

12th International Conference on Fun with Algorithms

FUN 2024, June 4–8, 2024, Island of La Maddalena, Sardinia, Italy

Edited by

Andrei Z. Broder

Tami Tamir



Editors

Andrei Z. Broder

Google
broder@acm.org

Tami Tamir

Reichman University, Herzliya, Israel
tami@runi.ac.il

ACM Classification 2012

Theory of computation → Design and analysis of algorithms

ISBN 978-3-95977-314-0

Published online and open access by

Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at <https://www.dagstuhl.de/dagpub/978-3-95977-314-0>.

Publication date

June, 2024

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 <https://portal.dnb.de>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0):
<https://creativecommons.org/licenses/by/4.0/legalcode>.



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

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

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.FUN.2024.0

ISBN 978-3-95977-314-0

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

LIPICs – Leibniz International Proceedings in Informatics

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

Editorial Board

- Luca Aceto (Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University, Brno, CZ)
- Meena Mahajan (*Chair*, Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)
- Pierre Senellart (ENS, Université PSL, Paris, FR)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Andrei Z. Broder and Tami Tamir</i>	0:ix
Conference Organization	
.....	0:xi–0:xii
Authors	
.....	0:xiii–0:xv

FUN with Algorithms

Baba Is Universal	
<i>Zachary Abel and Della Hendrickson</i>	1:1–1:15
Poset Positional Games	
<i>Guillaume Bagan, Eric Duchêne, Florian Galliot, Valentin Gledel,</i> <i>Mirjana Mikalački, Nacim Oijid, Aline Parreau, and Miloš Stojaković</i>	2:1–2:12
Snake in Optimal Space and Time	
<i>Philip Bille, Martín Farach-Colton, Inge Li Gørtz, and Ivor van der Hoog</i>	3:1–3:15
Uniform-Budget Solo Chess with Only Rooks or Only Knights Is Hard	
<i>Davide Bilò, Luca Di Donato, Luciano Gualà, and Stefano Leucci</i>	4:1–4:19
Swapping Mixed-Up Beers to Keep Them Cool	
<i>Davide Bilò, Maurizio Fiusco, Luciano Gualà, and Stefano Leucci</i>	5:1–5:18
Bottom-Up Rebalancing Binary Search Trees by Flipping a Coin	
<i>Gerth Stølting Brodal</i>	6:1–6:15
Physical Ring Signature	
<i>Xavier Bultel</i>	7:1–7:18
A Tractability Gap Beyond Nim-Sums: It’s Hard to Tell Whether a Bunch of Superstars Are Losers	
<i>Kyle Burke, Matthew Ferland, Svenja Huntemann, and Shanghua Teng</i>	8:1–8:14
The Steady-States of Splitter Networks	
<i>Basile Couëtoux, Bastien Gastaldi, and Guylain Naves</i>	9:1–9:14
How Did They Design This Game? SWISH: Complexity and Unplayable Positions	
<i>Antoine Dailly, Pascal Lafourcade, and Gaël Marcadet</i>	10:1–10:19
Hamiltonian Paths and Cycles in NP-Complete Puzzles	
<i>Marnix Deurloo, Mitchell Donkers, Mieke Maarse, Benjamin G. Rin, and Karen Schutte</i>	11:1–11:25
Card-Based Cryptography Meets Differential Privacy	
<i>Reo Eriguchi, Kazumasa Shinagawa, and Takao Murakami</i>	12:1–12:20

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples <i>Simone Faro, Francesco Pio Marino, Andrea Moschetto, Arianna Pavone, and Antonio Scardace</i>	13:1–13:17
PACKIT!: Gamified Rectangle Packing <i>Thomas Garrison, Marijn J. H. Heule, and Bernardo Subercaseaux</i>	14:1–14:19
Polyamorous Scheduling <i>Leszek Gąsieniec, Benjamin Smith, and Sebastian Wild</i>	15:1–15:18
Tetris Is Not Competitive <i>Matthias Gehnen and Luca Venier</i>	16:1–16:16
Computational Complexity of Matching Match Puzzle <i>Yuki Iburi and Ryuhei Uehara</i>	17:1–17:10
Advanced Spikes ‘n’ Stuff: An NP-Hard Puzzle Game in Which All Tutorials Are Efficiently Solvable <i>Christian Ikenmeyer and Dylan Khangure</i>	18:1–18:13
Anarchy in the APSP: Algorithm and Hardness for Incorrect Implementation of Floyd-Warshall <i>Jaehyun Koo</i>	19:1–19:11
Variations on the Tournament Problem <i>Fabrizio Luccio, Linda Pagli, and Nicola Santoro</i>	20:1–20:11
PSPACE-Hard 2D Super Mario Games: Thirteen Doors <i>MIT Hardness Group, Hayashi Ani, Erik D. Demaine, Holden Hall, and Matias Korman</i>	21:1–21:19
You Can’t Solve These Super Mario Bros. Levels: Undecidable Mario Games <i>MIT Hardness Group, Hayashi Ani, Erik D. Demaine, Holden Hall, Ricardo Ruiz, and Naveen Venkat</i>	22:1–22:20
ASP-Completeness of Hamiltonicity in Grid Graphs, with Applications to Loop Puzzles <i>MIT Hardness Group, Josh Brunner, Lily Chung, Erik D. Demaine, Della Hendrickson, and Andy Tockman</i>	23:1–23:20
Tetris with Few Piece Types <i>MIT Hardness Group, Erik D. Demaine, Holden Hall, and Jeffery Li</i>	24:1–24:18
Complexity of Planar Graph Orientation Consistency, Promise-Inference, and Uniqueness, with Applications to Minesweeper Variants <i>MIT Hardness Group, Della Hendrickson, and Andy Tockman</i>	25:1–25:20
Coordinating “7 Billion Humans” Is Hard <i>Alessandro Panconesi, Pietro Maria Posta, and Mirko Giacchini</i>	26:1–26:16
Arimaa Is PSPACE-Hard <i>Benjamin G. Rin and Atze Schipper</i>	27:1–27:24
No Tiling of the 70×70 Square with Consecutive Squares <i>Jiří Sgall, János Balogh, József Békési, György Dósa, Lars Magnus Hvattum, and Zsolt Tuza</i>	28:1–28:16

Achieving the Highest Possible Elo Rating <i>Rikhav Shah</i>	29:1–29:21
How to Covertly and Uniformly Scramble the 15 Puzzle and Rubik’s Cube <i>Kazumasa Shinagawa, Kazuki Kanai, Kengo Miyamoto, and Koji Nuida</i>	30:1–30:15
A Programming Language Embedded in <i>Magic: The Gathering</i> <i>Howe Choong Yin and Alex Churchill</i>	31:1–31:19

Salon des Refusés

Eating Ice-Cream with a Colander <i>Kien Huynh and Valentin Polishchuk</i>	32:1–32:4
Retrospective: Avoiding the Disk Bottleneck in the Data Domain Deduplication File System <i>Kai Li</i>	33:1–33:4
Short Programs for Functions on Curves: A STOC Rejection <i>Victor S. Miller</i>	34:1–34:4

■ Preface

FUN with Algorithms is dedicated to the use, design, and analysis of algorithms and data structures, focusing on results that provide amusing, witty, yet 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 realize 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; in La Maddalena Island, Sardinia, Italy; and in Island of Favignana, Sicily, Italy. Special issues of Theoretical Computer Science, Discrete Applied Mathematics, and Theory of Computing Systems were dedicated to them.

This volume contains the papers that will be presented at the 12th International Conference on Fun with Algorithms 2024, that will take place on June 4–8, 2024, on La Maddalena Island, Italy. The call for papers attracted 69 submissions from all over the world, addressing a wide variety of topics. These were reviewed by 18 Program Committee members. After a careful reviewing process and a thorough discussion, the committee decided to accept 31 papers. Extended versions of selected papers will appear in a special issue. This year, FUN will also host 4 talks in the Computer Science Salon des Refusés. This special track aims to be a “salon” in two senses of the word: First, “salon” as an exhibition of ideas in Computer Science that in their infancy were rejected from a major venue but later proven to be highly influential, and second “salon” as a meeting of convivial people getting together to share both amusement and learning. Finally, the program features invited talks by Paolo Ferragina and John Iacono.

We thank our invited speakers, Paolo Ferragina and John Iacono, all authors who submitted their work to FUN 2024, all Program Committee members for their expert assessments and the ensuing discussions, and all external reviewers for their kind assistance. 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 and the LIPIcs team for carefully overseeing the proceedings’ publication in the LIPIcs series. Finally, we appreciate the financial support from Università di Pisa under the “PRA – Progetti di Ricerca di Ateneo” (Institutional Research Grants) – Project no. PRA_2022_81.

