that *Metroid* games do restrict the ordering of locations that can be visited. However, many sequence-breaking glitches have been found in the last years, and this ordering restriction is often irrelevant. For example in *Super Metroid*, the *Norfair* boss *Ridley* is often defeated first in speedruns, whereas this boss is normally supposed to be reached last.

Figure 2 “This candle chicken certainly tastes great, but it is time-consuming...” - Richter Belmont going out of his way for meat.

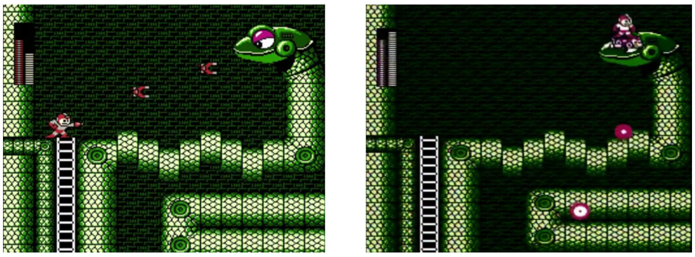


Figure 3 A event in *Mega Man 3*: facing Big Snakey. If Mega Man has Magnet Man’s weapon, Big Snakey can be defeated in 5 shots, which saves about 8 seconds. But if Mega Man has acquired Rush Jet from Needle Man, he can simply fly over Big Snakey, which saves about 15 seconds.

cleared beforehand. Bosses are not the only time-saving events in the game though, as the stages themselves also offer many opportunities. See Figure 3 for an example. Here, Mega Man must face Big Snakey and can achieve various time gains depending on the weapons currently at his disposal from the previous stages.

The problem of routing is to determine the order in which to clear the stages so as to save a maximum amount of time. If we represent stages as a graph, with an arc from S' to S weighted by the time saved in stage S by having cleared S' first, this is similar to the feedback arc set problem, which asks for an ordering of the vertices of a graph in a way that the number of forward edges is maximized. One notable difference is that each event within a stage can make use of a different previous stage.

Before proceeding, we introduce some notation that we will use throughout the paper. Given an ordered list $S = (s_1, \ldots, s_n)$, we write $s_i <_S s_j$ if $i < j$ (and $s_i \leq_S s_j$ if $i \leq j$). We denote by *head*(S) and *tail*(S) the first and last element of S, respectively. A *subsequence* of S is another ordered list $S' = (s'_1, \ldots, s'_k)$ in which $s'_i <_S s'_{i+1}$ for every $i \in [k-1]$. Suppose that each element in S is distinct. Let X be the set underlying S. We call S a *linear ordering* of X, or simply an *ordering* for short. Abusing notation slightly, we may treat S as the set X whenever convenient (e.g. we may write $s \in S$ if s occurs in S).

3 Damage boosting

We first study the damage boosting mechanics when losing a life is forbidden. That is, the player can only take damage or refill health, without ever letting health drop to zero. An *event* is an opportunity to gain time by taking damage, and is represented by a pair $e = (d, t)$ where d is the damage to take to save t units of time. We will assume that d and t are integers, possibly negative. If both d and t are negative, we call e a *chicken event*, as d

represents a health refill and t the time lost to regain this health. We denote by $d(e)$ and $t(e)$ the damage and time-save components of e, respectively.

A *stage* $S = (e_1, \dots, e_n)$ is an ordered list of events. A *solution* $\hat{S} = (\hat{e}_1, \dots, \hat{e}_k)$ to a stage S is a subsequence of S. We say that we *take* event e_i if $e_i \in \hat{S}$. A given integer hp represents the player's *hit points* (HP) at the start of the stage. The player's hit points can never exceed hp. Each event $\hat{e}_i \in \hat{S}$ leaves the player with a number of hit points $h_{\hat{S}}(\hat{e}_i)$ after being taken. We define $h_{\hat{S}}(\hat{e}_1) = \min(hp, hp - d(\hat{e}_1))$ and, for $i \in \{2, \dots, k\}$, $h_{\hat{S}}(\hat{e}_i) = \min(hp, h_{\hat{S}}(\hat{e}_{i-1}) - d(\hat{e}_i))$. A solution is *valid* if $h_{\hat{S}}(\hat{e}_i) > 0$ for every $\hat{e}_i$ in $\hat{S}$.

Given a stage S and maximum hit points hp, the objective in the DAMAGE BOOSTING problem is to find a valid solution $\hat{S}$ for S that maximizes $t(\hat{S}) = \sum_{e \in \hat{S}} t(e)$.

As mentioned before, DAMAGE BOOSTING can be viewed as a knapsack instance in which each item is given in order, and we have some opportunities to (partially) empty the sack (which corresponds to our chicken events). It is not hard to show that the well-known pseudo-polynomial time algorithm and FPTAS for knapsack can be adapted to DAMAGE BOOSTING. The proof is essentially the same as in the knapsack FPTAS - we include it here for the sake of completeness.

▶ **Theorem 1.** *DAMAGE BOOSTING can be solved in pseudo-polynomial time $O(n^2 T)$, where T is the maximum time gain of an event. Moreover, DAMAGE BOOSTING admits an FPTAS, and can be approximated within a factor $1 - \epsilon$ in time $\mathcal{O}(n^3/\epsilon)$ for any $\epsilon > 0$.*

Proof. Let (S, hp) be an instance of DAMAGE BOOSTING, $S = (e_1, \dots, e_n)$. Let $H(i, t)$ denote the highest HP value achievable when gaining a time of exactly t by taking a subset of the events $\{e_1, \dots, e_i\}$. Define $H(i, t) = -\infty$ if this is not possible, and define $H(0, 0) = hp$ and $H(0, t) = -\infty$ for $t > 0$. Then

$$H(i, t) = \min \{hp, \max \{H(i-1, t), H(i-1, t - t(e_i)\} - d(e_i))\}$$

$H(i, t)$ needs to be computed for each $i \in [n]$ and each $t \in [nT]$. We then look at the maximum value of t such that $H(n, t) > 0$, which leads to a dynamic programming algorithm with the claimed complexity.

To get an FPTAS, we scale the time gains as in the knapsack FPTAS. Let $\epsilon > 0$ and let $c = \epsilon T/n$. Let $S' = (e'_1, \dots, e'_n)$, where $e'_i = (d(e_i), \lfloor t(e_i)/c \rfloor)$. Let $\hat{S}$ (resp. $\hat{S}'$) be a subsequence of S (resp. S') that maximizes the time gain $t(\hat{S})$ (resp. $t(\hat{S}')$). Observe that for each $e_i \in S$, we have $t(e_i)/c - 1 \le t(e'_i) \le t(e)/c$ whether e_i is a chicken event or not. Hence, $t(\hat{S}') \ge \sum_{e_i \in \hat{S}} t(e'_i) \ge t(\hat{S})/c - n$ (where the first inequality is due to the optimality of $\hat{S}'$ on S'). Note that $\hat{S}'$ is a valid solution for S, since the damage values were unchanged from S to S'. The time gained by taking the events of $\hat{S}'$ as our solution for S is

$$\sum_{e'_i \in \hat{S}'} t(e_i) \ge \sum_{e'_i \in \hat{S}'} c \cdot t(e'_i) \ge c \cdot (t(\hat{S})/c - n) = t(\hat{S}) - cn = t(\hat{S}) - \epsilon T \ge (1 - \epsilon) t(\hat{S})$$

where we use $t(\hat{S}) \ge T$ in the last inequality. The algorithm takes time $\mathcal{O}(n^2 T/(\epsilon T/n)) = \mathcal{O}(n^3/\epsilon)$. ◀

From the point of view of parameterized complexity, Theorem 1 implies that DAMAGE BOOSTING is FPT in t, the total time that can be gained (due to results of [8]). However t is typically high, and alternative parameterizations are needed. In the context of video games, although stages can be large, the number of types of enemies and damage sources is usually limited. Likewise, there are usually only a few healing items in a stage. The time gained or lost per event can vary widely though.

We would therefore like to parameterize DAMAGE BOOSTING by the number k of values that $d(e)$ can take in the events of S. It was shown in [15] that knapsack can be solved in time $\mathcal{O}(2^{2.5k \log k} poly(n))$, where k is the number of distinct weights that appear in the input. The algorithm does not seem to extend directly to DAMAGE BOOSTING, and we leave the FPT status of the problem open for k. We do show, however, that if the number of chicken events is also bounded by some integer r, one can devise an FPT algorithm in $k + r$ based on the ideas of [15]. We make use of the result of Lokshtanov [23, Theorem 2.8.2], which improve upon Kannan's algorithm [21] and state that a solution to an Integer Linear Program (ILP) with ℓ variables can be found in time $\mathcal{O}(\ell^{2.5\ell} poly(n))$.

▶ **Theorem 2.** *DAMAGE BOOSTING is FPT in $k + r$, where k is the number of possible damage values and r the number of chicken events. Moreover, an optimal solution can be found in time $\mathcal{O}(2^r(2k(r+1)+r)^{2.5(2k(r+1)+r)} poly(n))$.*

Proof. Let C be the set of chicken events of S, and suppose $r = |C|$. We simply "guess" which of the 2^r subsets of C to take. That is, for each subset $C' \subseteq C$, we find the maximum time gain achievable under the condition that the chicken events taken are exactly C', hence the 2^r factor in the complexity. For the rest of the proof, assume $C = \{c_0, c_1, \ldots, c_r, c_{r+1}\}$ is a set of chicken events such that $c_i <_S c_{i+1}$ for $0 \leq i \leq r$, each of which must be taken. For notational convenience, we have added chicken $c_0 = c_{r+1} = (0, 0)$, where c_0 (respectively c_{r+1}) is a chicken event that occurs before (resp. after) every event of S.

Let $d_1, \ldots, d_k$ be the possible damage values. For $i \in [k]$ and $j \in \{0, \ldots, r\}$, let n_{ij} be the number of events of damage value d_i that occur after chicken c_j, but before chicken c_{j+1}. Note that to obtain a solution, it suffices to know how many events of damage value d_i we take for each i and j. That is, let $(e_{ij}^1, \ldots, e_{ij}^{n_{ij}})$ be the events of damage value d_i that occur between chickens c_j and c_{j+1} in S, sorted in non-increasing order of time gain. If we know that, say, $x_{ij} \in \{0, \ldots, n_{ij}\}$ events of damage value d_i must be taken between chickens c_j and c_{j+1}, then we simply take the first x_{ij} events of maximum time gain, i.e. $e_{ij}^1, \ldots, e_{ij}^{x_{ij}}$. The time gain with respect to x_{ij} is $f_{ij}(x_{ij}) := \sum_{h=1}^{x_{ij}} t(e_{ij}^h)$.

This lets us formulate an ILP with at most $2(r+1)k + r$ variables. For each $j \in [r]$, a variable h_j represents the player's HP right after taking the j-th chicken. We add the constant $h_0 := hp$ for convenience. For $i \in [k], j \in \{0, \ldots, r\}$, there is a variable x_{ij} for the number of events of damage value d_i to take between chicken c_j and c_{j+1}, and a variable g_{ij} for the time gained by events of damage value d_i within this range. The ILP is the following.

$$\begin{array}{lll}
\text{maximize} & \displaystyle\sum_{i=1}^{k}\sum_{j=0}^{r} g_{ij} & \\
\text{subject to} & h_{j+1} \leq h_j - \displaystyle\sum_{i=1}^{k} x_{ij} d_i - d(c_{j+1}) & j \in \{0, \ldots, r-1\} \\
& h_j \leq hp & j \in \{1, \ldots, r\} \\
& h_j - \displaystyle\sum_{i=1}^{k} x_{ij} d_i > 0 & j \in \{0, \ldots, r\} \\
& g_{ij} \leq f_{ij}(x_{ij}) & i \in [k], j \in \{0, \ldots, r\} \\
& h_j \in \mathbb{N} & j \in \{1, \ldots, r\} \\
& x_{ij} \in \{0, \ldots, n_{ij}\}, g_{ij} \in \mathbb{N} & i \in [k], j \in \{0, \ldots, r\}
\end{array}$$

The first constraint ensures that the player's HP after taking chicken c_{j+1} never exceeds the HP after taking chicken c_j and taking the damage boosts in-between. The second constraint ensures that we do not exceed the maximum hit points. The third constraint ensures that the player never dies. The fourth constraint bounds the total time gained by the damage boosts taken. The correctness of the above ILP is then straightforward to verify. The functions $f_{ij}(x_{ij})$ are however not guaranteed to be linear. But they are convex since they consist of the partial sums of a non-increasing sequence of integers. The authors of [15] have shown that the constraint $g_{ij} \leq f_{ij}(x_{ij})$ can easily be replaced (in polynomial time) by a set of linear constraints $g_{ij} \leq p_{ij}^{(\ell)}(x_{ij})$, for $\ell \in [n_{ij}]$. We refer the reader to [15, Lemma 2] for more details. The complexity follows from the aforementioned result of [23]. ◀

Damage boosting with lives

In the rest of this section, we consider the *death abuse* speedrunning strategy. In most games, when the player reaches 0 hit points, a life is lost and the player restarts with full health at the last predefined revival location traversed. We call such a location a *checkpoint*. The game is over once the player does not have any lives remaining. Death abusing is a common way of replenishing health, at the cost of having to re-traverse the portion of the stage from the last checkpoint to the location of death.

We modify the DAMAGE BOOSTING problem to incorporate death abuse as follows. A DAMAGE BOOSTING WITH LIVES instance is a 5-tuple (S, hp, ℓ, C, p) where $S = (e_1, \ldots, e_n)$ is a sequence of events, hp is the maximum hit points, ℓ is the starting number of lives, $C \subseteq \{e_1, \ldots, e_n\}$ is the set of checkpoints and $p : \{e_1, \ldots, e_n\} \to \mathbb{N}$ is the *death penalty*, where $p(e_i)$ is the time lost by dying at event e_i and having to re-do the stage from the last checkpoint to e_i. For an event $e_i \in S$, let $c(e_i) \in C$ be the latest checkpoint of S that occurs before e_i. If the player reaches 0 HP at event e_i, the game restarts right before event $c(e_i)$ (so that taking the event $c(e_i)$ is possible after dying). If $c(e_i) = e_j$, we assume that $p(e_i) \geq \sum_{h=j}^{i} t(e_h)$, as otherwise it might be possible to gain time by dying.

A solution to S is a list of $k \leq \ell$ event subsequences $(S_1, \ldots, S_k)$ that describes the events taken in each life used by the player. For $i \in [k-1]$, the i-th life of the player must end exactly after taking the last event of S_i. That is, $S_i = (e_1^i, \ldots, e_r^i)$ must satisfy $h_{S_i}(e_j^i) > 0$ for each $j \in [r-1]$ and $h_{S_i}(e_r^i) \leq 0$. As for S_k, it must simply be valid, since the player's hit points can never go below 0 in the last life. Finally, we require that for $i \in \{2, \ldots, k\}$, S_i starts at the checkpoint assigned to the event at which the player died in S_{i-1}. In other words, the first event of S_i must occur after the appropriate checkpoint, so that $c(tail(S_{i-1})) \leq_S head(S_i)$.

For $i < k$, the time gained $t(S_i)$ at life S_i is defined as before, except that $t(tail(S_i))$ is replaced by the penalty $p(tail(S_i))$ of dying at the last event. That is, $t(S_i) = \sum_{e \in S_i} t(e) - t(tail(S_i)) - p(tail(S_i))$. Our objective is to find a solution $(S_1, \ldots, S_k)$ to S that maximizes $\sum_{i \in [k-1]} t(S_i) + \sum_{e \in S_k} t(e)$.

In this section, we show that having even only one life to spare removes the possibility of having a PTAS for DAMAGE BOOSTING WITH LIVES (unless P = NP). Despite this, we show that the problem still admits a pseudo-polynomial time algorithm. Beforehand, we state a simple approximabiltiy result.

▶ **Proposition 3.** *For any $\epsilon > 0$, DAMAGE BOOSTING WITH LIVES can be approximated within a factor $\frac{1}{\ell} - \epsilon$ in time $\mathcal{O}(n^3/\epsilon)$.*

Proof. Let (S, hp, ℓ, C, p) be a given instance of DAMAGE BOOSTING WITH LIVES, and let t be the maximum time gain achievable in stage S without losing a single life. By Theorem 1, t can be approximated within a factor $1 - \epsilon$ for any $\epsilon > 0$. Now, each life of the player can be used to gain at most t time, implying that at most ℓt time can be gained. The Lemma follows, since $(1-\epsilon)t \geq \frac{1-\epsilon}{\ell} \cdot \ell t \geq (\frac{1}{\ell} - \epsilon)\ell t$. ◀

We then present our inapproximability result. Note that this implies that the above approximation is tight in the case that the player has two lives.

▶ **Theorem 4.** *DAMAGE BOOSTING WITH LIVES is hard to approximate within a factor* $1/2$*, even if the player has two lives, and there is no chicken event.*

Proof. We show that having an algorithm with approximation factor $1/2$ or better would allow solving SUBSET SUM in polynomial time. Let (B, s) be a SUBSET SUM instance, with $B = \{b_1, \dots, b_n\}$ a (multi)-set of n positive integers and s the target sum. Define a DAMAGE BOOSTING WITH LIVES instance (S, hp, ℓ, C, p) as follows. Put $hp = s + 1$ and $\ell = 2$. Also let $S = (e_1, \dots, e_n, x, y)$. Here the e_i events correspond to the b_i integers, and x and y are two additional special events. For each $i \in [n]$, put $e_i = (b_i, b_i)$, and put $x = (1, 0)$, $y = (s, s-1)$. Set x as the only checkpoint, i.e. $C = \{x\}$. The only relevant death penalties are $p(x) = 0$ and $p(y) = 10s$. We show that if (B, s) is a YES instance, then it is possible to gain a total of $2s - 1$ time units, and if (B, s) is a NO instance, then at most $s - 1$ time units can be gained.

Suppose that (B, s) is a YES instance, and that there is a subset $B' = \{b_{i_1}, \dots, b_{i_k}\}$ of B whose elements sum to s. Then the player can take the damage boosts $e_{i_1}, \dots, e_{i_k}$ before arriving at x. At this point, s time units have been gained and s damage has been taken. Hence there is only 1 HP remaining. The player can take the 1 damage at event x, lose a life, and reappear at event x at full health with no time penalty. With this new life, the player then skips x, and takes the y damage boost, saving an additional $s - 1$ time units. The total time gain is $2s - 1$.

Now suppose that (B, s) is a NO instance. Assume that the player uses an optimal strategy on the constructed DAMAGE BOOSTING WITH LIVES instance. Consider the situation when the player arrives at event x, before deciding whether to take it (for the first time, if more than one). Let h_x and t_x be the remaining HP of the player and the time gained at this point, respectively. Observe that since all the e_i events have equal damage and time gain, we have $h_x = hp - t_x$. We must have $t_x \neq s$, since otherwise the events taken so far would provide a solution to the SUBSET SUM instance. Moreover, we cannot have $t_x > s$, since otherwise $h_x = hp - t_x = s + 1 - t_x \leq 0$, i.e. the player would have died before event x, and would have restarted at the beginning of the stage. Thus, $t_x < s$, and therefore $h_x > 1$. Since only 1 HP can be lost at event x, the player cannot die at event x. Thus the player arrives at y with a time gain of at most $s - 1$. Note that there is no point in dying at the y event, as the time lost is too high. Moreover, the only way the player can gain time from the y event is by being at full health. Since the player did not die at event x and there is no chicken, full health is only possible if the player has taken no damage boost before getting to y. It follows that there are then only two possibilities: if the player takes some events prior to x, he can save at most $t_x < s$ time units, and otherwise, he can skip every damage boost prior to x and save $s - 1$ time units by taking event y. We conclude that the time gain is at most $s - 1$.

Now, observe that if there is a factor $1/2$ approximation algorithm, it returns a time gain of at least $(2s - 1)/2 = s - 1/2$ on YES instances, and a time gain of at most $s - 1$ on NO instances. This gap can be used to distinguish between YES and NO instances. ◀

We do not know whether there exists a constant-factor approximation algorithm for DAMAGE BOOSTING WITH LIVES that holds for all values of ℓ. However, the problem does admit a pseudo-polynomial time algorithm. The dynamic programming is not as straightforward as the one for the knapsack, since the player can die and come back at checkpoints. The idea is to optimize the first life for each possible death, then the second life depending on the first, and so on.

▶ **Theorem 5.** *DAMAGE BOOSTING WITH LIVES can be solved in time $\mathcal{O}(n^2 \cdot hp^2 \cdot \ell)$.*

Proof. Let (S, hp, ℓ, C, p) be a given DAMAGE BOOSTING WITH LIVES instance, with $S = (e_1, \ldots, e_n)$. For simplicity, we assume that $e_1 = (0, 0)$. Denote by $T(i, h, l)$ the maximum time gain that can be achieved by exiting event e_i (i.e. after deciding whether to take it or not) with exactly h hit points and l lives. Note that event e_i might have been visited in a previous life. Define $T(i, h, l) = -\infty$ if $h > hp$, $h \leq 0$, $l > \ell$ or $l \leq 0$. Our goal is to compute $\max_{1 \leq h \leq hp, 1 \leq l \leq \ell} T(n, h, l)$.

For $i = 1$, set $T(1, hp, \ell) = 0$ and $T(1, h, l) = -\infty$ whenever $h \neq hp$ or $l \neq \ell$ (we assume that we will never return to e_1 by losing a life, as this would be pointless).

For $i > 1$ such that $e_i \notin C$, note that we can only enter e_i through e_{i-1} with the same number of lives. If e_i is not a chicken event, we thus have

$$T(i, h, l) = \max \{T(i-1, h, l), T(i-1, h + d(e_i), l) + t(e_i)\}$$

(observe that invalid values of $h + d(e_i)$ yield a time gain of $-\infty$)

If e_i is a chicken event, the above recurrence applies unless taking event e_i would refill the player's health above hp. Thus $T(i, hp, l)$ is a special case, which we handle as follows (recall that $d(e_i)$ and $t(e_i)$ are now negative):

$$T(i, hp, l) = \max \left\{ T(i-1, hp, l), \max_{hp + d(e_i) \leq d \leq hp} \{T(i-1, hp-d, l)\} + t(e_i) \right\}$$

Now suppose that $i > 1$ is such that $e_i \in C$. We can either enter e_i through e_{i-1} with the same number of lives, or through some e_j with $j > i$ by dying while having $l + 1$ lives. In the latter case, we must enter e_i with health equal to hp. Therefore, if $h \notin \{hp, hp - d(e_i)\}$, it is impossible to enter e_i by dying and exiting with exactly h hit points. Hence, if $h \notin \{hp, hp - d(e_i)\}$, the above recurrence from the $i > 1$ case applies. Moreover, if $l = \ell$, the player cannot have died yet and the same recurrence also applies. Assume that $h \in \{hp, hp - d(e_i)\}$ and $l < \ell$. We must compute a temporary value for $T(i, h, l)$. Let e_k be the latest event that leads to checkpoint e_i upon death. That is, $c(e_k) = e_i$ but either $k = n$ or $c(e_{k+1}) \neq e_i$. Define

$$D_{i,l} = \max_{i \leq j \leq k} \left\{ \max_{h' \leq d(e_j)} \{T(j, h', l+1) - p(e_j)\} \right\}$$

which is the maximum time gain achievable by losing the player's $(l+1)$-th life and respawning at e_i. Then it follows that

$$T(i, hp, l) = \max\{T(i-1, hp, l), D_{i,l}\}$$

if e_i is not a chicken event. If e_i is a chicken event, then similarly as we did above,

$$T(i, hp, l) = \max \left\{ T(i-1, hp, l), \max_{hp + d(e_i) \leq d \leq hp} \{T(i-1, hp-d, l)\} + t(e_i), D_{i,l} \right\}$$

Finally, for the case $h = hp - d(e_i)$, we have

$$T(i, hp - d(e_i), l) = \max\{T(i-1, hp - d(e_i), l), D_{i,l} + t(e_i)\}$$

Note that to compute $T(i, h, l)$, one only needs values of $T(j, h', l')$ with either $j < i$ and $l' = l$, or with $l' = l + 1$. It is not difficult to see that one can compute the $T(i, h, l)$ values in decreasing order of values of l, starting at $l = \ell$, and in increasing order of i. Each $T(i, h, l)$ value depends on at most $\mathcal{O}(n \cdot hp)$ values. There are $(n + 2) \cdot |hp| \cdot |\ell|$ possible $T(i, h, l)$ values, resulting in a $\mathcal{O}(n^2 \cdot (hp)^2 \cdot \ell)$ time algorithm. ◀

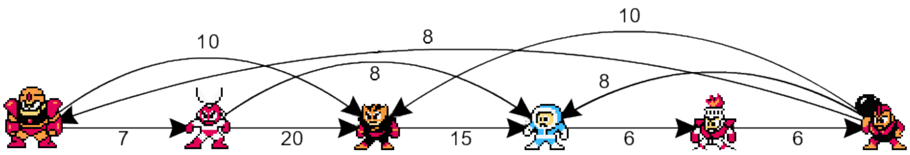


Figure 4 The dependency graph for the game Mega Man (with approximate time gains in seconds according to [1]), where the only event considered is defeating the boss.

4 Routing

We now turn to the problem of *routing*, in which the player may visit a set of locations or stages in any order. Clearing a stage yields a new weapon to the player. Each stage has a set of time-saving events, and each weapon can be used to gain some amount of time in an event. The time saved on an event depends on the best weapon available. Figure 4 represents this notion in Mega Man as a weighted directed graph. For instance, defeating Guts Man first (far left) allows saving 7 seconds against Cut Man (second), and 8 seconds could be gained by defeating Bomb Man (far right) before Guts Man.

In this section, a *game* is a set of stages $\mathbb{S} = \{S_1, \ldots, S_n\}$. A *stage* $S_i = \{e_1, \ldots, e_k\}$ is a set of *events*, where here an event $e_j : \mathbb{S} \to \mathbb{N}$ is a function mapping each stage to an integer. The event e_j is interpreted as follows: if stage S_i is cleared, then a time of $e_j(S_i)$ can be saved while going through e_j using the weapon gained from S_i. Let $C \subseteq \mathbb{S}$ and let e be an event. We will write $e(C) = \max_{S \in C} e(S)$. That is, if C is the set of cleared stages, we will assume that event e will be cleared using the best option available. Given C, the time gained in a stage S becomes $t(S, C) := \sum_{e \in S} e(C)$.

In the ROUTING problem, we are given a set of stages $\mathbb{S} = \{S_1, \ldots, S_n\}$. The objective is to find a linear ordering π of $\mathbb{S}$ such that $\sum_{i \in [n]} t(S_i, \{S_j : S_j <_\pi S_i\})$ is maximum. Later on, we shall consider the minimization version of ROUTING.

We define the notion of a *dependency digraph* $D(\mathbb{S})$ for a set of stages $\mathbb{S}$. The digraph $D(\mathbb{S}) = (\mathbb{S}, A, w)$ has one vertex for each stage, and for every ordered pair (i, j), an arc from S_i to S_j of weight $w(S_i, S_j) = \sum_{e \in S_j} e(S_i)$. The *underlying undirected graph* of $D(\mathbb{S})$ is the graph obtained by removing the arcs of weight 0, ignoring the other weights and the direction of the arcs.

We start with two easy special cases. The first case is when each stage contains only one event, which could for example correspond to the case in which we only consider the fastest way to defeat all bosses. This reduces to finding a *maximum weight branching* in $D(\mathbb{S})$, where a branching of a digraph D is an acyclic subdigraph of D in which every vertex has in-degree 0 or 1. The second case is when each event depends on only one stage. The Routing problem then becomes equivalent to finding a maximum weight directed acyclic sub-digraph of $D(\mathbb{S})$. This is the *maximum weight sub-DAG* problem, the maximization version of the feedback arc set problem.

▶ **Theorem 6.** *The following properties of Routing hold:*

1. *If each stage contains a single event, ROUTING can be solved in time $\mathcal{O}(|A| + |\mathbb{S}| \log |\mathbb{S}|)$.*
2. *If, for each event e, there is only one $S_i \in \mathbb{S}$ such that $e(S_i) > 0$, then ROUTING is equivalent to the maximum weight sub-DAG problem on $D(\mathbb{S})$.*

Proof. (1) For a stage S_i, denote by e_i the single event of S_i. Given an ordering π of $\mathbb{S}$ and a stage $S_i \neq tail(\pi)$, denote by $p_\pi(S_i)$ the stage prior to S_i that allows a maximum time gain on e_i, breaking ties arbitrarily. That is, $p_\pi(S_i) = \arg\max_{S_j <_\pi S_i} e_i(S_j)$.

Observe that for any ordering π and any stage $S_i \neq tail(\pi)$, because S_i has only one event there is at most one stage prior to S_i that can be useful to clear it, namely $p_\pi(S_i)$. Recalling that $D(\mathbb{S}) = (\mathbb{S}, A, w)$, the time gain for a given π is $t = \sum_{S_i \in \mathbb{S}} w(p_\pi(S_i), S_i)$. Consider the set of arcs $A' = \{(p_\pi(S_i), S_i) : 1 < i \leq n \text{ and } e_i(p_\pi(S_i)) > 0\}$. Then the subdigraph of $D(\mathbb{S})$ formed by the arc set A' contains no directed cycle, and each vertex has at most one incoming arc, with the exception of the first the vertex of π which has none. Thus A' forms a branching, and its weight is t. Conversely, let B be a branching of $D(\mathbb{S})$ with arc set A'. Then it is not hard to see that B can be converted to an ordering π of $\mathbb{S}$ such that the total time gained is $\sum_{(u,v) \in A'} w(u, v)$. Indeed, as B is acyclic, a topological sorting of B yield a linear ordering of $\mathbb{S}$ in which each event e_{S_i} can be completed using the in-neighbor of S_i in A' (if any). A maximum weight branching can be found in time $\mathcal{O}(|A| + |\mathbb{S}| \log |\mathbb{S}|)$ by reduction to the maximum weight spanning arborescence problem (see e.g. [10, Chapter 6]), and using Gabow & al.'s algorithm [18].

(2) If every event depends on exactly one stage, we show that ROUTING and maximum weight sub-DAG reduce to one another with the same optimality value. We start by reducing ROUTING to maximum weight sub-DAG. Consider the $D(\mathbb{S}) = (\mathbb{S}, A, w)$ digraph. Because each $e \in S_j$ depends only on one stage, $w(S_i, S_j)$ corresponds exactly to the time gain contribution of S_i to stage S_j if S_i is completed before S_j (which might not be the case if an event could be completed by more than one stage). Thus given an ordering π of $\mathbb{S}$, the total time gain is $t = \sum_{S_i <_\pi S_j} w(S_i, S_j)$ (where $w(S_i, S_j) = 0$ if $(S_i, S_j) \notin A$). Moreover, the arcs $\{(S_i, S_j) \in A : S_i <_\pi S_j\}$ cannot form a cycle in $D(\mathbb{S})$. It follows that an ordering π of time gain t can be used to find a sub-DAG of $D(\mathbb{S})$ of weight t. Conversely, a topological sorting of a sub-DAG of $D(\mathbb{S})$ with total weight t gives an ordering of the stages with total time gain t.

The reduction from the maximum weight sub-DAG problem to the routing problem goes along the same lines. Given a maximum weight sub-DAG instance $H = (V, A, w)$, it suffices to create a stage S_u for each $u \in V$, and add one event e^u_v in S_u for each v such that $(v, u) \in A$. We put $e^u_v(S_v) = w(v, u)$. It is easy to see that a total time of t can be gained if and only if H has a sub-DAG of weight t. ◀

The above implies that every known hardness result for the maximum weight sub-DAG problem transfers to ROUTING. In particular, ROUTING is NP-hard even if the maximum degree of the $D(\mathbb{S})$ is 4 (this follows from the hardness of vertex cover in cubic graphs [3]). Also, the maximum weight sub-DAG problem cannot be approximated within a ratio better than 1/2, assuming the Unique Games Conjecture [19]. On the positive side, it is trivial to attain this bound, just as in the maximum weight sub-DAG problem: take any ordering π. Either π or its reverse will attain 1/2 of the maximum possible time save.

▶ **Proposition 7.** *ROUTING admits a factor* 1/2 *approximation algorithm.*

Proof. Note that $\sum_{i \in [n]} \sum_{e \in S_i} e(\mathbb{S})$ is an obvious upper bound on the maximum time gain achievable. Pick a random ordering $(S_1, \ldots, S_n)$ of $\mathbb{S}$, and let $(S_n, \ldots, S_1)$ be the reverse ordering. One of these two must achieve a time gain of $e(\mathbb{S})$ for at least half the events e that are in $\mathbb{S}$. ◀

Minimizing time loss

We now turn to the minimization version of the ROUTING problem. That is, consider the upper bound $\mu := \sum_{i \in [n]} \sum_{e \in S_i} e(\mathbb{S})$ on the possible time gain. Ideally, one would like to get as close as possible to μ, which amounts to finding a time gain t that minimizes $\mu - t$. Given an ordering π of $\mathbb{S}$, denote by $cost(\mathbb{S}, \pi) := \mu - \sum_{i \in [n]} t(S_i, \{S_j : S_j <_\pi S_i\})$.

We define the MIN-ROUTING-LOSS as follows: given a set of n stages $\mathbb{S}$, find a linear ordering π of $\mathbb{S}$ that minimizes $cost(\mathbb{S}, \pi)$.

By Theorem 6, this is at least as hard as the *feedback arc set* (FAS) problem, where the goal is to delete a set of arcs of minimum weight from a digraph to obtain a DAG (these deletions correspond to time losses in $D(\mathbb{S})$). FAS is APX-hard [20], but determining if there is a constant factor approximation appears to be open. A factor $\mathcal{O}(\log n \log \log n)$ approximation algorithm is presented in [16], but does not appear to apply to MIN-ROUTING-LOSS.

We will show that MIN-ROUTING-LOSS cannot be approximated with a ratio better than $\mathcal{O}(\log n)$. As for parameterized complexity, FAS is known to be FPT in k, the weight of the edges to remove (assuming weights in $poly(n)$) [9]. FAS is also known to be FPT in the treewidth of the underlying undirected graph [6]. As we show here, both parameters are not applicable to MIN-ROUTING-LOSS.

▶ **Theorem 8.** *MIN-ROUTING-LOSS is W[2]-hard with respect to the time loss k and hard to approximate within a factor $\mathcal{O}(\log n)$. This holds even on instances in which the underlying undirected graph of $D(\mathbb{S})$ is a tree and only one stage has more than one event.*

Proof. We reduce from DOMINATING SET, which is known to be W[2]-hard for parameter k, the number of vertices in the dominating set [14]. Let (G, k) be an instance of DOMINATING SET. Denote $V(G) = \{v_1, \ldots, v_n\}$. Create a set of stages $\mathbb{S} = \{S_1, \ldots, S_n, X\}$. For $i \in [n]$, stage S_i has only one event e_i, whereas X has n events $\{x_1, \ldots, x_n\}$. For each edge $v_i v_j \in E(G)$, set $x_j(S_i)$ very high, say $x_j(S_i) = kn^{10}$. Also set $x_j(S_j) = kn^{10}$ for all $j \in [n]$. Then for each $i \in [n]$, set $e_i(X) = 1$. All other event completion times are set to 0. Note that the upper time bound on $\mathbb{S}$ is $\mu = n + (kn^{10})n$. We show that G has a dominating set of size at most k if and only if a time gain of at least $\mu - k$ is possible.