April, 2024

*Paolo Boldi
Andrei Z. Broder
Giuseppe Prencipe
Tami Tamir*



■ Conference Organization

Program Committee – FUN with Algorithms

Davide Bilò, University of L'Aquila
Erik D. Demaine, MIT
Miriam Di Ianni, University of Rome Tor Vergata
David Eppstein, University of California, Irvine
Guy Even, Tel-Aviv university
Michele Flammini, University of L'Aquila
Dorian Mazauric, Inria, France
William Kuszmaul, Harvard Univ.
Pascal Lafourcade, Université Clermont Auvergne, LIMOS
Anissa Lamani, University of Strasbourg
Neeldhara Misra, IIT Gandhinagar
Daiki Miyahara, University of Electro-Communications
Takaaki Mizuki, Tohoku University
Tim Oosterwijk, Vrije Universiteit Amsterdam
Adele A. Rescigno, University of Salerno
Daniel Schmand, Bremen University
Marc Schröder, Maastricht University
Tami Tamir, Reichman University (**chair**)

Program Committee – Salon de Refusés

Andrei Broder, Google (**chair**)
Ravi Kumar, Google

Steering Committee

Hiro Ito, U. of Electro-Communications, Japan
Stefano Leonardi, Sapienza U. di Roma, Italy
Linda Pagli, U. di Pisa, Italy
Giuseppe Prencipe, U. di Pisa, Italy
Geppino Pucci, U. degli Studi di Padova, Italy
Nicola Santoro, Carleton U., Canada

Organizers

Linda Pagli, U. di Pisa, Italy
Giuseppe Prencipe, U. di Pisa, Italy



External Reviewers

Aaron Putterman
Aditi Sethia
Alessandro Straziota
Anaïs Durand
Andrea D'Ascenzo
Anh Trieu
Antoine Dailly
Antonio Cruciani
Arcangelo Castiglione
Arman Rouhani
Ashkan Safari
Daniele Carnevale
Dipan Dey
Elias Pitschmann
Francesco Pasquale
Frédéric Hayek
Gianluca De Marco
Gianluca Rossi
Gianpiero Monaco
Giovanna Varricchio
Harshil M.
Jean-Romain Luttringer
Joe Politz
Joep van Sloun
John Kuszmaul
Jun Kawahara
Jyothi Krishnan
Laurent Beaudou
Laurent Théry
Leena Salmela
Leo Robert
Lola-Baie Mallordy
Luca Moscardelli
Luca Pepè Sciarria
Luciano Gualà
Malika More
Manoj Gupta
Manuela Flores
Moni Naor
Nicola Cotumaccio
Nicolas Martins
Nicole Schröder
Quentin Bramas
Ravi Kant Rai
Rebecca Lin
Rob van Stee
Roberto Bruno
Rocco Zaccagnino
Rosalba Zizza
Saraswati Nanoti
Simone Fioravanti
Stefano Leucci
Stéphane Devismes
Takeharu Shiraga
Torben Schürenberg
Vijayaragunathan Ramamoorthi
Will Ma
Xavier Bultel
Yan Gerard

■ List of Authors

- Zachary Abel (1)
Massachusetts Institute of Technology,
Cambridge, MA, USA
- Hayashi Ani (21, 22)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Guillaume Bagan (2)
Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS,
UMR5205, F-69622 Villeurbanne, France
- János Balogh  (28)
University of Szeged, Hungary
- Philip Bille  (3)
Department of Applied Mathematics and
Computer Science, Technical University of
Denmark, Lyngby, Denmark
- Davide Bilò  (4, 5)
Department of Information Engineering,
Computer Science and Mathematics, University
of L'Aquila, Italy
- Gerth Stølting Brodal  (6)
Department of Computer Science, Aarhus
University, Denmark
- Josh Brunner  (23)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Xavier Bultel  (7)
INSA Centre Val de Loire, Laboratoire
d'informatique fondamentale d'Orléans, Bourges,
France
- Kyle Burke  (8)
Florida Southern College, Lakeland, FL, USA
- József Békési  (28)
University of Szeged, Hungary
- Lily Chung  (23)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Alex Churchill  (31)
Independent Researcher, Cambridge, UK
- Basile Couëtoux (9)
Aix-Marseille Université, CNRS, LIS, Marseille,
France
- Antoine Dailly (10)
Université Clermont-Auvergne, CNRS, Mines de
Saint-Étienne, Clermont-Auvergne-INP, LIMOS,
63000 Clermont-Ferrand, France
- Erik D. Demaine  (21, 22, 23, 24)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Marnix Deurloo  (11)
Utrecht University, The Netherlands
- Luca Di Donato (4)
Department of Information Engineering,
Computer Science and Mathematics, University
of L'Aquila, Italy
- Mitchell Donkers (11)
Utrecht University, The Netherlands
- Eric Duchêne  (2)
Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS,
UMR5205, F-69622 Villeurbanne, France
- György Dósa  (28)
University of Pannonia, Veszprém, Hungary
- Reo Eriguchi  (12)
National Institute of Advanced Industrial
Science and Technology, Tokyo, Japan
- Martín Farach-Colton  (3)
New York University, NY, USA
- Simone Faro  (13)
Department of Mathematics and Computer
Science, University of Catania, Italy
- Matthew Ferland  (8)
University of Southern California, Los Angeles,
CA, USA
- Maurizio Fiusco (5)
Department of Enterprise Engineering,
University of Rome "Tor Vergata", Italy
- Florian Galliot  (2)
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI,
UMR 5800, F-33400 Talence, France
- Thomas Garrison  (14)
Carnegie Mellon University, Pittsburgh, PA,
USA
- Bastien Gastaldi (9)
Télécom SudParis, Institut Polytechnique de
Paris, Evry, France

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir





Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


Matthias Gehnen  (16)
RWTH Aachen University, Germany

Mirko Giacchini  (26)
Sapienza University of Rome, Italy

Valentin Gledel (2)
Université Savoie Mont Blanc, CNRS UMR5127,
LAMA, Chambéry, F-73000, France


Luciano Gualà  (4, 5)
Department of Enterprise Engineering,
University of Rome "Tor Vergata", Italy


Inge Li Gørtz  (3)
Department of Applied Mathematics and
Computer Science, Technical University of
Denmark, Lyngby, Denmark

Leszek Gąsieniec  (15)
University of Liverpool, UK

Holden Hall (21, 22, 24)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA

Della Hendrickson  (1, 23, 25)
Massachusetts Institute of Technology,
Cambridge, MA, USA


Marijn J. H. Heule  (14)
Carnegie Mellon University, Pittsburgh, PA,
USA


Svenja Huntemann  (8)
Mount Saint Vincent University, Halifax,
Canada

Kien Huynh (32)
Communications and Transport Systems, ITN,
Linköping University, Sweden


Lars Magnus Hvattum  (28)
Molde University College, Norway

Yuki Iburi (17)
The Digital Value, LTD., Tokyo, Japan


Christian Ikenmeyer  (18)
University of Warwick, UK


Kazuki Kanai  (30)
National Institute of Technology, Kure College,
Hiroshima, Japan

Dylan Khangure (18)
University of Warwick, UK


Jaehyun Koo  (19)
MIT EECS and CSAIL, Cambridge, MA, USA

Matias Korman (21)
Siemens Electronic Design Automation,
Wilsonville, OR, USA

Pascal Lafourcade  (10)
Université Clermont-Auvergne, CNRS, Mines de
Saint-Étienne, Clermont-Auvergne-INP, LIMOS,
63000 Clermont-Ferrand, France


Stefano Leucci  (4, 5)
Department of Information Engineering,
Computer Science and Mathematics, University
of L'Aquila, Italy

Jeffery Li (24)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA


Kai Li  (33)
Department of Computer Science, Princeton
University, NJ, USA


Fabrizio Luccio (20)
University of Pisa, Italy

Mieke Maarse (11)
Utrecht University, The Netherlands

Gaël Marcadet  (10)
Université Clermont-Auvergne, CNRS, Mines de
Saint-Étienne, Clermont-Auvergne-INP, LIMOS,
63000 Clermont-Ferrand, France

Francesco Pio Marino (13)
Department of Mathematics and Computer
Science, University of Catania, Italy; Univ
Rouen Normandie, INSA Rouen Normandie,
Université Le Havre Normandie, Normandie
Univ, LITIS UR 4108, CNRS NormaSTIC FR
3638, IRIB, Rouen, F-76000, France

Mirjana Mikalački  (2)
Department of Mathematics and Informatics,
Faculty of Sciences, University of Novi Sad,
Serbia

Victor S. Miller  (34)
Computer Science Laboratory, SRI, Menlo Park,
CA, USA

MIT Hardness Group (21, 22, 23, 24, 25)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA

Kengo Miyamoto  (30)
Ibaraki University, Japan

Andrea Moschetto (13)
Department of Mathematics and Computer
Science, University of Catania, Italy

- Takao Murakami  (12)
The Institute of Statistical Mathematics,
Tachikawa, Japan; National Institute of
Advanced Industrial Science and Technology,
Tokyo, Japan
- Guyslain Naves  (9)
Aix-Marseille Université, CNRS, LIS, Marseille,
France
- Koji Nuida  (30)
Institute of Mathematics for Industry (IMI),
Kyushu University, Fukuoka, Japan; National
Institute of Advanced Industrial Science and
Technology (AIST), Tokyo, Japan
- Nacim Ojjid  (2)
Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS,
UMR5205, F-69622 Villeurbanne, France
- Linda Pagli (20)
University of Pisa, Italy
- Alessandro Panconesi  (26)
Sapienza University of Rome, Italy
- Aline Parreau  (2)
Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS,
UMR5205, F-69622 Villeurbanne, France
- Arianna Pavone  (13)
Department of Mathematics and Computer
Science, University of Palermo, Italy
- Valentin Polishchuk (32)
Communications and Transport Systems, ITN,
Linköping University, Sweden
- Pietro Maria Posta  (26)
Sapienza University of Rome, Italy
- Benjamin G. Rin  (11, 27)
Utrecht University, The Netherlands
- Ricardo Ruiz (22)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Nicola Santoro (20)
Carleton University, Ottawa, Canada
- Antonio Scardace (13)
Department of Mathematics and Computer
Science, University of Catania, Italy
- Atze Schipper (27)
Utrecht University, The Netherlands
- Karen Schutte  (11)
Utrecht University, The Netherlands
- Jiří Sgall  (28)
Computer Science Institute of Charles
University, Faculty of Mathematics and Physics,
Prague, Czech Republic
- Rikhav Shah  (29)
University of California, Berkeley, CA, USA
- Kazumasa Shinagawa  (12, 30)
Ibaraki University, Japan; National Institute of
Advanced Industrial Science and Technology,
Tokyo, Japan
- Benjamin Smith  (15)
University of Liverpool, UK
- Miloš Stojaković  (2)
Department of Mathematics and Informatics,
Faculty of Sciences, University of Novi Sad,
Serbia
- Bernardo Subercaseaux  (14)
Carnegie Mellon University, Pittsburgh, PA,
USA
- Shanghai Teng  (8)
University of Southern California, Los Angeles,
CA, USA
- Andy Tockman  (23, 25)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Zsolt Tuza  (28)
University of Pannonia, Veszprém, Hungary;
HUN-REN Alfréd Rényi Institute of
Mathematics, Budapest, Hungary
- Ryuhei Uehara  (17)
School of Information Science, Japan Advanced
Institute of Science and Technology, Tokyo,
Japan
- Ivor van der Hoog  (3)
Department of Applied Mathematics and
Computer Science, Technical University of
Denmark, Lyngby, Denmark
- Luca Venier  (16)
RWTH Aachen University, Germany
- Naveen Venkat (22)
CSAIL, Massachusetts Institute of Technology,
Cambridge, MA, USA
- Sebastian Wild  (15)
University of Liverpool, UK
- Howe Choong Yin (31)
Independent Researcher, Singapore

Baba Is Universal

Zachary Abel ✉

Massachusetts Institute of Technology, Cambridge, MA, USA

Della Hendrickson ✉ 

Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We consider the computational complexity of constant-area levels of games which allow an unlimited number of objects in a fixed region. We discuss how to prove that such games are RE-hard (and in particular undecidable) and capable of universal computation, even on constant-area levels. We use the puzzle game Baba is You as a case study, showing that 8×17 levels are capable of universal computation by constructing a particular small universal counter machine within Baba is You.

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

Keywords and phrases Undecidability, Baba is You, RE-hardness, counter machines, universal computation

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.1

Acknowledgements This work was initiated at the 34th Bellairs Winter Workshop on Computational Geometry in March 2019 in Holetown, Barbados, which was organized by Erik Demaine and Godfried Toussaint. We thank the other participants for helpful discussion and contributing to a collaborative and productive research environment.

1 Introduction

There has recently been a surge of computational complexity results for games, especially video games [18, 1, 10, 9, 6]. Most commonly, games (technically, decision problems of the form “given a level of this game, is it possible to win?”) are shown to be NP- or PSPACE-complete, and frameworks have been developed to simplify such proofs [14, 18, 1, 7, 5]. There is not as much infrastructure for other complexity classes, so proofs of e.g. EXP-hardness are more bespoke, from more “primitive” problems like formula games [17].

In this paper, we work towards resolving this infrastructure gap for RE-hardness. We are aware of only a few RE-hardness (which implies undecidability) results for single-player video games: Braid [13], Recursed [8] New Super Mario Bros. [2], and very recent work extending the latter to other Mario games [12]. The result for Recursed uses a fundamentally different approach, but the others use counter machines, inspiring this work on Baba is You. Ani [2] begins to develop a framework for RE-hardness proofs, by extending the motion-planning gadget framework [7] to allow gadgets with infinitely many states, which can act as registers for a counter machine.

In this paper, we focus on *constant-area levels*, meaning instances of a game that have a fixed size in the game world, but may contain any number of objects. To obtain any kind of hardness in this setting, we need a reduction that encodes information in the number of objects present in a small area – for us, in a single cell. In particular, we explain how – and demonstrate with Baba is You – to prove RE-hardness for two decision problems:

1. Given a constant-area level, is it possible to win?
2. For a specific constant level, given a sequence of inputs, is it possible to win after performing those inputs?

These decision problems are always in RE (the class of recognizable problems), for reasonable single-player games (including Baba is You): we can enumerate over all possible input sequences, test whether each results in victory, and accept if so.



© Zachary Abel and Della Hendrickson;
licensed under Creative Commons License CC-BY 4.0

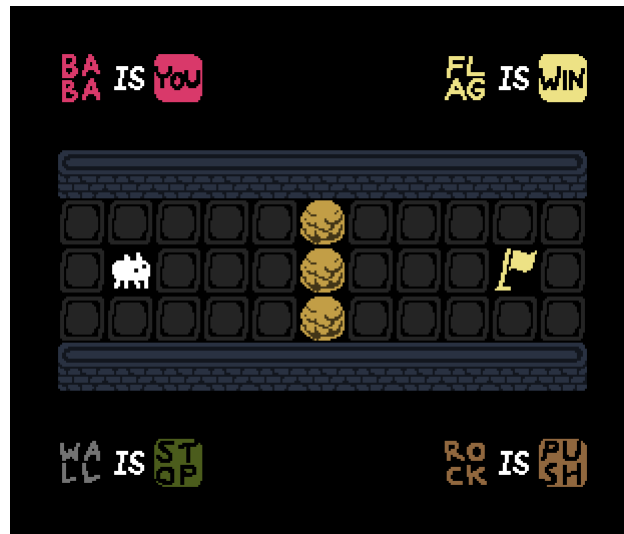
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 1; pp. 1:1–1:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** An introductory level from Baba is You. How many different solutions can you find?

Prior undecidability results use levels that scale with the size of the counter machine being constructed, and don't consider hardness on constant-area levels. However, the techniques we use can easily be applied to any similar proof (including Braid [13] and New Super Mario Bros. [2]), provided (for the first decision problem) that we allow levels to start with arbitrarily many objects in the same place – indeed, this has very recently been done for several Mario games [12].

The key idea to making counter-machine proofs work on constant-area levels is to reduce from a specific universal counter machine instead of a family of counter machines. The inputs are encoded in the initial state of the counter machine, which translate to starting the level with large piles of objects (which take constant space) or an input sequence that initializes the counters correctly.

1.1 Baba is You

Baba is You is a 2019 puzzle game by Hempuli in the genre of Sokoban. The player (typically) controls a white creature called BABA in a square grid, and can interact with the world by pushing objects. The main gimmick in Baba is You is that almost all of the rules are represented inside the game world, and can be pushed to change the game mechanics. A typical level, such as the one in Figure 1 will have rules like the following.

- BABA IS YOU: the player controls BABA with the arrow keys.
- FLAG IS WIN: the player wins if an object they control (e.g. BABA) touches a FLAG.
- WALL IS STOP: nothing can move into a WALL.
- ROCK IS PUSH: an object moving into a ROCK will push it.
- DOOR IS SHUT, KEY IS OPEN: if a KEY touches (or is pushed into) a DOOR, both are destroyed.

However, there is no fundamental difference between (non-text) objects; any “noun” can be paired with any “adjective”, and the challenge in a typical level is to construct useful rules by rearranging the available words.

Baba is You has a feature for sharing custom levels: each uploaded level is given a unique 8-character identifier, and can then be downloaded and played by anyone with the game. We have uploaded all levels in this paper; level codes are given with each level.



■ **Figure 2** The rules needed to reduce Pull!-*F [3] to Baba is You, and a level based on a gadget that proves PSPACE-hardness [4]. Level code: VMN9-1J93.

1.2 Prior work on Baba

In addition to being a video game, Baba is You is a programming language: with the right set of rules, you can embed all sorts computational system in it. Within weeks of the game’s release in March 2019, players had constructed Conway’s game of life,¹ the Rule 110 cellular automaton,² and a Turing machine.³ Often, these constructions are accompanied by a claim that Baba is You is “Turing-complete”, a term that doesn’t have a precise definition in this context.

From a complexity theory perspective, these constructions are reductions to (decision problems about) Baba is You. They show that Baba is You *with infinite space* is capable of arbitrary computation. In particular, provided they trigger victory under some appropriate condition, they show that the decision problem “can you win this level of infinite-space Baba is You” – or even “do you eventually win this fully deterministic level of infinite-space Baba is You” – is RE-hard.

However, levels in Baba is You are finite, and in complexity theory we typically generalize to large but still finite instances. This is especially problematic when extending the construction to an infinite level would require (without other changes) the level starting with infinitely many objects, as is the case for some of the above constructions.

For finite Baba is You, constructing one of these “Turing-complete” systems typically proves PSPACE-hardness – but Rule 110, a perennial favorite due to its simplicity, is not known to be PSPACE-hard, so constructing it doesn’t prove much in terms of complexity.

If the goal is PSPACE-hardness, it is simpler to reduce from a problem more similar to Baba is You: Figure 2 demonstrates an easy reduction to Baba is You from Pull!-*F, a PSPACE-complete block-pulling game [3]. But formally proving hardness generally isn’t the goal of constructions like those above – rather, they showcase the expressive power of Baba is You by pushing limits on what can be done from within the game.

Geller [11] explicitly considers Baba is You on an infinite board, and proves RE-hardness (though they don’t use that word) of determining whether a level is winnable using a reduction from the Post correspondence problem. The proof requires a finite region plus a two-cell-high empty infinite strip.

¹ <https://www.youtube.com/watch?v=YHONR1kmMUo>

² <https://twitter.com/mattar0d/status/1109987662608384000>

³ <https://www.youtube.com/watch?v=hsXpLx4soQY>



■ **Figure 3** A screenshot of Baba is You, after creating more than 2000 objects.

1.3 Our contributions

We consider Baba is You on bounded levels, and even on constant-area levels.

Unfortunately, the actual game has a practical limitation that puts Baba is You in PSPACE:⁴ there can only be six instances of an object in a single cell, and any additional copies will be destroyed. This means the total number of objects is bounded by a polynomial, so the game can be simulated by a polynomial-space nondeterministic machine. Thus Baba is You is in NPSPACE=PSPACE [16].

This constraint seems to be present for practical reasons, e.g. to prevent memory leaks or lag. But in an idealized theoretical version of Baba is You, there wouldn't be any limit on the number of copies of each object – this limit doesn't fit naturally with the other rules of the game, and it's never relevant to the levels in the game or taught to the player.⁵

We think that the more natural game to consider from a theoretical perspective is Baba is You without this limit. In particular, we believe this is a more natural generalization than infinite boards, so our undecidability result sticks to the spirit of the game better than the prior results which require unlimited space.

There is an additional limitation on constructing levels in practice: levels are described by three layers, and each layer can have only one object in each cell. So we are unable to make a level that starts with more than three objects in the same place. This is not a problem for constructing our counter machines, but it means a constant-area level can't encode the input to a counter machine as a large stack. This restriction seems to exist to make the interface for editing levels more convenient, but again we think the most natural generalization of Baba is You to study allows any pile of objects, even at the start of the level.

Finally, there is another rare practical failsafe in the game. If there are more than 2000 objects (or any of a few similar conditions are met), the entire level is replaced by the screen in Figure 3. While it demonstrates a surprising amount of self-awareness, this feature is inelegant for the same reasons as the limit to 6 copies.

For the remainder of this paper, we consider the version of Baba is You without any of these artificial limitations, which we call *Unlimited Baba is You* – however, with the exception of initialized registers, our levels use at most three objects per cell and thus can be built in the actual game. Our main result is Theorem 1: that Unlimited Baba is You is capable of universal computation, even on 8×17 levels. We prove this by constructing a specific universal counter machine called U_{22} [15] in Baba is You.