Let $B = \{v_{i_1}, \ldots, v_{i_k}\}$ be a dominating set of G of size k, and denote $\{v_{i_{k+1}}, \ldots, v_{i_n}\} = V(G) \setminus B$. Order the stages of $\mathbb{S}$ as follows: $\pi = (S_{i_1}, \ldots, S_{i_k}, X, S_{i_{k+1}}, \ldots, S_{i_n})$. For any $x_j \in X$, either $v_j \in B$ or there is some $v_i \in B$ such that $v_i v_j \in E(G)$. Since one of $S_j <_\pi X$ or $S_i <_\pi X$ holds, event x_j can be cleared with time gain kn^{10}. Also, every event in $S_{i_{k+1}}, \ldots, S_{i_n}$ can be cleared with a time gain 1 using stage X. Only the events in stages $S_{i_1}, \ldots, S_{i_k}$ do not yield a time gain, and the total time gain is therefore $\mu - k$.

Conversely, suppose that there is an ordering π of $\mathbb{S}$ that achieves a time gain of at least $\mu - k$. For this to be possible, every event of X must be cleared with a time gain kn^{10}. Consider the set $B = \{S_{i_1}, \ldots, S_{i_h}\}$ that precedes X in π. None of these stages can yield a time gain, which implies $h \leq k$. Moreover, B must be a dominating set, for if not, there is an event $x_j \in X$ that cannot be cleared with a time gain of kn^{10}.

As for the inapproximability result, DOMINATING SET is hard to approximate within a factor $\mathcal{O}(\log n)$ (see [4]). It is not hard to see that the above reduction is approximation preserving: from a dominating set of size k, one can obtain a time loss of at most k and vice-versa. As the number of stages in $\mathbb{S}$ is $n+1$, the $\mathcal{O}(\log n)$ inapproximability follows. ◀

Observe that in addition to treewidth, the number of stages with more than one event is also not an option for parameterization, as well as the maximum degree of $D(\mathbb{S})$ (due to Theorem 6 and the remark after). In the rest of this section, we show that Routing is FPT when combining the treewidth and maximum in-degree parameters.

Parameterization by treewidth and maximum in-degree

In this section, we assume that the in-degree of a vertex in $D(\mathbb{S})$ is bounded by d and the treewidth of the underlying undirected graph of $D(\mathbb{S})$ is bounded by t. We devise a more or less standard dynamic programming algorithm on the tree decomposition of $D(\mathbb{S})$. We introduce the essential notions here, and refer the reader to [14, 12] for more details.

A *tree decomposition* of a graph $G = (V, E)$ is a tree T in which each node x is associated with a *bag* $B_x \subseteq V$ such that $\bigcup_{x \in V} B_x = V$. Moreover, the two following properties must hold: (1) for any $uv \in E$, there is some $x \in V(T)$ such that $u, v \in B_x$, and (2) for any $v \in V$, the set $\{x \in V(T) : v \in B_x\}$ induces a connected component of T. The *width* of T is the size of the largest bag of T minus 1, and the *treewidth* of G is the minimum width of a tree decomposition of G.

A tree decomposition T for G is *nice* if each $x \in V(T)$ is of one of the following types:

- *Leaf node*: x is a leaf of T and $B_x = \emptyset$.
- *Introduce node*: x has exactly one child y and $B_x = B_y \cup \{v\}$ for some $v \in V(G)$.
- *Forget node*: x has exactly one child y and $B_x = B_y \setminus \{v\}$ for some $v \in V(G)$.
- *Join node*: x has exactly two children y, z and $B_x = B_y = B_z$.

We also assume that T is rooted at a vertex r such that $B_r = \emptyset$. The root defines the ancestor/descendant relationship between nodes of T. It is well-known that a nice tree decomposition T' of width t can be constructed from a tree decomposition T of width t in polynomial time (see [12, 14]).

The routing algorithm

Assume that we have constructed a nice tree decomposition T from $D(\mathbb{S}) = (V, A, w)$. For convenience, we shall treat stages of $\mathbb{S}$ as vertices (hence, each $v \in V$ is a set of events). For $v \in V$, denote by $N^-(v) = \{u \in V : (u, v) \in A\}$ and $N^-[v] = N^-(v) \cup \{v\}$. Under our assumptions, $|N^-(v)| \leq d$ for all $v \in V$. Roughly speaking, at each node $x \in V(T)$, we would like to "try" each ordering of B_x and compute a time cost for each stage $v \in B_x$ based on the children of x. This is essentially the idea in the bounded treewidth FPT algorithm for feedback arc set [6]. This however does not work directly, as the cost of a stage $v \in B_x$ depends on $N^-(v)$, which may or may not be included in B_x. To solve this problem, we also include all the in-neighbors of the stages in B_x in the set of orderings to consider. One way to do this would be to consider all orderings of $\bigcup_{v \in B_x} N^-[v]$ at every bag B_x and assign a cost to every vertex in B_x or in a bag below. This would lead to a relatively simple $\mathcal{O}((dt)!poly(n))$ algorithm. However, this complexity can be improved (at the expense of more technicality) by considering, instead of every permutation of $\bigcup_{v \in B_x} N^-[v]$, only the subsets of $N^-(v)$ that occur before v for each $v \in B_x$.

To formalize this notion, let $P = \{\pi_1, \ldots, \pi_s\}$ be a set of orderings of (possible different) subsets of V. We say that P is *realizable* if there exists an ordering π of V such that for each $i \in [s]$, $u <_{\pi_i} v$ implies $u <_\pi v$. We then say that π *realizes* P (or for short, π realizes π' if $P = \{\pi'\}$). Note that the existence of π can be verified in polynomial time.

Let V_x be the subset of vertices of V appearing in the bags under x, i.e. $v \in V_x$ if and only if x has a descendant y such that $v \in B_y$ (noting that x is a descendant of itself). For $x \in V(T)$, we denote by $\Pi(x)$ the set of all $|B_x|!$ possible orderings of B_x (with $\Pi(x) = \{()\}$ if $B_x = \emptyset$). Denote by $\Lambda(x)$ the set of all combinations of subsets of in-neighbors of vertices in B_x. That is, if $B_x = \{v_1, \ldots, v_s\}$ with $1 \leq s \leq t$, then

$$\Lambda(x) = \mathcal{P}(N^-(v_1)) \times \ldots \times \mathcal{P}(N^-(v_s))$$

where $\mathcal{P}(X)$ denotes the powerset of X. Let $\Lambda(x) = \{()\}$ contain the empty sequence if $B_x = \emptyset$. Observe that $|\Lambda(x)| = \mathcal{O}(2^{dt})$. For $P_x = (P_1, \ldots, P_s) \in \Lambda(x)$, we interpret P_i as "all elements of P_i occur before v_i, and those of $N^-(v_i) \setminus P_i$ occur after v_i". We thus denote the set of two-elements orderings implied by P_x by

$$s(P_x) = \bigcup_{v_i \in B_x} \{(u, v_i) : u \in P_i\} \cup \{(v_i, u) : u \in N^-(v_i) \setminus P_i\}$$

We now define a time cost $D(x, \mu_x, P_x)$ over all $x \in V(T)$, $\mu_x \in \Pi(x)$ and $P_x = (P_1, \ldots, P_s) \in \Lambda(x)$. Given an ordering π of V and $v \in V$, let $cost(v, \pi) = \sum_{e \in v_i} (e(V) - e(\{u : u <_\pi v\}))$ be the time lost in stage v. Let $V_x' = V_x \cup \bigcup_{v \in B_x} N^-(v)$. Then

$$D(x, \mu_x, P_x) := \min\{\sum_{v \in V_x} cost(v, \pi) : \pi \text{ is an ordering of } V_x' \text{ that realizes } \{\mu_x\} \cup s(P_x)\}$$

In words, $D(x, \mu_x, P_x)$ is the minimum cost for the set of stages in V_x in an ordering of V_x', with the obligation of using the partial orderings prescribed by μ_x and P_x. If r is the root of T, our goal is to compute $D(r, (), ())$ (recall that $B_r = \emptyset$). For $v \in V$ and $P \subseteq N^-(v)$, let $cost(v_i, P) = \sum_{e \in v_i} (e(V) - e(P))$ the time lost at stage v_i if precisely the elements of P occur before v. We claim that $D(x, \mu_x, P_x)$ can be computed as follows.

- If x is a leaf node, then V_x and B_x are empty and we simply set $D(x, (), ()) = 0$;
- If x is an introduce node with child y, let v_i be the new node in B_x and P_i be the subset of $N^-(v_i)$ present in P_x. Then

$$D(x, \mu_x, P_x) = \min\{D(y, \mu_y, P_y) : \mu_y \in \Pi(y), P_y \in \Lambda(y) \text{ and } s(P_x) \cup s(P_y) \cup \{\mu_x, \mu_y\} \text{ is realizable}\} + cost(v_i, P_i)$$

- If x is a forget node with child y, then

$$D(x, \mu_x, P_x) = \min\{D(y, \mu_y, P_y) : \mu_y \in \Pi(y), P_y \in \Lambda(y) \text{ and } s(P_x) \cup s(P_y) \cup \{\mu_x, \mu_y\} \text{ is realizable}\}$$

- If x is a join node with children y and z, then

$$D(x, \mu_x, P_x) = D(y, \mu_x, P_x) + D(z, \mu_x, P_x) - \sum_{v_i \in B_x} cost(v_i, P_i)$$

  where P_i is the ordering of P_x for $N^-[v_i]$, for each $v_i \in B_x$.

The above yield the following result. The main difficulty is to show that an ordering at node x can be obtained from the ordering of its child/children.

▶ **Theorem 9.** *ROUTING can be solved in time $\mathcal{O}(2^{t(d+\log t)}(2^d + md) \cdot nt)$, where m is the maximum number of events in a stage.*

Proof. We prove the complexity first, then proceed with the correctness of the dynamic programming recurrences. There are $\mathcal{O}(nt!2^{dt}) = \mathcal{O}(n2^{t \log t}2^{dt}) = \mathcal{O}(n2^{t(d+\log t)})$ possible x, μ_x and P_x combinations for the values of $D(x, \mu_x, P_x)$. To compute a specific $D(x, \mu_x, P_x)$, in the worst case we need to consider all the possible $D(y, \mu_y, P_y)$ values for the child y of x, in the case of introduce and forget nodes. However in these situations, μ_x and μ_y differ only by one element, and so given μ_x, there are only $\mathcal{O}(t)$ orderings of B_y such that $\{\mu_x, \mu_y\}$ are realizable. Similarly, $s(P_x) \cup s(P_y)$ are realizable only if they have the same sets of in-neighbors for each $v \in B_x \cap B_y$. Therefore, only one subset of $N^-(v_i)$ can differ between P_x and P_y, where here v_i is the introduced or forgotten vertex. It follows that only $\mathcal{O}(t2^d)$ combinations of μ_y and P_y need to be checked from $D(y, \mu_y, P_x)$. One still needs to check whether $\mu_x, \mu_y, s(P_x)$ and $s(P_y)$ are realizable. This can easily be done in time $O(n)$, for instance by constructing the directed graph on vertex set V and adding an arc from u to v whenever u is immediately before v in an ordering of $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$. This graph has $O(td)$ arcs and it suffices to check that it is acyclic. In the case of introduce nodes, the value of $cost(v_i, P_i)$ can be computed in time $\mathcal{O}(md)$. At most t such values need to be computed. It follows that the total complexity is $\mathcal{O}(n2^{t(d+\log t)}(t2^d + tmd))$.

It remains to show that our recurrences for $D(x, \mu_x, P_x)$ are correct, which we do by induction over the nodes of T from the leaves to the root. As a base case, this is true for the leaves, so assume $x \in V(T)$ is an internal node of T. For the remainder of the proof, given an ordering π of some set X, let $\pi|X'$ denote the ordering on $X' \subseteq X$ of π restricted to X' (i.e. $\pi|X'$ is the unique ordering of X' such that π realizes $\pi|X'$).

Before proceeding with the correctness, we first claim that for any child y of $x \in V(T)$, $V'_y \subseteq V'_x$. Suppose this is not the case. Because $V_y \subseteq V_x$, by the definition of V'_y and V'_x, there must be some $v \in B_y \setminus B_x$ and $u \in N^-(v)$ such that $u \notin V'_x$. In particular, $u \notin V_x$. Since T is a tree decomposition, there must be a node $z \in V(T)$ such that $u, v \in B_z$. But since $u \notin V_x$, z cannot be in the subtree rooted at x, as otherwise $u \in V'_x$ would hold. This is a contradiction, as this implies that the vertices with bags containing v do not form a connected component of T, which proves our claim.

We now treat each possible node type separately to prove our recurrences correct.

Introduce nodes. Suppose x is an introduce node with child y and new vertex v_i. For each $v_j \in B_x$, let $P_j \in P_x$ be the subset of $N^-(v_j)$ for v_j. Let $u \in N^-(v_i)$. It is straightforward to check that $u \notin V_y \setminus B_x$, since a bag of T must contain u and v, v_i was introduced in bag B_x and u is not in B_x. Similarly, let $u \in V_y \setminus B_y$. One can check that $N^-(u) \subseteq V_y$, as otherwise a neighbor of u outside of V_y would lead to the same type of contradiction.

We first show that $D(x, \mu_x, P_x) \geq \min_{y,\mu_y,P_y}\{D(y, \mu_y, P_y) + cost(v_i, P_i)\}$, where P_i is the subset of $N^-(v_i)$ for v_i in P_x, and $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$ are realizable. Let π be an ordering of V'_x such that $\sum_{v \in V_x} cost(v, \pi) = D(x, \mu_x, P_x)$ and such that π realizes μ_x and $s(P_x)$. Let $\pi_y := \pi|V'_y$ (note that π_y is well-defined since $V'_y \subseteq V'_x$), and let $\mu_y = \pi_y|B_y$ and $P_y \in \Lambda(y)$ be such that π_y realizes $s(P_y)$. Clearly, $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$ is realizable (as witnessed by π). Moreover, $\sum_{w \in V_y} cost(w, \pi_y) \geq D(y, \mu_y, P_y)$, by the definition of $D(y, \mu_y, P_y)$. As we also have $cost(v_i, \pi) = cost(v_i, P_i)$, it follows that

$$\sum_{v \in V_x} cost(v, \pi) \geq D(y, \mu_y, P_y) + cost(v_i, P_i) \geq \min_{y', \mu'_y, P'_y}\{D(y', \mu'_y, P'_y)\} + cost(v_i, P_i)$$

as desired.

As for the converse bound, take any ordering π_y of V'_y of cost $D(y, \mu_y, P_y)$ that realizes μ_y and P_y on B_y such that $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$ is realizable. We start from π_y and construct an ordering of V'_x. If v_i is not in π_y, insert v_i in π_y anywhere so that it realizes μ_x (this is possible since π_y realizes $\mu_y = \mu_x|(B_x \setminus \{v_i\})$. Then let $\pi'_y := \pi_y|(V_y \cup \{v_i\})$. Note that since any $u \in V_y \setminus B_x$ has no in-neighbor outside of V_y, the cost of u is entirely defined by π'_y, and hence unchanged from π_y. We now want to insert the elements of $V'_x \setminus (V_y \cup \{v_i\})$ so as to realize $s(P_x)$. Let $\hat{\pi}$ be any ordering of $\bigcup_{v_i \in B_x} N^-[v_i]$ that realizes $s(P_x) \cup s(P_y) \cup \{\mu_x\}$, and let $\pi' := \hat{\pi}|((V'_x \setminus V_y) \cup B_x)$. Since the elements of π'_y and π' coincide only on B_x and both realize μ_x, it is easy to see that there is some ordering π that realizes π'_y and π'. Note that π is an ordering of V'_x. Moreover, π realizes μ_x, and therefore also realizes μ_y.

We must now argue that π realizes $s(P_x)$ (which also implies that π realizes $s(P_y)$). First consider $P_j \in P_x$, where $i \neq j$ so that $v_j \in B_x \cap B_y$ and P_j is the subset of $N^-(v_j)$ for v_j in P_x. Let $u \in P_j$. If $u \in V_y$, then $u <_\pi v_j$ since π realizes π'_y (which is a subordering of π_y that realizes $s(P_y)$). If $u \in V'_x \setminus V_y$, then $u <_\pi v_j$ because π realizes π' (which is a subordering of $\hat{\pi}$ that realizes $s(P_x)$). By a similar argument, one can check that all $u \in N^-(v_j) \setminus P_j$ occur after v_j. Now consider $P_i \in P_x$, the subset of $N^-(v_i)$ for v_i. Let $u \in P_i$, and recall that $u \notin V_y \setminus B_x$. If $u \in B_x$, then $u <_\pi v_i$ because π realizes μ_x (and we may assume $u <_{\mu_x} v_i$ as otherwise μ_x and $s(P_x)$ are not possibly realizable together). If $u \in V'_x \setminus B_x$, then $u <_{\mu_x} v_i$ because π realizes π', as above. The case $u \in N^-(v_i) \setminus P_i$ can be verified in a similar manner.

Since the costs of the $v \in V_y$ are unchanged from π_y to π, it follows that $D(x, \mu_x, P_x) \leq \sum_{v \in V_x} cost(v, \pi) = \sum_{v \in V_y} cost(v, \pi_y) + cost(v_i, \pi_y) = D(y, \mu_y, P_y) + cost(v_i, P_i)$, which yields the complementary bound.

Forget nodes. Suppose that x is a forget node with child y. In this case, $V_x = V_y$ and $V'_x = V'_y$. It is not hard to see that it suffices to inherit the time costs computed at the y node.

Join nodes. Suppose x is a join node with children y, z, in which case $B_x = B_y = B_z$. Denote $B_x = \{v_1, \ldots, v_s\}$. For each $v_i \in B_x$, let $P_i \in P_x$ be the subset of $N^-(v_i)$ for v_i. Note that if $v \in V_y \setminus B_x$, then $v \notin V_z$ (otherwise, the bags containing v would not be connected). Similarly, if $v \in V_z \setminus B_x$ then $v \notin V_y$. Hence $V_y \cap V_z = B_x$. Let π be an ordering of V'_x that realizes μ_x and $s(P_x)$ of cost $D(x, \mu_x, P_x)$. Let $\pi_y := \pi | V'_y$ and $\pi_z := \pi | V'_z$. Note that both π_y and π_z must realize μ_x and $s(P_x)$. Hence $\sum_{v \in V_y} cost(v, \pi_y) \geq D(y, \mu_x, P_x)$ and $\sum_{v \in V_z} cost(v, \pi_z) \geq D(z, \mu_x, P_x)$. Since $V_y \cap V_z = B_x$, it follows that $D(x, \mu_x, P_x) \geq D(y, \mu_x, P_x) + D(z, \mu_z, P_x) - \sum_{v_i \in B_x} cost(v_i, P_i)$.

For the converse bound, let π_y (respectively π_z) be orderings of V'_y (V'_z) that realize μ_x and $s(P_x)$ of cost $D(y, \mu_x, P_x)$ ($D(z, \mu_x, P_x)$). Note that if $u \in V_z \setminus B_x$, then $N^-(u) \subseteq V_z$ (using tree decomposition arguments) and if $u \in V_y \setminus B_x$, then $N^-(u) \subseteq V_y$. Let $\pi'_y := \pi_y | (V'_y \setminus V_z) \cup B_x$. Then for all $u \in V_y \setminus B_x$, the cost of u is unchanged from π_y to π'_y. Then, let $\pi'_z := \pi_z | V_z$, with the same remark on $u \in V_z \setminus B_x$. Let π be an ordering of V'_x that realizes π'_y and π'_z. Note that π exists, since π'_y and π'_z coincide only on B_x and both realize μ_x.

We argue that π realizes $s(P_x)$. Let $v_i \in B_x$ and $u \in P_i$. If $u \in B_x$, then $u <_\pi v_i$ because π realizes μ_x (which, as we may assume, is realizable with $s(P_x)$). If $u \in V'_y \setminus V_z$, then $u <_\pi v_i$ because π realizes π'_y (which is a subordering of π_y which realizes $s(P_x)$). Finally if $u \in V_z \setminus B_x$, then $u <_\pi v_i$ because π realizes π'_z (which is a subordering of π_z which realizes $s(P_x)$). A similar argument shows that $v_i <_\pi u$ for $u \in N^-(v_i) \setminus P_i$.

It remains to argue that $D(x, \mu_x, P_x) \leq \sum_{v \in V_x} cost(v, \pi) = D(x, \mu_x, P_x) + D(y, \mu_x, P_x) - \sum_{v \in B_x} cost(v, P_i)$. For $v \in V_y \setminus B_x$ or $v \in V_z \setminus B_x$, the cost is unchanged from π_y and π_z, respectively, as we mentioned above. If $v \in B_x$, the cost is the same as in π_y and π_z, since π, π_y and π_z all realize $s(P_x)$. Therefore, $\sum_{v \in V_x} cost(v, \pi) = \sum_{v \in V_y} cost(v, \pi_y) + \sum_{v \in V_z} cost(v, \pi_z) - \sum_{v_i \in B_x} cost(v, \pi)$ (as we double-counted the B_x elements). The correctness follows, since $\sum_{v \in V_y} cost(v, \pi_y) = D(y, \mu_x, B_x)$ and $\sum_{v \in V_z} cost(v, \pi_z) = D(z, \mu_x, B_x)$. ◀

5 Conclusion

The hardness results presented in this work apply to any game that allows damage boosting or routing in its speedrunning mechanics. However, the positive results ignore other possible aspects of the game, which could be incorporated in our problem models in the future. For instance, some games may offer multiple possible paths that in turn offer different sets of events. Also, role-playing games such as *Final Fantasy* are notorious for the calculations needed for manipulating the game's random number generator, which leads to other optimization problems. We also leave the problems of approximating damage boosting with lives and minimum-loss routing open, as well as determining their precise FPT status.

References

1 Classic damage data charts mega man 1 damage data chart. `http://megaman.wikia.com/wiki/Mega_Man_1_Damage_Data_Chart`. Accessed: 2018-02-21.

2 Games done quick. `https://gamesdonequick.com/`. Accessed: 2018-02-21.

3 Paola Alimonti and Viggo Kann. Hardness of approximating problems on cubic graphs. In *Italian Conference on Algorithms and Complexity*, pages 288–298. Springer, 1997.

4 Noga Alon, Dana Moshkovitz, and Shmuel Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms (TALG)*, 2(2):153–177, 2006.

5 Greg Aloupis, Erik D Demaine, Alan Guo, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.

6 Marthe Bonamy, Lukasz Kowalik, Jesper Nederlof, Michal Pilipczuk, Arkadiusz Socala, and Marcin Wrochna. On directed feedback vertex set parameterized by treewidth. *arXiv preprint arXiv:1707.01470*, 2017.

7 Jeffrey Bosboom, Erik D Demaine, Adam Hesterberg, Jayson Lynch, and Erik Waingarten. Mario kart is hard. In *Japanese Conference on Discrete and Computational Geometry and Graphs*, pages 49–59. Springer, 2015.

8 Liming Cai and Jianer Chen. On fixed-parameter tractability and approximability of NP optimization problems. *Journal of Computer and System Sciences*, 54(3):465–474, 1997.

9 Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM (JACM)*, 55(5):21, 2008.

10 William Cook, László Lovász, Paul D Seymour, et al. *Combinatorial optimization: papers from the DIMACS Special Year*, volume 20. American Mathematical Soc., 1995.

11 Graham Cormode. The hardness of the lemmings game, or oh no, more NP-completeness proofs. In *Proceedings of Third International Conference on Fun with Algorithms*, pages 65–76, 2004.

12 Marek Cygan, Fedor V Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 4. Springer, 2015.

13 Erik D Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder/easier than we thought. In *Proceedings of Third International Conference on Fun with Algorithms*, 2016.

14 Rodney G Downey and Michael Ralph Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.

15 Michael Etscheid, Stefan Kratsch, Matthias Mnich, and Heiko Röglin. Polynomial kernels for weighted problems. *Journal of Computer and System Sciences*, 84:1–10, 2017.

16 Guy Even, J Seffi Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.

17 Michal Forišek. Computational complexity of two-dimensional platform games. In *International Conference on Fun with Algorithms*, pages 214–227. Springer, 2010.

18 Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

19 Venkatesan Guruswami, Rajsekar Manokaran, and Prasad Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 573–582. IEEE, 2008.

20 Viggo Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Royal Institute of Technology Stockholm, 1992.

21 Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, 12(3):415–440, 1987.

22 Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

23 Daniel Lokshtanov. New methods in parameterized algorithms and complexity. *University of Bergen, Norway*, 2009.

24 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

25 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

Gender-Aware Facility Location in Multi-Gender World

Valentin Polishchuk
Communications and Transport Systems, ITN, Linköping University, Sweden
valentin.polishchuk@liu.se

Leonid Sedov
Communications and Transport Systems, ITN, Linköping University, Sweden
leonid.sedov@liu.se

Abstract

This interdisciplinary (GS and CS) paper starts from considering the problem of locating restrooms or locker rooms in a privacy-preserving way, i.e., so that while following the path to one's room, one cannot peek into another room; the rooms are meant for a multitude of genders, one room per gender. We then proceed to showing that gender inequality (non-uniform treatment of genders by genders) makes the room placement hard. Finally, we delve into specifics of gender definition and consider locating facilities for the genders in a "perfect" way, i.e., so that navigating to the facilities involves only quick binary decisions; on the way, we indicate that there is room for interpretation the facilities under consideration (we outline several possibilities, depending on the application).

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases visibility, Strahler number, perfect tree, interval graphs, gender studies

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.28

Acknowledgements We thank the anonymous reviewers for their helpful comments and Dr. Marietta Radomska from the Gender studies unit of Linköping University for bringing our attention to gender issues. The authors are partially supported by Swedish Transport Administration.

1 Introduction

The future progressive mankind, realizing and recognizing existence of more than two genders [5,14,22,39,43], faces social, political, economic, as well as algorithmic and scientistic[1] challenges of adjusting to practices of the multi-gender world and catering to the multiple genders needs. For instance, in the TEDx talk [32] on gendered innovations – a hot topic on both sides of the Atlantic [10,35] – a possibility is suggested to "require sophisticated sex and gender analysis when selecting papers for publication" (it would be fair to mention that the suggestion referred to doing it for journals and not for FUN). A lot of other inspiring material is available, produced not only by academics but also by recreational mathematicians and science popularizers [17].

One place where gender considerations come into play are locker rooms in a gym and public restrooms in a mall. As locking the locker room and restroom usage to biological sex

[1] We define a *scientistic*, or *scientistical* challenge as a non-scientific challenge faced by a scientist. Examples of scientistic challenges abound; one example relevant to this paper is learning to speak about genders without referring to sexes.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 28; pp. 28:1–28:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

is being questioned (see e.g., [21]), solutions are explored both in terms of extreme gender segregation (cf. women-only gyms) and integration (cf. unisex public toilets).

For concreteness, let us speak about the gym application. We will often call the locker rooms just rooms. Privacy preservation stipulates that each gender has its own room, and that no gender g can see into another gender's room while going to g's room. We assume that the undesired peeking into other rooms may happen only from the path to the locker room, and not from anywhere in the gym (as no one wants to embarrass oneself by standing at a known spot of other-genders spotting, pretending to exercise; in addition, when exercising, people are busy with their own bodies).

Going deeper into the physics (optics) of peeking, observe that undesired exposures happen through rooms doors.[2] Locker room areas are a scarce resource, especially given the multitude of genders, so there may be people, in any locker room, arbitrarily close to the door (in fact, this is the case in many existing space-constrained gyms – those doubtful are welcome to visit a popular fitness facility at 6pm). Therefore, we ignore the locker room shapes, and require that no gender g sees another gender's door while going to g's door.

We model the gym by a simple polygon P. Let V be the vertices of P. One of the vertices $s \in V$ is the entrance to the gym. The potential locker room doors are a subset $D \subset V$ (we model doors as vertices, and not edges of P, because doors are small). For a point $p \in P$, let $\pi(s, p)$ denote the shortest s-p path. Our goal is to pick a maximum-cardinality subset $L \subseteq D$ of locker room doors so that no point on the shortest path $\pi(s, l)$ from s to a door $l \in L$ sees another door $l' \in L, l' \neq l$.

1.1 Related work

Genders (and perfectness – the other theme of this paper, developed below) are in focus of Hall's Marriage Theorem [42]: a bipartite graph, whose parts are two genders and whose edges indicate compatible pairs, has a perfect matching if and only if the vertices from any size-k subset of one part are collectively connected to at least k vertices (from the other part). Genders are central in research on Stable Marriage [16], where individuals rank members of the other gender and the goal is to match people so that there exists no pair of opposite-gender individuals each of which ranks the other higher than the current partner. Last but not least, three genders are the subject in 3D Matching – one of the six basic NP-complete problems [11, Section 3.1], in which the input is a set of compatible triples, each containing one member of each of the three genders, and the goal is to choose a subset of the triples so that each individual belongs to exactly one triple.

Hiding genders from the other genders is related to People Hiding, which is the problem of selecting a largest set of pairwise-invisible polygon vertices, or the Maximum Independent Set (MIS) in the visibility graph on V. Our problem is related to People Hiding because a feasible set L of locker room doors must be an independent set (IS) in the graph. People Hiding was proved NP-hard in [33], and [8] showed hardness of giving a PTAS.

More generally, both visibility and path planning are textbook subjects in geometric computing – see, e.g., the respective chapters in the handbook [13] and the books [12, 27]. Visibility meets path planning in a variety of computational-geometry tasks. Historically, the first approach to finding shortest paths was based on searching the visibility graph of the domain. Visibility is vital also in computing minimum-link paths, i.e., paths with fewest edges [23, 25, 37]. Last but not least, "visibility-driven" route planning is the subject in

[2] In addition, the door must be open, but the sensitive study of the birth–death process of people inside locker rooms is outside the scope of this paper; anyway, doors are regularly opened for drying.

Watchman Route problems [2, 6, 7, 24, 28] where the goal is to find the shortest path (or a closed loop) from which every point of the domain is seen (our door-picking algorithm uses the “essential cut” from watchman-route solutions). Apart from the above-mentioned theoretical considerations, visibility and motion planning are closely coupled in practice: computer vision and robot navigation go hand-in-hand in many courses and real-world applications.

1.2 Contributions and Roadmap

We open up the door (pun intended) to algorithmic gender studies. Specifically,

Section 2 gives an algorithm for choosing the maximum number of doors so that no door is seen from the shortest path between the entrance and another door. Note the contrast with the hardness of hiding people: while just picking a maximum subset of pairwise-invisible doors is equivalent to the NP-hard People Hiding (see Section 1.1), adding the restriction that no door is seen also from the *entire path* to another door makes our problem solvable in polynomial time (usually, adding restrictions makes problems harder). To make our solution efficient, we give a simple algorithm for MIS in “interval nest digraphs” (an extension of interval graphs), improving the currently-best cubic runtime [19] to quadratic; our algorithm also works for the weighted MIS, improving the currently-best runtime from quatric [18] to quadratic (the algorithms in [18, 19] work for more general class of graphs, however). We also give a simple reduction from People Hiding to our problem in *domains with holes*.

Section 3 proves (or, rather, gives another justification to the apparent fact) that non-equal treatment of genders by genders makes things hard(er): the door picking problem becomes NP-hard when every gender has a list of genders by whom it does not want to be seen (and the list does not simply contain all the other genders).

Section 4 considers picking facilities for gender segregation *without* taking into account the (in)visibility of genders to genders. We introduce the view on a gender as a (binary) string of attributes, formulate the Axiom of (Gender) Choice and derive from it the first result of the Gender Number Theory – the Fundamental Theorem On Multiple Genders. The theorem, along with navigation convenience, motivates us to consider choosing a subset of intervals on the polygon boundary so that the shortest paths from the entrance to the chosen intervals form a *perfect* binary tree. We give an algorithm for the problem, combining the greedy solution for MIS in an interval graph (which is perfect) with the bottom-up computation of *Strahler number* [41], or the *dimension* [9] of a tree, which is the height of the largest perfect tree minor of the given tree. We thus compute the *gender dimension* of a set of intervals on the boundary of a polygon – the number of genders that can be accommodated in a perfect way. The geometric nature of our MIS instances is used to bound the amount of information propagated up the tree by the algorithm; for general graphs, the requirement to have a perfect tree on the IS does not make the problem easier than vanilla MIS.

2 The gender number of a gym

We show how to find the maximum number of genders that can have the “pathview-independent” locker rooms in a gym. Recall the notation from Section 1: we are given a subset $D \subset V$ of vertices of a simple polygon P (the potential locker room doors) and a vertex $s \in V$ (the entrance), and want to pick a maximum-cardinality subset $L \subseteq D$ so that for any door $l \in L$, no point on the shortest s-l path $\pi(s, l)$ sees another door $l' \in L, l' \neq l$.

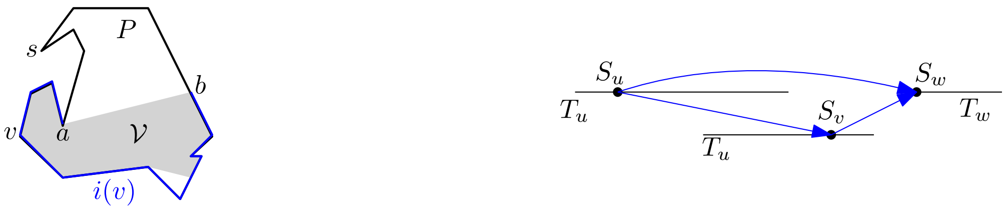


Figure 1 Left: $\mathcal{V}$ is shaded, $i(v)$ is blue; v is seen from $\pi(s, u)$ if and only if $u \in i(v)$. Right: Any IS is a path in the complement DAG, and vice versa: two consecutive vertices are connected in the DAG by definition; moreover, if uv, vw are edges, then S_v is right of T_u (since S_v is right of S_u, and there is no uv edge in the original graph) and hence S_w is also right of T_u (since S_w is right of S_v), implying that uw is also an edge, and inductively – that the whole path is a clique in the complement (the path coincides with its transitive closure).

Consider the visibility polygon $\mathcal{V}$ of a vertex $v \in D$, which is the set of points seen by v (Fig. 1, left). The polygon is bounded by edges and chords of P, with each chord connecting a vertex of P to a point on the boundary of P. One immediate observation is that if a vertex is visible from s, then the vertex cannot serve as the door to a room; thus it can be assumed w.l.o.g. that no vertex in D sees s.

For a vertex v that does not see s, there is a unique chord ab of $\mathcal{V}$ separating v from s. The chord is called the *essential cut* of v [2]. Let $i(v)$ denote the interval a-b along the boundary of P, to which v belongs. The vertex v is seen from the path $\pi(s, u)$ to a vertex u if and only if the path crosses ab, or, equivalently if and only if $u \in i(v)$. Thus, our problem reduces to the following: given the set of pairs $(v, i(v))_{v \in D}$ where v is a vertex and $i(v)$ is an interval on the boundary of P, find a maximum-cardinality subset $L \subseteq D$ so that there are no two vertices $l, l' \in L$ with $l \in i(l')$.

The above problem is an extension of MIS in interval graphs. Specifically, the *intersection graph* of a family of geometric objects has a vertex for every object and an edge for every pair of intersecting objects; the most prominent example, relevant for us, is the class of *interval graphs* in which the objects are intervals on a line. (A generalization of interval graphs are circular-arc graphs in which the objects are arcs on a circle; however, we have an interval graph, not circular-arc graph, because none of our intervals contains s, as s is separated by the essential cuts.) The following generalization of intersection graphs was proposed in [4]: the *intersection digraph* of a family $(S_v, T_v)_{v \in D}$ of ordered pairs of sets has D as the vertex set and a (directed) arc from vertex v to vertex u whenever $S_v \cap T_u \neq \emptyset$; in the *interval digraph* the sets S_v, T_v are intervals of the real line. Our problem reduces to finding MIS in an interval digraph (or more precisely, in the undirected *underlying* graph in which every arc is turned into undirected edge), simply by setting $S_v = v, T_v = i(v)$; in fact, we have an *interval catch digraph* [29], which are interval digraphs where S_v is a point and $S_v \subseteq T_v$. It was shown in [29] that interval catch digraphs are weakly triangulated (contain no chordless cycle of length at least 5 and no complement of such a cycle), and weakly triangulated graphs are perfect [20]. A well known property of perfect graphs (see, e.g., [15, Chapter 9]) is that MIS in them can be found in polynomial time.