⁴ Baba is You has randomness, which complicates this – it's not even obvious what the decision problem should be in the presence of randomness. Here we consider the deterministic fragment of Baba is You.

⁵ We only learned of its existence when we started building counter machines, and they failed in mysterious ways.

As a consequence of building U_{22} , we obtain Corollaries 2 and 4, that the two decision problems mentioned above are RE-complete for Baba is You.

Beyond Baba, the ideas we use can likely be applied to many games that allow arbitrarily many objects in a bounded region, and we hope to see many similar results for other games in the future.

1.4 Baba is You mechanics

We now describe the specific mechanics of Baba is You that our reductions use. We do not attempt to cover all of the mechanics in the game, or even to define the ones we need in full detail. The game contains *text* objects, which have a single word. These words can be read left-to-right or top-to-bottom to form *rules*, which affect the game. The words in Baba is You can (roughly) be partitioned into *nouns*, *verbs*, *adjectives*, and *prepositions*, which approximately correspond to the usual meanings of those terms.

Nouns refer to kinds of objects, most of which are interchangeable. Verbs are the cores of rules; by far the most common is **IS**, which usually assigns an adjective to a noun. Adjectives represent mechanics, which affect the behavior of nouns they apply to. Prepositions are used to make *noun phrases* like “BABA ON FRUIT”, which define a new set of objects and are used in mostly the same way as nouns.

We use n , a , and N as variables for nouns, adjectives, and noun phrases (including nouns), respectively. In addition to nouns (which are listed in Figure 6), our constructions use the words in Table 1.

In Section 2, we discuss counter machines and how they can be used to prove RE-hardness, even for constant-area instances of a problem like Unlimited Baba is You. In Section 3, we construct counter machines in Unlimited Baba is You, and obtain RE-hardness for 8×17 levels.

2 Count is Universal

A counter machine is a finite state machine with access to a constant number of *registers* which each store an arbitrary nonnegative integer. A counter machine interacts with its registers via *instructions* such as “add 1” or “transition to a different state depending on whether the register is 0”.

For instance, Figure 4 shows a simple counter machine with four registers which multiplies two numbers. The four registers are called 0 through 3. It uses two kinds of instructions which act on a particular register:

- Increment (inc): increase the register by 1.
- Jump if zero and decrement (dec): go to one state if the register is positive and a different state if it’s 0, and decrease it by 1 in the former case.

Korec [15] calls these RiP and RiZM, respectively.

We draw increment instructions as rectangles and jump-and-decrement instructions as diamonds. The outgoing path from a diamond marked with a 0 is the one taken when the register is 0. To make them convenient to build in Baba is You, we draw our counter machines so that paths go straight through increment instructions, go straight through jump-and-decrement instructions to take the zero branch, and turn to take the nonzero branch. We write R_i for the value of register i .

To understand the counter machine in Figure 4, start the registers with $R_1 = m$, $R_2 = n$, and $C = D = 0$. The machine runs the main loop n times, decrementing R_2 on each loop and stopping when $R_2 = 0$. Each time, it moves R_1 to both R_0 and R_3 , then moves R_3 back to R_1 , for a net effect of increasing R_0 by m . When it stops, $R_0 = mn$.

■ **Table 1** The words in Baba is You we use other than specific nouns.

Word	Part of speech	Behavior
IS	verb	The “rule N IS a ” applies the mechanic corresponding to a to all instances of N . The rule “ N IS n ” replaces each instance of N with an n .
MAKE	verb	Each time step, the rule “ N MAKE n ” creates an n in each cell with an N that doesn’t already have an n .
YOU	adjective	This object is controlled by the player’s inputs. There can be zero or multiple such objects. The player can also choose to <i>pass</i> , which lets other rules process one time step.
WIN	adjective	If this object is in the same cell as an object that’s YOU, the player wins.
MOVE	adjective	Each time step, this object moves one unit in the direction it’s facing. If it can’t move forward (e.g. because it’s the edge of the level), it bounces off (the details won’t matter for our reduction).
SHIFT	adjective	Each time step, move all other objects in the same cell one unit in the direction this object is facing, and set their facing directions to match.
FLOAT	adjective	This object is on a second “layer”, which is independent of the default layer for some adjectives including WIN and SHIFT.
STOP	adjective	Other objects can’t move into the cell containing this object.
PUSH	adjective	If another object attempts to move into the cell containing this object, this object is pushed in the same direction.
PULL	adjective	This object is STOP, and if an object adjacent to it moves away, this object follows.
OPEN SHUT	adjectives	If an OPEN object is in the same cell as a SHUT object, destroy both. This always destroys objects in pairs – it will destroy the same number of OPEN and SHUT objects, up to whichever there are fewer of.
ON	preposition	The noun phrase “ n ON n' ” refers to each n which is in the same cell as an n' .
AND	preposition	The noun phrase “ n AND n' ” refers to both each n and each n' , and similarly for conjunctions of more than two nouns. AND can also be used to combine adjectives, e.g. “BABA IS YOU AND FLOAT”.
NOT	preposition	The noun phrase “NOT n ” refers to object that aren’t n . A rule of the form “ n IS NOT a ” overrules the rule “ n IS a ”.
TEXT	noun	Refers to all text objects.

There are many possible instructions we can allow counter machines to use. From the perspective of using counter machines to prove undecidability, there is a tradeoff between simplicity and efficiency: more powerful instruction sets require more work to implement, but may allow us to use a counter machine that has fewer registers or states, or that simulates a Turing machine more efficiently.

Korec [15] constructs explicit universal counter machines for a variety of instruction sets at different points on this tradeoff. For Baba is You, we will use counter machines with the two instructions described above.

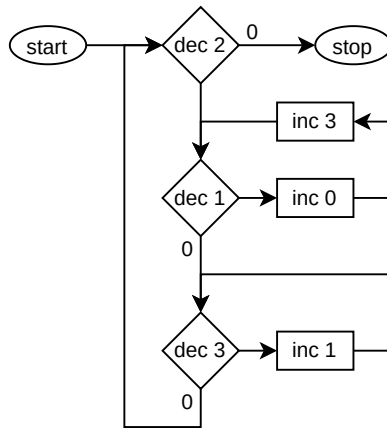


Figure 4 A counter machine which inputs two numbers A and B and outputs their product on register D . Rectangles are increment instructions; diamonds are jump if zero and decrement instructions, with 0 indicating the branch to take when the register is 0.

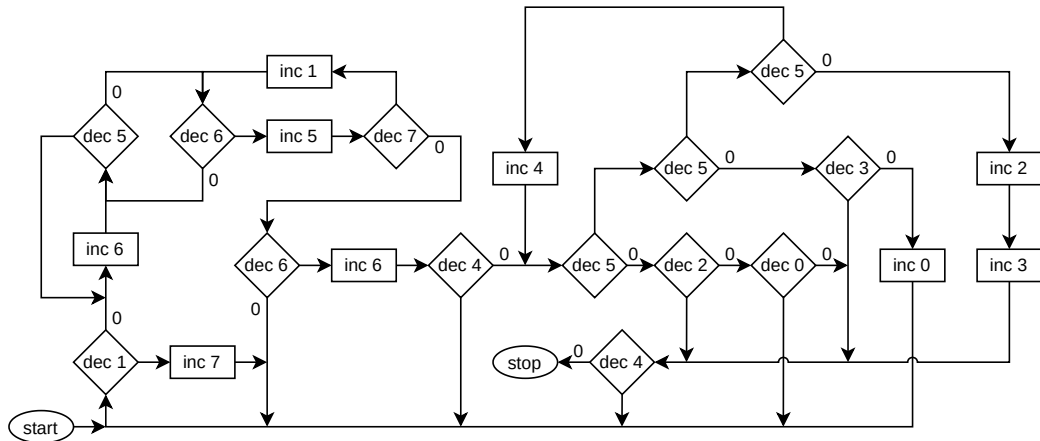


Figure 5 Korec’s strongly universal counter machine U_{22} [15].

The counter machine we will use is the 22-state machine U_{22} [15], which is shown in Figure 5 laid out in a way that suits our purposes. This machine is *strongly universal*, meaning the following. Let Φ_i be an enumeration of partial computable functions $\mathbb{N} \rightarrow \mathbb{N}$, e.g. as Turing machines. There is a computable function g such that, if we run U_{22} starting from $R_1 = g(i)$, $R_2 = n$, and all other registers 0, it halts if and only if $\Phi_i(n)$ is defined, and if so when it halts $R_0 = \Phi_i(n)$. That is, if we input $g(i)$ and n , this machine computes $\Phi_i(n)$.

We now attempt to summarize how U_{22} works. The input on register 1 is actually a description of a particular kind of counter machine with three very specific instructions, which U_{22} simulates – the first step of computing g is to convert your Turing machine into such a counter machine. Registers 0, 2, and 3 are for the simulated machine. The description and current state of the simulated machine are kept in registers 1 and 4, respectively. The left half of U_{22} parses these two registers, unpacking the encoding of the counter machine at the current state. The result is that R_5 encodes both the next instruction to execute (which determines where the machine exits the loop with three dec 5 instructions) and the next state to switch to (which becomes R_4). The bottom right section carries out the instruction, possibly decrementing R_4 to allow the simulated machine to branch, before moving to the next step or halting.

2.1 Game Make Count

Consider a game capable of building counters in bounded space, typically by stacking arbitrarily many objects in the same place. Suppose that we can also implement the instructions used by U_{22} , or some other sufficiently powerful instruction set. Games that fit these criteria include Braid [13], several Mario games [10, 12], and Baba is You (as we will show).

After designing these components, it is straightforward to prove RE-hardness for such a game: convert any Turing machine to an appropriate counter machine, and then construct the counter machine in the game. But we can do better, by using a universal counter machine to obtain results for constant-area levels.

Specifically, construct a level which simulates U_{22} (or another universal counter machine). Then that level is capable of universal computation: if it is initialized with the appropriate set of objects to represent $R_1 = g(i)$ and $R_2 = n$, when it halts it will encode $R_0 = \Phi_i(n)$.