▶ **Theorem 1.** *The maximum number of genders, for which a gym can set up locker rooms so that the shortest path from the entrance to any locker room does not see the door to another locker room, can be computed in polynomial time.*

▶ Remark. Instead of having $D \subset V$, we could consider a more general version where the doors are segments on the boundary of P. We would then redefine $\mathcal{V}$ to be the *weak visibility*

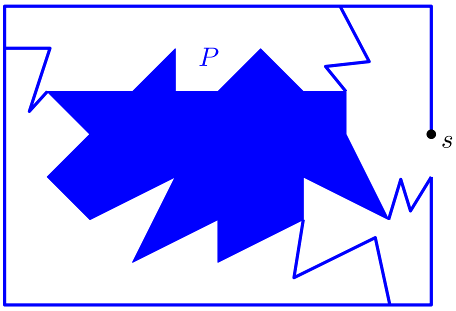


Figure 2 The free space (blue) is P, the corridor and the zigzags (zigzags are drawn not to all vertices); everything else (white) is obstacles.

polygon of v, i.e., the set of points seen from some point of v, or the union of the visibility polygons of points of v – as above, if s is inside $\mathcal{V}$, then v cannot be a locker room door. Otherwise, a point of the door is seen from a path iff the path crosses the essential cut of v, so the problem again reduces to MIS; this time – MIS in an *interval nest digraph*, in which S_v is a segment and $S_v \subseteq T_v$ (not interval catch digraph, as we had above, when the doors were vertices). When the doors are non-point objects, other models could be possible for the unwanted exposure: when a gender's door l is *fully* seen from a point on the path $\pi(s, l')$ to another gender's door, when all of l is seen (collectively) from a subset of $\pi(s, l')$ (i.e., when there exists $\pi' \subseteq \pi(s, l') : l \subset \cup_{p \in \pi(s,l')} \mathcal{V}(p)$ where $\mathcal{V}(p)$ is the visibility polygon of a point p), when π' has to be a contiguous subpath of $\pi(s, l')$, etc. – we leave these open.

▶ Remark. In non-simple gyms the problem becomes NP-hard. Indeed, given an instance of People Hiding (finding MIS in the visibility graph of vertices of a simple polygon P), we set the potential doors D to be the vertices of P, put a corridor around P and set the entrance s to the obtained domain at the start of the corridor; we connect the corridor to each vertex of P with a thin zigzag, so that no vertex sees any part of the corridor – this way, the path to any door may see another door if and only if the two doors are visible in P (Fig. 2). Thus, the maximum number of people that can hide from each other at vertices of P equals the maximum number of genders that can have the "pathview-independent" locker rooms in the domain.

2.1 A more efficient, combinatorial solution

General algorithms for MIS in perfect graphs [15, Chapter 9] are non-combinatorial and may have high complexity; therefore a lot of effort has been devoted to designing faster algorithms for special classes of perfect graphs. In particular, existing algorithms for interval catch and interval nest digraphs look for cliques in the complement [30]; the same is true also about general weakly triangulated graphs [1, 18, 19, 31, 34]. The fastest existing solution to our problem (MIS in an interval catch digraph) would run in worst-case cubic time [19]; here we give a simple quadratic algorithm. Our algorithm generalizes verbatim to interval nest graphs and to finding *weighted* MIS, improving from the currently-best quartic-time solution for the weighted case [18] (algorithms in [18, 19] work for more general classes of graphs though).[3]

[3] While going down from quartic to quadratic is a drastic theoretical improvement, we do not foresee an application of weighing the genders unequally. A (different) gender-unequal situation is considered in

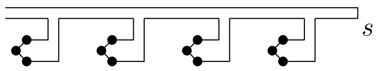


Figure 3 Solid circles are the doors; a gender will see the doors of the genders in the same "leg" of the caterpillar.

(Similarly to earlier algorithms,) we build the complement graph. (However, instead of looking for a clique in it,) we turn the complement into a DAG using S_v's for the vertices and directing the edges from left to right (if S_v is to the right of S_u, then there is a directed edge uv; Fig. 1, right), and find a longest path in it. The (quadratic) runtime is dominated by building the DAG and finding the path.

3 Gender inequality leads to hardness

In the previous section, the genders were gender-oblivious: it did not matter, for any gender, which other gender would see it naked – the exposure was undesirable regardless. In this section we consider the case when each gender has a list of genders to which it does not want to be exposed; being seen by the genders outside the list is not an issue. This generalization makes our problem NP-hard:

▶ **Theorem 2.** *If genders have lists of genders by whom they do not want to be seen, it is NP-hard to decide whether it is possible to set up locker rooms so that there are no unwanted exposures from the shortest paths between the entrance and the locker rooms.*

Proof. The reduction is from Partition Into Triangles [11, Problem GT11] (a close relative of 3D Matching, see Section 1.1): Can vertices of a graph be partitioned into triples so that the induced subgraph on each triple is a triangle? Given an instance of the problem, create a caterpillar-like polygon which consists of a long corridor to which short corridors are attached (Fig. 3), each ending with 3 doors; the total number of doors equals the number of vertices in the graph. We have as many genders as there are doors – thus, our problem is not to choose the doors (all doors need to be chosen), but to match perfectly the genders with the doors (in Section 2, the matching was not an issue since all genders were equal).

We associate each vertex of the graph with a gender. For every edge in the graph, the genders on the edge endpoints *do not* care about being exposed to each other, i.e., they are *not* in each other's lists (even though we are no longer in a gender-equal world, we at least have gender-symmetric lists: if one gender does not want to be seen by another, the latter does not want to be seen by the former). Thus, the vertices of the graph can be split into triangle-inducing triples if and only if the rooms can be assigned to the genders, avoiding undesirable exposures. ◀

4 Gender dimension and perfectness

In this section we stop taking (in)visibility between genders into account, consider premises other than gyms and go deeper into understanding what a gender is.

the next section.

4.1 Setting up the perfect scene

Women-only gyms are an extreme way to address (if at all) gender issues; an arguably milder approach to gender segregation, practiced e.g., in fast-food chains and airports in gender-segregating societies, is to have separate counters for the different genders. The ancient segregation tradition is unlikely to fade away, but maybe the modern ideas of acknowledging multiple genders existence could make their way into gender-segregating cultures? Being proactive, in anticipation of this, we consider the algorithmic question of picking disjoint counters for the many genders; here, hiding genders from genders is not an issue (as it was in Section 2) since people appear in the restaurant/airport well dressed.

Similar problems may arise e.g., in a ballet class where different genders use different barres lined up along the walls, and may even be supervised by different(-gender) coaches. A usual reason for separating genders in ballet classes is that they practice different, often complementary, parties. In ballet, gender questions have always been answered in the most progressive ways (and even were subjects of publications [26]); e.g., more-than-lightweight ballerinas and their spectacular performance are recurring topics in ballet circles [38,40]. We may look forward ballet pieces in which presence of multiple genders is an essential part of the composition (as the two genders currently are) in the foreseeable future (directing such compositions may lead to interesting geometric questions). That is, solving the problem of optimally setting up the barres for multiple genders may be even more pressing than the above-discussed question of segregating genders at counters. Again, no issue of hiding (from) genders is present (on the contrary, genders might even want to show off to other genders).

Examples like above bear the common abstract gist, which may be formalized in the following problem statement:

Picking Intervals for Gender Segregation (PIGS) problem We are given a set D of intervals on the boundary of a simple polygon P – representing potential counters in a shop or restaurant, (sequences of) ballet barres, etc.; also given is a vertex s of P – the entrance. None of the intervals in D contains s, but otherwise, the intervals are not connected to P in any way (they may start and end in the middle of P's edges, may span multiple edges, etc.). It can be assumed w.l.o.g. that no interval is a subset of another interval. The goal is to pick a maximum-cardinality subset $L \subseteq D$ of non-overlapping intervals.

Giving a polynomial-time algorithm for PIGS is not interesting, since the intervals form an interval graph which is perfect (as in Section 2, we have an interval graph, and not circular-arc graph, since s belongs to no interval, implying that the boundary of P may be punctured at s). In the reminder of the section we take perfection one step further and require also that the shortest paths from s to the picked intervals form a *perfect* binary tree (which we will often call just "perfect" tree) – i.e., a binary tree in which every internal node has degree 2 (after vertices along paths are smoothed) and all leaves are at the same level (such a tree is sometimes called "complete" [44]). The motivation for the requirement comes from a deeper study of gender nature, presented next.

4.2 Gender definition and formal GS foundations

A moment of reflection suggests that a gender must be characterized by certain attributes – of social, biological, or any other nature. To expand and formalize this thought, we define gender as a string:

▶ **Definition 3.** Let $A_1, \dots, A_K$ be a set of attributes; for $k = 1 \dots K$, let $\mathcal{A}_k$ be the set of values that A_k may assume. A *gender* is a string $a_1 \dots a_K$ where $a_k \in \mathcal{A}_k$.

Algorithmic gender studies, for which this paper makes the first steps, will become an essential part of Gender Science (GS). As any science, GS must be built on a solid axiomatic foundation (currently missing, which is forgivable for a science as young as GS). In particular, we believe that it would be unfair to deprive GS of the Axiom of Choice (while the good old sciences like mathematics enjoy the axiom in its full generally). The GS axiom reads:

▶ **Axiom 4** (Axiom of Choice)**.** Given a set of attributes, there is a gender for each choice of values for the attributes.

For instance, assume that $a_K = \textit{baldness}$ is a binary attribute (equal to 1 for a bald person and 0 otherwise), and let $S = a_1 \dots a_{K-1}$ be a length-$(K-1)$ string with $a_k \in \mathcal{A}_k, k = 1 \dots K-1$. Then both $S0$ and $S1$ should be genders, for it would be discriminative if bald people could identify themselves with a separate gender, while non-bald could not, or vice versa (boldness, or another attribute could be used in the example just as well). As computer scientists (who are adepts of the binary view of the world, reducing everything to 0s and 1s – cf. the classical joke about 10 types of people), we postulate that every attribute can assume only the values of 0 and 1 ($\forall k = 1 \dots K, \mathcal{A}_k = \{0, 1\}$); if not, just represent any kind of non-binary attribute by a set of binary attributes of appropriate size. With this interdisciplinary GS–CS postulate, we are ready to prove the first (and maybe the last) result in Gender Number Theory:

▶ **Theorem 5** (Fundamental Theorem On Multiple Genders (Fundamental OMG Theorem))**.** *The number of genders is a power of 2.*

Proof. By the Axiom of Choice, the number of genders is 2^K. ◀

We emphasize that the number of attributes K (and hence the number of genders) is not a constant; as the society gender awareness evolves, K may grow (and possibly decrease – but this is outside our scope).

4.3 Perfect counter/barre choice

We now return to choosing disjoint intervals for the genders (PIGS). By the Fundamental OMG Theorem, the number of picked intervals, $|L|$, must be a power of 2. As the number of genders will grow exponentially with new attributes being introduced/acknowledged/fashonable, it may become cumbersome to navigate from the entrance, s, to the intervals corresponding to people's genders. Returning for a moment to our gym application (Section 2), we ran extensive experiments on ourselves, in which the subject arrives to a new gym and finds the way to the locker room of the subject's gender; it was observed that, unsurprisingly, the task is much easier to perform in gyms with legible signs which first point in the general direction of the locker rooms, and then clearly mark the point where the paths to the different-gender rooms diverge.[4] Back to PIGS, we envision that in multi-gender society, it will also be convenient to have a "binary-split" shortest path tree (an extension of the classical "Boys to the left, girls to the right" directions from field trips to the wild), for navigation from s to the genders target intervals, with clearly marked split nodes. It might be confusing, however, to have a sign listing, say, 17 genders whose intervals are in one direction and 15 genders whose targets are in another – in fact, it could be embarrassing to stand near such a sign as

[4] Detailed report of the experimental results is deferred to a submission to the future sister conference on FUN with *Experimental* Algorithms.

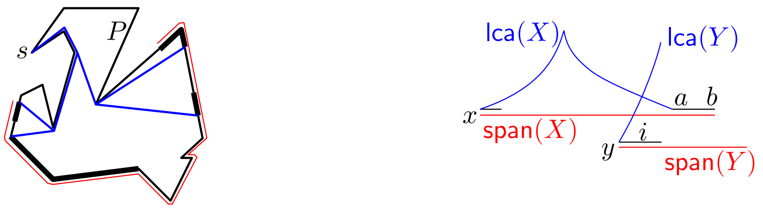


Figure 4 Left: Bold intervals are a perfect set Y of height 2. Red is the span of the set (Definition 6 below) and blue is $\mathsf{SPT}(Y)$; $\mathsf{lca}(Y) = s$. Right: Intervals are black; paths in $\mathsf{SPT}(D)$ are blue.

if confused about own gender (like the gender-confused Wolf from Shrek 1, mentioned as such in Shrek 2). On the contrary, it would be more natural (and gender-respectful) to split the genders paths based on the attributes, with each split involving only one attribute.

Thus, a gender-caring facility owner may face PIGS with the additional requirement that the shortest paths tree from s to the chosen intervals has maximum degree 3 (i.e., when viewed as rooted tree with s as the root, its every node has either 1 or 2 children). To state the full problem formally, let us introduce some notation.

We keep the assumptions from PIGS; in particular, the assumption that s does not belong to any interval from D. For an interval $i \in D$, we define the *left* endpoint $l(i)$ of i to be its (end)point encountered first when going from s counterclockwise around ∂P; more generally, we say that for two points $a, b \in \partial P$, a is to the left of b if a is between s and b when following the boundary counterclockwise from s. For a vertex v of P, the *shortest path from v to i* will mean the shortest path $\pi(v, l(i))$ to $l(i)$. (Alternatively, we could have used any other fixed points on the intervals to define the shortest paths or even use the “true” shortest paths to the intervals – i.e., shortest paths to their points that are closest to s– but dealing with paths to points makes the exposition cleaner; in terms of our applications, we may assume that the left endpoints are where the cashiers are at the counters or where the ballet dancers drop their stuff before using the barres.) For a subset $X \subseteq D$ of intervals, let $\mathsf{SPT}(X)$ be the shortest paths tree from s to the intervals in X, and let $\mathsf{lca}(X)$ be the intervals’ least common ancestor in $\mathsf{SPT}(D)$. It can be assumed w.l.o.g. that all intervals in D are leaves of $\mathsf{SPT}(D)$ (otherwise, if an interval i corresponding to an internal node of the tree is picked, the interval can be replaced by a leaf interval i', the shortest path to which goes via i).

Let $Y \subseteq D$ be a set of pairwise-disjoint intervals; we say that intervals in Y are *independent* (since they form an IS in the interval graph). Let tree T be the union of the shortest paths from $\mathsf{lca}(Y)$ to intervals in Y (Fig. 4, left). Abusing the terminology, we say that T is *perfect* if it becomes a perfect binary tree after its degree-2 vertices are repeatedly smoothed (i.e., after every path of degree-2 vertices in T is replaced by a single edge). If T is perfect, we say that Y is a *perfect* IS, or simply a *perfect (sub)set.* Clearly, the cardinality of a perfect subset is a power of 2. We say that a size-2^k perfect set Y has *height* k (because k is the height of $\mathsf{lca}(Y)$ when T becomes perfect after the smoothing).

With the above notation, our problem may be stated as:

Perfect PIGS We are given a set D of intervals on the boundary of a simple polygon P; also given is a vertex s of P. The goal is to pick the largest perfect subset L of D (i.e., the largest subset $L \subseteq D$ such that $\mathsf{SPT}(L)$ is perfect).

That is, solution to Perfect PIGS gives the maximum number of genders for which the intervals can be picked so that people can navigate to their intervals in a “perfect” way – with only binary decisions at the branching points of $\mathsf{SPT}(L)$.

Perfect PIGS is an extension of not only PIGS but also of the problem of finding the *dimension* [9], or the *Strahler number* [41] of a tree – the height of the largest perfect minor of the tree (it is assumed that the tree is rooted and that the perfect minor has the same root). We therefore define the *gender dimension* of D as the height of the perfect tree $\mathsf{SPT}(L)$; with this definition, Perfect PIGS becomes the problem of computing the gender dimension of a set of intervals.

Our solution for Perfect PIGS, presented in Section 4.4, is based on two simple algorithms:

- the greedy Earliest-Endpoint algorithm for MIS in interval graphs: iteratively pick the interval with leftmost right endpoint and remove the intervals overlapping the picked one;
- the recursive procedure to compute the Strahler number of a tree (and in fact, the Strahler number of each subtree – the height of the largest perfect minor of the subtree): assign Strahler number $d(l) = 0$ to each leaf l, and then for an internal node v, let k be the maximum of the Strahler numbers of v's children – if the maximum is unique, assign $d(v) = k$, otherwise, $d(v) = k + 1$.

For Perfect PIGS, to make sure the leaves of $\mathsf{SPT}(L)$ are pairwise-disjoint, we propagate more information up the tree $\mathsf{SPT}(D)$: for every node v of $\mathsf{SPT}(D)$ we list, for every k, all height-k "tight" perfect sets of intervals from the subtree of v (where tight is an analog of earliest-endpoint). The details follow.

4.4 Algorithm for Perfect PIGS

We start from showing that when merging perfect sets from sibling nodes of $\mathsf{SPT}(D)$, it suffices to look at "spans" of the sets. Specifically, let $X \subseteq D$ be a perfect set.

▶ **Definition 6.** The *span* of X, denoted $\mathsf{span}(X)$, is the smallest interval containing all intervals in X (refer to Fig. 4, left).

Clearly, if spans of two independent sets do not overlap, their union is also an IS. The next lemma shows that if the sets live in different subtrees of $\mathsf{SPT}(D)$, the converse is also true:

▶ **Lemma 7.** *Let $X, Y \subset D$ be independent sets such that none of them is a subset of the other and $\mathsf{lca}(X) \neq \mathsf{lca}(Y)$. If $X \cup Y$ is independent, then $\mathsf{span}(X) \cap \mathsf{span}(Y) = \emptyset$.*

Proof. Let x, y be the left endpoints of $\mathsf{span}(X)$ and $\mathsf{span}(Y)$ resp. (Fig. 4, right); assume w.l.o.g. that y is to the right of x, and let $i \in Y$ be the interval whose left endpoint is y ($y = l(i)$). Let ab be the rightmost interval in X (so b is the right endpoint of $\mathsf{span}(X)$). The paths from $\mathsf{lca}(X)$ to x and a, together with the part of the boundary of P from x to a form a closed loop; since none of $\mathsf{lca}(X)$, $\mathsf{lca}(Y)$ is in the subtree of the other (for otherwise one of X, Y would be a subset of the other), $\mathsf{lca}(Y)$ is outside the loop. We claim that if the spans of X and Y intersect, then y is to the left of a, implying that the path $\pi(\mathsf{lca}(Y), y)$ from $\mathsf{lca}(Y)$ to i would intersect the loop, contradicting planarity of $\mathsf{SPT}(D)$ (which follows from the triangle inequality). Indeed, for the spans to intersect, y must be to the left of b, but if y is also to the right of a, then i intersects ab – contradicting that $X \cup Y$ is independent. ◀

Lemma 7 allows us to work with spans of perfect sets (instead of the sets themselves) when merging sets up $\mathsf{SPT}(D)$; moreover, where it creates no confusion, we will identify X with $\mathsf{span}(X)$, *and* with $\mathsf{lca}(X)$. E.g., we will speak about perfect sets X, Y (or the spans $\mathsf{span}(X)$, $\mathsf{span}(Y)$) being *siblings* – meaning that $\mathsf{lca}(X)$, $\mathsf{lca}(Y)$ are siblings in $\mathsf{SPT}(D)$. In addition, unless stated otherwise, whenever we speak about two perfect sets X, Y, we will assume that they are *siblings and have the same height* (i.e., the same number of intervals) – this is because we separately maintain perfect sets of each possible height and merge up the

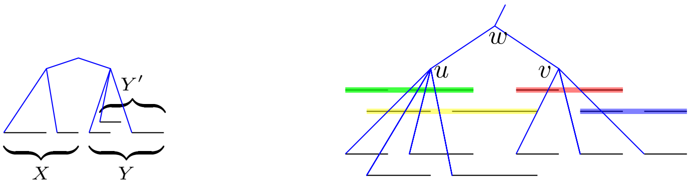


Figure 5 Left: $\mathsf{SPT}(D)$ is blue; Y is tight w.r.t. X, but Y' is not. Right: Tight height-1 sets at u are green and yellow, and the sets at v are red and blue; tight height-2 sets at w will be green+red and yellow+blue.

tree only pairs of sibling sets with the same height. Finally, whenever two spans have been merged in the course of the algorithm, we do a cleanup by removing "dominating" spans among siblings ($\mathsf{span}(Y)$ dominates a sibling span $\mathsf{span}(X)$ if $\mathsf{span}(X) \subseteq \mathsf{span}(Y)$) – this lets us speak about one set being *to the left* of another sibling set: X is to the left of a sibling Y if the left endpoint of $\mathsf{span}(X)$ is to the left of the left endpoint of $\mathsf{span}(Y)$ (due to removal of dominating spans, the right endpoint of X is also to the left of the right endpoint of Y).

Suppose now that X is to the left of Y and their spans do not overlap, or, to spell out all the assumptions, let $X, Y \subset D$ be perfect sets such that $|X| = |Y|$, (any point of) $\mathsf{span}(X)$ is to the left of $\mathsf{span}(Y)$, and $\mathsf{lca}(X)$ and $\mathsf{lca}(Y)$ are siblings in $\mathsf{SPT}(D)$.

▶ **Definition 8.** Y is *tight* w.r.t. X if there is no perfect set Y' such that (Fig 5, left): Y' is in the same subtree ($\mathsf{lca}(Y') = \mathsf{lca}(Y)$); $|Y'| = |Y|$; $X \cup Y'$ is an IS; Y' is to the left of Y.

Let $v = \mathsf{lca}(\mathsf{lca}(X), \mathsf{lca}(Y))$ be the common parent of $\mathsf{lca}(X)$ and $\mathsf{lca}(Y)$. Since (even) the spans of X and Y do not intersect, $Z = X \cup Y$ is an IS. Since X and Y have the same number of intervals and are siblings, Z is perfect: the height of the perfect tree rooted at v is by one larger than the height of the perfect trees rooted at its children; with our terminology, if each of X, Y had height k, the height of Z is $k + 1$.

Our algorithm merges, for each possible height k of perfect sets, only tight sets (Fig. 5, right). We initialize by making at each leaf of $\mathsf{SPT}(D)$ the list containing the interval at the leaf ($k = 0$). We then go up the tree. First of all, an internal node v of $\mathsf{SPT}(D)$ inherits the lists of its children for all k. In addition, we go through the lists of v's children for k, searching for tight merges of perfect sets. We do it brute force: for every child x of v, for every set X in the list of x, we go through each set Y in the list of every other child y checking whether $\mathsf{span}(X)$ and $\mathsf{span}(Y)$ are disjoint; out of the found sets, we keep only the tight one – for a fixed X, there is at most one tight union $X \cup Y$. The tight union becomes part of the list of v for $k + 1$. If no pair (X, Y) with disjoint spans is found, then v's list for $k + 1$ is empty.

▶ **Theorem 9.** *There is a polynomial-time algorithm for Perfect PIGS.*

Proof. Any tight set, after being created at a node v, appears also in the lists of all nodes on the path from v to the root s (unless the set is removed due to domination) – altogether at most $|V|$ times. To bound the number of sets created through the algorithm, note that since we remove dominating sets, at any node of $\mathsf{SPT}(D)$, for any k, we have at most one tight set "starting" at any interval i, i.e., the set whose span's left endpoint is $l(i)$ (in any case, the total number of possible spans is $O(|D|^2)$, as the spans endpoints come from endpoints of the original intervals). Since k is logarithmic in $|D|$, there are $O(\log |D|)$) lists at any node, so we can afford the propagation for all k – the total amount of propagated information is polynomial in $|D|$ and $|V|$, and the algorithm runs in polynomial time.

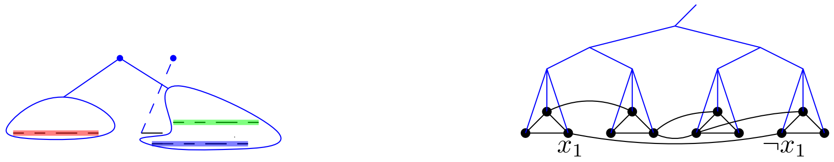


Figure 6 Left: Red and green spans are not tight, while red and blue are; using the tight union cannot lead to overlap with an interval i from another subtree, as such overlap would imply a crossing in $\mathsf{SPT}(D)$. Right: (Some of) the graph edges are drawn black; the inter-clause edges connect any variable to its negation.

Correctness of the solution can be shown by arguing that all "recursively tight sets" appear in our algorithm in the sets lists and that it suffices to look only at such sets. Specifically, let $\mathcal{T}$ be a perfect tree and let $L \subseteq D$ be its leaves. For a set $Z \subseteq L$ let $v = \mathsf{lca}_{\mathcal{T}}(Z)$ be the node of $\mathcal{T}$ whose subtree has Z as the leaves, and let X and Y be the sets at $\mathcal{T}$'s children of v (so $Z = X \cup Y$), with Y being tight w.r.t X. Say that Z is *recursively tight* if the sets at each internal node of the subtree of v is a tight union (i.e., if recursively, each of X and Y is a tight union, and – for the base of the recursion – a single interval is assumed to be a tight union). By induction on the node height, our algorithm lists all recursively tight sets, as it goes through all possible pairs of sets at each node of $\mathsf{SPT}(D)$. At the same time, there must exist a recursively tight optimal solution. Indeed, any feasible solution can be made recursively tight by (recursively) shuffling the spans to the left: start at the root of the perfect tree, and if the set $Z = X \cup Y$ at the root is not a tight union, then there exists a set Y' such that $Z' = X \cup Y'$ is a tight union – the solution with Y replaced by Y' is still an IS; recursively, if any of X, Y' is not a tight union, it can be fixed in the same way – the solution will remain feasible, which can be seen by a planarity argument analogous to the proof of Lemma 6 (Fig. 6, left). ◀

▶ Remark. Unsurprisingly, in general, putting a tree on top of an arbitrary graph and requiring to have a perfect tree minor on the IS from the graph, does not help finding MIS. To see this, use CLRS [3, p.1087] reduction from 3-SAT to MIS in the graph with a vertex for every literal, the literals in each clause connected into triangle (so at most one vertex from every clause may enter MIS) and edges between two literals whenever they are a variable and its negation (so only one can be in MIS); MIS size equals the number of clauses if and only if the 3-SAT instance is feasible (Fig. 6, right). It is easy to make the number of clauses a power of 2 and add a tree on the graph nodes, with vertices in each clause triangle being siblings of a height-1 parent, and the perfect binary tree up the leaves parents – any MIS will be perfect, so demanding the tree perfectness does not make the problem easier. It could be interesting to explore for which graphs, if any, the perfectness requirement changes the hardness of finding MIS; one question relevant to the gender study is whether it is possible to find perfect MIS in an interval catch digraph, to which reduces the problem of picking genders' locker room doors (like in Section 2) in the perfect way (like in Section 4) – we answer the question affirmatively in Appendix A.

5 Conclusion

We gave first algorithmic results for assessing how fit a gym is (i.e., how many genders it can accommodate) and related questions, presenting some polynomially-solvable and g-hard (for gender-hard, i.e., hard w.r.t. the number of genders) problems. Many extensions are possible:

- For a *given* number of genders, if it is not feasible to choose doors so that there is no exposure (Section 2), one may want to minimize various measures of the exposure – the number of undesirable peeks, the total length of the parts of the shortest paths from which the other(s') doors are seen, the total area seen behind the doors, the "depth" of the exposures (how far into locker rooms other genders see), etc. In the gender-oblivious scenario (when any gender does not want to be seen by all the others) the assignment of genders to the doors does not matter (as in Section 2); what matters is which vertices are chosen to be the doors.
- The gender-unequal setting in Section 3 may be generalized to the case when there is a whole matrix of numbers (possibly with positive entries, for exhibitionist genders) signifying (un)desirability of one gender being seen by every other. Here, one may want to minimize the *weighted* exposure.
- Last but not least, it would be interesting to know hardness of finding maximum perfect IS in a general perfect graph with a tree over its vertices. E.g., can the math programming algorithms for MIS in perfect graphs be made to work?

More generally, further mathematical studies on genders are to come, e.g., extending the differential equations for love [36] to many genders.

References

1 Srinivasa Arikati and C. Rangan. An efficient algorithm for finding a two-pair, and its applications. *Discrete Applied Mathematics*, 31(1):71–74, 1991.

2 Svante Carlsson, Håkan Jonsson, and Bengt J. Nilsson. Finding the shortest watchman route in a simple polygon. *Discrete & Computational Geometry*, 22(3):377–402, 1999. doi:10.1007/PL00009467.

3 Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

4 Sandip Das, M Sen, AB Roy, and Douglas B West. Interval digraphs. *J. Graph Theory*, 13(2):189–202, 1989.

5 Donna Dean. Changing ones: 3rd & 4th genders in native America. *Women and Military*, 18(2):54–54, 2000.

6 Moshe Dror, Alon Efrat, Anna Lubiw, and Joseph Mitchell. Touring a sequence of polygons. In Lawrence L. Larmore and Michel X. Goemans, editors, *SToC'03*, pages 473–482. ACM, 2003.

7 Adrian Dumitrescu and Csaba D. Tóth. Watchman tours for polygons with holes. *Comput. Geom.*, 45(7):326–333, 2012. doi:10.1016/j.comgeo.2012.02.001.

8 Stephan Eidenbenz. Inapproximability of finding maximum hidden sets on polygons and terrains. *CGTA*, 21(3):139–153, 2002.

9 Javier Esparza, Michael Luttenberger, and Maximilian Schlund. Fpsolve: A generic solver for fixpoint equations over semirings. *Intl J Foundations of Computer Science*, 26(07):805–825, 2015.

10 European Commission. Gendered Innovations.

11 Michael R. Garey and David S. Johnson. *Computers and Intractability*. Freeman & Co., New York, NY, USA, 1979.

12 Subir Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, New York, NY, USA, 2007.

13 J.E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry*. Discrete Mathematics and Its Applications. Taylor & Francis, 2nd edition, 2004.

14 Sharyn Graham. Sulawesi's fifth gender. *Inside Indonesia*, 66, 2001.

15 Martin Grötschel, László Lovász, and Alexander Schrijver. *Combinatorial optimization*, volume 2. Springer Science & Business Media, 2012.

16 Dan Gusfield and Robert Irving. *The Stable Marriage Problem: Structure and Algorithms.* MIT Press, Cambridge, MA, USA, 1989.

17 Vi Hart. On Gender, 2015.

18 Ryan Hayward, Chính Hoàng, and Frédéric Maffray. Optimizing weakly triangulated graphs. *Graphs and Combinatorics*, 5(1):339–349, Dec 1989.

19 Ryan Hayward, Jeremy Spinrad, and R. Sritharan. Weakly chordal graph algorithms via handles. In *SODA'00*, pages 42–49, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

20 Ryan B Hayward. Weakly triangulated graphs. *J Comb Theory, Series B*, 39(3):200–208, 1985.

21 Aaron Homer. Man Uses Women's Locker Room At Seattle Pool, Says It's Legal Because Of Anti-Transgender Discrimination Laws, 2016.

22 M Kay Martin and Barbara Voorhies. *Female of the Species.* Columbia University Press, 1975.

23 J. Mitchell, G. Rote, and G. Woeginger. Minimum-link paths among obstacles. *Alg-ca'92*, 8(1):431–459, 1992.

24 Joseph Mitchell. Approximating watchman routes. In Sanjeev Khanna, editor, *Proc. 24th Annual ACM-SIAM Symp. on Discrete Algorithms, SODA'13, New Orleans, Louisiana, USA*, pages 844–855. SIAM, 2013.

25 Joseph S. B. Mitchell, Valentin Polishchuk, and Mikko Sysikaski. Minimum-link paths revisited. *Comput. Geom.*, 47(6):651–667, 2014. doi:10.1016/j.comgeo.2013.12.005.

26 Cynthia J Novack. Ballet, gender and cultural power. In *Dance, gender and culture*, pages 34–48. Springer, 1993.

27 Joseph O'Rourke. *Art Gallery Theorems and Algorithms.* The International Series of Monographs on Computer Science. Oxford University Press, New York, NY, 1987.

28 Eli Packer. Computing multiple watchman routes. In Catherine C. McGeoch, editor, *SEA'08*, volume 5038 of *Lecture Notes in Computer Science*, pages 114–128. Springer, 2008.

29 Erich Prisner. A characterization of interval catch digraphs. *Discrete Math*, 73(3):285–289, 1989.

30 Erich Prisner. Algorithms for interval catch digraphs. *Discrete Appl Math*, 51(1-2):147–157, 1994.

31 Arvind Raghunathan. Algorithms for weakly triangulated graphs, 1989. Tech Rep, UC Berkeley. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/1989/5196.html.

32 Londa Schiebinger. Gendered Innovations, 2013.

33 T Shermer. Hiding people in polygons. *Computing*, 42(2):109–131, 1989.

34 Jeremy P. Spinrad and R. Sritharan. Algorithms for weakly triangulated graphs. *Discrete Applied Mathematics*, 59(2):181–191, 1995. doi:10.1016/0166-218X(93)E0161-Q.

35 Stanford University. Gendered Innovations.

36 Steven H Strogatz. Love affairs and differential equations. *Mathematics Magazine*, 61(1):35, 1988.

37 Subhash Suri. A linear-time algorithm for minimum link paths inside a simple polygon. *Computer Vision, Graphics and Image Processing*, 35(1):99–110, 1986.

38 Erica Tempesta. Stereotypes are made to be broken!, 2017.

39 Randolph Trumbach. *From 3 sexes to 4 genders in the making of modern culture.* Routledge, 1991.

40 Peter Walker. Men need lots of energy to lift ballet dancers because women are getting taller, 2017.

41 Wikipedia contributors. Strahler number, 2017.

42 Wikipedia contributors. Hall's marriage theorem, 2018.
43 Wikipedia contributors. Third gender, 2018.
44 Yuming Zou and Paul Black. Perfect binary tree. In Vreda Pieterse and Paul E. Black, editors, *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology, 2008.

A Maximum perfect subset of doors

This section extends the algorithm for perfect MIS in interval graphs (Section 4) to interval *catch digraphs*, thus solving the problem of picking genders' locker room doors (like in Section 2) in the perfect way (like in Section 4). The extension follows the same approach of merging "tight" sets up the tree, after appropriate modifications of definitions for span, being to the left, tightness, etc. The technical details follow.