To prove RE-hardness for constant-area levels, we need two additional conditions: the counter machine terminating should trigger victory, and the initial state of a level must be allowed to have arbitrarily many objects to encode an arbitrary initial register value. Then we can reduce from the halting problem. Given a number encoding a Turing machine i and an input n , build the level that simulates U_{22} initialized as above, so the player wins if and only if $\Phi_i(n)$ is defined.

For RE-hardness of a constant level after a given sequence of inputs, we must build something that allows the player to choose the initial values of the input registers. Then we can again reduce from the halting problem: start by making a sequence of moves that initializes the counter machine as described above and then transitions to deterministically simulating the machine. After this sequence, the player will win if and only if $\Phi_i(n)$ is defined.

In the next section, we will do all of this for Unlimited Baba is You. We will build counter machines by representing each register as the number of copies of an object in a single cell. Then we will describe a level that runs U_{22} with no player choices, and a version of it that allows the player to set input values during the initialization phase.

3 Baba is Count

We now start building counter machines in Unlimited Baba is You. There are many possible approaches to doing this; we choose one which is space-efficient and will result in levels visually matching diagrams like Figure 4.

In our construction, the player will never be able to make choices – nothing will be YOU until the frame the player wins, so all the player can do is wait for the counter machine. We will never move text or have rules change. Each register will be represented by a particular objects, with the number of copies matching the current value. All of these registers will be in the same cell, and will move around the counter machine together. When the pile encounters certain other objects, it will gain or lose register objects and be redirected to simulate the counter machine.

The moving pile will be “carried” by a **CART**, with the rule “**CART IS MOVE**”. It will be routed using **BELTS**, with “**BELT IS SHIFT**”. We will decrement registers by using a **GATE** to consume an item, with ‘**GATE IS SHUT**’.

Each register will use three types of object, which we call **REG**, **INC**, and **DEC** to explain a generic register. **REG** is the object that forms a pile to keep track of the value of the register. We have the rule “**REG ON CART IS MOVE**” so that it follows the cart, but other instances of **REG** waiting to be picked up stay where they are.

To increment, we have “INC MAKE REG”. Each copy of INC will have one REG sitting on it, and if the CART drives over that cell the REG will be added to the pile. The facing direction of the waiting REG matches that of the INC, so we need place the INC facing the direction the CART will cross it.

To decrement, we have “DEC MAKE GATE” and “REG ON DEC IS OPEN”. When the pile of registers crosses DEC, the GATE on it will consume exactly one REG. Having REG only be OPEN when on DEC ensures that we won’t consume an item from the wrong register.

To branch, we again use DEC. We have the rule “DEC ON REG IS SHIFT”, with the DEC facing a direction orthogonal to the direction the CART is moving. If the register is zero, the DEC is not on a REG, so the CART continues straight. Otherwise, the DEC deflects it.

There is a potential conflict between decrementing and branching: if the pile MOVES onto the DEC and is immediately SHIFTEd, it fails to consume one of the REGS—SHIFT is applied before OPEN and SHUT. We can avoid this by having a BELT directly into every DEC, so that the pile MOVES onto the BELT, is SHIFTEd onto the DEC, and then the GATE can consume a REG (but the pile’s direction will change to match the DEC if the register is nonzero).

When the counter machine halts, the CART arrives at a FLAG. So that this results in victory, we have the rules “CART ON FLAG IS YOU” and “FLAG IS WIN”.

To summarize: the objects we use are CART, BELT, GATE, FLAG, and three objects (REG, INC, DEC) for each register. The rules are

- CART IS MOVE
- BELT IS SHIFT
- GATE IS SHUT
- CART ON FLAG IS YOU
- FLAG IS WIN
- For each register:
 - REG ON CART IS MOVE
 - INC MAKE REG
 - DEC MAKE GATE
 - REG ON DEC IS OPEN
 - DEC ON REG IS SHIFT

Figure 6 shows these objects, including the objects we’ll use for the eight registers of U_{22} .

Figure 7 shows an Unlimited Baba is You level implementing the simple counter machine from Figure 4. The actual counter machine is in the top left, below it are the “generic” rules, and to the right is a square of rules for each of the four registers. The CART is facing up, and start by driving into the top belt and then the SWORD (it needs a bit of space at the start so the SWORD has time to MAKE a GATE).

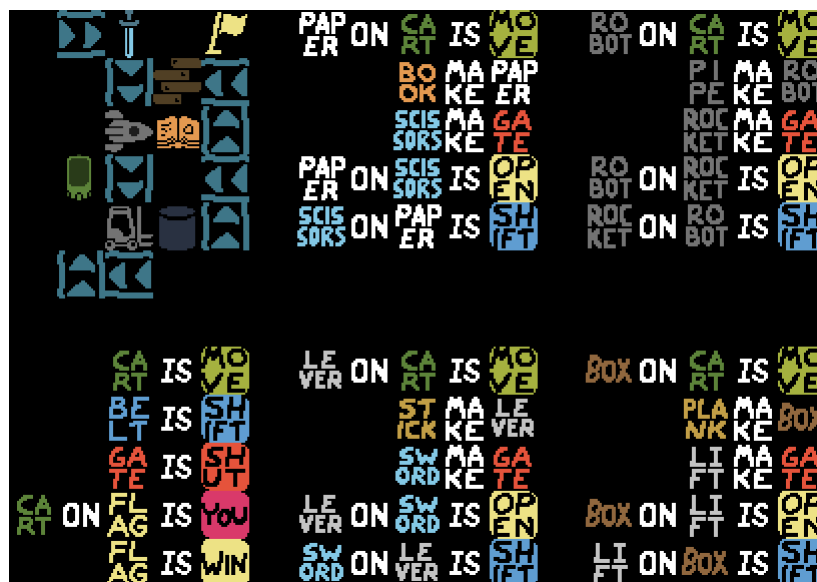
To supply inputs, we start the level with a pile of m ROBOTS and n LEVERs in the same cell as the CART – for readability, the figure shows $m = n = 0$. The CART will eventually reach the FLAG carrying mn PAPERs (and still m ROBOTS), at which point the player wins.

It’s somewhat inelegant that we have to encode the input in the level: if a player simply wants to know what the product of two numbers is, they have to load up the level editor and stick in some ROBOTS and LEVERs (which in practice requires adding more space). We modify the level as shown in Figure 8 to allow the player to choose the inputs to the counter machine.

We have changed the rules a bit, added a TILE above the CART and created the cage on the left. Now the counter machine doesn’t start running immediately, and the player doesn’t win when it halts. Instead, the player controls BABA, who is stuck in the cage.



■ **Figure 6** The Baba is You objects used in our counter machines. The objects at the top are not connected to a specific register, and each register 0 through 7 uses the items in one of the rows, listed in the order REG, INC, and DEC. The DECs are chosen so that the direction they’re facing is visible – all objects have a facing direction, but some don’t have directional sprites. We have attempted to use semantically-related items to make the registers somewhat intuitive, but the limited inventory of objects makes this challenging.



■ **Figure 7** A counter machine that multiplies two numbers (Figure 4) built in Unlimited Baba is You. Level code: EUQT-TCZJ.



■ **Figure 8** The same Unlimited Baba is You counter machine as in Figure 7, but with infrastructure that lets the player choose inputs. Level code: Q1V8-PRIE.

The player will eventually create `CART IS MOVE`, which starts the counter machine and requires putting `MAKE` somewhere it can't be used. Before this, the player chooses the inputs to R_1 and R_2 , which are the initial numbers of `ROBOTS` and `LEVERS`, respectively. They do so by creating `TILE MAKE ROBOT` and `TILE MAKE LEVER` for any desired number of time steps: push `TILE` down, and then push the nouns on the right down three times, waiting in between for the appropriate number of steps while the created objects `SHIFT` off the `TILE`. The player can use a slightly different sequence to make either or both registers start at 0 – the cage gives just enough freedom to allow this without letting the player create any unintended rules or `MAKE` any more objects after starting the counter machine.

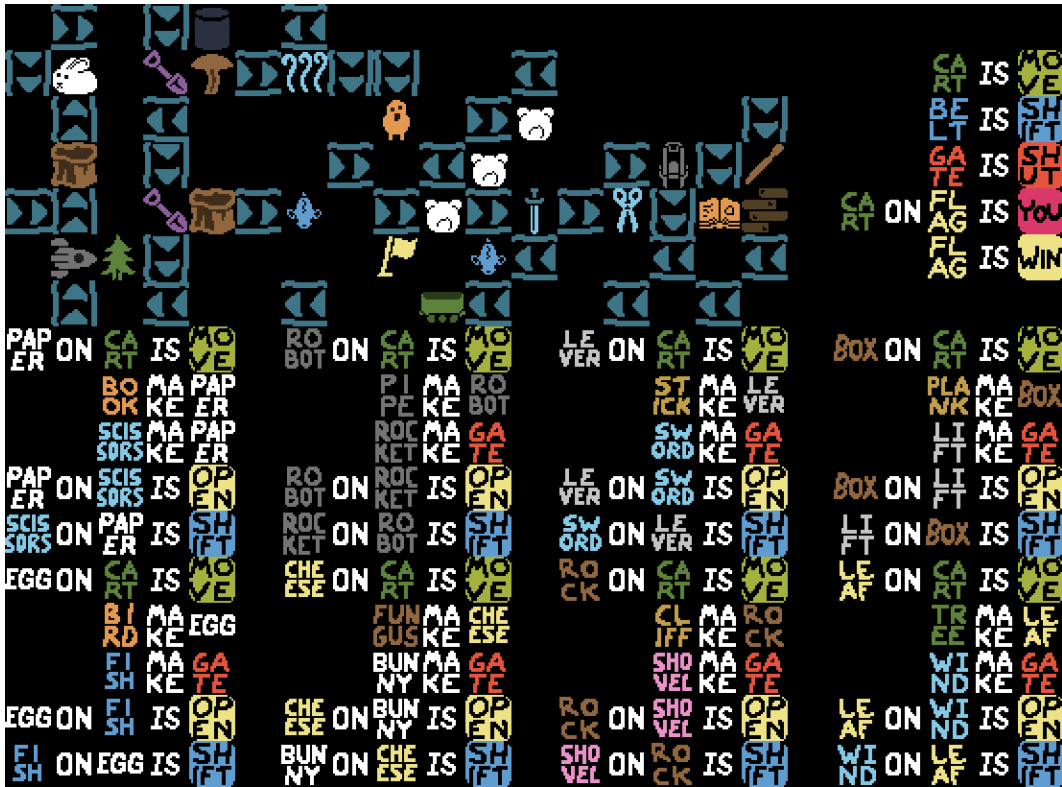
To read out the product the machine computes, the player must be attentive when the `CART` reaches the `FLAG`: it turns into `SCISSORS`, which create `GATES` to consume one `PAPER` each time step – they decrement register 0. The `WALL` prevents the `SCISSORS` from `SHIFTING` the pile of counters away. The player receives the desired product by counting how many times they hear the distinctive noise of mutual annihilation of `OPEN` and `SHUT`.

Finally, the player moves to the `FLAG` at the top of their cage. This allows them to complete the level after it multiplies the numbers of their choosing, or to abandon it early should they decide to use a more efficient calculator.

3.1 U_{22} in Baba

Since we've described how to construct general counter machines in Unlimited Baba is You, we can build Korec's U_{22} [15]. This is shown in Figure 9. The actual counter machine is about a third of the space of the level, with the rest being taken up by rules.

As before, the input is given as a stack of `ROBOTS` and `LEVERS` on the cart, representing the initial values of R_1 and R_2 , respectively. If we wish to compute $\Phi_i(n)$, we should set up the level with $g(i)$ `ROBOTS` and n `LEVERS`. Then the player wins exactly when $\Phi_i(n)$ exists, and if so the number of `PAPERS` on the `CART` just before victory is $\Phi_i(n)$. We now apply the construction we used to create Figure 8, to allow us to perform computation in a level that is not only constant-area, but truly constant. This level is shown in Figure 10.



■ **Figure 9** U_{22} (Figure 5) in Unlimited Baba is You. Level code: 7LKG-WKBJ (initialized with one ROBOT so that it terminates).

The cage is now in the top right, and we have moved some rules to make space. There are two differences from the cage in Figure 8: there isn't an accessible FLAG, and the CART becomes BABA (instead of SCISSORS) when it reaches the FLAG, triggering victory.

As before, the player can choose how many ROBOTs and LEVERs to start the counter machine with, setting the initial values of R_1 and R_2 . Once the player makes CART IS WIN, there is nothing of use they can do but wait, and will eventually win if the counter machine halts on the provided input.

3.2 Smaller levels

We now compact our counter machine construction to show RE-hardness for smaller constant-area levels. The actual counter machine in Figure 9 is already pretty compact; our goal here is to make the rules take less space. In addition to leaving less empty space, there are a few tricks we employ:

- We place rules in the same region as the counter machine. We need TEXT IS FLOAT so that text isn't affected by BELT's, and TEXT IS NOT PUSH so that it isn't affected by the moving CART.
- Multiple words can go in the same cell: to write ROCKET MAKE GATE and SWORD MAKE GATE, we have cell with both words ROCKET and SWORD in front of MAKE GATE.
- Another way to make that pair of rules is with AND: the rule ROCKET AND SWORD MAKE GATE is more efficient than having both rules separately.



■ **Figure 10** The same construction of U_{22} as in Figure 9, but now the player can control BABA to specify inputs to the counter machine. Level code: ZFGG-WH6Z.

- Combining the two previous points, we make rules like SWORD/ROCKET AND FISH/BUNNY MAKE GATE, which is parsed as four separate rules of the form X AND Y MAKE GATE. We end up with multiple copies of SWORD MAKE GATE – this is fine for MAKE, but is an issue for other verbs like MOVE, where two copies of CART IS MOVE means the CART moves two cells per time step.

The level editor allows us to place multiple objects in the same cell using layers, but there is a limit of 3 objects per cell.

The result is the unreadable mess in Figure 11, which is an 8×17 level. It behaves the same as in Figure 9. In particular, inputs are given as piles of left-facing ROBOTS and LEVERS in the seventh row, and the output is the number of PAPERS when the player wins.

► **Theorem 1.** *An 8×17 level of Unlimited Baba is You is capable of universal computation, where inputs and outputs are encoding as the number of identical objects in a single cell.*

By *universal computation*, we mean that with the inputs $g(i)$ and n in the format specified for some computable function g , the output is $\Phi_i(n)$, again in the format specified.

► **Corollary 2.** *Deciding whether an 8×17 level of Unlimited Baba is You is solvable is RE-complete (and in particular undecidable).*

We can also compactify the level in Figure 10, which lets the player specify inputs to the counter machine. The only additional complication is that we need TEXT IS NOT PUSH to not exist until after the player is done initializing. We also place the TILE under the CART to avoid needing TILE IS SHIFT.



■ **Figure 11** U_{22} in Baba is You (Figure 9), after compactifying at the cost of clarity. Level code: 4ZGV-BPQ3 (initialized with one ROBOT so that it terminates).



■ **Figure 12** U_{22} in Baba is You with inputs given by the player (Figure 10), after compactifying in the same way as Figure 11. Level code: 7GG1-1PTH.

► **Corollary 3.** *There is a specific 8×21 level of Unlimited Baba is You which is capable of universal computation, where inputs are given by a sequence of moves the player makes and the output is the number of identical objects in a single cell when the player wins.*

To compute $\Phi_i(n)$, the player performs a fixed sequence of moves to make TILE MAKE ROBOT, waits $g(i) - 1$ times, performs another fixed sequence of moves to destroy that rule and create TILE MAKE LEVEL, waits $n - 1$ times, and finally destroys that rule as well while creating CART IS MOVE. If $g(i)$ or n is zero, the player must use a slightly different initialization sequence.

► **Corollary 4.** *There is a specific 8×21 level of Unlimited Baba is You for which the decision problem “if the player starts by performing a given sequence of inputs, can they go on to win (without undoing)?” is RE-complete.*

Results about constant-area levels have another small advantage: our levels are buildable in the Baba is You level editor, which has a maximum size of 33×18 (and three objects per cell) – and you can play them at the level codes given. If you enter initial values for the levels where that’s possible (Figures 8, 10 and 12), you can then run the counter machine the level simulates, with only the caveat that the values of registers are capped at 6.

References

- 1 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.
- 2 Hayashi Ani. *Unsimulability, Universality, and Undecidability in the Gizmo Framework*. PhD thesis, Massachusetts Institute of Technology, 2023.
- 3 Hayashi Ani, Sualeh Asif, Erik D. Demaine, Jenny Diomidov, Della Hendrickson, Jayson Lynch, Sarah Scheffler, and Adam Suhl. PSPACE-completeness of pulling blocks to reach a goal. *Journal of Information Processing*, 28:929–941, 2020.
- 4 Hayashi Ani, Jeffrey Bosboom, Erik D. Demaine, Jenny Diomidov, Della Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In *10th International Conference on Fun with Algorithms (FUN 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 5 Hayashi Ani, Lily Chung, Erik D. Demaine, Jenny Diomidov, Della Hendrickson, and Jayson Lynch. Pushing blocks via checkable gadgets: PSPACE-completeness of Push-1F and Block-/Box Dude. In *11th International Conference on Fun with Algorithms (FUN 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.
- 6 Jeffrey Bosboom, Josh Brunner, Michael Coulombe, Erik D. Demaine, Della Hendrickson, Jayson Lynch, and Elle Najt. The Legend of Zelda: The complexity of mechanics: Discrete and computational geometry, graphs, and games. *Thai Journal of Mathematics*, 21(4):687–716, 2023.
- 7 Erik D. Demaine, Della Hendrickson, and Jayson Lynch. Toward a general complexity theory of motion planning: Characterizing which gadgets make games hard. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 8 Erik D. Demaine, Justin Kopinsky, and Jayson Lynch. Recursed is not recursive: A jarring result. In *31st International Symposium on Algorithms and Computation (ISAAC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 9 Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The computational complexity of Portal and other 3D video games. In *9th International Conference on Fun with Algorithms (FUN 2018)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2018.
- 10 Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder/easier than we thought. In *8th International Conference on Fun with Algorithms (FUN 2016)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2016.
- 11 Jonathan Geller. Baba is You is undecidable. *arXiv preprint arXiv:2205.00127*, 2022.
- 12 MIT Hardness Group, Hayashi Ani, Erik D. Demaine, Holden Hall, Ricardo Ruiz, and Naveen Venkat. You can't solve these Super Mario Bros. levels: undecidable Mario games. In *12th International Conference on Fun with Algorithms (FUN 2024)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
- 13 Linus Hamilton. Braid is undecidable. *arXiv preprint arXiv:1412.0784*, 2014.
- 14 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. CRC Press, 2009.
- 15 Ivan Korec. Small universal register machines. *Theoretical Computer Science*, 168(2):267–301, November 1996. doi:10.1016/S0304-3975(96)00080-1.
- 16 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of computer and system sciences*, 4(2):177–192, 1970.
- 17 Matthew Stephenson, Jochen Renz, and Xiaoyu Ge. The computational complexity of Angry Birds. *Artificial Intelligence*, 280:103232, 2020.
- 18 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54:595–621, 2014.

Poset Positional Games

Guillaume Bagan ✉

Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France

Eric Duchêne ✉ 

Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France

Florian Galliot ✉ 

Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France

Valentin Gledel ✉

Université Savoie Mont Blanc, CNRS UMR5127, LAMA, Chambéry, F-73000, France

Mirjana Mikalački ✉ 

Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia

Nacim Oijid ✉ 

Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France

Aline Parreau ✉ 

Univ Lyon, CNRS, UCBL, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France

Miloš Stojaković ✉ 

Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, Serbia

Abstract

We propose a generalization of positional games, supplementing them with a restriction on the order in which the elements of the board are allowed to be claimed. We introduce poset positional games, which are positional games with an additional structure – a poset on the elements of the board. Throughout the game play, based on this poset and the set of the board elements that are claimed up to that point, we reduce the set of available moves for the player whose turn it is – an element of the board can only be claimed if all the smaller elements in the poset are already claimed.

We proceed to analyze these games in more detail, with a prime focus on the most studied convention, the Maker-Breaker games. First we build a general framework around poset positional games. Then, we perform a comprehensive study of the complexity of determining the game outcome, conditioned on the structure of the family of winning sets on the one side and the structure of the poset on the other.