We first recollect some definitions from Section 2. An *interval digraph* has two intervals (S_v, T_v) for each vertex v; two vertices u, v are connected if either $S_u \cap T_v \neq \emptyset$ or $S_v \cap T_u \neq \emptyset$ (or both). Strictly speaking, the graph is directed, but, similarly to almost all work on interval digraphs, we will look at the underlying undirected graph and often say simply "graph" for "digraph". An interval *nest* digraph has $S_v \subseteq T_v \, \forall v$; we call S_v and T_v the *inner* and the *outer* intervals, resp. An interval nest graph is an interval *catch* graph if S_v is a single point for all v. As usual, we will identify graph vertices with their intervals. The problem of picking maximum perfect subset of locker rooms reduces to the following: given an interval catch graph on the potential doors D, with $S_v = v, T_v = i(v) \, \forall v \in D$ (see Section 2 for notation), find a maximum independent set $L \subseteq D$ such that the shortest paths from s to L form a perfect tree (see Section 4 for terminology).

Our algorithm for picking a maximum perfect pathview-independent subset of locker room doors reuses two ideas from the algorithm of [30] for maximum clique in interval catch digraphs:

- instead of solving the problem for interval catch graphs, solve it for (more general) interval nest graphs
- if two vertices u, v of an interval nest graph are replaced by their "span", the span will be connected to those and only those vertices to which (at least) one of u, v was connected ([30, Theorem 3.6]; see also [18] for an application of the similar idea to finding maximum cliques in general weakly triangulated graphs).

We now define the span formally. Extending Definition 6 (Section 4) of the span of a set of segments, we obtain the span of two vertices in an interval nest digraph by taking the spans of both the inner and the outer intervals:

▶ **Definition 10.** The *span* of two vertices u, v consists of a pair of intervals (S, T) where S is the smallest interval containing S_u, S_v and T is the smallest interval containing T_u, T_v.

Clearly, if $S_v \subseteq T_v, S_u \subseteq T_u$, then $S \subseteq T$; thus, we can speak about the new interval nest graph in which u and v are replaced by their span (in [18, 30] such graph is denoted by $G(uv >> w)$ where G is the original graph and w is the vertex for the span of u, v). The span of more than two vertices is defined analogously, using associativity of taking the span. We retain all conventions from Section 4: a "set" will usually mean a perfect IS, two sets will generally have the same cardinality and be siblings in $\mathsf{SPT}(D)$, we will identify sets with their spans, and sometimes say "vertices" instead of "spans" – meaning that the spans are vertices in the interval nest graph (obtained by applying the $G(uv >> w)$ operation, possibly several times).

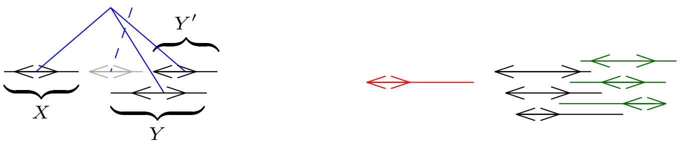


Figure 7 The inner intervals are represented by the arrows. Left: Y is tight w.r.t. X, but Y' is not; as in the proof of Lemma 7, if Y' were to be picked together with a set Z (grey) from another subtree, then $\mathsf{SPT}(D)$ (blue) would have had a crossing – to see this, ignore the outer intervals $T_X, T_{Y'}, T_Z$ and note that the paths to vertices in X, Y' and Z end inside $S_X, S_{Y'}, S_Z$ resp. (and for Z to be independent from both X and Y', S_Z must not overlap with any of $S_X, S_{Y'}$ – so S_Z would have to lie between S_X and $S_{Y'}$, implying the crossing). Right: Any black set is tight w.r.t. the red; no black sets "dominates" another black set because, depending on which green set is present, any black set may be the only one compatible with the green.

Analogously to merging intervals in Section 4, we merge (tight spans of) vertices up the tree $\mathsf{SPT}(D)$. The only difference is that for interval nest graphs the tightness is defined w.r.t. both inner and outer intervals. Specifically, let X and Y be two non-connected vertices, i.e., spelling out all our assumptions, two same-cardinality perfect independent sets such that $\mathsf{lca}(X)$, $\mathsf{lca}(Y)$ are siblings and $X \cup Y$ is a perfect IS (or two spans of such sets); assume w.l.o.g. that X is "to the left" of Y– i.e., T_X is to the left of S_Y and (hence) S_X is to the left of T_Y.

▶ **Definition 11.** Y is *tight* w.r.t. X if there is no perfect set Y' such that (Fig 7, left):

- Y' is in the same subtree ($\mathsf{lca}(Y') = \mathsf{lca}(Y)$);
- $|Y'| = |Y|$;
- $X \cup Y'$ is an IS;
- the right endpoint of $S_{Y'}$ is to the left of the right endpoint of S_Y, or the right endpoint of $T_{Y'}$ is to the left of the right endpoint of T_Y.

The first three items are the same as in the tightness definition for interval graphs (Definition 8 in Section 4), while the last is slightly more involved since for interval nest graphs we have to look at both the inner and outer intervals (Fig. 7, right).

As in Section 4, considering only tight unions is enough, since any feasible solution can be made recursively tight by the same (recursive) procedure as in the proof of Theorem 9 – replace any non-tight set Y with a set Y' that certifies non-tightness if Y; also as in the theorem's proof, our algorithm lists all recursively tight sets, as it goes through all possible tight sets at every node of $\mathsf{SPT}(D)$. The only difference is that for interval nest graphs, more tight sets are created because there may be more than one tight set "starting" at any vertex $v \in D$ (i.e., the set whose span's inner interval has left endpoint at $l(S_v)$ or whose span's outer interval has left endpoint at $l(T_v)$); refer to Fig. 7, right. Still, for every k, at any node of $\mathsf{SPT}(D)$ there are $O(|D|^4)$ spans – this is because a span is defined by 4 points (endpoints of the inner and outer intervals), each of which is either $v \in D$ or an endpoint of $i(v)$; thus there are $O(|D|^4)$ different spans overall.

Card-Based Zero-Knowledge Proof for Sudoku

Tatsuya Sasaki
Graduate School of Information Sciences, Tohoku University
6–3–09 Aramaki-Aza-Aoba, Aoba, Sendai 980–8579, Japan
tatsuya.sasaki.p2@dc.tohoku.ac.jp

Takaaki Mizuki
Cyberscience Center, Tohoku University
6–3 Aramaki-Aza-Aoba, Aoba, Sendai 980–8578, Japan
tm-paper+cardsudk@g-mail.tohoku-university.jp

Hideaki Sone
Cyberscience Center, Tohoku University
6–3 Aramaki-Aza-Aoba, Aoba, Sendai 980–8578, Japan

Abstract

In 2009, Gradwohl, Naor, Pinkas, and Rothblum proposed physical zero-knowledge proof protocols for Sudoku. That is, for a puzzle instance of Sudoku, their excellent protocols allow a prover to convince a verifier that there is a solution to the Sudoku puzzle and that he/she knows it, without revealing any information about the solution. The possible drawback is that the existing protocols have a soundness error with a non-zero probability or need special cards (such as scratch-off cards). Thus, in this study, we propose new protocols to perform zero-knowledge proof for Sudoku that use a normal deck of playing cards and have no soundness error. Our protocols can be easily implemented by humans with a reasonable number of playing cards.

2012 ACM Subject Classification Security and privacy → Information-theoretic techniques

Keywords and phrases Zero-knowledge proof, Card-based cryptography, Sudoku

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.29

Funding This work was supported by JSPS KAKENHI Grant Number JP17K00001.

Acknowledgements The authors would like to express their deepest appreciation to Kazumasa Shinagawa for helpful discussions during this work.

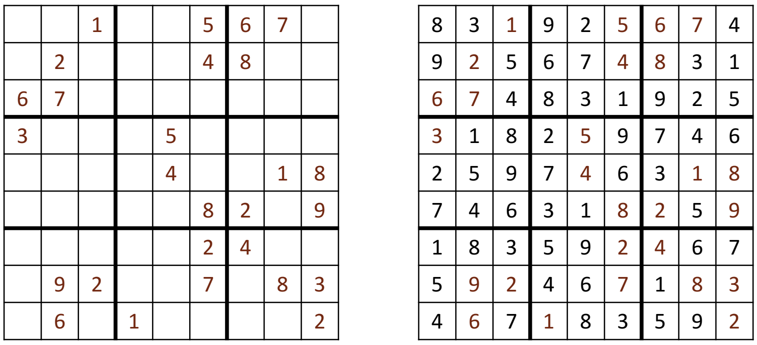
1 Introduction

Sudoku is one of the most famous puzzles. In a standard challenge, a 9×9 grid is used, which is divided into 3×3 subgrids. Some of the cells are already filled with numbers between 1 and 9. The goal of Sudoku is to fill all the empty cells with numbers so that each row, each column, and each subgrid contains all the numbers from 1 to 9. Figure 1 shows an example of a standard Sudoku challenge, and its solution.

We address a generalized version of Sudoku in this study. That is, a Sudoku puzzle where a grid is $n \times n$ cells, a subgrid is $k \times k$ cells, and numbers from 1 to n are used. Note that $n = k^2$; the standard size of a Sudoku puzzle corresponds to $n = 9$ and $k = 3$.

We solicit zero-knowledge proof protocols for Sudoku. That is, for a certain Sudoku puzzle, we assume a prover P who knows the solution to the Sudoku puzzle and a verifier V who does not know it, and suppose that P wants to convince V of the following without revealing any information about the solution.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 29; pp. 29:1–29:10
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



		1			5	6	7	
	2				4	8		
6	7							
3				5				
				4			1	8
					8	2		9
					2	4		
	9	2			7		8	3
	6		1					2

8	3	1	9	2	5	6	7	4
9	2	5	6	7	4	8	3	1
6	7	4	8	3	1	9	2	5
3	1	8	2	5	9	7	4	6
2	5	9	7	4	6	3	1	8
7	4	6	3	1	8	2	5	9
1	8	3	5	9	2	4	6	7
5	9	2	4	6	7	1	8	3
4	6	7	1	8	3	5	9	2

Figure 1 Example of the standard Sudoku challenge, and its solution.

- There is a solution to the puzzle;
- P knows the solution.

Unlike in conventional zero-knowledge proof (see, e.g., [4]), in our setting we do not want to use electronic devices such as computers. Instead, we want to use only everyday items to execute a protocol manually. The prover P and the verifier V are assumed to be in the same place. Such a restricted zero-knowledge proof is called a *physical zero-knowledge proof* [1, 2, 5].

In 2009, Gradwohl, Naor, Pinkas, and Rothblum proposed a few physical zero-knowledge proof protocols for Sudoku [5]. Among them, Protocol 3 (hereinafter referred to as *GNPR Protocol 3*) utilizes a deck of cards having numbers on their faces, such as playing cards[1]. This protocol needs $3n^2$ cards and has a soundness error with a non-zero probability. In contrast, Protocol 5 (hereinafter referred to as *GNPR Protocol 5*) avoids soundness error by utilizing special cards (namely, scratch-off cards that allow the colors to be covered) and scissors[2]. Unfortunately, GNPR Protocol 5 consumes non-reusable scratch-off cards at every execution of the protocol. Therefore, it is preferable to construct a protocol that can be implemented with only reusable everyday objects such as playing cards.

Thus, in this paper, we propose zero-knowledge proof protocols that satisfy the following: (i) they utilize the same items as GNPR Protocol 3, namely a standard deck of playing cards, (ii) they are implementable with fewer cards than GNPR Protocol 3, and (iii) they have no soundness error. The main idea behind our protocols is to apply techniques of card-based cryptography (see, e.g., [6], [8]). In particular, copy computation, which is an important primitive in the field of card-based cryptography, prevents the prover P from inputting incorrect numbers.

The remainder of this paper is organized as follows. In Section 2, we review zero-knowledge proof, GNPR Protocol 3, and GNPR Protocol 5. In Section 3, we present our proposed protocols. In Section 4, we compare our protocols with the existing ones, and conclude this paper.

2 Preliminaries

In this section, we first review zero-knowledge proof and then introduce two existing protocols, GNPR Protocols 3 and 5.

[1] Protocols 1 and 2 presented in [5] are conventional (non-physical) zero-knowledge proof protocols.
[2] Protocol 4 in [5] is a variation of GNPR Protocol 3.

2.1 Zero-Knowledge Proof

A zero-knowledge proof is an interactive proof between a prover P and a verifier V. They both have an instance of problem x and only P knows w, which is some information about a solution or a witness. The verifier V is computationally bounded so that V cannot obtain w from x. Under these assumptions, P wants to convince V that he/she knows w without revealing any information about w. Such a proof is called a *zero-knowledge proof*, which must satisfy the following three properties.

Completeness If P knows w, P is able to convince V.

Soundness If P does not know w, P cannot convince V (with a high probability).

Zero-knowledge V cannot obtain any information about w.

The probability that V will be convinced although P does not know w is called the *soundness error*. If we have a zero-knowledge proof protocol, the soundness error of which is $\delta > 0$, repeating the protocol ℓ times allows V to detect that P does not know w with a probability $1 - \delta^\ell$. Therefore, in general, even if the soundness error of a protocol is not 0, we can in practice establish zero-knowledge proof with a negligible soundness error by repeating the protocol. However, since we assume that a protocol is executed by human hands, it is impractical to repeat the protocol many times. Therefore, it is indispensable to design a protocol with no soundness error.

A zero-knowledge proof was first defined by Goldwasser, Micali, and Rackoff [4], and it was proved that (computational) zero-knowledge proofs exist for any NP problems [3]. Because it is known that Sudoku is NP-complete [9], we can construct conventional (computational) zero-knowledge proof protocols for it [5]. Remember, however, that this paper is focused not on a conventional zero-knowledge but on a physical zero-knowledge proof for Sudoku. Hence, we introduce the existing physical protocols, GNPR Protocols 3 and 5, in the following two subsections.

2.2 Gradwohl, Naor, Pinkas, and Rothblum Protocol 3

Here, we review GNPR Protocol 3 [5]. This protocol utilizes physical cards, the face side of each of which has one number between 1 and n, such as $\boxed{1}$ $\boxed{2}$ ... $\boxed{n}$; all the back sides are identical, for example, $\boxed{?}$ $\boxed{?}$ ... $\boxed{?}$. The protocol uses $3n$ sets of such n cards, namely, $3n^2$ cards in total.

Before presenting the protocol, we define a shuffle operation for cards. Given a sequence of ℓ cards $(c_1, c_2, c_3, ..., c_\ell)$, a *shuffle* results in a sequence

$$\left(c_{r^{-1}(1)}, c_{r^{-1}(2)}, c_{r^{-1}(3)}, ..., c_{r^{-1}(\ell)}\right),$$

where $r \in S_\ell$ is a uniformly random permutation and S_ℓ is the symmetric group of degree ℓ.

GNPR Protocol 3 proceeds as follows.

- The prover P places three face-down cards on each cell according to the Sudoku solution. On the filled-in cells, P places three face-up cards corresponding to the numbers filled in. After V confirms the values of the face-up cards, P turns them over.
- The verifier V picks one card randomly from each cell of a row to make a packet of n cards corresponding to the row. Because there are n rows, n packets are created. The same procedure is applied for each column and each subgrid. Thus, V makes $3n$ packets in total and passes them to P.
- P who received the packets from V applies a shuffle to the cards in each packet and returns the n shuffled packets to V.
- V opens all the cards in all the packets and checks that each packet contains all the numbers from 1 to n.

This is GNPR Protocol 3, which satisfies the three properties of zero-knowledge proof, as follows.

Completeness If P places the face-down cards correctly according to the solution, every packet made by V must contain all the numbers from 1 to n. By checking them, V is convinced that all the cards have been placed according to the solution. Furthermore, V is convinced that the packets are not a solution to another puzzle instance, because V sees the face-up cards corresponding to the values of the filled-in cells.

Soundness Consider a situation where V is convinced, in spite of an illegal input by P. Such a situation occurs when the three cards placed on each cell are not identical. The soundness error was shown to be at most 1/9 [5].

Zero-knowledge Assume a simulator S that simulates the conversation between P and V. Although S does not have any information about the witness w, S is allowed to replace packets with arbitrary packets. S acts as follows.

- The simulator S places three arbitrary face-down cards on each cell. On filled-in cells, S places three face-up cards, according to the filled-in cells. After V confirms the values, S turns them over.
- V makes $3n$ packets using the same procedure as in GNPR Protocol 3 and passes them to S.
- S shuffles the cards in each packet. Before passing the packets to V, S replaces them all by new ones, each of which contains all the cards numbered from 1 to n.
- V opens all the packets and checks that each packet contains all the numbers from 1 to n.

Since the conversation of S is indistinguishable from that of P, the protocol satisfies the zero-knowledge property.

In this protocol, P places three cards on each cell, and hence, the protocol uses $3n^2$ cards in total. For example, in the case of a Sudoku puzzle consisting of a 9 × 9 grid, the protocol needs 243 cards. Because a physical zero-knowledge proof protocol is supposed to be executed by human hands, it is preferable that the number of cards used in a protocol is as small as possible. In addition, as mentioned in Section 2.1, a protocol with no soundness error is also preferable.

2.3 Gradwohl, Naor, Pinkas, and Rothblum Protocol 5

As mentioned in the previous subsection, a soundness error during an execution of GNPR Protocol 3 would occur if the prover P did not place three identical cards on a cell. Therefore, if we could guarantee that all three cards placed on each cell are identical, a soundness error would never occur. This can be realized by using the following special scratch-off cards [5].

Consider scratch-off cards that cover any one of n colors. Assume that P and V agree on a one-to-one correspondence from a color to a Sudoku number. Suppose that only P knows which scratch-off card covers which color. Under these assumptions, P places such a scratch-off card on each cell according to the Sudoku solution. Next, V cuts every scratch-off card into three pieces with scissors so that they have three small cards having the same shape and size. Because the verifier V cuts the cards him/herself, it is possible to guarantee that the three small obtained cards are identical and cover the same color. Thus, scratch-off cards and scissors provide a protocol with no soundness error; this is GNPR Protocol 5 [5].

However, such scratch-off cards do not seem to be ordinary everyday items, and in addition, they are non-reusable. Therefore, in the next section, we propose a method to guarantee that the three cards placed on each cell are identical without any use of special cards.

3 Our Protocols

In this section, we propose efficient zero-knowledge proof protocols for Sudoku with no soundness error in which card-based cryptography perspectives are applied. Our protocols utilize the same type of cards as GNPR Protocol 3, but require fewer cards. We first design a fundamental protocol, and then modify it to attain more efficient protocols.

The outline of our fundamental protocol is as follows.

- P places exactly one face-down card on each cell corresponding to the solution (as seen in Section 3.1).
- V checks that the format of the packet of face-down cards placed on each subgrid is correct while making two identical copies of the packet (as seen in Section 3.2), which will be used for verifying rows and columns.
- V verifies that each row and each column contains all the numbers from 1 to n (as seen in Sections 3.3 and 3.4).

In Section 3.5, we show that our protocols satisfy the zero-knowledge proof properties.

3.1 Commitment as Input

In our protocols, the prover P places a single face-down card on each cell according to the solution. On filled-in cells, P places face-up cards. After V confirms the value of the face-up cards, P turns them over. Now, there are exactly n^2 cards placed on the grid. We call a sequence of n face-down cards corresponding to each subgrid a *commitment*.

For example, in the case of the top-left subgrid in Figure 1, P places nine cards according to the solution; after V confirms the value of the face-up cards, P turns them over:

$$\begin{array}{ccc} \boxed{?} & \boxed{?} & \boxed{1} \\ \boxed{?} & \boxed{2} & \boxed{?} \\ \boxed{6} & \boxed{7} & \boxed{?} \end{array} \quad \rightarrow \quad \begin{array}{ccc} \boxed{?} & \boxed{?} & \boxed{?} \\ \boxed{?} & \boxed{?} & \boxed{?} \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \; .$$

This is a commitment corresponding to this subgrid, and we regard it as a sequence:

$$\overbrace{\boxed{?}\;\boxed{?}\;\boxed{?}}^{\text{1st row}}\;\overbrace{\boxed{?}\;\boxed{?}\;\boxed{?}}^{\text{2nd row}}\;\overbrace{\boxed{?}\;\boxed{?}\;\boxed{?}}^{\text{3rd row}} \; .$$

Thus, P and V generate n commitments corresponding to n subgrids.

3.2 Subgrid Copy

After n commitments, each of which corresponds to a subgrid, have been generated, P and V want to copy these commitments to verify each row and each column. In addition, they also want to make sure that each commitment contains all the numbers from 1 to n. Therefore, in this subsection, we propose a method to verify that the format of a given commitment is correct, which involves making two copied commitments.

To this end, we first introduce a well-known shuffle operation called *pile-scramble shuffle* [7]. Assume that there are m piles, each of which consists of the same number of face-down cards; we denote this by

$$(pile_1, pile_2, pile_3, ..., pile_m).$$

For such a sequence of piles, applying a pile-scramble shuffle results in

$$(pile_{r^{-1}(1)}, pile_{r^{-1}(2)}, pile_{r^{-1}(3)}, ..., pile_{r^{-1}(m)}),$$

where $r \in S_m$ is a uniformly distributed random permutation. A pile-scramble shuffle can be implemented with the help of clips, envelopes, or similar items.

We now borrow two existing ideas: (i) a method for regarding a commitment as a permutation and a technique for inverting a permutation, which were given by Hashimoto, Shinagawa, Nuida, Inamura, and Hanaoka [6], and (ii) a technique for checking the format of a sequence of face-down cards, which was given by Mizuki and Shizuya [8]. That is, we regard a commitment consisting of n cards as a permutation $v \in S_n$:

$$\boxed{?}\ \boxed{?}\ \boxed{?}\ \cdots\ \boxed{?} \quad (v),$$

where a card having number i, $1 \leq i \leq n$, on its face side is placed at the $v(i)$-th position, and a permutation with parentheses, such as (v), means that the permutation is hidden (because the cards are face-down). Given a commitment to a permutation $v \in S_n$, we construct a method to check whether the commitment consists of all the numbers from 1 to n that involves making two identical copied commitments. We call this method *subgrid copy* and it operates as follows.

1. V puts n cards numbered from 1 to n in this order to generate a card sequence corresponding to the identity permutation id under the commitment to v:

$$\begin{array}{lllllc} \boxed{?} & \boxed{?} & \boxed{?} & \cdots & \boxed{?} & (v) \\ \boxed{1} & \boxed{2} & \boxed{3} & \cdots & \boxed{n} & \mathsf{id}\ . \end{array}$$

2. V turns over all the face-up cards in the bottom row and stacks the cards in each column so that there are n two-card piles:

$$\begin{array}{c|c|c|c} \boxed{?} & \boxed{?} & & \boxed{?} \\ \boxed{?} & \boxed{?} & \cdots & \boxed{?} \end{array} \quad \begin{array}{c} (v) \\ \mathsf{id}\ . \end{array}$$

   P applies a pile-scramble shuffle to them and obtains a commitment to $rv \in S_n$ and a commitment to $r \in S_n$, where $r \in S_n$ is a uniformly distributed random permutation:

$$\left[\ \begin{array}{c|c|c|c} \boxed{?} & \boxed{?} & & \boxed{?} \\ \boxed{?} & \boxed{?} & \cdots & \boxed{?} \end{array}\ \right] \rightarrow \begin{array}{lllll} \boxed{?} & \boxed{?} & \cdots & \boxed{?} & (rv) \\ \boxed{?} & \boxed{?} & \cdots & \boxed{?} & (r)\ . \end{array}$$

3. V turns over all the cards in the top row and checks the opened cards. If there are all cards numbered from 1 to n, V is convinced that the face-down cards placed by P on the subgrid are compatible with the puzzle solution. Since V learns only the value of rv, which is also a random permutation, no information about v leaks.
4. P sorts the n columns so that the top row becomes id. This means that a permutation $(rv)^{-1}$ is multiplied to each row, and hence, the bottom row becomes a commitment to $(rv)^{-1}r = v^{-1}$, i.e., the inverse of v:

$$\begin{array}{lllllc} \boxed{1} & \boxed{2} & \boxed{3} & \cdots & \boxed{n} & \mathsf{id} \\ \boxed{?} & \boxed{?} & \boxed{?} & \cdots & \boxed{?} & (v^{-1}). \end{array}$$

5. From now on, we make two copied commitments to v. V places two identity permutations id under the commitment to v^{-1};

$$\begin{array}{cccccc} \boxed{?} & \boxed{?} & \boxed{?} & \ldots & \boxed{?} & (v^{-1}) \\ \boxed{1} & \boxed{2} & \boxed{3} & \ldots & \boxed{n} & \mathsf{id} \\ \boxed{1} & \boxed{2} & \boxed{3} & \ldots & \boxed{n} & \mathsf{id} \ . \end{array}$$

6. By applying a procedure similar to Steps 2 and 3, the bottom-most two rows become commitments to v:

$$\begin{array}{cccccc} \boxed{1} & \boxed{2} & \boxed{3} & \ldots & \boxed{n} & \mathsf{id} \\ \boxed{?} & \boxed{?} & \boxed{?} & \ldots & \boxed{?} & (v) \\ \boxed{?} & \boxed{?} & \boxed{?} & \ldots & \boxed{?} & (v) \ . \end{array}$$

Thus, we can verify that a given commitment corresponding to a subgrid contains all the numbers from 1 to n, while making two copied commitments. This requires $2n$ cards in addition to the input commitment. The copied commitments are used for verifying rows and columns, as we describe in the complete protocol in the next subsection.

3.3 Fundamental Protocol

We are now ready to describe our fundamental protocol. The protocol proceeds as follows.

1. P places a commitment on each subgrid (as already described in Section 3.1).
2. V and P apply the subgrid copy (as explained in Section 3.2) to all subgrids. Then, there are two cards on each cell of the grid. Note that the verification that every commitment contains all cards numbered from 1 to n has been completed.
3. As in a similar way to GNPR Protocol 3, P makes $2n$ packets corresponding to n rows and n columns. Each packet is shuffled.
4. V opens all the packets and checks that each packet includes all the numbers from 1 to n.

Let us count how many cards we use in this protocol. Immediately before applying the subgrid copy to the final subgrid, there are $2(n^2 - n) + n$ cards on the grid. To apply the subgrid copy to the n-th subgrid, we need $2n$ more additional cards, and hence, we need $2n^2 + n$ cards in total. This is the maximum number of required cards during any execution. Therefore, the protocol requires $2n^2 + n$ cards.

3.4 Compact Protocol

In the fundamental protocol presented in Section 3.3, a subgrid copy was applied to all the subgrids before the verifications of each row and each column were performed. However, we do not have to wait until all the copy actions of n subgrids are complete; when verification of rows or columns becomes applicable, we can stop the subgrid copy action, and instead, start to verify rows or columns so that we have reusable opened cards, and consequently, it is possible to reduce the number of required cards. Thus, we have a compact protocol, as follows.

As mentioned in Section 1, an $n \times n$ grid of Sudoku can be regarded as a subgrid matrix of $k \times k$. We refer to such rows and columns of subgrids as subgrid-rows and subgrid-columns, respectively. The protocol proceeds as follows.

1. P places a commitment for each subgrid (as explained in Section 3.1).
2. V and P apply subgrid copy (as explained in Section 3.2) to all the subgrids in the first subgrid-row.
3. After Step 2, two cards are placed on each cell in the first subgrid-row. V verifies that each of the first k rows (which constitute the first subgrid-row) contains all the numbers from 1 to n. After the verification is complete, there is one card on every cell.
4. P and V repeat the same procedure as Steps 2 and 3 for every subgrid-row from the second to the (k-1)-th.
5. In the k-th subgrid-row, V and P will operate a similar procedure to verify the columns. First, the subgrid copy action is applied to the first subgrid in the k-th subgrid-row. Then, P and V verify the first k columns.
6. P and V repeat the same procedure as Step 5 for every subgrid-column from the second to the k-th.
7. Finally, P and V verify the rows in the k-th subgrid-row, so that the verification is complete for all rows, columns, and subgrids.

Let us consider at which point the number of used cards becomes largest. It is when the subgrid copy is applied to the last subgrid in Step 4, and at that point we use $n^2 + (k+1)n$ cards. Therefore, this compact protocol requires $n^2 + (k+1)n$ cards.

3.5 Correctness of Proposed Protocols

In this subsection, we show that our protocols proposed in Sections 3.3 and 3.4 satisfy the properties of zero-knowledge proof.

Completeness A prover P who knows the solution can place cards so that each row, each column, and each subgrid contains all the numbers. Whether the format of each subgrid is correct can be checked by using the subgrid copy, and whether the format of each row and each column is correct can be checked by using the copied commitments. Further, V is convinced that P's input is not a solution to another problem by comparing the face-up cards and the corresponding value of the filled-in cells.

Soundness Since P and V use copied commitments, it is guaranteed that the cards placed on each cell are identical. Therefore, the protocol has no soundness error.

Zero-Knowledge Assume a simulator that simulates the conversation as in Section 2.2. When verifying each row and each column, information about knowledge w does not leak for the same reason as in GNPR Protocol 3. Thus, it is sufficient to show the zero-knowledge property of the subgrid copy.

- The simulator S places one arbitrary face-down card on each cell. On filled-in cells, S places face-up cards. After V confirms them, S turns them over.
- V makes a sequence of n piles using the same procedure as Step 2 in Section 3.2, and passes them to S.
- S replaces the sequence by the following two identity permutations id. S applies a pile-scramble shuffle to the replaced sequence, and passes it to V.

$$\boxed{?}\ \boxed{?}\ \boxed{?}\ \ldots\ \boxed{?}\quad (\mathsf{id})$$

$$\boxed{?}\ \boxed{?}\ \boxed{?}\ \ldots\ \boxed{?}\quad (\mathsf{id})$$

- V opens the cards in the top row, and checks whether it contains all the numbers from 1 to n. Then, V operates the same procedure as described in Section 3.2, and outputs the bottom row.

- V operates Step 5 in Section 3.2, makes n packets, and passes them to S.
- S applies a pile-scramble shuffle to the sequence and passes it to V. In this case, S does not need to replace packets.

Since the conversation of S is indistinguishable from that of P, the protocol satisfies the zero-knowledge property.

4 Conclusion

In this paper, we proposed two card-based zero-knowledge proof protocols for Sudoku. We now compare our protocols with the existing protocols, GNPR Protocols 3 and 5, in terms of the number of cards, the number of shuffles, and the soundness error. Table 1 shows the performance of the protocols.

Our fundamental protocol and our compact protocol use $2n^2+n$ and $n^2+(k+1)n$ cards, respectively, as described in Sections 3.3 and 3.4. Let us count the number of shuffles in our proposed protocols. The subgrid copy procedure requires two shuffles, and this procedure is performed for each of n subgrids. The packet shuffle is performed once in the verification of each row and each column, and their number is $2n$. Therefore, the total number of shuffles is $2n+2n=4n$. Furthermore, as shown previously, our protocols have no soundness error. See Table 1 again.

In Sudoku's standard size $n=9$, the compact protocol can be implemented with less than half the number of cards used in GNPR Protocol 3 (117 versus 241). Further, when GNPR Protocol 3 is executed more than once, the number of shuffles is larger than that in our protocol. As compared to GNPR Protocol 5, the number of shuffles in our protocol is larger; however, in our opinion, a protocol that uses no special cards is superior.

Finally, we attempt to reduce the number of cards and shuffles further (but it looks crafty).

Further reduction of the number of cards

Thus far, we assumed that P places his/her inputs on all the cells simultaneously. If we allow P to input at multiple timings, it is possible to construct a protocol with fewer cards. The outline is as follows.

1. P places a commitment on one subgrid.
2. P and V apply the subgrid copy action to the subgrid.
3. P and V apply Steps 1 and 2 also to the other subgrids, and, as in the compact protocol, perform verification when it becomes possible to verify the rows and columns.

This protocol uses n^2+n cards. For example, when $n=9$, the number of required cards is 90.

Reduction of the number of shuffles[3]

In the subgrid copy action, P performs a pile-scramble shuffle twice; the output commitment obtained by the first pile-scramble shuffle is an inverse of v, and hence, P needs to shuffle again to obtain a commitment v. However, if P places a commitment of an inverse as input, P can omit one pile-scramble shuffle.

The performance of this crafty protocol is also shown in Table 1.

In the literature, for several puzzles other than Sudoku, physical zero-knowledge proof protocols have been proposed [1, 2]. Therefore, interesting future work is to design more efficient zero-knowledge proof protocols for those puzzles with the help of card-based cryptography.

[3] The idea was introduced by Kazumasa Shinagawa.

Table 1 Comparison of protocols

	# of cards	# of shuffles	Soundness error
GNPR Protocol 3	$3n^2$	$3n \times \ell$	at most $(\frac{1}{9})^\ell$
GNPR Protocol 5	n^2 (special cards)	$3n$	0
Fundamental Protocol	$2n^2 + n$	$4n$	0
Compact Protocol	$n^2 + (k+1)n$	$4n$	0
Crafty Protocol	$n^2 + n$	$3n$	0

References

1 Xavier Bultel, Jannik Dreier, Jean-Guillaume Dumas, and Pascal Lafourcade. Physical zero-knowledge proofs for akari, takuzu, kakuro and kenken. In Erik D. Demaine and Fabrizio Grandoni, editors, *8th International Conference on Fun with Algorithms, FUN 2016, June 8-10, 2016, La Maddalena, Italy*, volume 49 of *LIPIcs*, pages 8:1–8:20. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016. doi:10.4230/LIPIcs.FUN.2016.8.

2 Yu-Feng Chien and Wing-Kai Hon. Cryptographic and physical zero-knowledge proof: From sudoku to nonogram. In Paolo Boldi and Luisa Gargano, editors, *Fun with Algorithms, 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings*, volume 6099 of *Lecture Notes in Computer Science*, pages 102–112. Springer, 2010. doi:10.1007/978-3-642-13122-6_12.

3 Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991. doi:10.1145/116825.116852.

4 Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM J. Comput.*, 18(1):186–208, 1989. doi:10.1137/0218012.

5 Ronen Gradwohl, Moni Naor, Benny Pinkas, and Guy N. Rothblum. Cryptographic and physical zero-knowledge proof systems for solutions of sudoku puzzles. *Theory Comput. Syst.*, 44(2):245–268, 2009. doi:10.1007/s00224-008-9119-9.

6 Yuji Hashimoto, Kazumasa Shinagawa, Koji Nuida, Masaki Inamura, and Goichiro Hanaoka. Secure grouping protocol using a deck of cards. In Junji Shikata, editor, *Information Theoretic Security - 10th International Conference, ICITS 2017, Hong Kong, China, November 29 - December 2, 2017, Proceedings*, volume 10681 of *Lecture Notes in Computer Science*, pages 135–152. Springer, 2017. doi:10.1007/978-3-319-72089-0_8.

7 Rie Ishikawa, Eikoh Chida, and Takaaki Mizuki. Efficient card-based protocols for generating a hidden random permutation without fixed points. In Cristian S. Calude and Michael J. Dinneen, editors, *Unconventional Computation and Natural Computation - 14th International Conference, UCNC 2015, Auckland, New Zealand, August 30 - September 3, 2015, Proceedings*, volume 9252 of *Lecture Notes in Computer Science*, pages 215–226. Springer, 2015. doi:10.1007/978-3-319-21819-9_16.

8 Takaaki Mizuki and Hiroki Shizuya. Practical card-based cryptography. In Alfredo Ferro, Fabrizio Luccio, and Peter Widmayer, editors, *Fun with Algorithms - 7th International Conference, FUN 2014, Lipari Island, Sicily, Italy, July 1-3, 2014. Proceedings*, volume 8496 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2014. doi:10.1007/978-3-319-07890-8_27.

9 Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions*, 86-A(5):1052–1060, 2003.