2012 ACM Subject Classification Theory of computation → Algorithmic game theory

Keywords and phrases Positional games, Maker-Breaker games, Game complexity, Poset, Connect 4

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.2

Related Version *Full Version:* <https://arxiv.org/abs/2404.07700>

Funding This research was partly supported by the french ANR project P-GASE (ANR-21-CE48-0001-01), by Provincial Secretariat for Higher Education and Scientific Research, Province of Vojvodina (Grant No. 142-451-2686/2021) and by Ministry of Science, Technological Development and Innovation of Republic of Serbia (Grants 451-03-66/2024-03/200125 & 451-03-65/2024-03/200125).

1 Introduction

1.1 General motivation

Positional games. Positional games are a class of combinatorial games that have been extensively studied in recent literature – see books [6] and [12] for an overview of the field. They include popular recreational games like Tic-Tac-Toe, Hex and Sim. Structurally, a



© Guillaume Bagan, Eric Duchêne, Florian Galliot, Valentin Gledel, Mirjana Mikalački, Nacim Oijid, Aline Parreau, and Miloš Stojaković;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 2; pp. 2:1–2:12

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

positional game is a pair (X, \mathcal{F}) , where X is a (finite) set that we call *the board*, and $\mathcal{F} \subseteq 2^X$ is a collection of sets that we call *the winning sets*. The pair (X, \mathcal{F}) is referred to as the *hypergraph of the game*. The game is played in the following way: two players alternately claim unclaimed elements of the board, until all the elements are claimed. There are several standard conventions, defining the way the winner is determined: Maker-Maker games, Maker-Breaker games, Avoider-Enforcer games, etc.

In *Maker-Maker games*, also known as strong making games, players compete to fill up one of the winning sets i.e. claim all its elements, and whoever does it first wins. If there is no winner by the time all the board elements are claimed, the game is declared a draw. Tic-Tac-Toe (“3-in-a-line”) is a notable representative of this type of games and every child knows the game ends in a draw, provided that both players play optimally. It can be generalized to the $n \times n$ board, going under the name “ n -in-a-line”, and it is also known to be a drawn game [6] for all $n \geq 3$. More generally, Tic-Tac-Toe can be played on the hypercube $[n]^d$, where the winning sets are all the geometric lines of cardinality n . The Hales-Jewett Theorem [11] states that, for every n , there exists a positive number d , also called the Hales-Jewett number $HJ(n)$, such that for all $k \geq d$, the game $[n]^k$ is a first player win.

Generally speaking, the strong making games are natural to introduce and study, and many recreational games are of this type. Hence, it is no wonder that numerous questions about them have been asked in the literature. Yet, it may come as a surprise that the majority of those questions remain unanswered. The thing is, in a typical strong making game, each player’s goal can be seen as two-fold: they are simultaneously trying to claim a winning set and to block their opponent’s attempts at claiming a winning set. This makes most such games notoriously hard to analyze, and hardly any general tools are known for this purpose. As a consequence, there are very few published results about this convention, when compared to Maker-Breaker games.

In *Maker-Breaker games*, we call the players Maker and Breaker, and they have different goals – Maker wins if she fills up a winning set, while Breaker wins otherwise, i.e. if he claims at least one element in each winning set. Note that no draw is possible in this convention. Maker-Breaker games are the most researched convention of positional games, ever since Erdős and Selfridge [9] first introduced them in 1973. Looking at Tic-Tac-Toe on the standard 3×3 board in the Maker-Breaker convention, it is straightforward to convince oneself that Makers wins when playing first.

The main problem on positional games consists in determining the *outcome* i.e. the identity of the player who has a winning strategy (possibly depending on who starts), if there is one, assuming that both players are playing optimally. The study of the complexity of computing the outcome of a given positional game can be traced back to Schaefer [14], who was first to prove that Maker-Breaker games are PSPACE-complete, even when the winning sets are of size at most 11. This was later improved by Rahman and Watson [13], requiring only winning sets of size 6. On the other hand, Galliot et al. [10] proved that the outcome of any Maker-Breaker game with winning sets of size at most 3 can be determined in polynomial time. Maker-Maker games are also known to be PSPACE-complete, as shown by Byskov in [7].

Poset positional games. Let us now take a closer look at a popular recreational game, Connect-4, which has a lot in common with the Tic-Tac-Toe family of positional games. In Connect-4, two players play on a board that is 7-wide and 6-high. They move alternately by placing a token of theirs in a column of their choice. Each placed token drops down with

“gravity”, landing on top of the last previously placed token in that column, or heading all the way to the bottom of the column if it is empty. The first player with tokens on four consecutive positions in a line (vertical, horizontal or diagonal) wins, and, if neither of the players manages to do that by the time all the columns are full, then a draw is declared. Although the game is similar to Tic-Tac-Toe at first glance, there is one crucial difference – the players cannot choose freely from all the empty positions of the board, as at any point a column offers at most one available position (the lowest token-free position).

The outcome of the game is known as a first player win, as shown by Allis in [3], and independently by Allen [1] who later wrote a whole book [2] about this game. Connect-4 was solved in [17] for some nonstandard board sizes, and for some others in [16].

Recently, Avadhanam and Jena [4] studied *Connect-Tac-Toe*, where they introduce a restriction on which unclaimed elements can be claimed, combining the nature of Connect-4 with the positional game of Tic-Tac-Toe. They also look at the generalized $[n]^d$ Connect-Tac-Toe, played on a hypercube $[n]^d$, relating it to the restricted version of the $[n]^d$ Tic-Tac-Toe. Additionally, they give a lower bound on an analogue of the Hales-Jewett number, $HJ(n)$, in this setting.

In the present paper, we propose a new framework which, in full generality, enables us to combine the move restrictions with positional games (like it is done in Connect-4). Namely, we introduce *poset positional games*, which are positional games with an additional structure – a poset on the elements of the board. Throughout the game play, based on this poset and the set of elements that are claimed up to that point, we reduce the set of available moves for the player whose turn it is – an element of the board can only be claimed if all the smaller elements (in the poset) are already claimed.

We proceed to analyze these games in more detail, with a prime focus on the most studied convention, the Maker-Breaker games. After setting out a formal introduction of poset positional games, we go on to build a general framework around them. Then, we perform a comprehensive study of the complexity of determining the game outcome, conditioned on the structure of the family of winning sets on the one side and the structure of the poset on the other.

1.2 Framework of poset positional games

A *poset* P on a set X is defined by a partial order relation \leq . On top of that, we use $<$ to denote the same relation acting on distinct elements. In this paper, posets will be depicted by directed graphs in the usual way, where the elements are vertices, and two elements x and y satisfy $x \leq y$ if and only if there exists a directed path from x to y . Two elements are deemed *incomparable* if there exists no directed path between them.

In addition, we will use the following standard definitions about posets. A *chain* is a set of elements that are pairwise comparable. An *antichain* is a set of elements that are pairwise incomparable. The *height* of a poset is the cardinality of its longest chain. The *width* of a poset is the cardinality of its largest antichain. Given an element x , we say that y is a *predecessor* (resp. a *successor*) of x if $y < x$ (resp. $x < y$) and there is no other element between x and y . An element x is said to be *maximal* (resp. *minimal*) in the poset if it has no successor (resp. no predecessor).

We are now able to formally define a poset positional game as follows. A *poset positional game* is a triple (X, \mathcal{F}, P) where X is a finite set of elements (also called *vertices*), \mathcal{F} is a collection of subsets of X corresponding to the *winning sets*, and P is a poset on X .

The game is played by two players that alternately claim an unclaimed vertex x of X such that all vertices smaller than x have already been claimed. The rest stays the same as before. In the Maker-Maker convention, the first player that fills up a winning set $S \in \mathcal{F}$ wins. If

no player manages to fill up a winning set, the game ends in a draw. In the Maker-Breaker convention, Maker wins if she fills up a winning set at some point during the game, otherwise Breaker wins. During a game, when a player claims a single vertex, we call that a *move*, whereas a *round* corresponds to a pair of moves made successively by both players.

Note that a standard positional game is a poset positional game where all the vertices are pairwise incomparable. Furthermore, in poset positional games, from any given position, the set of moves that are available to the next player forms an antichain of the poset. In particular, there are at most w available moves at any point of a game, where w is the width of the poset.

It is a well-known result that, for any standard positional game played in the Maker-Maker convention, the second player cannot have a winning strategy, so that the only possible outcomes are a first player win \mathcal{FP} or a draw \mathcal{D} . Similarly, for any standard positional game played in the Maker-Breaker convention, there are only three possible outcomes (if we do not specify who starts):

- \mathcal{M} if Maker has a winning strategy no matter who starts the game,
- \mathcal{B} if Breaker has a winning strategy no matter who starts the game,
- \mathcal{N} if the next player (i.e. the player whose turn it is) has a winning strategy.

When switching to the framework of poset positional games, the property asserting that the second player never wins is not true any more. As a consequence, there are game positions for which the next player may have interest in skipping their turn. The corresponding outcome will be denoted by \mathcal{P} (standing for “Previous player wins”, or equivalently meaning that the second player has a winning strategy). Such phenomena are generally called *zugzwangs* in the literature of combinatorial games [8].

From now on, unless explicitly stated, we will only consider the Maker-Breaker convention of poset positional games. Though the computation of the outcome is generally the main issue when investigating such games, this study can, under certain circumstances, be reduced to the case of a particular player starting the game. Indeed, starting with a game $\mathcal{G} = (X, \mathcal{F}, P)$ where Maker (resp. Breaker) is the first player and claims a vertex x , we get a resulting game $\mathcal{G}' = (X', \mathcal{F}', P')$ where Breaker (resp. Maker) is the first player, defined by $X' = X - x$, $P' = P - x$ and $\mathcal{F}' = \{W \setminus \{x\} \mid W \in \mathcal{F}\}$ (resp. $\mathcal{F}' = \{W \mid W \in \mathcal{F}, x \notin W\}$). Therefore, when studying a class of games which is stable under Maker’s (resp. Breaker’s) moves in terms of that update, we may freely assume that Breaker (resp. Maker) is the first player, up to considering all possibilities for their opponent’s first move otherwise. In this paper, all studied classes will be stable under Breaker’s moves, so we will always assume that Maker is the first player. Therefore, the decision problem that will be mainly investigated is the following:

MB POSET POSITIONAL GAME

Input: A poset positional game $\mathcal{G} = (X, \mathcal{F}, P)$.