The Complexity of Escaping Labyrinths and Enchanted Forests

Florian D. Schwahn[1]
Department of Mathematics, University of Kaiserslautern, Paul-Ehrlich-Str. 14,
D-67663 Kaiserslautern, Germany
fschwahn@mathematik.uni-kl.de

Clemens Thielen
Department of Mathematics, University of Kaiserslautern, Paul-Ehrlich-Str. 14,
D-67663 Kaiserslautern, Germany
thielen@mathematik.uni-kl.de
https://orcid.org/0000-0003-0897-3571

Abstract

The board games *The aMAZEing Labyrinth* (or simply *Labyrinth* for short) and *Enchanted Forest* published by Ravensburger are seemingly simple family games.

In *Labyrinth*, the players move though a labyrinth in order to collect specific items. To do so, they shift the tiles making up the labyrinth in order to open up new paths (and, at the same time, close paths for their opponents). We show that, even without any opponents, determining a shortest path (i.e., a path using the minimum possible number of turns) to the next desired item in the labyrinth is strongly NP-hard. Moreover, we show that, when competing with another player, deciding whether there exists a strategy that guarantees to reach one's next item faster than one's opponent is PSPACE-hard.

In *Enchanted Forest*, items are hidden under specific trees and the objective of the players is to report their locations to the king in his castle. Movements are performed by rolling two dice, resulting in two numbers of fields one has to move, where each of the two movements must be executed consecutively in one direction (but the player can choose the order in which the two movements are performed). Here, we provide an efficient polynomial-time algorithm for computing a shortest path between two fields on the board for a given sequence of die rolls, which also has implications for the complexity of problems the players face in the game when future die rolls are unknown.

2012 ACM Subject Classification Theory of computation → Shortest paths, Theory of computation → Representations of games and their complexity

Keywords and phrases board games, combinatorial game theory, computational complexity

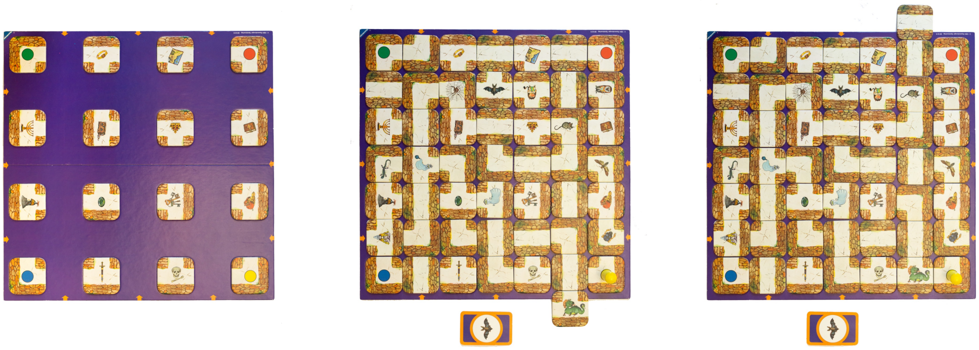
Digital Object Identifier 10.4230/LIPIcs.FUN.2018.30

1 Introduction

Computational complexity questions related to games and puzzles have received considerable interest among mathematicians and computer scientists within the last decades. For an introduction to the topic and an overview of known results, we refer to [1, 3, 5]. While many one-player puzzles are NP-complete, two-player games often turn out to be PSPACE-complete or even EXPTIME-complete.

[1] The work of this author was partially supported by Anne M. Schwahn by explaining the two studied board games to him back in the dark winters of 1990 and 1991.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 30; pp. 30:1–30:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) The empty board. **(b)** Before the shift. **(c)** After the shift.

Figure 1 The aMAZEing board.

In this paper, we study the computational complexity of several natural decision problems arising in the two board games *(The aMAZEing) Labyrinth* [6] and *Enchanted Forest* [8]. In both games, the players move on a board subject to specific movement rules in order to find certain items. In Labyrinth, this involves shifting the moving tiles on the board in order to open up paths through the labyrinth. In Enchanted Forest, movement is performed by rolling two dice, resulting in two numbers of fields the player has to move subject to the constraint that each of the two movements must be executed consecutively in one direction.

In the following sections, we first consider Labyrinth and show that – even without any opponents – deciding whether a given tile on the board is reachable within a given number of turns is strongly NP-complete. When competing with another player, the natural extension of this shortest path problem that asks whether a player has a strategy that guarantees to reach a given target tile faster than her opponent reaches their (possibly different) target tile is shown to be PSPACE-hard. For Enchanted Forest, on the other hand, we provide an efficient polynomial-time algorithm for deciding whether a given field on the board can be reached in a given number of turns for a given sequence of die rolls.

2 The aMAZEing Labyrinth

The game *(The aMAZEing) Labyrinth*, developed by Max J. Kobbert, was originally published by Ravensburger in 1986 under the German title “Das verrückte Labyrinth” [6] (“verrückt” is a play on two possible meanings, similar to *disarranged*). The game is a huge success all over the world with about 30 million sold units in over 60 countries so far.

In the game, one to four players[2] try to collect a sequence of items on a board consisting of moving tiles. The board (see Figure 1 (a)) features spots for 7 × 7 square tiles, with some of those fixed to it. The tiles have three different shapes, which can be rotated in two/four orientations: *I*-tile (│, ─), *L*-tile (└, ┌, ┐, ┘), *T*-tile (┬, ┤, ┴, ├).

The board is randomly filled with the movable tiles and one additional tile is placed aside. In a player’s turn, she has to execute a *shift* and a *move* action (see Figures 1 (b) and (c)).

[2] This refers to the original release of the game, where playing alone was actually allowed. Now, it is sold as a game for *two* to four players.

Shift: The (mandatory) shift action is performed by pushing in the surplus tile - in any orientation - from any border, such that all tiles in this row/column are shifted by one place and the border tile on the other end of the row/column is pushed out of the board (obviously, the rows and columns containing fixed tiles cannot be shifted). This tile is left in the place in order to mark the last shift action for the next player, who is not allowed to directly reverse the previous shift. If some player is standing on the tile that is pushed out of the board, the player is instead placed on the tile that has just been pushed in (*wrap-around rule*).

Move: After shifting, a player may either move to any tile reachable in the labyrinth (e.g. the requested bat tile in Figure 1 (c)), or choose not to move at all. An adjacent tile is (directly) reachable if both tiles feature open space on their common edge.

Goal: Some tiles feature the symbol of an item, animal, or mythical creature. At the beginning of the game, each player gets a stack of cards requesting to collect some of those objects. At each point in time, each player only knows about the object she is requested to collect next, but not about the further objects she has to collect. The currently requested object is collected if the move action of the player ends on the tile featuring this object. She may then look at her next card, showing the next object to collect. Once a player collects the last object requested from her, the last goal is to return to her starting tile. Whoever is the first to accomplish this is the winner of the game.

2.1 Formal Problem Definition

In order to analyze the game mathematically, we consider a board of arbitrary size. Hence, the rectangle board contains $a \times b$ *spots* for $(a \cdot b) + 1$ *tiles*, some of which may be fixed. The kinds of tiles considered and the rules of the game are as described above.

At each point in time during the game, a player only knows about the next object she is requested to collect. Hence, the fundamental problem faced at each point in time when a player plays alone is reaching the next object (or her starting tile in case that she has already collected the last object) in the minimum number of turns. The decision version of this single-player problem is formally defined as follows:

▶ **Definition 1** (SP-Labyrinth).

INSTANCE: The initial board setting, the shape of the current surplus tile, two distinct tiles s and t on the board, and an allowed number of turns $k \in \mathbb{N}$.

QUESTION: Can a single player starting at tile s reach tile t in at most k turns?

When two opposing players play alternately, the fundamental problem each player faces amounts to reaching the object she is requested to collect next (or her starting tile if she has already collected all required objects) faster than her opponent reaches their next object (or starting tile). The decision version of the two-player problem is, thus, defined as follows:

▶ **Definition 2** (SP-Versus-Labyrinth).

INSTANCE: The initial board setting, the shape of the current surplus tile, and two pairs (s_1, t_1), (s_2, t_2) of tiles on the board.

QUESTION: Is there a strategy for player 1 (who has the first turn and starts at tile s_1) that allows her to reach tile t_1 before player 2 (starting at tile s_2) can reach tile t_2?

In the following subsections, we first show that already the single-player problem SP-Labyrinth in strongly NP-complete (this problem has already been used as a benchmark problem for testing ASP solvers, cf. [2]). Afterwards, we consider the two-player problem SP-Versus-Labyrinth and show that this problem is PSPACE-hard.

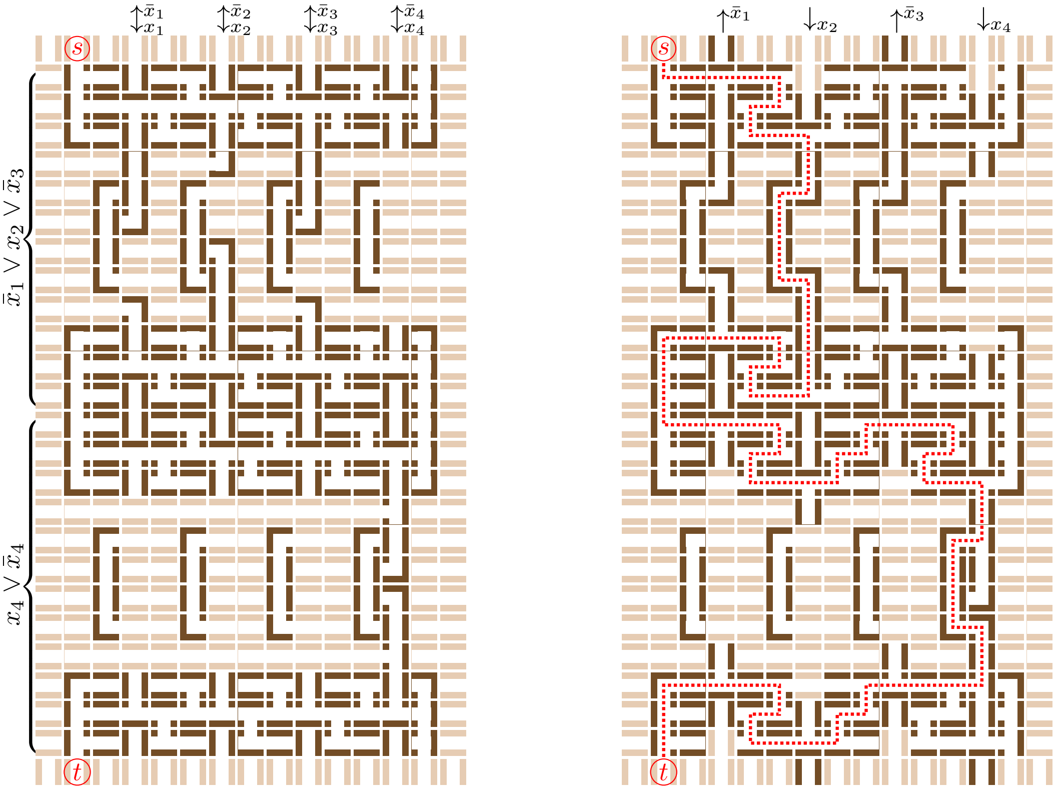


Figure 2 Example of a clause and a variable gadget (initial and final board setting).

2.2 Escaping the Labyrinth is hard . . .

Poor little Alice got lost in the aMAZEing Labyrinth and has to find her way to the exit before the batteries of her flashlight run out. Unfortunately (for her) we now show that deciding whether she can get out in time is strongly NP-complete:

▶ **Theorem 3.** *SP-Labyrinth is strongly NP-complete.*

Proof. Since it is straightforward to check that the player reaches t from s with a given sequence of shifts and moves, the problem is clearly contained in NP.

To show NP-hardness, we use a reduction from 3SAT (a problem that Alice has probably encountered when reading [4]). Given an instance $\mathcal{I}$ of 3SAT with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, we construct an instance of SP-Labyrinth in which the player can reach t in at most $k := n$ turns if and only if $\mathcal{I}$ is satisfiable.[3]

Figure 2 shows an example of a clause and a variable gadget we use in this reduction. In total, there is one gadget for each variable and one gadget for each clause, i.e., there are $n+m$ gadgets with twelve rows each. Together with the top row containing s and the bottom row containing t, this makes $12 \cdot (n+m)+2$ rows. To reach t from s, the player needs to move through or get shifted through the $12 \cdot (n+m)$ rows between s and t. As each shift of the avatar itself can only contribute a single row, she has to move along vertical paths to reach t fast enough. To avoid problems with the wrap-around rule, we augment the board on

[3] This also shows that it is sometimes helpful to have a SAT solver with you when entering a labyrinth.

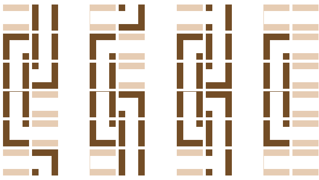


Figure 3 The four hook gadgets (negated, unnegated, either-or, empty).

each side by $2n$ columns / rows that are filled with ▌▐-tiles and ☰-tiles in alternating order. Thus, none of the central tiles shown in Figure 2 can be shifted out of the board and the player cannot reach any boundary of the augmented board in n turns. Moreover, even with those additional tiles, the resulting board has size polynomial in n and m, so the instance can be constructed in polynomial time.

The complete board setting includes one gadget for each clause and one gadget for each variable. The gadgets vary only in the special variable columns (marked with $\updownarrow^{\bar{x}_i}_{x_i}$). Here, shifting the variable column corresponding to a variable x_i downwards will correspond to setting x_i to `true`, while shifting the column upwards will correspond to setting x_i to `false`.[4] The connection to the clauses is made by using the four different kinds of *hook gadgets* shown in Figure 3. Each clause gadget contains the corresponding hook gadget for each variable (depending on whether the variable is contained in the clause negated, unnegated, both negated and unnegated, or not at all). For example, the clause gadget for $\bar{x}_1 \vee x_2 \vee \bar{x}_3$ presented in Figure 2 contains a negated hook gadget for x_1 and x_3, and unnegated hook gadget for x_2, and an empty hook gadget for x_4 (since x_4 is not contained in the clause). As one can see, the clause gadget can be crossed if and only if the variable column corresponding to either x_1 or x_3 is shifted upwards, or the variable column corresponding to x_2 is shifted downwards. Since x_4 is not contained in the clause, shifting the corresponding variable column cannot make the gadget crossable.

The first and last three rows of every clause gadget (above and below the hook gadgets) ensure that, starting from the top left of the gadget, we can choose to cross by any variable column (given that the corresponding variable was set to satisfy the clause) and reach the bottom left of the gadget in order to enter the next gadget.

Below the clause gadgets, there is an analogously constructed gadget for each variable (i.e., the gadget for variable x_i corresponds to a clause gadget for the clause $x_i \vee \bar{x}_i$, see Figure 2). Thus, the variable gadget corresponding to variable x_i will be crossable if and only if the column of variable x_i was shifted either upwards or downwards. Hence, all variable gadgets will be crossable if and only if each variable was set to either `true` or `false`.

We now show that the constructed instance is equivalent to the given instance $\mathcal{I}$ of 3SAT, i.e., that the player can reach t in at most n turns if and only if $\mathcal{I}$ is satisfiable.

First assume that $\mathcal{I}$ is satisfiable. Then, shifting the variable columns according to some satisfying variable assignment yields a path from s to t through which the player can move after the n-th shift (in the previous $n-1$ turns, the player does not move at all). Hence, the player can reach t in at most n turns.

[4] Instead, one could equivalently shift the column to the left of the variable column in the opposite direction, which has the same effect (but could be prevented by putting a fixed tile on top of this column). In the following, we will assume that the variable column itself is always shifted instead of the column to its left.

In order to prove the other direction, we observe that, in every solution to the constructed SP-Labyrinth instance, each variable column must be shifted exactly once. This follows since the corresponding variable gadget is not crossable in its initial form and the only possible way to make it crossable by using only a single (row or column) shift is to shift the corresponding variable column exactly once (as there are n variable gadgets in total and only n shifts are available, we cannot use more than one shift to make a single variable gadget crossable). Hence, each solution to the constructed SP-Labyrinth instance corresponds to a truth-assignment for the variables in the given 3SAT-instance $\mathcal{I}$ by setting x_i to `true` (`false`) if and only if the variable column corresponding to x_i is shifted downwards (upwards). Moreover, since all clause gadgets are crossable in the solution to the SP-Labyrinth instance, all clauses in $\mathcal{I}$ must be satisfied by this truth-assignment. ◀

2.3 ... but doing it faster than someone else is even harder.[5]

The famous archaeologists Lara and Henry Jr. play a game of the aMAZEing Labyrinth and each of them only needs to return to their starting tile. It is Lara's turn – but will she manage to arrive first and, thus, win the game? Who will resort to a destructive strategy? Does it pay off?

▶ **Theorem 4.** *SP-Versus-Labyrinth is PSPACE-hard.*

Proof. We use a reduction from *Quantified Satisfiability* (QSAT) (also known as *Quantified Boolean Formula*, cf. [9]). Given an instance $\mathcal{I}$ of QSAT with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$, we construct an instance of SP-Versus-Labyrinth in which player 1 (Lara) has a winning strategy if and only if $\mathcal{I}$ is a yes-instance.[6] Without loss of generality, we assume that the number n of variables in $\mathcal{I}$ is odd.

This time, we translate "setting variables" into "shifting variable *rows*" by using similar clause gadgets as in the proof of Theorem 3 (rotated clockwise by 90 degrees). However, we now use only a single variable gadget corresponding to variable x_n - the other variables have no corresponding variable gadgets. Instead, the board contains a distinct order preserving gadget for each player, which is used to make sure that the player has to set her corresponding variables x_i (the ones with odd i for Lara and the ones with even i for Henry) in the correct order. In between each pair of adjacent gadgets and at the sides of the board, we need some buffer columns to avoid any unintentional interaction of the gadgets as well as using the wrap-around rule. Since we will show that at least one of the players will always reach their goal after at most $n+1$ turns, $n+2$ buffer columns consisting of alternating ║ and ═-tiles are sufficient between each pair of adjacent gadgets and at the left and the right of the board.[7]

In order to reach her goal, Lara has to cross her order gadget as well as the clause gadgets, whereas Henry only has to cross his order gadget and the variable gadget of variable x_n (but *not* the clause gadgets). An overview of the structure of the board is given in Figure 4. The pink row at the top in Figure 4 is used in order to connect the different gadgets and will be referred to as the *wiring row* throughout the rest of the proof. The blue row at the bottom is used as an *escape path* that makes a player reach her goal directly in case that the other

[5] Assuming NP $\neq$ PSPACE.

[6] This shows that, as expected, beating an archaeologist as intelligent as Henry Jr. is very challenging.

[7] As will become clear later in the proof, two buffer columns (instead of $n+2$) are actually sufficient at each of these places since no row will ever be shifted more than twice before one of the players can reach their goal.

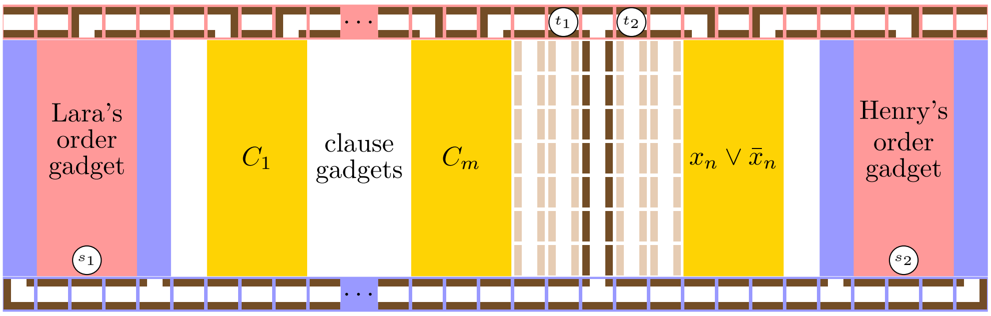


Figure 4 Overview of the board used in the proof of Theorem 4.

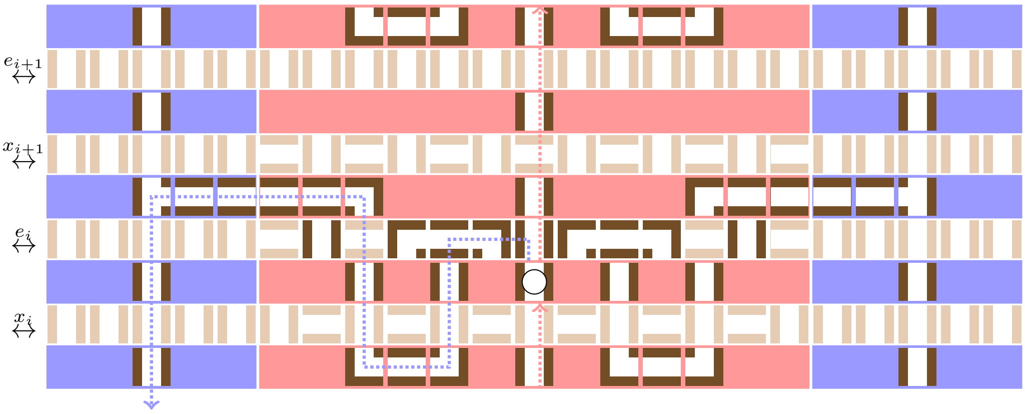


Figure 5 A segment of a player's order gadget (before turn i of the other player).

player does not shift their corresponding variable rows in the correct order. All tiles of the wiring row and the escape path row are fixed, which implies that only rows can be shifted and the wrap-around rule cannot be used on the top or the bottom of the board (where no buffer was added).

In Figure 5, we illustrate a segment of a player's order gadget. Here, exactly the variable rows of Henry's variables (i.e., the ones with even index) are crossable via the pink path at the beginning of the game in Lara's order gadget, while the variable rows of Lara's variables (i.e., the ones with odd index) are crossable via the pink path in Henry's order gadget at the beginning. Moreover, Lara starts in the middle of the row *below* the variable row corresponding to x_1 in her order gadget, while Henry starts in the middle of the row *above* the variable row of x_1. The lowermost shiftable row corresponds to x_1 and the uppermost shiftable row corresponds to x_n (i.e., there is no row e_n). We note that, compared to the (rotated) structure of the clause gadgets and the variable gadget of x_n shown in Figure 2, the row e_i represents an additional row (column in the figure) between the variables x_i and x_{i+1} for each $i \in \{1, \ldots, n-1\}$. Within the clause gadgets and the variable gadget of x_n, this additional row is simply filled by ▌▐-tiles (which would correspond to 〓-tiles in Figure 2 due to the rotation) so that it can always be crossed and does not change the structure of the gadgets.

In total, the board contains $4n$ rows (2 for each x_i for $i \in \{1, \ldots, n\}$, 2 for each e_i for $i \in \{1, \ldots, n-1\}$, the wiring row, and the escape path row). Since each of the two order gadgets, each of the m clause gadgets, and the variable gadget corresponding to x_n only requires a constant number of columns and $n + 2$ buffer columns are inserted in between each pair of adjacent gadgets as outlined above, the constructed board is of size poly(n, m), so the described instance can be constructed from the given instance $\mathcal{I}$ of QSAT in polynomial time.

We now show that the constructed instance is equivalent to the given instance $\mathcal{I}$ of QSAT, i.e., that Lara has a winning strategy if and only if $\mathcal{I}$ is a yes-instance. To do so, we first assume that both players only shift their corresponding variable rows in the correct order, i.e., that Lara always sets the next odd variable available in each of her turns and that Henry always sets the next even variable available in each of his turns. As we show afterwards, this behavior of the players will be enforced since each player can reach the escape path at the bottom of the board (and, thus, win) if the other player deviates from this rule.

Lara starts the game and, according to the above assumption, shifts the variable row corresponding to x_1 in turn one, thereby setting x_1 to either `true` or `false`. Moreover, this closes the blue path in Henry's order gadget (Figure 5 and, thus, prevents Henry from using this path in order to reach the escape path at the bottom of the board. After her shift, Lara can move past the variable row of x_1 and also the untouched variable row of x_2 in her order gadget. However, she cannot yet cross the variable row corresponding to x_3 in her order gadget. Thus, in order to still have the blue path after the variable row of x_2 available in case that Henry should not set x_2 in the next turn, Lara will stop in the row between x_2 and e_2.

Again according to the above assumption, Henry will now shift the variable row corresponding to x_2 in turn two (which is Henry's first turn), thereby setting x_2 to either `true` or `false`. Moreover, this closes the blue path in Lara's order gadget and prevents her from entering the escape path at the bottom of the board in her next turn. After his shift, Henry can move past the variable row of x_2 and also the untouched variable row of x_3 in his order gadget. However, he cannot yet cross the variable row corresponding to x_4 in his order gadget. Thus, in order to still have the blue path after the variable row of x_3 available in case that Lara should not set x_3 in her second turn, Henry will stop in the row between x_3 and e_3.

It is then Lara's turn again and she will now set variable x_3 and so on. At the end, in turn $n-1$, Henry will set variable x_{n-1} due to the assumption that n is odd. Thus, Henry reaches the entry to the wiring row at the top of his order gadget in turn $n-1$. However, he cannot yet reach his goal t_2 since he cannot yet cross the variable gadget of the still unset variable x_n. Turn n is then Lara's turn and she only has to set variable x_n in order to reach the entry to the wiring row at the top of her order gadget. Since she has to cross the clause gadgets in order to reach her goal t_1, she will, thus, arrive at t_1 in turn n (i.e., before Henry reaches t_2) if and only if all the clauses are satisfied by the variable assignment determined by both players in the n turns. Since, in any case, Henry will reach t_2 in turn $n+1$, this shows that Lara has a winning strategy (i.e., a strategy of setting the odd variables in her corresponding turns so that all clauses will be satisfied no matter how Henry sets the even variables in his turns) if and only if the given instance $\mathcal{I}$ of QSAT is a yes-instance.

It remains to show that whoever violates the order first loses the game. So assume that both players always set their corresponding variables as desired during the first $i-1$ turns, but in turn i, the corresponding player (say Lara) decides not to shift the variable row corresponding to x_i. For Lara, this means that the variable row corresponding to x_i is still blocked in her order gadget and she cannot proceed. For $i = n$, Lara loses in this case since Henry will reach t_2 in turn $n+1$ as seen above. For $i < n$, Henry finds his order gadget as in Figure 5 before his next turn (standing in the circled spot in the row between x_i and e_i) and, with a left or right shift of row e_i, he can move onto the blue path (which is not blocked since Lara did not shift the variable row corresponding to x_i). Note that, even though Lara may have shifted row e_i in turn i, Henry can always enter the blue path either to the left or to the right since any single shift in row e_i opens the blue path for him. Consequently, Henry can enter the escape path row, which directly takes him to his goal t_2, and Lara loses the game. Similarly, Lara can enter the escape path row if Henry deviates first, which finishes the proof. ◀

Note that our proof of PSPACE-hardness did not rely on the rule that prevents a player from directly reversing the previous shift – the hardness result in Theorem 4 holds both with and without this rule.

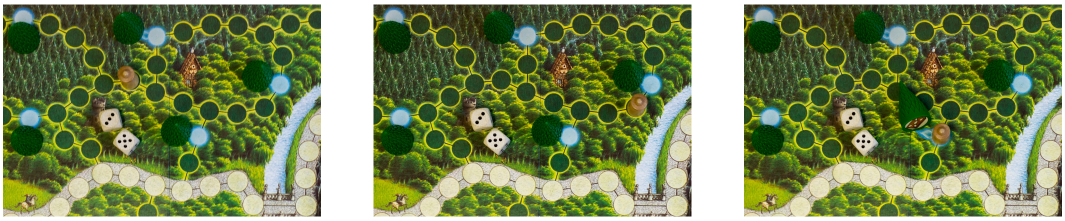
Concerning upper bounds on the computational complexity of SP-Versus-Labyrinth, note that the number of possible positions for each of the avatars and goals of the two players is bounded by the number of spots on the board, there are only twenty different tiles possible at each spot (counting different orientations and whether the tile is fixed or not), and only three possible shapes exit for the current surplus tile. Consequently, the number of possible game states in SP-Versus-Labyrinth is bounded from above by an exponential function of the board size $a \cdot b$ and it follows by standard arguments that SP-Versus-Labyrinth $\in$ EXPTIME (see, for example, [12]). Furthermore, it can easily be seen that SP-Versus-Labyrinth $\in$ PSPACE when upper bounding the number of turns by some value k polynomial in the encoding length of the game and then rating which player has achieved the better situation after k turns, i.e., has fewer turns left to reach her goal when continuing to play alone (similar to the turn-restricted version of Go analyzed in [9]). Since our proof of PSPACE-hardness works also for this turn-restricted version of SP-Versus-Labyrinth, it follows that the turn-restricted version of the problem is PSPACE-complete. Whether the actual (not turn-restricted) version of SP-Versus-Labyrinth belongs to PSPACE, however, remains an interesting open question.

3 Enchanted Forest

The game *Enchanted Forest* developed by Alex Randolph and Michel Matschoss was originally published by Ravensburger in 1981 under the German title “Sagaland” [8]. In 1982, the game earned the prestigious award “Spiel des Jahres” (engl. *Game of the Year*) [10].

In the game, two to six players move around in an enchanted forest in order to find items from popular fairy tales that are hidden under special trees. At the beginning of the game, all players start in the village next to the enchanted forest. A player gets to know which item is hidden under one of the trees if she ends her move on the blue field next to the tree. The king in the castle requests the location of the different items and whoever moves up to the castle and reports the correct location of the currently requested item earns a point. The player standing in the castle may then continue to name (guess if necessary) the locations of the next requested items to earn additional points. When she names a wrong location for the first time, she must return to the starting point in the village. The first player to earn three points wins the game.

A player’s turn consists of rolling two dice, which results in two numbers of fields she has to move. Each of the two movements must be executed consecutively in one direction, but the player can choose the order in which the two movements are performed (see Figure 6 for an example). If the roll is a double, the player may alternatively choose to either move to any blue field adjacent to a tree or to the castle, or to shuffle the cards determining the order in which the king requests the items. If a player moves to a field already occupied by another player, the other player is moved back to the starting point in the village.



(a) Initial position of the player and the result of rolling the two dice.

(b) The player's position after her first movement (five fields).

(c) After the second movement (three fields), the player reaches the tree and observes the item hidden underneath it.

Figure 6 A player's turn in Enchanted Forest.

3.1 Formal Problem Definition

In order to analyze the game mathematically, we model the board as an undirected, connected, simple graph $G = (V, E)$, where each vertex corresponds to a field on the board and there is a unit-length edge between each pair of adjacent fields.[8] As usual, we let $n := |V|$ and $m := |E|$. The special fields to which a player can move instantaneously when rolling a double (the blue fields adjacent to the trees and the castle) are given as a subset $V' \subseteq V$. Moreover, we consider two arbitrary d-sided dice for the moves (where $d > n$ is possible since the graph may contain cycles). If a player rolls $(x, \bar{x})$ with $x, \bar{x} \in \{1, \dots, d\}$ and starts at $s \in V$, she can decide on two (not necessarily simple) paths to follow, where the first one starts at s and the second one starts at the end vertex of the first one. One of these two paths must have length x and the other one length $\bar{x}$. Moreover, the requirement that each movement has to be executed consecutively in one direction means that the two paths are not allowed to contain cycles of length 2 as subpaths. The special rules used when rolling a double mean that, if $x = \bar{x}$, the player may alternatively choose to move to any vertex in the subset V'.

As in *Labyrinth*, we consider the fundamental problem of reaching a certain location on the board (e.g., the castle or a specific tree) in a minimum number of turns. Here, we assume that the player has complete knowledge of the sequence of die rolls for her future turns. The decision version of this problem is formally defined as follows:

▶ **Definition 5** (SP-EnchantedForest).

INSTANCE: The simple graph $G = (V, E)$, the subset $V' \subseteq V$, the maximum die value $d \in \mathbb{N}$, the die rolls $(x_1, \bar{x}_1), \dots, (x_k, \bar{x}_k)$ with $x_i, \bar{x}_i \in \{1, \dots, d\}$, and two vertices $s, t \in V$.

QUESTION: Can a player starting at vertex s reach vertex t in at most k turns using the given rolls of the dice?

The encoding length of an instance of SP-EnchantedForest (when storing the graph G in adjacency list representation) is $\mathcal{O}(n + m + k \cdot \log_2 d)$.

In the following subsection, we show that SP-EnchantedForest can be solved efficiently in polynomial time. This result also has implications for the complexity of problems the players face in Enchanted Forest when the outcomes of future die rolls are unknown. For example, the polynomial-time solvability of SP-EnchantedForest directly implies that, when

[8] Note that, even though G models an enchanted *forest*, the graph may contain cycles (as does the original board).

the outcomes of future die rolls are unknown, the problem of choosing two movements for the current turn that maximize the probability of reaching a desired location in (at most) k turns for a given constant k can be solved in polynomial time.

3.2 Finding one's way in the Enchanted Forest is easy

After living a long, rich, and joyful life, Gretel wants to relive her childhood memories and eat some gingerbread from the gingerbread house in the Enchanted Forest. However, the ancient enchantments do not allow her to simply follow the path of pebbles laid out. Before making the next step, she has to roll two dice and move accordingly. At least, with all the lucky charms she obtained from the witch's heritage, she can predict the rolls. Can she find the delicious gingerbread or will the journey be too long to bear the appetite?

Gretel rolls (or rather predicts to roll) $(x_1, \bar{x}_1), \ldots, (x_k, \bar{x}_k)$ and decides that it is probably a good idea to first compute which vertices in G she can reach from a given position in the enchanted forest by using a single die roll. Thus, for any $x \in \{1, \ldots, d\}$, she defines the (symmetric) $(n \times n)$-matrix D_x with entries in $\{0, 1\}$ such that, for each pair $(u, v) \in V \times V$, the matrix has an entry 1 at the position corresponding to u and v if and only if there exists a (not necessarily simple) path of length x from u to v in G that does not contain any cycles of length 2 as subpaths. However, in order to compute D_x efficiently in polynomial time, Gretel cannot explore the graph G step-by-step by always moving to an adjacent vertex since this would lead to a time requirement polynomial in d, but *not* in $\log_2 d$. Instead, she uses the following procedure provided by two friendly scholars:

Computing $\mathbf{D_x}$ efficiently: In order to make sure that only paths that do not contain cycles of length 2 as subpaths are considered, Gretel constructs a directed graph $H = (N, A)$ from G by setting

$$N := \{v_u : \ \{u, v\} \in E\}, \ \text{ and } \ A := \{(v_u, w_v) : \ v_u, w_v \in N, \{v, w\} \in E, w \neq u\}.$$

Here, the vertex set N of H contains a copy v_u of each vertex $v \in V$ for every neighbor u of v in G. The arcs in H are constructed such that the copy v_u of v corresponding to u is connected by a directed arc (of unit length) to all copies w_v of the neighbors w of v that are *different from* u. Thus, there exists a path of length x without cycles of length 2 as subpaths from a node $\tilde{u}$ to another node $\tilde{v}$ in G if and only if there exists a directed path of the same length from *some copy* of $\tilde{u}$ to *some copy* of $\tilde{v}$ in H. Hence, the problem of determining whether a node $\tilde{v}$ in G can be reached from a given node $\tilde{u}$ in G by a path of length x without cycles of length 2 as subpaths reduces to determining whether *some copy* $\tilde{v}_w$ of $\tilde{v}$ can be reached from *some copy* $\tilde{u}_z$ of $\tilde{u}$ by a directed path of length x in H.

To compute which vertices in H are reachable from which other vertices, Gretel computes the x-th power of the adjacency matrix M of H (cf. [11]). This can be done in $\mathcal{O}(|N|^{2.373} \log_2 x) = \mathcal{O}(m^{2.373} \log_2 d)$ time by computing M^{2^α} for $\alpha = 1, \ldots, \lfloor \log_2 x \rfloor$ via the square matrix multiplication algorithm from [7] (since $|N| = 2 \cdot |E| = 2m$ and $x \in \{1, \ldots, d\}$). Then, D_x has a 1 at the position corresponding to $u \in V$ and $v \in V$ exactly if M^x has a positive entry at some position corresponding to a copy of u and a copy of v.

Computing which vertices are reachable in a single turn: With the knowledge of the next k rolls $(x_1, \bar{x}_1), \ldots, (x_k, \bar{x}_k)$ and the above method for computing the sets D_x, Gretel can now compute efficiently which vertices v she can reach from any given vertex u in the graph G with any single pair $(x_i, \bar{x}_i)$ of die rolls. To do so, she defines the $(n \times n)$-matrix $D_{(x_i, \bar{x}_i)}$

with entries in $\{0,1\}$ such that, for each pair $(u,v) \in V \times V$, the matrix has an entry 1 at the position corresponding to u and v if and only if vertex v can be reached from vertex u with the pair $(x_i, \bar{x}_i)$ of die rolls. Gretel now wants to compute $D_{(x_i,\bar{x}_i)}$ efficiently. If $(x_i, \bar{x}_i)$ is not a double (i.e., if $x_i \neq \bar{x}_i$), she notes that, by definition of the matrices D_{x_i} and $D_{\bar{x}_i}$, the matrix $D_{(x_i,\bar{x}_i)}$ has a 1 at the position corresponding to u and v if and only if at least one of the matrices $D_{x_i} \cdot D_{\bar{x}_i}$ and $D_{\bar{x}_i} \cdot D_{x_i}$ has a positive entry at this position. If $(x_i, \bar{x}_i)$ is a double (i.e., if $x_i = \bar{x}_i$), Gretel has to take into account that she can also decide to move instantaneously to any vertex in the subset $V' \subseteq V$. Hence, in this case, the matrix $D_{(x_i,\bar{x}_i)}$ also has a 1 at the position corresponding to u and v whenever $v \in V'$. If Gretel again uses the square matrix multiplication algorithm from [7] to compute $D_{x_i} \cdot D_{\bar{x}_i}$ and $D_{\bar{x}_i} \cdot D_{x_i}$, this shows that she can obtain $D_{(x_i,\bar{x}_i)}$ from the matrices D_x and $D_{\bar{x}_i}$ in time $\mathcal{O}(n^{2.373})$.

Turn-expanded network: Similar to a time-expanded network, Gretel constructs the (directed) *turn-expanded network* $F = (N_F, A_F)$ with

$$N_F := \{v^i : \ v \in V, i = 0, \dots, k\} \cup \{t^*\}, \text{ and}$$

$$A_F := \bigcup_{i=1}^{k} \{(u^{i-1}, v^i) : \ D_{(x_i,\bar{x}_i)} \text{ has entry 1 at position } (u,v)\} \cup \{(t^1, t^*), \dots, (t^k, t^*)\}.$$

She can now compute a shortest path from s_0 to t^* by breadth-first search in $\mathcal{O}(|N_F|+|A_F|) = \mathcal{O}(k \cdot n^2)$ time and decide whether she can reach her favorite dish in k turns (the required number of turns equals the length of a shortest s_0-t^*-path minus one). As computing the at most $2k$ matrices D_x is the most time consuming step, the overall running time of Gretel's procedure is $\mathcal{O}(k \cdot m^{2.373} \cdot \log_2 d)$, which is polynomial in the input length. This shows:

▶ **Theorem 6.** *SP-EnchantedForest can be solved in polynomial time* $\mathcal{O}(k \cdot m^{2.373} \cdot \log_2 d)$. ◀

As noted before, the result of Theorem 6 also has implications for the complexity of problems that Gretel faces when she cannot use her lucky charms in order to predict the outcomes of future die rolls. For example, the polynomial-time solvability of SP-EnchantedForest directly implies that, when the outcomes of future die rolls are unknown, Gretel only needs polynomial time in order to compute two movements for her current turn that maximize the probability of reaching the gingerbread house in (at most) k turns for a given constant k.

References

1 E. D. Demaine and R. A. Hearn. Playing games with algorithms: Algorithmic combinatorial game theory, 2001. `http://arxiv.org/abs/cs.CC/0106019`.

2 C. Dodaro, M. Alviano, W. Faber, N. Leone, F. Ricca, and M. Sirianni. The birth of a WASP: Preliminary report on a new ASP solver. In *Proceedings of the 26th Italian Conference on Computational Logic (CILC)*, pages 99–113, 2011.

3 D. Eppstein. Computational complexity of games and puzzles. `https://www.ics.uci.edu/~eppstein/cgt/hard.html`. Accessed: 2018-04-16.

4 M. R. Garey and D. S. Johnson. *Computers and Intractability (A Guide to the Theory of NP-Completeness).* W.H. Freeman and Company, New York, 1979.

5 G. Kendall, A. J. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

6 M. J. Kobbert. Das verrückte Labyrinth. Ravensburger, 1986.

7 F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 296–303, 2014.

8 M. Matschoss and A. Randolph. Sagaland. Ravensburger, 1981.

9 C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

10 Spiel des Jahres e.V. Spiel des Jahres. `http://www.spiel-des-jahres.com/en`.

11 R. P. Stanley. *Enumerative Combinatorics*. Wadsworth Publ. Co., 1986.

12 L. J. Stockmeyer and A. K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979.

Card-based Protocols Using Triangle Cards

Kazumasa Shinagawa
Tokyo Institute of Technology, Tokyo, Japan
Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan
shinagawakazumasa@gmail.com

Takaaki Mizuki
Tohoku University, Sendai, Japan
tm-paper+triacard@g-mail.tohoku-university.jp

Abstract

Suppose that three boys and three girls attend a party. Each boy and girl have a crush on exactly one of the three girls and three boys, respectively. The following dilemma arises: On one hand, each person thinks that if there is a mutual affection between a girl and boy, the couple should go on a date the next day. On the other hand, everyone wants to avoid the possible embarrassing situation in which their heart is broken "publicly." In this paper, we solve the dilemma using novel cards called *triangle cards*. The number of cards required is only six, which is minimal in the case where each player commits their input at the beginning of the protocol. We also construct multiplication and addition protocols based on triangle cards. Combining these protocols, we can securely compute any function $f : \{0, 1, 2\}^n \to \{0, 1, 2\}$.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols

Keywords and phrases Cryptography without computer, Secure computation, Card-based protocols, Triangle cards, Three-valued computation, Secure matching problem

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.31

Funding Supported by JSPS KAKENHI Grant Numbers 17J01169 and 17K00001.

1 Introduction

Three girls, Alice, Carol, and Ellen, and three boys, Bob, Dave, and Frank, are having a good time at a party. Assume that each boy has a crush on exactly one of the three girls, and each girl has a crush on exactly one of the three boys. On one hand, everyone thinks that if there is a mutual affection between a girl and a boy the couple should go on a date the next day. On the other hand, each of the six people is too shy to announce the person they have in mind: Everyone wants to avoid the possible embarrassing situation in which their heart is broken "publicly." Are there any solutions to this dilemma?

A cryptographic technique called secure multiparty computation provides a viable solution. This enables participants holding private inputs to securely compute the value of a desired function, without revealing their input information. In order to solve the social dilemma described above, it suffices to design a secure multiparty computation protocol for the function $f : \{0, 1, 2\}^6 \to \{0, 1\}^9$:

$$f(a_0, a_1, a_2, b_0, b_1, b_2) = (c_{0,0}, c_{0,1}, c_{0,2}, c_{1,0}, c_{1,1}, c_{1,2}, c_{2,0}, c_{2,1}, c_{2,2}),$$

where $c_{i,j} = 1$ if $(a_i = j) \wedge (b_j = i)$ and $c_{i,j} = 0$ otherwise. We call this function f the $(3, 3)$-*matching function*, and refer to the problem of securely computing f as the *secure* $(3, 3)$-*matching problem*. Because it is well-known in the field of cryptography that any function can be securely computed [2], the secure $(3, 3)$-matching problem can clearly be

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 31; pp. 31:1–31:13
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

resolved somehow. However, conventional secure multiparty computation protocols tend to be based on a deep mathematical perspective, and hence it seems to be unlikely that all participants executing a given protocol will concretely understand its correctness and security. Because the secure $(3, 3)$-matching problem would typically arise in everyday life, such as in the party scenario mentioned above, it is desirable to have a more simple and convenient solution. Thus, this paper solicits a solution to the secure matching problem using physical cards. The cards used in this paper are novel ones, called *triangle cards*, which can easily be constructed using sheets of paper and seals.

1.1 Triangle card

In this paper, we propose a novel card called a triangle card, whose shape is a regular triangle, where its front/back sides have the same symbol (e.g., ♠). We use the following encoding rules:

$\leftrightarrow 0$, $\leftrightarrow 1$, $\leftrightarrow 2 = -1$.

In an execution of a protocol, both faces of a card can be hidden by placing seals as follows:

.

We will formally define triangle cards in Section 2.

1.2 Our results

In this paper, we design a protocol for solving the secure $(3, 3)$-matching problem using six triangle cards. We also design a protocol for any function $f : (\mathbb{F}_3)^n \to \mathbb{F}_3$.

1.2.1 Secure $(3, 3)$-matching protocol

As shown in Table 1, we construct a protocol for the secure $(3, 3)$-matching problem using six cards and six shuffles. Our protocol is efficient in terms of both the number of cards and shuffles, because a straightforward solution based on the five-card trick [1] requires 42 cards and 15 shuffles, as explained in Section 3. Our protocol is optimal in terms of the number of cards when each party submits a (tuple of) card(s) as input at the beginning of a protocol, because the number of parties in $(3, 3)$-matching is six.

1.2.2 Protocol for any function

As shown in Table 1, we design a multiplication protocol over $\mathbb{F}_3$ using four cards. Regular 3-sided cards, proposed by Shinagawa et al. [5], also enable a secure multiplication protocol over $\mathbb{F}_3$, while requiring 15 cards. Although the previous work in [5] and ours are based on different types of cards, and thus incomparable, this paper implies that triangle cards are effective for computing a function over $\mathbb{F}_3$ compared with regular 3-sided cards. Based on the previous addition protocol in [5], we also design an addition protocol for triangle cards. Because our protocols output a card with *seals*, an output card for our protocol can be used as an input card for another protocol, i.e., our protocols are *composable*. By combining our protocols, we can securely compute any function over $\mathbb{F}_3$.

Table 1 Comparison between our protocols and previous ones.

	Type of cards	Number of cards	Number of shuffles
◦ Protocol for Secure $(3,3)$-Matching Problem			
Based on [1]	♣, ♡	42	15
Ours	triangle	6	6
◦ Multiplication over $\mathbb{F}_3$			
[5]	regular 3-sided	15	2
Ours	triangle	4	2
◦ Addition over $\mathbb{F}_3$			
[5]	regular 3-sided	2	1
Ours	triangle	2	1

1.3 Related work

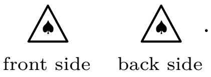
While most existing card-based protocols (cf. [1,4]) utilize a binary pair of cards ♣ ♡, there are some studies based on different types of cards, such as cards with rotationally symmetric backs [3,7], polarizing cards [6], and regular n-sided polygon cards [5]. Part of our technique is motivated by this previous works. In particular, we employ a rotation shuffle [3,5] and a turning shuffle [6].

2 Triangle card

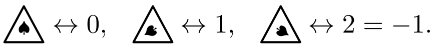
In this section, we define triangle cards and the operations used in our protocols.

2.1 Definition of triangle cards

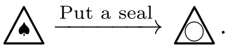
A *triangle card* is a card of regular triangular shape, whose front and back sides show the same symbol, as follows:



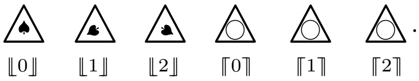
We use the following encoding rule:



The value of a card can be hidden from participating parties by placing seals on both sides, as follows:

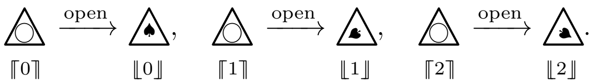


For a value $a \in \mathbb{F}_3$, we call a card of a without seals an *opened card*, denoted by $\lfloor\!\lfloor a \rfloor\!\rfloor$, and a card of a with seals a *closed card*, denoted by $\lceil\!\lceil a \rceil\!\rceil$. That is, there are six states for a triangle card, as follows:

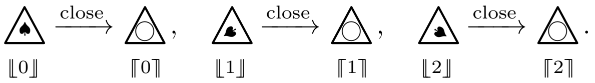

$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
$\lfloor\!\lfloor 1 \rfloor\!\rfloor$
$\lfloor\!\lfloor 2 \rfloor\!\rfloor$
$\lceil\!\lceil 0 \rceil\!\rceil$
$\lceil\!\lceil 1 \rceil\!\rceil$
$\lceil\!\lceil 2 \rceil\!\rceil$

We define operations for a single triangle card as follows:

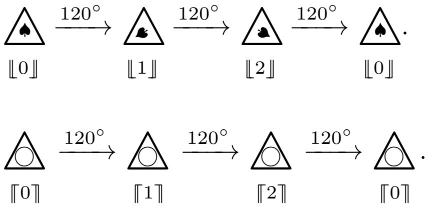
- Open/Close: An *open* operation transforms a closed card $\lceil\!\lceil a \rceil\!\rceil$ into the opened card $\lfloor\!\lfloor a \rfloor\!\rfloor$, as follows:


open
open
open
$\lceil\!\lceil 0 \rceil\!\rceil$
$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
$\lceil\!\lceil 1 \rceil\!\rceil$
$\lfloor\!\lfloor 1 \rfloor\!\rfloor$
$\lceil\!\lceil 2 \rceil\!\rceil$
$\lfloor\!\lfloor 2 \rfloor\!\rfloor$

Conversely, a *close* operation transforms an opened card $\lfloor\!\lfloor a \rfloor\!\rfloor$ into the closed card $\lceil\!\lceil a \rceil\!\rceil$, as follows:

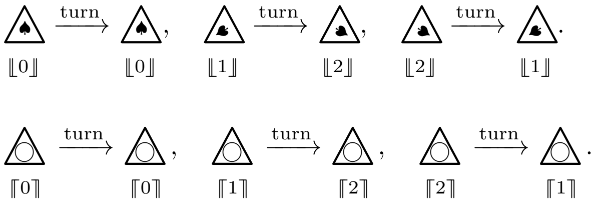

close
close
close
$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
$\lceil\!\lceil 0 \rceil\!\rceil$
$\lfloor\!\lfloor 1 \rfloor\!\rfloor$
$\lceil\!\lceil 1 \rceil\!\rceil$
$\lfloor\!\lfloor 2 \rfloor\!\rfloor$
$\lceil\!\lceil 2 \rceil\!\rceil$

- Rotate: A *rotation* by 120° transforms a card of value a into a card of value $a+1$, as follows:


120°
120°
120°
$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
$\lfloor\!\lfloor 1 \rfloor\!\rfloor$
$\lfloor\!\lfloor 2 \rfloor\!\rfloor$
$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
120°
120°
120°
$\lceil\!\lceil 0 \rceil\!\rceil$
$\lceil\!\lceil 1 \rceil\!\rceil$
$\lceil\!\lceil 2 \rceil\!\rceil$
$\lceil\!\lceil 0 \rceil\!\rceil$

Consequently, a rotation by $(n \times 120)^\circ$ transforms a card of value a into a card of value $a+n$. We call this operation a *rotation n-times*.

- Turn: A *turn* operation turns over a card. That is, it transforms a card of value a into a card of value $-a$, as follows:

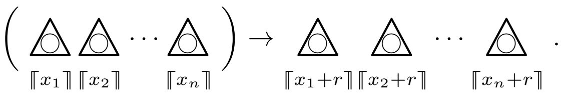

turn
turn
turn
$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
$\lfloor\!\lfloor 0 \rfloor\!\rfloor$
$\lfloor\!\lfloor 1 \rfloor\!\rfloor$
$\lfloor\!\lfloor 2 \rfloor\!\rfloor$
$\lfloor\!\lfloor 2 \rfloor\!\rfloor$
$\lfloor\!\lfloor 1 \rfloor\!\rfloor$
turn
turn
turn
$\lceil\!\lceil 0 \rceil\!\rceil$
$\lceil\!\lceil 0 \rceil\!\rceil$
$\lceil\!\lceil 1 \rceil\!\rceil$
$\lceil\!\lceil 2 \rceil\!\rceil$
$\lceil\!\lceil 2 \rceil\!\rceil$
$\lceil\!\lceil 1 \rceil\!\rceil$

2.2 Shuffles for triangle cards

A *shuffle* is a probabilistic operation on a sequence of cards. In this paper, we use three types of shuffle: rotation shuffle, turning shuffle, and flower shuffle. We present example implementations of these shuffles in the appendix.

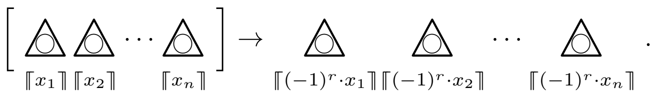
2.2.1 Rotation shuffle

This takes a sequence of n closed cards $(\lceil\!\lceil x_1 \rceil\!\rceil, \lceil\!\lceil x_2 \rceil\!\rceil, \cdots, \lceil\!\lceil x_n \rceil\!\rceil)$ and outputs a sequence of n closed cards $(\lceil\!\lceil x_1 + r \rceil\!\rceil, \lceil\!\lceil x_2 + r \rceil\!\rceil, \cdots, \lceil\!\lceil x_n + r \rceil\!\rceil)$, where r is a uniformly random number over $\mathbb{F}_3$ that is independent from the inputs and other randomness, and information-theoretically hidden from all parties. (See Figure 2 in the appendix.) We denote this shuffle as follows:


$\lceil\!\lceil x_1 \rceil\!\rceil$
$\lceil\!\lceil x_2 \rceil\!\rceil$
$\lceil\!\lceil x_n \rceil\!\rceil$
$\lceil\!\lceil x_1{+}r \rceil\!\rceil$
$\lceil\!\lceil x_2{+}r \rceil\!\rceil$
$\lceil\!\lceil x_n{+}r \rceil\!\rceil$

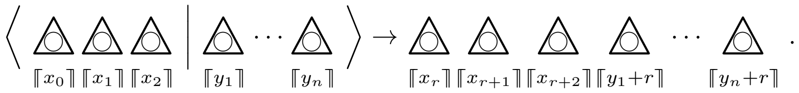
2.2.2 Turning shuffle

This takes a sequence of n closed cards ($[\![x_1]\!], [\![x_2]\!], \cdots, [\![x_n]\!]$) and outputs a sequence of n closed cards ($[\![(-1)^r \cdot x_1]\!], [\![(-1)^r \cdot x_2]\!], \cdots, [\![(-1)^r \cdot x_n]\!]$), where r is a uniformly random number over $\{0,1\}$ that is independent from the inputs and other randomness, and information-theoretically hidden from all parties. (See Figure 3 in the appendix.) We denote this shuffle as follows:


$[\![x_1]\!]$
$[\![x_2]\!]$
$\cdots$
$[\![x_n]\!]$
$\rightarrow$
$[\![(-1)^r \cdot x_1]\!]$
$[\![(-1)^r \cdot x_2]\!]$
$\cdots$
$[\![(-1)^r \cdot x_n]\!]$
.

2.2.3 Flower shuffle

This takes a sequence of $3+n$ closed cards ($[\![x_0]\!], [\![x_1]\!], [\![x_2]\!], [\![y_1]\!], \cdots, [\![y_n]\!]$) and outputs a sequence of $3+n$ closed cards ($[\![x_r]\!], [\![x_{r+1}]\!], [\![x_{r+2}]\!], [\![y_1+r]\!], \cdots, [\![y_n+r]\!]$), where r is a uniformly random number over $\mathbb{F}_3$ that is independent of the inputs and other randomness, and information-theoretically hidden from all parties. (See Figure 4 in the appendix.) We denote this shuffle as follows:



3 A solution based on existing methods

In this section, we briefly describe a straightforward solution for the secure $(3,3)$-matching problem using a deck of cards employed in previous studies, where their front sides show $\boxed{\clubsuit}$, $\boxed{\heartsuit}$ and their back sides show the same $\boxed{?}$. A pair of face-down cards is called a *commitment* to 0 (resp. 1) if its face-up symbols are $(\clubsuit, \heartsuit)$ (resp. $(\heartsuit, \clubsuit)$). The main tool used is the *five-card trick*, proposed by den Boer [1], which takes two commitments to $a, b \in \{0,1\}$ and a single additional card, and outputs the value $c = a \wedge b$ with five free cards as follows:

$$\underbrace{\boxed{?}\boxed{?}}_{a}\underbrace{\boxed{?}\boxed{?}}_{b}\boxed{\heartsuit} \rightarrow c, \boxed{\clubsuit}\boxed{\clubsuit}\boxed{\heartsuit}\boxed{\heartsuit}\boxed{\heartsuit}.$$

This requires a single shuffle called a *random cut*. Let Alice (resp. Bob) input 1 if she likes Bob (resp. Alice), and 0 otherwise. This solves a secure matching problem between two parties, where either both parties like each other or one party does not like the other. A straightforward solution for the secure $(3,3)$-matching problem is to apply the five-card trick for every boy-girl pair. That is, each girl (indexed by i) submits a tuple of three commitments to $(a_{i,0}, a_{i,1}, a_{i,2}) \in \{0,1\}^3$, where $a_{i,j} = 1$ if and only if she likes the j-th boy. Similarly, each boy submits a tuple of three commitments in the same manner. Then, for every $i, j \in \mathbb{F}_3$ we apply the five-card trick to the commitments to $a_{i,j}$ and $b_{j,i}$. We set $c_{i,j} = 1$ if the output is 1, and 0 otherwise. (Recall that $c_{i,j} = 1$ means that the i-th girl and j-th boy have a mutual affection.) This idea solves the secure $(3,3)$-matching problem using $36+1$ cards and 9 shuffles.

One weakness of the above idea is that each party may deviate from the input rule, i.e., each party may submit two or more commitments to 1. In order to prevent a deviation from the input rule, we should check the input format in a zero-knowledge manner. Based on the idea of Shinagawa et al. [5] (Section 5, a voting protocol), we can check the validity

$v \in \{\mathsf{true}, \mathsf{false}\}$ of three commitments using additional six cards, without destroying the input, as follows:

$$\underbrace{\boxed{?}\,\boxed{?}}_{a_{i,0}}\underbrace{\boxed{?}\,\boxed{?}}_{a_{i,1}}\underbrace{\boxed{?}\,\boxed{?}}_{a_{i,2}}\boxed{\clubsuit}\,\boxed{\clubsuit}\,\boxed{\clubsuit}\,\boxed{\heartsuit}\,\boxed{\heartsuit}\,\boxed{\heartsuit} \to v,\ \underbrace{\boxed{?}\,\boxed{?}}_{a_{i,0}}\underbrace{\boxed{?}\,\boxed{?}}_{a_{i,1}}\underbrace{\boxed{?}\,\boxed{?}}_{a_{i,2}}\boxed{\clubsuit}\,\boxed{\clubsuit}\,\boxed{\clubsuit}\,\boxed{\heartsuit}\,\boxed{\heartsuit}\,\boxed{\heartsuit}.$$

This also requires a single shuffle. Thus, the number of cards required is 42 $(= 36 + 6)$, and the number of shuffles required is 15 $(= 9 + 6)$.

4 Our main solution

In this section, we solve the $(3, 3)$-matching problem using six triangle cards. Recall that the $(3, 3)$-matching function $f : (\mathbb{F}_3)^6 \to \{0, 1\}^9$ is defined follows:

$$f(a_0, a_1, a_2, b_0, b_1, b_2) = (c_{0,0}, c_{0,1}, c_{0,2}, c_{1,0}, c_{1,1}, c_{1,2}, c_{2,0}, c_{2,1}, c_{2,2}),$$

where $c_{i,j} = 1$ if $(a_i = j) \wedge (b_j = i)$ and $c_{i,j} = 0$ otherwise.

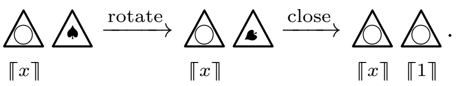
We first design a protocol for checking $x \overset{?}{=} 0$ in Section 4.1. Then, we construct a $(3, 3)$-matching protocol in Section 4.2.

4.1 IsZero protocol

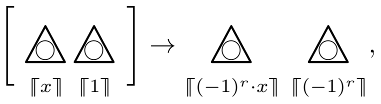
The IsZero protocol takes two cards $(\llbracket x \rrbracket, \llbracket 0 \rrbracket)$, and outputs a predicate $c \in \{0, 1\}$ for $x \overset{?}{=} 0$ without changing the sequence or revealing information of x beyond the predicate c.

The protocol proceeds as follows.

1. Rotate and close the right card as follows:

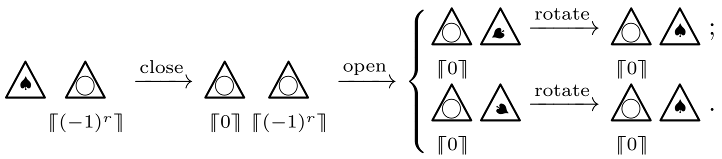


2. Apply a turning shuffle to these.

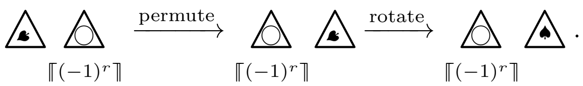


where r is a random value generated in the shuffle.

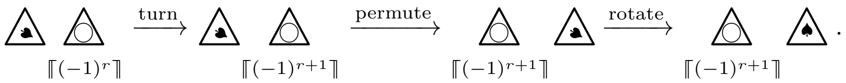
3. Open the left card. Let $v = (-1)^r \cdot x$ be the opened value.
   a. The case $v = 0$: Close the left card, open the right card, and rotate the right card 2 (resp. 1) times if the opened value is 1 (resp. 2), as follows:



   b. Case $v = 1$: Exchange the two cards, and rotate the right card 2-times, as follows:



c. Case $v = 2$: Turn the right card, exchange the two cards, and rotate the right card 1-time, as follows:



4. Output $c = 1$ if $v = 0$ and $c = 0$ otherwise.

4.1.1 Correctness

First, let us check that the resulting sequence is equal to the original sequence $(⟦x⟧, ⟦0⟧)$. When $v = 0$, it holds that $x = 0$. Thus, the resulting sequence $(⟦0⟧, ⟦0⟧)$ is equal to the original sequence. When $v = 1$, it is either the case that $(x, r) = (1, 0)$ or $(x, r) = (2, 1)$. Thus, the resulting sequence $(⟦(-1)^r⟧, ⟦0⟧)$ is equal to the original sequence. When $v = 2$, either $(x, r) = (2, 0)$ or $(x, r) = (1, 1)$. Thus, the resulting sequence $(⟦(-1)^{r+1}⟧, ⟦0⟧)$ is equal to the original sequence. Moreover, we can observe that c is equal to the predicate for $x \overset{?}{=} 0$, because $v = 0$ if and only if $x = 0$.

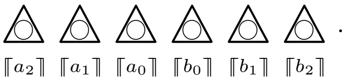
4.1.2 Security

When $c = 1$, i.e., $x = 0$, the opened value v is always 0. When $c = 0$, the opened value $v = 1$ (or 2) with a probability of exactly 1/2. Thus, the distribution of v is statistically independent from x given c.

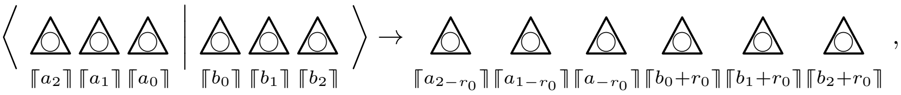
4.2 Secure $(3, 3)$-matching protocol

The protocol proceeds as follows.

1. Place the six cards as follows:

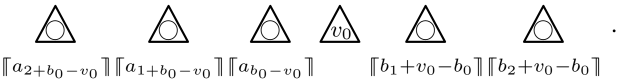


2. Apply a flower shuffle to the sequence:

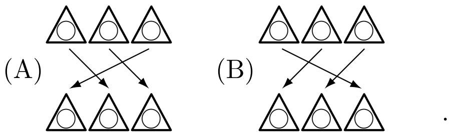


where r_0 is a random value generated in the shuffle. Note that the first three cards of the resulting sequence are $(a_{2-r_0}, a_{1-r_0}, a_{-r_0})$, because those of the original sequence are arranged in reverse order (a_2, a_1, a_0).

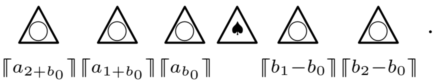
3. Open the fourth card. Let $v_0 = b_0 + r_0$ be the opened value. Now, the current sequence can be expressed as follows:



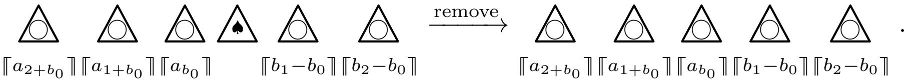
4. Permute the first three cards according to (A) if $v_0 = 1$ and (B) if $v_0 = 2$:



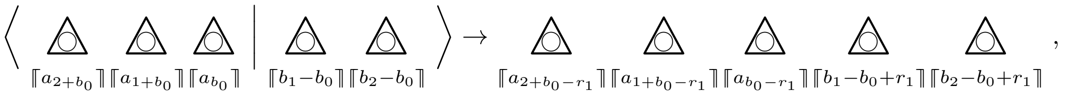
Rotate the fourth/fifth/sixth cards $(3-v_0)$-times. Now, the current sequence is as follows:


$\llbracket a_{2+b_0} \rrbracket$
$\llbracket a_{1+b_0} \rrbracket$
$\llbracket a_{b_0} \rrbracket$
$\llbracket b_1-b_0 \rrbracket$
$\llbracket b_2-b_0 \rrbracket$
.

5. Apply the IsZero protocol to the third/fourth cards, and let c_0 be the output of this.
6. Remove the fourth card.

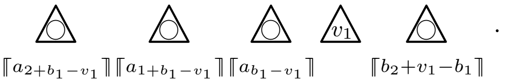

$\llbracket a_{2+b_0} \rrbracket$
$\llbracket a_{1+b_0} \rrbracket$
$\llbracket a_{b_0} \rrbracket$
$\llbracket b_1-b_0 \rrbracket$
$\llbracket b_2-b_0 \rrbracket$
remove
$\llbracket a_{2+b_0} \rrbracket$
$\llbracket a_{1+b_0} \rrbracket$
$\llbracket a_{b_0} \rrbracket$
$\llbracket b_1-b_0 \rrbracket$
$\llbracket b_2-b_0 \rrbracket$
.

7. Apply a flower shuffle to the sequence:

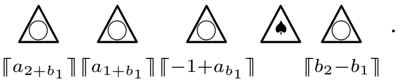

$\llbracket a_{2+b_0} \rrbracket$
$\llbracket a_{1+b_0} \rrbracket$
$\llbracket a_{b_0} \rrbracket$
$\llbracket b_1-b_0 \rrbracket$
$\llbracket b_2-b_0 \rrbracket$
$\rightarrow$
$\llbracket a_{2+b_0-r_1} \rrbracket$
$\llbracket a_{1+b_0-r_1} \rrbracket$
$\llbracket a_{b_0-r_1} \rrbracket$
$\llbracket b_1-b_0+r_1 \rrbracket$
$\llbracket b_2-b_0+r_1 \rrbracket$
,

where r_1 is a random value generated in the shuffle.

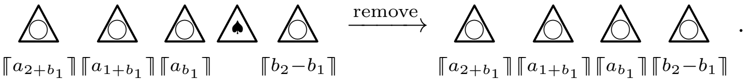
8. Open the fourth card. Let $v_1 = b_1 - b_0 + r_1$ be the opened value. Now, the current sequence can be expressed as follows:


v_1
$\llbracket a_{2+b_1-v_1} \rrbracket$
$\llbracket a_{1+b_1-v_1} \rrbracket$
$\llbracket a_{b_1-v_1} \rrbracket$
$\llbracket b_2+v_1-b_1 \rrbracket$
.

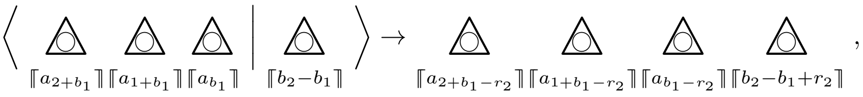
9. Permute the first three cards according to (A) in Step 4 if $v_1 = 1$ and (B) in Step 4 if $v_1 = 2$. Rotate the fourth/fifth cards $(3 - v_1)$-times, and rotate the third card 2-times. Now, the current sequence is as follows:


$\llbracket a_{2+b_1} \rrbracket$
$\llbracket a_{1+b_1} \rrbracket$
$\llbracket -1+a_{b_1} \rrbracket$
$\llbracket b_2-b_1 \rrbracket$
.

10. Apply the IsZero protocol to the third/fourth cards, and let c_1 be the output of this.
11. Remove the fourth card.

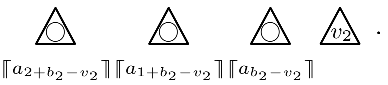

$\llbracket a_{2+b_1} \rrbracket$
$\llbracket a_{1+b_1} \rrbracket$
$\llbracket a_{b_1} \rrbracket$
$\llbracket b_2-b_1 \rrbracket$
remove
$\llbracket a_{2+b_1} \rrbracket$
$\llbracket a_{1+b_1} \rrbracket$
$\llbracket a_{b_1} \rrbracket$
$\llbracket b_2-b_1 \rrbracket$
.

12. Apply a flower shuffle to the sequence:

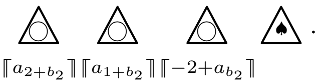

$\llbracket a_{2+b_1} \rrbracket$
$\llbracket a_{1+b_1} \rrbracket$
$\llbracket a_{b_1} \rrbracket$
$\llbracket b_2-b_1 \rrbracket$
$\rightarrow$
$\llbracket a_{2+b_1-r_2} \rrbracket$
$\llbracket a_{1+b_1-r_2} \rrbracket$
$\llbracket a_{b_1-r_2} \rrbracket$
$\llbracket b_2-b_1+r_2 \rrbracket$
,

where r_2 is a random value generated in the shuffle.

13. Open the fourth card, and let $v_2 = b_2 - b_1 + r_2$ be the opened value. Now, the current sequence can be expressed as follows:


v_2
$\llbracket a_{2+b_2-v_2} \rrbracket$
$\llbracket a_{1+b_2-v_2} \rrbracket$
$\llbracket a_{b_2-v_2} \rrbracket$
.

14. Permute the first three cards according to (A) in Step 4 if $v_2 = 1$ and (B) in Step 4 if $v_2 = 2$. Rotate the fourth/fifth cards $(3 - v_2)$-times, and rotate the third card 1-time. Now, the current sequence is as follows:


$\llbracket a_{2+b_2} \rrbracket$
$\llbracket a_{1+b_2} \rrbracket$
$\llbracket -2+a_{b_2} \rrbracket$
.

15. Apply the IsZero protocol to the third/fourth cards, and let c_2 be the output of this.
16. For $j = 0, 1, 2$, do the following: If $c_j = 1$, the $(j + 3)$-th input b_j is publicly announced[1] by the $(j + 3)$-th party, and we set $c_{j,b_j} = 1$ and $c_{i,j} = 0$ for $i \neq b_j$. Otherwise, set $(c_{0,j}, c_{1,j}, c_{2,j}) = (0, 0, 0)$.
17. Output $(c_{0,0}, c_{0,1}, c_{0,2}, c_{1,0}, c_{1,1}, c_{1,2}, c_{2,0}, c_{2,1}, c_{2,2})$.

4.2.1 Correctness

Let $j \in \{0, 1, 2\}$ be any index. From the correctness of the IsZero protocol, the value $c_{i,j}$ is 1 if it holds that both $i = b_j$ and $-j + a_{b_j} = 0$, and 0 otherwise. That is, $c_{i,j} = 1$ if $(a_i = j) \wedge (b_j = i)$, and $c_{i,j} = 0$ otherwise. Therefore, the above protocol correctly computes the $(3, 3)$-matching function.

4.2.2 Security

From the security of the IsZero protocol, the opened values of the IsZero protocol are statistically independent from the inputs $a_{b_0}, a_{b_1}, a_{b_2}$ given the outputs c_0, c_1, c_2. The opened values $v_0 = b_0 + r_0, v_1 = b_1 + r_1, v_2 = b_2 + r_2$ in Steps 3, 8, and 13, respectively, are masked by a uniformly random number $r_0, r_1, r_2 \in \mathbb{F}_3$. Thus, the distribution of the opened values and the distribution of the inputs are statistically independent given an output value.

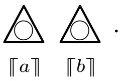
5 Secure computation for any function

In this section, we construct addition and multiplication protocols. Given two closed cards $[\![a]\!], [\![b]\!]$ as inputs, the former protocol outputs $[\![a + b]\!]$, and the latter outputs $[\![ab]\!]$. Because any function over $\mathbb{F}_3$ can be expressed using additions and multiplications, we can securely compute any function over $\mathbb{F}_3$ by combining these protocols.

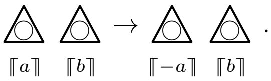
5.1 Addition protocol

The protocol proceeds as follows:

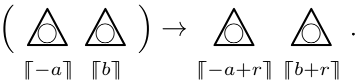
1. Place the two cards as follows:



2. Apply the turning operation to the left card:



3. Apply a rotation shuffle to the cards:

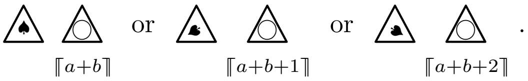


[1] We note that the j-th boy such that $c_j = 1$ can maliciously announce the i-th girl such that $i \neq b_j$, and this is not noticed when the i-th girl is attracted to the j-th boy. Thus, the levels of trust between boys and girls are different. To avoid this asymmetry of boys and girls, we can use one additional card. That is, by using an additional card, we can copy the input of each boy and verify the girl they have in mind.

Table 2 All possibilities of the final sequence in our addition protocol.

(a,b)	$a+b$	s_0	s_1	s_2
$(0,0)$	0	$(0,0)$	$(1,1)$	$(2,2)$
$(0,1)$	1	$(0,1)$	$(1,2)$	$(2,0)$
$(0,2)$	2	$(0,2)$	$(1,0)$	$(2,1)$
$(1,0)$	1	$(2,0)$	$(0,1)$	$(1,2)$
$(1,1)$	2	$(2,1)$	$(0,2)$	$(1,0)$
$(1,2)$	0	$(2,2)$	$(0,0)$	$(1,1)$
$(2,0)$	2	$(1,0)$	$(2,1)$	$(0,2)$
$(2,1)$	0	$(1,1)$	$(2,2)$	$(0,0)$
$(2,2)$	1	$(1,2)$	$(2,0)$	$(0,1)$

4. Open the first card. Then, a commitment to $a+b$ is obtained as follows:


$\llbracket a+b \rrbracket$
or
$\llbracket a+b+1 \rrbracket$
or
$\llbracket a+b+2 \rrbracket$
.

5.1.1 Correctness

After applying a rotation shuffle to the sequence $(\llbracket -a \rrbracket, \llbracket b \rrbracket)$ in Step 3, the resulting sequence is one of s_0, s_1, s_2 as follows:

$$
\begin{aligned}
s_0 &= (\llbracket -a \rrbracket, \llbracket b \rrbracket),\\
s_1 &= (\llbracket -a+1 \rrbracket, \llbracket b+1 \rrbracket),\\
s_2 &= (\llbracket -a+2 \rrbracket, \llbracket b+2 \rrbracket).
\end{aligned}
$$

Table 2 shows all possibilities of the final sequence in our addition protocol. We can observe that in each sequence, the right value is $a+b+\ell$, where ℓ is the left value. Therefore, our addition protocol is correct.

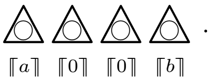
5.1.2 Security

Let v be the opened value in Step 4. Owing to the rotation shuffle in Step 3, v is equal to $-a+r$, where $r \in \mathbb{F}_3$ is a uniformly random value that is hidden from all parties and independent from the inputs (a,b). Thus, the distribution of v and the distribution of the inputs are statistically independent.

5.2 Multiplication protocol

The protocol proceeds as follows:

1. Place the four cards $(\llbracket a \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \llbracket b \rrbracket)$ as follows:


$\llbracket a \rrbracket$
$\llbracket 0 \rrbracket$
$\llbracket 0 \rrbracket$
$\llbracket b \rrbracket$
.

2. Apply our addition protocol to the leftmost three cards $(\llbracket a \rrbracket, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket)$:

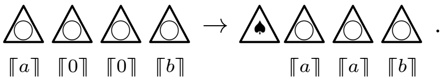
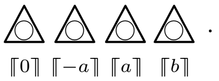

$\llbracket a \rrbracket$
$\llbracket 0 \rrbracket$
$\llbracket 0 \rrbracket$
$\llbracket b \rrbracket$
→
$\llbracket a \rrbracket$
$\llbracket a \rrbracket$
$\llbracket b \rrbracket$
.

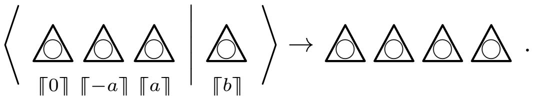
Table 3 All possibilities of the final sequence in our multiplication protocol. (An underlined value corresponds to the output.)

(a,b)	ab	s_0	s_1	s_2
$(0,0)$	0	$(\underline{0},0,0,0)$	$(0,0,\underline{0},1)$	$(0,\underline{0},0,2)$
$(0,1)$	0	$(0,0,\underline{0},1)$	$(0,\underline{0},0,2)$	$(\underline{0},0,0,0)$
$(0,2)$	0	$(0,\underline{0},0,2)$	$(\underline{0},0,0,0)$	$(0,0,\underline{0},1)$
$(1,0)$	0	$(\underline{0},2,1,0)$	$(1,2,\underline{0},1)$	$(2,\underline{0},1,2)$
$(1,1)$	1	$(0,2,\underline{1},1)$	$(2,\underline{1},0,2)$	$(\underline{1},0,2,0)$
$(1,2)$	2	$(0,\underline{2},1,2)$	$(\underline{2},1,0,0)$	$(1,0,\underline{2},1)$
$(2,0)$	0	$(\underline{0},1,2,0)$	$(1,2,\underline{0},1)$	$(2,\underline{0},1,2)$
$(2,1)$	2	$(0,1,\underline{2},1)$	$(1,\underline{2},0,2)$	$(\underline{2},0,1,0)$
$(2,2)$	1	$(0,\underline{1},2,2)$	$(\underline{1},2,0,0)$	$(2,0,\underline{1},1)$

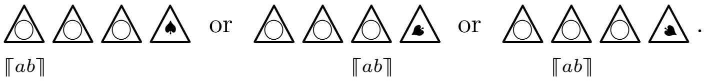
3. Close the first card, and apply a turning operation to the second card:



4. Apply a flower shuffle to the sequence:



5. Open the fourth card. Then, the closed card $\llbracket ab \rrbracket$ is obtained as follows:



5.2.1 Correctness

After applying a flower shuffle to the sequence $(\llbracket 0 \rrbracket, \llbracket -a \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket)$ in Step 4, the resulting sequence is one of s_0, s_1, s_2 as follows:

$$\begin{aligned} s_0 &= (\llbracket 0 \rrbracket, \llbracket -a \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket), \\ s_1 &= (\llbracket -a \rrbracket, \llbracket a \rrbracket, \llbracket 0 \rrbracket, \llbracket b+1 \rrbracket), \\ s_2 &= (\llbracket a \rrbracket, \llbracket 0 \rrbracket, \llbracket -a \rrbracket, \llbracket b+2 \rrbracket). \end{aligned}$$

Table 3 shows all the possibilities of the final sequence in our multiplication protocol. We can observe that in each sequence, the underlying value is equal to ab, and the first/second/third value is underlined if the fourth value is 0/2/1, respectively. Therefore, our multiplication protocol is correct.

5.2.2 Security

Let v_1, v_2 be the opened values in Steps 2 and 5, respectively. From Section 5.1.2, v_1 is statistically independent from the inputs. Owing to the flower shuffle in Step 4, v_2 is equal to $b + r$, where $r \in \mathbb{F}_3$ is a uniformly random value that is hidden from all parties and independent from v_1 and the inputs (a, b). Thus, the distribution of the opened values and the distribution of the inputs are statistically independent.

6 Conclusion

In this paper, we proposed novel cards called triangle cards, and solved the secure $(3, 3)$-matching problem using six triangle cards. We also designed a protocol for any function over $\mathbb{F}_3$. One could ask if our technique applies to $\mathbb{F}_n$ for any n. A straightforward generalization to this case does not work. This is because for a regular n-sided polygon with $n > 3$, a rotation over $n - 1$ points with a single fixed point cannot be implemented by a physical operation. In contrast, in the triangle case a turning operation corresponds to an operation of this type: a rotation between 1 and 2 with a fixed point 0. Our protocols exploits this property. This is why we concentrate on a triangle rather than a general polygon. An interesting open question is that of finding new physical objects that enable the construction of an efficient protocol for the secure (n, m)-matching problem for any integers n, m.

References

1 Bert den Boer. More efficient match-making and satisfiability: *The Five Card Trick*. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings*, volume 434 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 1989. doi:10.1007/3-540-46885-4_23.

2 Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987. doi:10.1145/28395.28420.

3 Takaaki Mizuki and Hiroki Shizuya. Practical card-based cryptography. In Alfredo Ferro, Fabrizio Luccio, and Peter Widmayer, editors, *Fun with Algorithms - 7th International Conference, FUN 2014, Lipari Island, Sicily, Italy, July 1-3, 2014. Proceedings*, volume 8496 of *Lecture Notes in Computer Science*, pages 313–324. Springer, 2014. doi:10.1007/978-3-319-07890-8_27.

4 Takaaki Mizuki and Hideaki Sone. Six-card secure AND and four-card secure XOR. In Xiaotie Deng, John E. Hopcroft, and Jinyun Xue, editors, *Frontiers in Algorithmics, Third International Workshop, FAW 2009, Hefei, China, June 20-23, 2009. Proceedings*, volume 5598 of *Lecture Notes in Computer Science*, pages 358–369. Springer, 2009. doi:10.1007/978-3-642-02270-8_36.

5 Kazumasa Shinagawa, Takaaki Mizuki, Jacob C. N. Schuldt, Koji Nuida, Naoki Kanayama, Takashi Nishide, Goichiro Hanaoka, and Eiji Okamoto. Multi-party computation with small shuffle complexity using regular polygon cards. In Man Ho Au and Atsuko Miyaji, editors, *Provable Security - 9th International Conference, ProvSec 2015, Kanazawa, Japan, November 24-26, 2015, Proceedings*, volume 9451 of *Lecture Notes in Computer Science*, pages 127–146. Springer, 2015. doi:10.1007/978-3-319-26059-4_7.

6 Kazumasa Shinagawa, Takaaki Mizuki, Jacob C. N. Schuldt, Koji Nuida, Naoki Kanayama, Takashi Nishide, Goichiro Hanaoka, and Eiji Okamoto. Secure multi-party computation using polarizing cards. In Keisuke Tanaka and Yuji Suga, editors, *Advances in Information and Computer Security - 10th International Workshop on Security, IWSEC 2015, Nara, Japan, August 26-28, 2015, Proceedings*, volume 9241 of *Lecture Notes in Computer Science*, pages 281–297. Springer, 2015. doi:10.1007/978-3-319-22425-1_17.

7 Kazumasa Shinagawa, Koji Nuida, Takashi Nishide, Goichiro Hanaoka, and Eiji Okamoto. Committed AND protocol using three cards with more handy shuffle. In *2016 International Symposium on Information Theory and Its Applications, ISITA 2016, Monterey, CA, USA, October 30 - November 2, 2016*, pages 700–702. IEEE, 2016. URL: http://ieeexplore.ieee.org/document/7840515/.

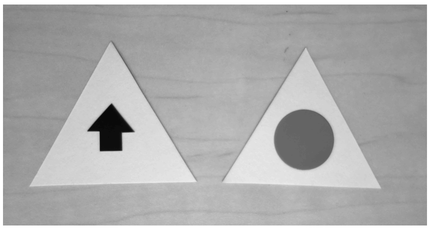


Figure 1 Triangle cards: an opened card (left) and a closed card (right).

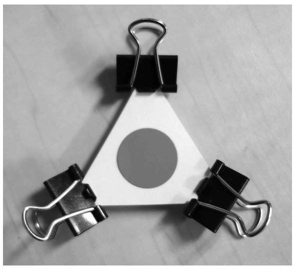


Figure 2 A rotation shuffle.

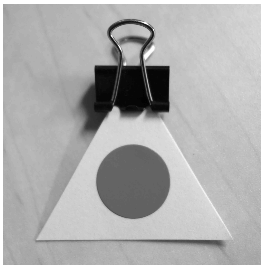


Figure 3 A turning shuffle.

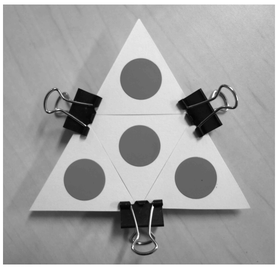


Figure 4 A flower shuffle.

8 Itaru Ueda, Akihiro Nishimura, Yu-ichi Hayashi, Takaaki Mizuki, and Hideaki Sone. How to implement a random bisection cut. In Carlos Martín-Vide, Takaaki Mizuki, and Miguel A. Vega-Rodríguez, editors, *Theory and Practice of Natural Computing - 5th International Conference, TPNC 2016, Sendai, Japan, December 12-13, 2016, Proceedings*, volume 10071 of *Lecture Notes in Computer Science*, pages 58–69, 2016. doi:10.1007/978-3-319-49001-4_5.

A Implementation of cards and shuffles

Figure 1 shows an example implementation of a triangle card. The left card is an opened card, and the right card is a closed card. A rotation shuffle is implemented by using three clips: all cards are stacked using three clips as in Figure 2, and then the stack is spun like a "roulette wheel." A turning shuffle is implemented by using a single clip: all cards are stacked using a clip as in Figure 3, and then the stack is thrown in a spinning manner, like a coin toss (this technique is called a *spinning throw* [8]). A flower shuffle for $(\llbracket x_0 \rrbracket, \llbracket x_1 \rrbracket, \llbracket x_2 \rrbracket, \llbracket y_1 \rrbracket, \cdots, \llbracket y_n \rrbracket)$ is implemented by using three clips: As in Figure 4, the last n cards $\llbracket y_1 \rrbracket, \cdots, \llbracket y_n \rrbracket$ are stacked and placed on the center of the flower, $\llbracket x_0 \rrbracket$ is placed on the top petal, $\llbracket x_1 \rrbracket$ is placed on the right petal, and $\llbracket x_2 \rrbracket$ is placed on the left petal. Then, the flower is spun like a "roulette wheel."

The Power of One Secret Agent

Tami Tamir
School of Computer Science, The Interdisciplinary Center (IDC), Herzliya, Israel
tami@idc.ac.il

Abstract

I am a job. In job-scheduling applications, my friends and I are assigned to machines that can process us. In the last decade, thanks to our strong employee committee, and the rise of algorithmic game theory, we are getting more and more freedom regarding our assignment. Each of us acts to minimize his own cost, rather than to optimize a global objective.

My goal is different. I am a secret agent operated by the system. I do my best to lead my fellow jobs to an outcome with a high social cost. My naive friends keep doing the best they can, each of them performs his best-response move whenever he gets the opportunity to do so. Luckily, I am a charismatic guy. I can determine the order according to which the naive jobs perform their best-response moves. In this paper, I analyze my power, formalized as the *Price of a Traitor* (PoT), in cost-sharing scheduling games – in which we need to cover the cost of the machines that process us.

Starting from an initial *Nash Equilibrium* (NE) profile, I join the instance and hurt its stability. A sequence of best-response moves is performed until I vanish, leaving the naive jobs in a new NE. For an initial NE assignment, S_0, the PoT measures the ratio between the social cost of a worst NE I can lead the jobs to, starting from S_0, and the social cost of S_0. The PoT of a game is the maximal such ratio among all game instances and initial NE assignments.

My analysis distinguishes between instances with unit- and arbitrary-cost machines, and instances with unit- and arbitrary-length jobs. I give exact bounds on the PoT for each setting, in general and in symmetric games. While it turns out that in most settings my power is really impressive, my task is computationally hard (and also hard to approximate).

2012 ACM Subject Classification Theory of computation → Algorithmic game theory

Keywords and phrases Job scheduling games, Cost sharing, Equilibrium inefficiency

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.32

1 Introduction

I am a job. In job-scheduling applications, my friends and I are assigned to machines that can process us. The authorities that assign us to machines like to analyze the way we are assigned. They treat us as instances of combinatorial optimization problems, and our assignment became a major discipline in operations research. In the old days, we were all controlled by a centralized scheduler who assigned us in a way that achieves an effective use of the system's resources, or a target quality of service [20]. In the last decade, thanks to our strong employees committee, and also the rise of algorithmic game theory, we are getting more and more freedom regarding our assignment. Many modern systems provide service to multiple strategic users, whose individual payoff is affected by the decisions made by other users of the system. As a result, non-cooperative game theory has become an essential tool in the analysis of our assignment [21, 15, 24, 4, 12, 3]. Each of us has strategic considerations and acts to minimize his own cost, rather than to optimize any global objective. Practically, this means that we *choose* a machine instead of being assigned to one by a centralized scheduler.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 32; pp. 32:1–32:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

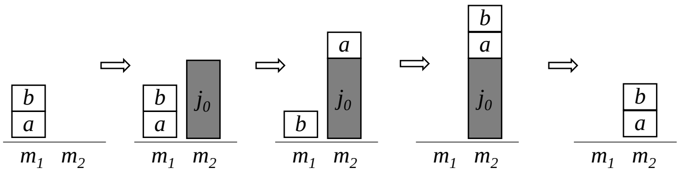


Figure 1 A simple example of a traitor BR-sequence with PoT= 2.

My goal is different, I am not the regular job you are used to analyze. Already in my childhood I was a problematic kid and my parents were invited regularly to school to discuss my behavior[1]. Recently, I started to work as a secret agent, operated by the system. My mission is to join a stable assignment of other jobs, perturb its stability, and lead a sequence of best-response moves, whose outcome is as poor as possible. When I'm done, I vanish, leaving the other jobs in a new stable profile, whose cost is hopefully higher. My naive friends keep doing the best they can, each of them performs his best-response move whenever he gets the opportunity to do so. Luckily, I am a charismatic guy; I can determine the order according to which the naive jobs deviate.

In this paper, I analyze my power, formalized as the *Price of a traitor* (PoT), in cost-sharing scheduling games. In these games every job has a subset of the machines on which it can be assigned, and the cost of every utilized machine is shared by the job assigned to it, where the share is proportional to the load generated by the jobs. My goal is to lead the jobs into a stable assignment in which the total cost of utilized machines is maximal. Before diving into the details, let me demonstrate my mission on a small example.

Example 1: Consider an instance with two machines m_1 and m_2 of costs 1 and 2 respectively. Assume that two naive jobs of length 1 are assigned on m_1 (see leftmost assignment in Figure 1). The cost of each of them in this initial profile is 1/2. Assume that my length is $3+\epsilon$, and I appear and assign myself on m_2 (I am Job 0 - the gray guy in the figure). Since $2/(4+\epsilon) < 1/2$ each of the naive jobs will benefit from joining me. So they join me one after the other. Once we are all on m_2, I vanish. The jobs are left on the more expensive machine (rightmost assignment in Figure 1), and their assignment is stable, since they each pay 1, and a unilateral deviation to m_1 will also result in this cost. My mission is completed with a NE whose cost is doubled.

1.1 Preliminaries

An instance of a cost-sharing game with a traitor (CST) is given by a tuple $G = \langle \mathcal{J}, \mathcal{M}, \{M_j\}_{j\in\mathcal{J}}, p_0\rangle$, where $\mathcal{M}$ is a set of m machines, and $\mathcal{J}$ is a set of k *naive* jobs. Not all machines are feasible to all jobs. For each $j \in \mathcal{J}$, the machines that may process Job j are given by the set $M_j \subseteq \mathcal{M}$. Every job $j \in \mathcal{J}$ has processing time p_j which is independent of the machine on which it is assigned. Every machine $i \in \mathcal{M}$ has an activation cost, $c(i)$. The last component of the tuple specifies my length - the processing time of the *traitor*. Throughout this paper, I am denoted Job 0.

Every job is a player, where the strategy space of Job j is the set of machines in M_j. A profile of a CST game is a vector $S = \langle s_0, s_1, \ldots, s_k\rangle \in ((\mathcal{M} \cup \{\perp\}) \times M_1 \times \ldots \times M_k)$,

[1] Enthusiastic fans of the conference *FUN with algorithms* may recognize me as a bully job in [23].

describing the machines selected by the jobs. My strategy, s_0, is in $\mathcal{M} \cup \{\perp\}$, meaning that I can go to any machine and also be away, in which case $s_0 = \perp$. A profile in which $s_0 = \perp$ is denoted a *traitor-free* profile. For a machine $i \in \mathcal{M}$, the *load* on i in S, denoted $L_i(S)$, is the total processing time of the jobs assigned to machine i in S, that is, $L_i(S) = \sum_{\{j|s_j=i\}} p_j$. When S is clear from the context it is omitted.

A machine i is *utilized* in a profile S if $L_i(S) > 0$. The cost of a utilized machine is covered by the jobs assigned to it, where the share is proportional to the load generated by the jobs. Formally, the cost of Job j in the profile S is $cost_j(S) = c(s_j) \cdot \frac{p_j}{L_{s_j}(S)}$. This cost-sharing scheme fits the commonly used proportional cost-sharing rule for weighted players, (e.g., [21, 1, 11]).

Consider a game G. For a profile S, a job j, and a strategy $s'_j \in M_j$, let (S_{-j}, s'_j) denote the profile obtained from S by replacing the strategy of Job j by s'_j. That is, the profile resulting from a migration of Job j from machine s_j to machine s'_j. A profile s is a *pure Nash equilibrium* (NE) if no job can benefit from unilaterally deviating from his strategy in S to another strategy; i.e., for every job j and every strategy $s'_j \in M_j$ it holds that $cost_j((S_{-j}, s'_j)) \geq cost_j(S)$. This paper considers only *pure* strategies. Unlike mixed strategies, pure strategies may not be random or drawn from a distribution.

Given a profile S, the *best response* (BR) of Job j is $BR_j(S) = \arg\min_{s'_j \in M_j} cost_j(S_{-j}, s'_j)$; i.e., a machine i such that Job j's cost will be minimized if he is assigned to machine i, fixing the assignment of all other jobs. If there are several such machines, each of them is considered a best-response. *Best-Response Dynamics* (BRD) is a local-search method where in each step some player is chosen and plays his BR.

A naive job j is said to be *suboptimal* in a profile S if he can reduce his cost by migrating to another machine, i.e., if $s_j \notin BR_j(S)$. Given an initial profile S_0, a *traitor BR-sequence* from S_0 is a sequence of profiles $\langle S_0, S_1, \ldots S_T \rangle$ in which for every $t = 0, 1, \ldots$, either there exists a naive job j such that $S_{t+1} \in (S_{t_{-j}}, BR_j(S_t))$, or $S_{t+1} = (S_{t_{-0}}, s'_0)$. In other words, either a naive job performs a BR move or I perform a move of my choice – even if it is not beneficial for me. I am interested in traitor BR-sequences in which both S_0 and S_T are traitor-free NEs. The stability of S_0 is perturbed once I arrive and select some machine. Formally, in S_0 my strategy is $\perp$, and no naive job is suboptimal. Then, $S_1 = (S_{0_{-0}}, s'_0)$ for $s'_0 \in \mathcal{M}$. The last profile in a traitor BR-sequence is also traitor-free, that is $s_0(S_T) = \perp$. For a profile S_0, let TNE(S_0) be the set of Nash equilibria reachable from S_0 via a traitor BR-sequence. If my departure leaves the naive jobs in a non-stable profile, they will keep forming BR-moves until they converge to a NE (by [2] this will surely happen).

The social cost of a profile S is the total cost of resources utilized in S, which is equal to the total cost of the players. Formally, $cost(S) = \sum_{j \in \mathcal{J} \cup \{0\}} cost_j(S) = \sum_{i \in \cup_j s_j} c(i)$. Note that I pay my part in utilizing a machine that I share with others – this is essential also to keep my reliability among the naive jobs. However, the fact that the final NE in the sequence is traitor-free guarantees that I cannot force a very expensive outcome by selecting an expensive machine for myself.

Let NE(G) be the set of Nash equilibria in a CST game G. Being a weighted cost-sharing game with singleton strategies, it is well known that NE$(G) \neq \emptyset$ and that BRD converges to a NE [2]. Recall that TNE(S_0) is the set of traitor-free Nash equilibria reachable from a traitor-free NE S_0 via a traitor BR-sequence.

The *Price of a Traitor* in a game G, denoted PoT(G), is defined as the worst ratio, among all initial traitor-free NE profiles S_0, between the social cost of a NE in TNE(S_0) and the social cost of S_0. I.e.,

$$PoT(G) = \sup_{S_0 \in \mathrm{NE}(G)} \max_{S \in \mathrm{TNE}(S_0)} \frac{cost(S)}{cost(S_0)}.$$

For a class of games $\mathcal{G}$, the price of a traitor with respect to $\mathcal{G}$ is defined as the worst-case PoT over all games in $\mathcal{G}$. That is, $\mathrm{PoT}(\mathcal{G}) = \sup_{G \in \mathcal{G}}\{\mathrm{PoT}(G)\}$.

It is well known that NE profiles may be sub-optimal. Let $OPT(G)$ denote the minimal possible social cost of a feasible assignment of $\mathcal{J}$, i.e., $OPT(G) = \min_S cost(S)$. The inefficiency incurred due to self-interested behavior is quantified according to the *price of anarchy* (PoA) [15, 19] and *price of stability* (PoS) [1] measures. The PoA is the worst-case inefficiency of a pure Nash equilibrium, while the PoS measures the best-case inefficiency of a pure Nash equilibrium. Formally, $\mathrm{PoA}(G) = \max_{S \in NE(G)} cost(S)/OPT(G)$, and $\mathrm{PoS}(G) = \min_{S \in NE(G)} cost(S)/OPT(G)$.

The following observation bounds my power for any game instance.

▶ **Observation 1.** *For every game G, $1 \leq PoT(G) \leq \frac{PoA(G)}{PoS(G)}$.*

Proof. For every initial NE profile, S_0, it holds that $cost(S_0) \geq OPT(G) \cdot \mathrm{PoS}(G)$. Also, for every $S \in \mathrm{TNE}(S_0)$, it holds that $cost(S) \leq OPT(G) \cdot \mathrm{PoA}(G)$. Therefore,

$$\mathrm{PoT}(G) \leq \max_{S \in \mathrm{TNE}(S_0)} \frac{cost(S)}{cost(S_0)} \leq \frac{OPT(G) \cdot \mathrm{PoA}(G)}{OPT(G) \cdot \mathrm{PoS}(G)} = \frac{\mathrm{PoA}(G)}{\mathrm{PoS}(G)}.$$

Also, since $S_0 \in \mathrm{TNE}(S_0)$, it holds that $\mathrm{PoT}(G) \geq 1$. ◀

Related work: I am not a young job. I participated in many assignments in my life, and I always tried to analyze the performance of these assignments. In addition, I'm trying to follow the huge effort done by researchers in analyzing our assignments. Before the rise of algorithmic game theory, most of the study dealt with achieving a global objective of the assignment such as load balancing, minimizing our total completion time, or the makespan (corresponding to the maximal cost of some job) [20].

In the last decade, game-theoretic analysis became an important tool for analyzing our assignments, as many other systems in which a set of resources is shared by selfish users. *Congestion games* consist of a set of resources and a set of players who need to use these resources. Players' strategies are subsets of resources. Each resource has a latency function which, given the load generated by the players on the resource, returns the cost of the resource [21, 1]. CST games are congestion games with singleton strategies, in which each resource has an activation cost that is shared by the players using it according to some sharing mechanism. A generalized, traitor-free, model of this game, in which the processing times of jobs depend on the machines they are assigned to was studied in [17, 2].

Best-Response dynamics corresponds to actual dynamics in real life applications. They are therefore starring in the study of non-cooperative game theory [1, 13, 8]. The important questions are whether BRD converges to a NE, if one exists [17, 12]; what is the converges time [5, 6, 22, 13]; and what is the quality of the solution [8]. The paper [22] studies the complexity of equilibria in a wide range of cost sharing games.

Other related work deal with games in which some of the players are not selfish. For example, in the Stackelberg model [14, 10, 7], a fraction of the jobs are selfish, while the rest are willing to obey a centralized authority. A Stackelberg strategy assigns the controllable jobs, trying to minimize the inefficiency caused by the others.

Table 1 The non-restricted jobs and their initial profile in the CST game constructed in the reduction.

Job j	p_j	M_j	$S_0(j)$	$cost_0(j)$
a	1	$\{m_1, m_2, m_3\}$	m_1	1/4
b	1	$\{m_2, m_4\}$	m_2	1/2
c	2	$\{m_3, m_5\}$	m_5	2/5
d	2	$\{m_3, m_5\}$	m_5	2/5
$1, \ldots, n$	a_j	$\{m_1, m_2\}$	m_1	$a_j/4$

Games in which some players are adversarial were defined and study in the areas of Cryptography [16, 18] and Rational Synthesis [9]. However, the goals and the allowed actions of the malicious players in these games are different, and their analysis is not relevant to my power.

2 Unit-cost Machines

In this section I study my power in an environment of unit-cost machines. The cost of a profile is simply the number of utilized machines. My goal is therefore to activate as many new machines, and keep them utilized also after my departure. Unfortunately, it turns out that my ambition exceeds my ability: in order to achieve my goal, I need to solve an NP-hard problem. Moreover, the reduction below presents an instance for which (i) $cost(S_0) = 4$, (ii) for every $S_T \in TNE(S_0)$ it holds that $cost(S_T) \in \{4, 5\}$, and (iii) an NP-hard problem should be solved in order to lead the jobs to a profile of cost 5. This implies that it is unlikely to have an algorithm that approximates my potential damage with ratio better than 4/5, and thus, my mission is APX-hard.

▶ **Theorem 2.** *My task is APX-hard even with a constant number of unit-cost machines.*

Proof. I show a reduction from the Partition problem. The input is a set I of positive numbers $\{a_1, a_2, ..., a_n\}$ such that $\forall j,\ 0 < a_j < 1$ and $\sum_j a_j = 2$. The goal is to decide whether there exists a subset $I_1 \subset I$, such that $\sum_{j \in I_1} a_j = \sum_{j \in I \setminus I_1} a_j = 1$. Given an instance of Partition, consider the CST game and initial profile depicted in Figure 2(a). The game is played on $\mathcal{M} = \{m_1, m_2, m_3, m_4, m_5\}$, where $\forall i, c(m_i) = 1$. The are $n + 8$ naive jobs. Four jobs of length 1 are *restricted*. Each of them is restricted to go to a different single machine, m_1, m_2, m_4 or m_5. These are the gray jobs in the figure. Since $|M_j| = 1$ for each of these jobs, they will not participate in the BR-sequence. The restricted jobs guarantee that the cost of every profile is at least 4. The lengths, possible strategies, and initial assignments of the other jobs are given in Table 1. My length is $p_0 = 2 + \epsilon$. Note that the last n jobs are originated from the Partition instance. Let J_I denote this set, whose total length is 2.

The initial profile, S_0, depicted in Figure 2(a) is indeed a NE, as jobs can only migrate to machines with a lower or equal load. My goal is to utilize m_3 and keep it utilized after I vanish. As I show, I must be able to solve the Partition problem in order to do it.

▶ **Claim 3.** *I can lead the jobs to a NE on 5 machines if and only if a partition exists.*

Proof. Assume first that a partition exists. Let J_A be a set of jobs $J_A \subset J_I$ such that $\sum_{j \in J_A} a_j = 1$, and let $J_B = J_I \setminus J_A$. Here is a traitor BR-sequence that ends with a NE on 5 machines: First, I'll migrate to m_2 and let the jobs in J_A perform BR. Recall that my length is $2 + \epsilon$. The loads on m_1 and m_2 are 4 and $4 + \epsilon$, respectively. Since $\frac{a_j}{4} > \frac{a_j}{4+\epsilon+a_j}$,

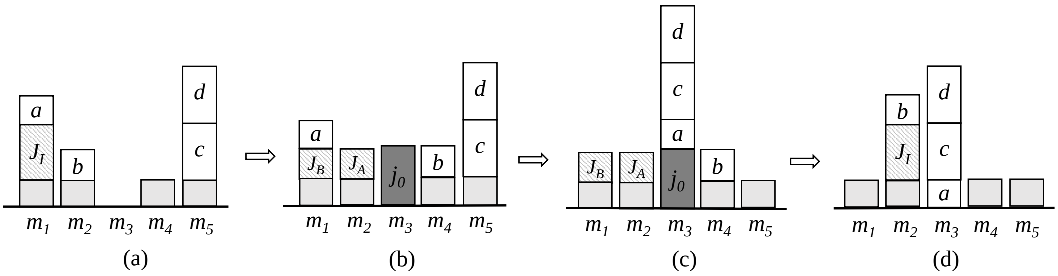


Figure 2 The CST game constructed in the reduction from Partition. (a) The initial profile S_0, (b) The profile before Job a performs BR, (c) The profile after c and d join us on m_3, and (d) the final NE if a Partition exists.

the jobs in J_A will move to m_2. Once the jobs of J_A are all on m_2, I'll migrate to m_4, and let Job b perform BR. He will join me, since $\frac{1}{3} > \frac{1}{4+\epsilon}$. Then, I'll move to m_3. The profile at this time-point is depicted in Figure 2(b). Let's analyze the possible strategies of Job a: If he stays on m_1 or move to m_2 his cost will be 1/3, while if he joins me on m_3 his cost will be $1/(3+\epsilon)$. Thus, joining me on m_3 is his BR. This is exactly what I wanted - now I can attract additional jobs to this machine! After Job a joins me, I will let Jobs c and d perform BR. These guys are required in order to keep Job a on m_3 after I leave. Job c currently pays 2/5. He will join us since $2/5 > 2/(5+\epsilon)$. Job d will clearly follow since $2/3 > 2/(7+\epsilon)$. The profile at this time-point is depicted in Figure 2(c). Stay tuned, we are getting closer to the end of our sequence. Next, I will let the jobs in J_B move, and join their friends in J_A on m_2; Job b will also join them. My mission is now completed - I can vanish, leaving the naive jobs in the profile depicted in Figure 2(d). This profile is a traitor-free NE: Jobs c and d will not return to m_5 since their cost will increase to 2/3. Job a is staying with them since his cost would be 1/5 also on m_2. The jobs of J_I are clearly happy together, and all other four jobs are restricted. The cost of this traitor-free NE is 5.

Assume now that a partition of I does not exist. I argue that I have no chance to keep m_3 utilized. First, note that Jobs c and d will not join me on m_3 if it's only me there, since $2/(4+\epsilon) > 2/5$. Next, let's analyze the conditions for Job a to join me on m_3. In order for m_3 to be his BR, the load on each of m_1 and m_2 must be less than my length, $2+\epsilon$. In any assignment, the jobs of J_I are partitioned such that for some $0 \leq \alpha \leq 2$, jobs of total length α are on m_1 and jobs of total length $2-\alpha$ are on m_2. However, for a small enough ϵ, we have that $\max(1+\alpha, 3-\alpha) \leq 2+\epsilon$ only for $\alpha = 1$. Therefore, if I has no partition, I will not be able to attract anyone to join me on m_3, and exactly 4 machines will be utilized in any traitor-free NE. ◀

◀

2.1 PoT in Games with Unit-length Jobs

In this simplest setting, of unit-cost machines and unit-length jobs, my power is very limited. The price of anarchy in this setting is k. PoA= k is achieved by an instance in which one machine can process all the jobs, but in the NE each job is assigned on a different machine that is only capable to process him. The price of stability in this setting is 1 since an optimal assignment is stable - no job will utilize a new machine. Thus, by Observation 1, my potential power is k. Unfortunately, independent of S_0, I will never be able to lead the naive jobs to a more expensive profile. Formally,

▶ **Theorem 4.** *For $\mathcal{G} = \{$CST games with unit-cost machines and unit-length jobs$\}$, it holds that $PoT(\mathcal{G}) = 1$.*

Proof. Let S_0 be an initial NE. Along any traitor BR-sequence, no naive job activates a new machine, since activating a new machine costs 1, which is the maximal cost of a job in any profile. Assume that I move to a new machine and someone joins me. Each of us now pays 1/2. Such a migration is beneficial only if the other guy was on a machine by himself, implying that some machine was abandoned when he joined me. Moreover, additional jobs may join us, meaning that additional machines may be abandoned. Therefore, whenever I activate a new machine, if someone joins me, then the cost of at least one machine is saved, and if no one joins me then the total cost of the machines for the other guys does not change. Therefore, I will never be able to increase the number of machines that accommodate naive jobs. ◀

2.2 PoT in Games with Arbitrary-length Jobs

In games with unit-cost machines and arbitrary-length jobs, I can do much better. My power varies depending on k and m, and is equal to the price of anarchy [2].

▶ **Theorem 5.** *Let $\mathcal{G} = \{$CST games on unit-cost machines$\}$. (i) For $\mathcal{G}_1 \subseteq \mathcal{G}$ with $k < m$, $PoT\ (\mathcal{G}_1) = k$, (ii) For $\mathcal{G}_2 \subseteq \mathcal{G}$ with $m \leq k \leq 2m - 2$, $PoT\ (\mathcal{G}_2) = m - 1$, (iii) For $\mathcal{G}_3 \subseteq \mathcal{G}$ with $k \geq 2m - 1$, $PoT\ (\mathcal{G}_3) = m$.*

Proof. (i) Assume that $k < m$. Clearly, $cost(S_0) \geq 1$ and in any NE profile the naive jobs may need to cover the cost of at most k machines. Thus, $PoT \leq k$. For the lower bound, given m and k such that $k < m$, consider a game G on $\mathcal{M} = \{m_0, \ldots, m_{m-1}\}$. My length is $p_0 = 2^k$, and for $1 \leq j \leq k$, Job j has length $p_j = 2^j$ and $M_j = \{m_0, m_j\}$. In the initial profile, S_0, the naive jobs are all together on m_0. This is clearly a NE, since a deviating job will need to pay for a new machine.

Here is a traitor BR-sequence that will lead us to a NE on k machines: starting from S_0, I will move to m_k. Since $2^k > \sum_{j=1}^{k-1} 2^j$, poor Job k needs to pay a bit more than 1/2. Since we have the same length, he will gladly join me to share the cost of m_k. I will then move to m_{k-1}. It is now the turn of Job $k-1$ to contribute a bit more than half of the load on m_0, and join me on m_{k-1}. Well, I'm sure you can now complete the sequence by yourself. Eventually, I will be together with Job 1 on m_1, while m_0 is empty and each of $m_2, \ldots, m_k$ is assigned only one job. This is the right time for me to vanish. The resulting profile is a TNE. The only alternative of each naive job is returning to m_0; however, m_0 is empty so it does not attract anyone. Since $cost(S_0) = 1$, and in the final NE the naive jobs are on k machines, we have that $\text{PoT}(G) = k$.

(ii) Assume that $m \leq k \leq 2m-2$. Let me show you that $\text{PoT} = m-1$. The lower bound is similar to the case in which $k < m$. My length is $p_0 = 2^k$, while for $1 \leq j \leq m-1$, Job j has length $p_j = 2^j$ and $M_j = \{m_0, m_j\}$. For $m \leq j \leq k$, Job j has length $p_j = \epsilon$ and his capable machines are $\{m_0, m_{m-1}\}$. In S_0 they are all assigned to m_0. As in case (i), I can lead the jobs to a profile in which they are assigned on $m-1$ machines, by attracting them, one by one starting from the longest job to their 'second' machine. Since $cost(S_0) = 1$, The PoT in this game is $m-1$.

For the upper bound, assume by contradiction that there is a game $G \in \mathcal{G}_2$ such that $\text{PoT}(G) > m-1$. Since S_0 uses at least one machine and any profile uses at most m machines, in an instance with $\text{PoT} > m-1$ it must be that $cost(S_0) = 1$ and some NE costs m. Since there is just one active machine in S_0, this machine is capable for all naive jobs. Denote this machine m_a. Assume that $S_T \in TNE(S_0)$ and $cost(S_T) = m$. It must be that there is at least one naive job on every machine. Since $k \leq 2m - 2$, by the pigeonhole principle, there are at least two machines that are used by exactly one naive job. Let $m_b \neq m_a$ be a machine that is used only by some Job j. Thus, $cost_j(S_T) = 1$. Since there is at least one naive job on m_a and $m_a \in M_j$, by deviating to m_a, Job j can reduce his cost, contradicting the assumption that S_T is a NE, and thus, contradicting the assumption that $\mathrm{PoT}(G) > m - 1$.

(iii) Assume that $k \geq 2m - 1$. Let me present a game $G \in \mathcal{G}_3$ in which $cost(S_0) = 1$ and some NE in $\mathrm{TNE}(S_0)$ uses m machines, thus, $\mathrm{PoT}(G) = m$. This is clearly a tight bound as $cost(S_0) \geq 1$ and any schedule uses at most m machines. Given k, m, consider a game G on $\mathcal{M} = \{m_0, \ldots, m_{m-1}\}$. My length is $p_0 = 2^{2m-2}$, and for $1 \leq j \leq 2m - 2$, Job j has length $p_j = 2^j$ and $M_j = \{m_0, m_{\lceil j/2 \rceil}\}$. Let J_R be the set of jobs $\{j \geq 2m - 1\}$. Since $k \geq 2m - 1$, the set J_R is not empty. The jobs of J_R have total length 1, and are restricted to m_0. In the initial profile, which is clearly a NE, all naive jobs are on m_0.

Here is a traitor BR-sequence that will lead the naive jobs to a NE on m machines: starting from S_0, I will assign myself on m_{m-1}. The load on m_0 is $\sum_{i=0}^{2m-2} 2^i = 2^{2m-1} - 1$. Poor Job $2m - 2$, whose length is 2^{2m-2}, needs to pay a bit more than $1/2$. Since we have the same length, he will gladly join me to share the cost of m_{m-1}. The remaining load on m_0 is $\sum_{i=0}^{2m-3} 2^i = 2^{2m-2} - 1$. It is now the turn of Job $2m - 3$ to contribute a bit more than half of the load on m_0. He will gladly join us on m_{m-1}. I will then move to m_{m-2} and attract the next pair of long jobs on m_0 to join me one after the other. The sequence continues - in turn, I attract the pair with the largest length to their 'second' machine. Eventually, only jobs from J_R, of total length 1, remain on m_0. At this time point, I will vanish. The resulting profile uses m machines and is a NE. The jobs of J_R have no alternative strategy, and the only alternative of the other naive jobs is returning to m_0. However, their current cost is either $1/3$ or $2/3$, so they prefer it over returning to m_0 and share it with J_R. Since $cost(S_0) = 1$, we get that $\mathrm{PoT}(G) = m$. ◀

3 Arbitrary-cost Machines

CST games with arbitrary-cost machines and unit-cost jobs fit the classic model of fair cost-sharing with singleton strategies. For games without a traitor it is known that the PoA is k and the PoS is $\mathcal{H}_k$ were $\mathcal{H}_0 = 0$, and $\mathcal{H}_i = 1 + 1/2 + \ldots + 1/i$. As I show, my power is quite limited, and moreover - my mission is computationally hard. On the other hand, as detailed in Section 4, when jobs may have arbitrary lengths, then my power equals the PoA already in symmetric games.

Let me start with the hardness result.

▶ **Theorem 6.** *My task is APX-hard, even with unit-length jobs and if for every naive job j, $|M_j| \leq 4$.*

Proof. I show a reduction from the maximum 3-bounded 3-dimensional matching problem (3DM-3). The input to the 3DM-3 problem is a set of triplets $U \subseteq X \times Y \times Z$, where $|X| = |Y| = |Z| = n$. The number of occurrences of every element of $X \cup Y \cup Z$ in U is at most 3. The number of triplets is $|U| \geq n$. The desired output is a 3-dimensional matching in U of maximal cardinality; i.e., a subset $U' \subseteq U$, such that every element in $X \cup Y \cup Z$ appears at most once in U', and $|U'|$ is maximal. It is known that 3DM-3 is APX-hard.

Given an instance of 3DM-3, construct the following CST game with unit length jobs. $\mathcal{J} = \{x_1, x'_1 \ldots, x_n, x'_n, y_1, y'_1, \ldots, y_n, y'_n, z_1, z'_1, \ldots, z_n, z'_n\}$, that is, $3n$ pairs of jobs, one pair for every element of $X \cup Y \cup Z$. Let $\mathcal{M} = M_X \cup M_Y \cup M_Z \cup M_U$, where each of M_X, M_Y, M_Z is

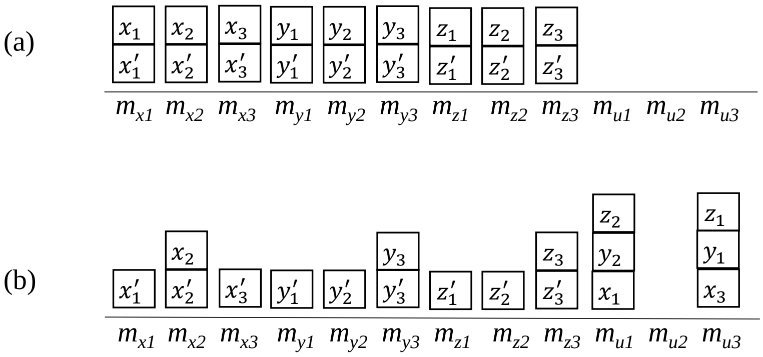

(a)
x_1 x_2 x_3 y_1 y_2 y_3 z_1 z_2 z_3
x'_1 x'_2 x'_3 y'_1 y'_2 y'_3 z'_1 z'_2 z'_3
m_{x1} m_{x2} m_{x3} m_{y1} m_{y2} m_{y3} m_{z1} m_{z2} m_{z3} m_{u1} m_{u2} m_{u3}
(b)
z_2 z_1
x_2 y_3 z_3 y_2 y_1
x'_1 x'_2 x'_3 y'_1 y'_2 y'_3 z'_1 z'_2 z'_3 x_1 x_3
m_{x1} m_{x2} m_{x3} m_{y1} m_{y2} m_{y3} m_{z1} m_{z2} m_{z3} m_{u1} m_{u2} m_{u3}

Figure 3 The CST game constructed for $n = 3$ and $U = \{\{x_1, y_2, z_2\}, \{x_1, y_3, z_3\}, \{x_3, y_1, z_1\}\}$. (a) The initial profile S_0, (b) The NE corresponding to the matching $\{\{x_1, y_2, z_2\}, \{x_3, y_1, z_1\}\}$.

a set of n *element-machines*, one machine per element, and M_U is a set of $|U|$ *triplet-machines*, one machine per triplet.

Every machine in M_X costs $3 + \epsilon$, every machine in $M_Y \cup M_Z$ costs $2 + \epsilon$, and every machine in M_U costs 3. The feasible machines for the naive jobs are as follows: For the pair x_i and x'_i, Job x_i can choose between m_{x_i} and any machine of a triplet he belongs to, while Job x'_i is restricted to go to machine m_{x_i}. Formally, $M_{x_i} = \{m_{x_i}\} \cup \{m_{u_\ell} | x_i \in u_\ell\}$, and $M_{x'_i} = \{m_{x_i}\}$. Similarly, for the pair y_j and y'_j, $M_{y_j} = \{m_{y_j}\} \cup \{m_{u_\ell} | y_j \in u_\ell\}$, and $M_{y'_j} = \{m_{y_j}\}$, and for the pair z_k and z'_k, $M_{z_k} = \{m_{z_k}\} \cup \{m_{u_\ell} | z_k \in u_\ell\}$, and $M_{z'_k} = \{m_{z_k}\}$.

In the initial assignment, S_0, every pair is assigned on his dedicated machine. See an example in Figure 3(a). It is easy to verify that S_0 is a NE. The cost for every job corresponding to an X-element is $(3 + \epsilon)/2$, the cost for every other job is $(2 + \epsilon)/2$. Any migration of a naive job is associated with an activation of a triplet-machine and is therefore not beneficial. It holds that $cost(S_0) = 7n + 3n\epsilon$.

▶ **Claim 7.** *I can lead to a NE whose cost is $cost(S_0) + 3w$ if and only if a matching of size w exists.*

Proof. Let $W = \{u_1, \ldots, u_w\}$ be a 3-dim matching of size w. The traitor BR-sequence I will lead from S_0 consists of w iterations, in each of them I attract the elements of one triplet to their triplet-machine. Assume $\langle x_i, y_j, z_k \rangle = u_\ell \in W$. After I move to m_{u_ℓ}, I offer Job x_i to perform a BR-move. His current cost, on m_{x_i}, is $(3 + \epsilon)/2$, and he can reduce it to $3/2$ by joining me. All other triplet-machines that are capable to process him are empty, and therefore, joining me is his BR. Once he joins me, I offer y_j to perform a BR-move. Since $(2 + \epsilon)/2 > 3/3$, he would join us. Next, z_k will join us since $(2 + \epsilon)/2 > 3/4$. I will then move to attract the next triplet to their triplet-machine. After w such iterations, I will vanish. The resulting profile (see Figure 3(b)) is a traitor-free NE - the jobs assigned to the w triplet-machines each pays $3/3 = 1$ while returning to their element-machines will cost them $(3 + \epsilon)/2$ or $(2 + \epsilon)/2$. The other jobs are either restricted to their machine (jobs of type x'_i, y'_j or z'_k), or can move to an empty triplet-machine - which is not beneficial.

For the other side of the reduction assume that there is a traitor BR-sequence that ends in a NE S_T whose cost is $cost(S_0) + 3w$. For every element-machine there is one job (the 'prime'-job) who is restricted to it. Also, all element-machines are utilized in S_0; therefore, in order to achieve cost $cost(S_0) + 3w$, exactly w triplet-machines are utilized in S_T. I claim that each such machine is assigned all its corresponding triplet. If u_ℓ is assigned only two jobs, then the cost for each of them is 1.5. Since at least one of the two jobs is a Y-job or a Z-job, he can migrate from m_{u_ℓ} to join his pair in S_0 for cost $1 + \epsilon/2$. Therefore, S_T is stable only if w machines are assigned their corresponding triplets - inducing a matching of size w. ◀

◀

3.1 PoT in Games with unit-length jobs

In order to bound my power in this setting, let us consider a stronger model, in which the jobs are allowed to perform *better*-response moves, and not only best-response ones. An *upgraded traitor* has the ability to select the next job to deviate and also his next strategy, as long as it is better than his current strategy. My power in this model is at least as high as in the regular model, since every BR-sequence is also a better-response one. The upper bound for the PoT in CST games with unit-length jobs is valid also for the stronger model, while the lower bound in my analysis below is achieved already by a traitor BR-sequence.

Let $\mathcal{G} = \{$CST games with unit-length jobs$\}$. Let $G \in \mathcal{G}$ and let S_0 be an initial profile in G, such that PoT($\mathcal{G}$) is achieved by G starting from S_0. The proof of the following lemma is based on the fact that the set $\mathcal{M}$ can be tailored to include machines that can only accommodate one specific job, and I can attract the jobs to these machines.

▶ **Lemma 8.** *W.l.o.g., in the worst traitor better-response sequence from S_0, all the initial machines are emptied, and every job migrates exactly once.*

Proof. Let S_T be the most expensive profile in TNE(S_0). Assume by contradiction that there is a machine m_a that was utilized in S_0 and is not empty in S_T. Assume that $L_a(S_T) = \ell$. Consider a game G' in which $\mathcal{M}' = \mathcal{M} \cup \{m'_1, \dots m'_\ell\}$. For $1 < z \leq \ell$, define $c(m'_z) = 2c(m_a)/z - \epsilon$. For m'_1 define $c(m'_1) = c(m_a)$. Let $j_1, j_2, \dots, j_\ell$ be the jobs on m_a in S_T according to the order they joined m_a. It is possible that some of them were on m_a in S_0, in which case their enumeration is arbitrary. In G', for every $1 \leq z \leq \ell$ define $M'_{j_z} = M_{j_z} \cup \{m'_z\}$. Clearly, for all z, $cost_{j_z}(S_T) = c(m_a)/\ell$. I claim that I can lead G' from S_0 to a NE of cost $cost(S_0) + \sum_{z=1}^{\ell} c(m'_z)$. Thus, G' has a higher PoT. The traitor better-response sequence from S_0 in G' will be identical to the sequence in G with the following suffix: Before I vanish, I will migrate to machine m'_ℓ. Since $c(m'_\ell)/2 < c(m_a)/\ell$, joining me is attractive for j_ℓ. I will continue in a similar way to evacuate m_a. Note that $c(m'_1) = c(m_a)$. The machine m'_1 is essential, since it is important to make sure that j_1 also leaves m_a: this guarantees that the resulting profile is a NE - for all $1 \leq z \leq \ell$, we have $c(m'_z) \leq c(m_a)$, and therefore none of them has an incentive to return to m_a and attract the other jobs back after I'm gone. Also, since m'_z is only capable to process Job j_z, no other job is affected. The above extension of the traitor better-response sequence can be applied for any machine which is utilized in S_0 but not emptied along sequence.

Using a similar extension of the sequence, I can show that there exists a worst traitor better-response sequence from S_0, in which every utilized machine accommodates exactly one naive job. Finally, let me show that there exists a worst sequence in which every naive job j migrates exactly once - from machine $s_j(S_0)$ to some new machine: By the above, there exists a sequence in which all the machines that were active in S_0 are empty in S_T and every utilized machine accommodates a single job. Therefore, every job migrates at least once. Assume by contradiction that there are naive jobs who migrate more than once. Let j be the naive job who performed the last before-last migration. By the choice of j, after his before-last migration there were migrations only to final destinations, and according to the properties above, these migrations are into new dedicated machines - each accommodating a single job.

Assume that in his before-last migration, Job j moved from m_a to m_b. Let ℓ_a and ℓ_b denote the loads on m_a and m_b, respectively, before the migration of Job j from m_a to m_b. The migration is beneficial, therefore, $c(m_a)/\ell_a > c(m_b)/(\ell_b+1)$. Given that m_b is about to be evacuated, there exists some job on m_b who will be the first to migrate to some dedicated machine m'. His move would be beneficial, so his cost on m' will be less than $c(m_b)/(\ell_b+1)$. Define a game G' in which $M'_j = M_j \cup \{m'\}$. Consider the sequence in which before the migration of job j I move to m' and then let Job j perform a BR move. Joining me will be j's best-response. In G' I need to permute the dedicated machines allowed for each of the jobs currently on M_b - such that it will be emptied as before, maybe in a different order. This permutation however does not hurt the total cost of the machines activated due to the jobs leaving m_b. Therefore, there exists a sequence in which the before-last migration of j is saved, and the total cost of machines utilized is not hurt. The above process can be repeated as long as there are jobs migrating more than once – to end up with a sequence fulfilling the properties stated in the lemma. ◀

Based on the above characterization, the PoT can be bounded as follows.

▶ **Theorem 9.** *For $\mathcal{G} = \{$CST games with unit-length jobs $\}$, it holds that PoT$(\mathcal{G}) = 2H_k - 1$.*

Proof. By Lemma 8, the PoT is achieved by emptying one by one the machines in S_0, where every job is attracted to a new machine. Thus, for every machine m_i, the load on m_i reduces during the traitor BR-sequence from $L_i(S_0)$ to 0. A naive job j that leaves his machine m_a when the load on it is ℓ will be attracted to join me on a new machine only if its cost is less than $\frac{2c(m_a)}{\ell}$. Also, if $\ell = 1$ then I can attract j to a machine of cost at most $c(m_a)$, as otherwise, j will return to m_a after I'm gone, and will also attract the other jobs back to it. For $\ell > 1$, the new machines have cost lower than $\frac{2c(m_a)}{\ell} \leq c(m_a)$, and therefore, the trapped jobs will not have an incentive to return to m_a after I vanish. The total cost of machines I will utilize in order to empty m_a is therefore less than $c(m_a) + \sum_{\ell=2}^{L_a(S_0)} \frac{2c(m_a)}{\ell} = c(m_a)(2\mathcal{H}_{L_a(S_0)} - 1)$. Summing over all the machines in S_0, we get that for the final NE S_T, $cost(S_T) < \sum_{i|L_i(S'_0)>0} c(i)(2H_{L_i(S'_0)} - 1)$. For at least one machine, $L_i(S_0) \leq k$, implying that $cost(S_T) < cost(S_0)(2H_k - 1)$.

For the lower bound, let me describe a CST game in which I can lead the jobs to a NE whose cost is arbitrarily close to $cost(S_0)(2H_k - 1)$. The game is played on $m = k+1$ machines, $\mathcal{M} = \{0, 1, \ldots, k\}$. The cost of machine m_0 is $1 + \epsilon$, for $1 \leq i < k$, we have $c(m_i) = \frac{2}{k-i+1}$, and $c(m_k) = 1$. There are k unit-length jobs, where for $1 \leq j \leq k$, $M_j = \{m_0, m_j\}$. In the initial profile, S_0, all the jobs are on m_0. Since the cost for each naive job is $\frac{1+\epsilon}{k}$ and the cheapest empty machine has cost $\frac{2}{k}$, S_0 is a NE. Here is a traitor BR-sequence I can initiate: First, I appear on m_1, whose cost is $\frac{2}{k}$. Joining me is beneficial and possible for Job 1. Once he migrates and joins me, I move further to m_2 and let Job 2 perform best-response. His current cost is $\frac{1+\epsilon}{k-1}$ and I offer him a cheaper alternative. I continue to attract the jobs one after the other until eventually, m_0 is empty, m_i accommodates Job i, for all $1 \leq i \leq k-1$, and m_k is shared by Job k and myself. My mission is completed. The resulting profile is a NE: the naive players have no incentive to activate m_0, since each of them has current cost at most 1.

The cost of this NE is $\sum_{i=1}^{k} c(m_i) = \sum_{i=1}^{k-1} \frac{2}{k-i+1} + c(m_k) = 2(H_k - 1) + 1 = 2H_k - 1$. Since $cost(S_0) = 1 + \epsilon$, the PoT is arbitrarily close to $2H_k - 1$. ◀

4 Symmetric Games

In a symmetric game, all the players have the same set of feasible machines. W.l.o.g., for all j, $M_j = \mathcal{M}$. It is well-known that in symmetric games, all the players use the same strategy in every NE. Indeed, if two naive players use different strategies, then at least one of them would benefit from joining the other. It is also known that PoA= k and PoS=1 in this settings [1], where the high PoA is achieved even with unit-length jobs.

In this section I show that with unit-length jobs or with unit-cost machines I have no power, that is, I cannot lead the players to a NE worse than S_0. I then suggest an efficient way to increase my power: I consider the lightest relaxation of the unit-length condition, and show that allowing me to have an arbitrary length (while all other jobs have unit-length), is sufficient to achieve PoT=PoA= k. I then provide a tight bound for my power with arbitrary-length jobs and arbitrary-cost machines.

The first theorem follows trivially from the fact that in symmetric games all the players use a single machine in every NE.

▶ **Theorem 10.** *For $\mathcal{G} = \{$symmetric CST games with unit-cost machines$\}$, it holds that $PoT(\mathcal{G}) = 1$.*

Next, let me show that I cannot be of any help also with unit-length jobs.

▶ **Theorem 11.** *For $\mathcal{G} = \{$symmetric CST games with unit-length jobs$\}$, it holds that $PoT(\mathcal{G}) = 1$.*

Proof. Since the game is symmetric, in S_0 all the jobs are on the same machine, say m_0. Assume w.l.o.g., that $cost(S_0) = c(m_0) = 1$. For $k = 1$, I will be able to attract the single naive job only to a machine whose cost is less than 2. However, once I vanish, he would return to m_0. Therefore, for every $S_T \in TNE(S_0)$, $cost(S_T) = cost_1(S_T) \leq 1 = cost(S_0)$.

Assume next that $k > 1$. Clearly, I am the only job who may initiate the use of a machine whose cost is more than 1. In order to end up with a more expensive profile, some job must join me on an expensive machine. Assume by contradiction that there exists a traitor BR-sequence in which I attract someone to join me on an expensive machine. Let m_a be the first expensive machine in which a job j joins me. When job j migrates, since $k > 1$, apart from m_a, there is at least one active machine m_b utilized by jobs in $\mathcal{J} \setminus \{0, j\}$. Since m_a is the first expensive machine to accommodate a naive job, it must be that $c(m_b) \leq 1$. Therefore, Job j has an alternative strategy, m_b, in which his cost would be at most 1/2, contradicting the assumption that his BR-move is to join me on m_a. We conclude that naive jobs will only migrate to machines of cost at most 1. Since they will end-up on a single machine, we get that also for $k > 1$, $PoT = 1$. ◀

To increase my power in symmetric games with unit-length jobs, I asked my operators to increase my processing time. Gladly, this works above and beyond everyone's expectations:

▶ **Theorem 12.** *For $\mathcal{G} = \{$symmetric CST games with unit-length jobs and arbitrary-length traitor$\}$, it holds that $PoT(\mathcal{G}) = k$.*

Proof. The upper bound follows from Observation 1 and the fact that in cost sharing symmetric games $PoA \leq k$ [15]. The lower bound is a generalization for arbitrary k of Example 1. Consider an instance with two machines m_1 and m_2 of costs 1 and k respectively. Assume that k unit-length jobs are assigned on m_1. Assume now that I appear and assign myself on m_2. My length is $k^2 - 1 + \epsilon$, thus, a job that joins me would pay $k \cdot \frac{1}{k^2+\epsilon}$, which is less than $\frac{1}{k}$, his current cost on m_1. The other jobs will follow, and once they are all on m_2, I will be gone, leaving them on in a traitor-free NE of cost k. ◀

Table 2 My power in various environments. In entries marked by [⋆], the PoT is lower than the PoA/PoS-bound and the $PoA = k$. In all other entries, $PoT = PoA$.

Jobs \ Machines	unit-cost		arbitrary-cost	
	general	symmetric	general	symmetric
unit-length	1 [⋆]	1	$\Theta(\mathcal{H}_k)$ [⋆]	1 [⋆]
arbitrary-length	$\min(m, k)$	1	k	$\frac{\sum_j p_j}{\max_j p_j}$

Let's consider next instances with arbitrary-length jobs and arbitrary-cost machines. Let $L(\mathcal{J}) = \sum_{j \in \mathcal{J}} p_j$ be the total length of the naive jobs, and let $\alpha(\mathcal{J}) = max_{j \in \mathcal{J}} p_j / L(\mathcal{J})$. Theorem 12 analyzes the case $\alpha(\mathcal{J}) = 1/k$. The following theorem generalizes it for arbitrary $\alpha(\mathcal{J})$.

▶ **Theorem 13.** *For $\mathcal{G} = \{$symmetric CST games$\}$, it holds that $PoT(\mathcal{G}) = PoA(\mathcal{G}) = 1/\alpha(\mathcal{J})$.*

Proof. Let me first bound the PoA. Assume that Job 1 determines the value of α, and that $OPT = 1$. Let S be a NE profile. Since the game is symmetric, all the jobs are on a single machine, and the cost of Job 1 is $\alpha \cdot cost(S)$. He can deviate to the machine of cost 1 utilized in the optimal profile, therefore, in order for S to be a NE, it must hold that $\alpha \cdot cost(S) \leq 1$, implying $cost(S) \leq 1/\alpha$.

The lower bound is a generalization of Example 1. Given a set of naive jobs $\mathcal{J}$, let $L = L(\mathcal{J}), \alpha = \alpha(\mathcal{J})$. Assume that Job 1 determines the value of α. Consider a symmetric CST game, on two machines, where $c(m_1) = 1$ and $c(m_2) = \frac{1}{\alpha} = \frac{L}{p_1}$. Let S_0 be the initial stable profile in which all the jobs are on m_1. Assume now that I appear and assign myself on m_2. My length is $\frac{L^2}{p_1} - p_1 + \epsilon$. Note that if Job 1 joins me, his cost would be $\frac{L}{p_1} \cdot \frac{p_1}{\frac{L^2}{p_1} + \epsilon}$ which is less than $\frac{p_1}{L}$, his current cost on m_1. It is easy to see that the other jobs will follow. Once they are all on m_2, I will be gone. The resulting profile has cost $c(m_2) = \frac{1}{\alpha}$. The cost for a job of length p_j is $\frac{L}{p_1} \cdot \frac{p_j}{L} \leq 1$, thus, no one will migrate back to m_1 and this profile is a traitor-free NE. ◀

5 Conclusions and Plans for My Retirement

Being the evil guy is not an easy task, but a rewarding one. My power is summarized in Table 2. I'm a bit disappointed from my limited power in instances with unit-length jobs, which is significantly lower than the PoA/PoS bound. However, if you run a system that processes arbitrary-length jobs and would like to boost your revenue, you should definitely hire me! If you deal with symmetric jobs then you will greatly enjoy my services if you process arbitrary-length jobs on arbitrary-cost machines.

I am exploring several ways to increase my power. One clear direction is to employ additional secret agents to work with me. I want to analyze the power of several traitors, who coordinate their moves trying to lead the naive jobs to a poor outcome. In this general setting, the number of traitors is γk for some fraction γ. Fooling the naive jobs by a bunch of secret agents could be really fun and rewarding!

I would also like to devise algorithms that calculate, for a given initial profile, a traitor BR-sequence with high PoT. In this paper I proved that the problem is NP-hard, but I believe that there are interesting classes of instances for which it is possible to come up with an optimal sequence, or at least an approximated one. Another interesting problem is to consider the power of a traitor in other congestion games. Specifically, after my retirement, I hope to volunteer in networks, and be in charge of routing messages. After gaining the trust of other players there, I will challenge myself harming the social cost in network formation games.

Alternatively, I may enter the world of congestion games – in which the cost associated with using a resource increases with the load on it. It seems that a totally different approach is required in such games, because I will no longer attract naive players to join me, but to get away from me. In general, almost every congestion game becomes more interesting when a single or multiple traitors are involved.

References

1 E. Anshelevich, A. Dasgupta, J. Kleinberg, E. Tardos, T. Wexler, and T. Roughgarden. The price of stability for network design with fair cost allocation. *SIAM Journal on Computing*, 38(4):1602–1623, 2008.

2 G. Avni and T. Tamir. Cost-sharing scheduling games on restricted unrelated machines. *Theoretical Computer Science*, 646:26–39, 2016.

3 V. Bilò and C. Vinci. On the impact of singleton strategies in congestion games. In *Proc. 25th Annual European Symposium on Algorithms*, pages 17:1–17:14, 2017.

4 Ioannis Caragiannis, Michele Flammini, Christos Kaklamanis, Panagiotis Kanellopoulos, and Luca Moscardelli. Tight bounds for selfish and greedy load balancing. *Algorithmica*, 61(3):606–637, 2011.

5 E.Even-Dar, A.Kesselman, and Y.Mansour. Convergence time to nash equilibria. In *Proc. 30th Int. Colloq. on Automata, Languages, and Programming*, pages 502–513, 2003.

6 A. Fabrikant, C. Papadimitriou, and K. Talwar. The complexity of pure nash equilibria. In *Proc. 36th ACM Symp. on Theory of Computing*, pages 604–612, 2004.

7 A. Fanelli, M. Flammini, and L. Moscardelli. Stackelberg strategies for network design games. In *Proc. of the 3rd International Conference on Algorithmic Game Theory*, pages 222—233, 2010.

8 M. Feldman, Y. Snappir, and T. Tamir. The efficiency of best-response dynamics. In *Proc. of the 10th International Symposium on Algorithmic Game Theory (SAGT)*, 2017.

9 D. Fisman, O. Kupferman, and Y. Lustig. Rational synthesis. In *The 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 190–204, 2010.

10 D. Fotakis. Stackelberg strategies for atomic congestion games. *Theory of Computing Systems*, 47(1):218–249, 2010.

11 Vasilis Gkatzelis, Konstantinos Kollias, and Tim Roughgarden. Optimal cost-sharing in general resource selection games. *Operations Research*, 64(6):1230–1238, 2016.

12 T. Harks and M. Klimm. On the existence of pure nash equilibria in weighted congestion games. *Math. Oper. Res.*, 37(3):419–436, 2012.

13 Samuel Ieong, Robert McGrew, Eugene Nudelman, Yoav Shoham, and Qixiang Sun. Fast and compact: A simple class of congestion games. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, AAAI'05, 2005.

14 Yannis A. Korilis, Aurel A. Lazar, and Ariel Orda. Achieving network optima using stackelberg routing strategies. *IEEE/ACM Trans. Netw.*, 5(1):161–173, 1997.

15 E. Koutsoupias and C. Papadimitriou. Worst-case equilibria. *Computer Science Review*, 3(2):65–69, 2009.

16 Anna Lysyanskaya and Nikos Triandopoulos. Rationality and adversarial behavior in multiparty computation. In *Advances in Cryptology - CRYPTO 2006*, pages 180–197, 2006.

17 I. Milchtaich. Congestion games with player-specific payoff functions. *Games and Economic Behavior*, 13(1):111–124, 1996.

18 Shien Jin Ong, David C. Parkes, Alon Rosen, and Salil Vadhan. Fairness with an honest minority and a rational majority. In Omer Reingold, editor, *Theory of Cryptography*, pages 36–53, 2009.

19 C. H. Papadimitriou. Algorithms, games, and the internet. In *Proc. 33rd ACM Symp. on Theory of Computing*, pages 749–753, 2001.

20 M. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 2008.

21 R. W. Rosenthal. A class of games possessing pure-strategy nash equilibria. *International Journal of Game Theory*, 2:65–67, 1973.

22 V. Syrgkanis. The complexity of equilibria in cost sharing games. In *WINE*, volume 6484, pages 366–377. Springer, 2010.

23 Tami Tamir. Scheduling with bully selfish jobs. In *Proceedings of the 5th International Conference on Fun with Algorithms*, FUN'10, pages 355–367, 2010.

24 B. Vöcking. *Algorithmic Game Theory*, chapter 20: Selfish Load Balancing. Cambridge University Press, 2007.