Output: The player having a winning strategy (i.e. Maker or Breaker) when Maker starts.

Since standard positional games are included in poset positional games, the problem MB POSET POSITIONAL GAME is PSPACE-complete from a result of Schaefer [14].

1.3 Exposition of the results

The main objective of this paper is to consider the complexity of MB POSET POSITIONAL GAME related to some parameters of the instance. More precisely, we have chosen to focus on the properties of the poset, as it is the main distinctiveness of the current contribution,

comparing the results we obtain to the previous results about standard positional games. In Section 2, we will firstly examine the problem depending on the height of the poset. As it is already known that, even for height 1 (i.e. all the elements are pairwise incomparable) and winning sets of size 6, the problem is PSPACE-complete [13], we will also refine our classification according to the number and/or the size of the winning sets. The main contribution of this section addresses the case where there is only one winning set of size 1. By adding such a condition, the problem admits a complexity jump between instances of height 2, proved to be polynomial, and height 3, proved to be NP-hard.

Section 3 deals with the width of the poset. As the case of width 1 is straightforward (P is made of only one chain, so the order of the moves is completely predetermined), we start by considering instances with width 2. We show that MB POSET POSITIONAL GAME is PSPACE-hard in this case, even if all the winning sets are of size 3. This illustrates a major difference with standard positional games, which are known to be tractable in this setting. We then give a polynomial-time algorithm that solves the general case where the width of the poset and the number of winning sets are fixed.

Section 4 is devoted to the case where the poset is a union of disjoint chains. This case is a direct generalization of the game Connect-4. Our first result is a full characterization of the outcome when all the winning sets are of size 1. When the winning sets are of size at most 2, things are more tricky, but we do provide a polynomial-time algorithm when the width of the poset (i.e. the number of chains in the union) is fixed. Finally, by adding the restriction that the height of each chain is at most 2, we get a polynomial-time algorithm regardless of the number of chains.

Table 1 summarizes all the results about the complexity of MB POSET POSITIONAL GAME, referring either to the literature of standard positional games, or to results proved in the current paper. In the table, recall that the complexity class XP defines the class of problems parameterized by a parameter k and that can be solved in time $O(|X|^{f(k)})$, where $|X|$ is the size of the instance and f is a computable function.

In order to satisfy the page limit, some of our statements are given without a proof. All the proofs are available in the full version of our paper [5].

2 Posets of small height

2.1 Posets of height 2

We first look at what happens when all the winning sets are of size 1. This case is rarely straightforward, and often deserves to be studied depending on some parameters of the poset. In addition, it is closely correlated to the case where there is a unique winning set (of any size). The following remark explains the link between both situations.

► **Remark 1.** Up to switching the roles of the players, a poset positional game with a single winning set of size k is equivalent to a poset positional game with k winning sets of size 1.

In the above remark, the equivalence means that the two games have the same outcome. Indeed, Maker wins when there is a single winning set of size k if and only if she manages to claim all the elements of this set. In the second game, a win of Breaker consists in claiming all the winning sets (of size 1), thus corresponding to the equivalence. Note that Breaker is starting in one of the two games, but this is not a pitfall since both these classes are stable under Breaker's moves.

■ **Table 1** Complexity of MB POSET POSITIONAL GAME depending on some parameters of the poset and of the collection of winning sets.

winning sets \ poset	general	height h	width w	disjoint chains
general	PSPACE-c [13]	$h = 1$: PSPACE-c [13]	$w = 1$: P $w = 2$: PSPACE-c (Th. 7)	$h = 1$: PSPACE-c [13]
size s	$s = 3$: PSPACE-c (Th. 7)	$h = 1, s = 6$: PSPACE-c [13] $h = 2, s = 3$: NP-h (Th. 5)	$w = 2, s = 3$: PSPACE-c (Th. 7)	$s = 1$: P (Th. 11) $s = 2$: XP by w (Th. 14) $h = 2, s = 2$: P (Th. 15)
number m	NP-hard (Th. 6)	$m = 1, h = 3$: NP-hard (Th. 6)	XP (Th. 8)	$m = 1$: P (Th. 11 and Remark 1)
number m and size s	$m = 1, s = 1$: NP-hard (Th. 6)	$h = 2, m = 1, s = 1$: P (Th. 3) $h = 3, m = 1, s = 1$: NP-hard (Th. 6)	XP (Th. 8)	XP (Th. 8)

We continue with a minor result which is useful when dealing with winning sets of size 1. Given a poset P and a vertex x , we denote by $p(x)$ the number of vertices that are not greater or equal to x i.e. $p(x) := |X| - |\{y \mid y \geq x\}|$. This quantity is used in the following lemma, which yields a necessary and sufficient condition for Maker to win when there is a unique winning set of size 1.

► **Lemma 2.** *Let $\mathcal{G} = (X, \{\{x\}\}, P)$ be a poset positional game. Suppose that, after some rounds of play in \mathcal{G} , all predecessors of x have been claimed apart from one, which we denote by y . Then, in this position, Maker wins if and only if $p(y)$ is odd.*

► **Theorem 3.** *MB POSET POSITIONAL GAME can be solved in polynomial time when: the poset is of height 2, there is only one winning set, and this winning set is of size 1.*

Sketch of the proof. Let $\{x\}$ be the winning set. We partition the predecessors of x into two sets M and B , where M (resp. B) contains the predecessors y of x such that $p(y)$ is even (resp. odd). Then Maker wins if and only if $|M| \leq |B|$. ◀

The situation is already much more complicated with two winning sets of size 1 – we leave this as an open problem. However, for winning sets of size 1 and a poset of height 2 whose nonminimal elements are all winning sets, one can compute the outcome in polynomial time, as we now show. Note that the same situation is PSPACE-complete in the Maker-Maker convention (see the full version [5]).

► **Theorem 4.** *MB POSET POSITIONAL GAME can be solved in polynomial time when: the poset is of height 2, all the winning sets are of size 1, and all nonminimal elements are winning sets.*

Sketch of the proof. Let P be a poset of height 2 on a set X . We say the minimal elements of P are the “bottom” vertices, and the others are the “top” vertices. We say that x is a *private predecessor* of y if y is the only successor of x .

The poset positional game we consider is $\mathcal{G} = (X, \{\{y\}, y \text{ top vertex}\}, P)$. We prove that Maker wins if and only if (1) $|X|$ is odd or (2) there exists a top vertex y which does not have more private predecessors than non-private ones.

If (1) holds then, since Maker starts, she also plays last, which means she can win by waiting for Breaker to claim the last predecessor of some top vertex. If (2) holds, then Maker can ensure that all the private predecessors of y are claimed before all the non-private ones are, after which she waits for Breaker to claim the last predecessor of y . She might have to claim it herself, but that will make at least two top vertices available at once, meaning she will win with her next move.

If none of the conditions hold, then Breaker can maintain the property that “all the top vertices have strictly more private predecessors than non-private ones” during the game. This allows Breaker to claim all the top vertices, meaning he wins. ◀

For posets of height $h = 2$, as the size s of the winning sets increases, the value $s = 3$ is the smallest one for which we have identified an algorithmic complexity jump.

► **Theorem 5.** MB POSET POSITIONAL GAME is NP-hard, even when restricted to instances where the poset has height 2 and all the winning sets are of size 3.

Sketch of the proof. The proof is a reduction from 3-SAT. Let ϕ be a 3-SAT formula. We build a poset positional game $\mathcal{G} = (X, \mathcal{F}, P)$ with P of height 2 and winning sets of size 3 as follows:

- For any variable x_i of ϕ , we add four vertices u_i, v_i, \bar{v}_i and \tilde{v}_i in X .
- For any clause $C_j = l_{j_1} \vee l_{j_2} \vee l_{j_3}$ in ϕ , we add the winning set S_j with, for $1 \leq k \leq 3$, $v_{j_k} \in S_j$ if $l_{j_k} = x_{j_k}$, and $\bar{v}_{j_k} \in S_j$ if $l_{j_k} = \neg x_{j_k}$.
- For any variable x_i of ϕ , we add the relations $u_i < v_i$, $u_i < \bar{v}_i$ and $u_i < \tilde{v}_i$ in P .

We prove that ϕ is satisfiable if and only if Breaker has a winning strategy in \mathcal{G} . ◀

2.2 Posets of height 3

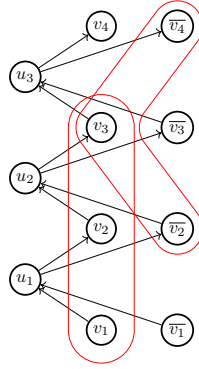
We now consider posets of height 3. Using Lemma 2, we prove that MB POSET POSITIONAL GAME is NP-hard even if there is only one winning set and that winning set is of size 1. The reduction is done from the problem SET COVER.

► **Theorem 6.** MB POSET POSITIONAL GAME is NP-hard even when restricted to instances where: the poset has height 3, there is only one winning set, and that winning set is of size 1.

3 Posets of bounded width

In this section, we consider posets of bounded width. For width 1, all moves are forced, so the outcome is obviously computed in polynomial time. However, for width 2, we prove that MB POSET POSITIONAL GAME is already PSPACE-hard even if the winning sets are of size 3.

► **Theorem 7.** MB POSET POSITIONAL GAME is PSPACE-complete even when restricted to instances where the poset is of width 2 and the winning sets are of size 3.



■ **Figure 1** The poset positional game obtained by reduction of $\forall x_1 \exists x_2 \forall x_3 \exists x_4 (x_1 \vee x_2 \vee x_3) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$.

Sketch of the proof. The reduction is done from 3-QBF, which has been proved PSPACE-complete by Stockmeyer and Meyer [15]. In 3-QBF, the input is a formula of the form $\psi = \forall x_1 \exists x_2 \dots \forall x_{2n-1} \exists x_{2n} \phi(x_1, \dots, x_{2n})$ where ϕ is a 3-CNF formula. The output is the truth value of ψ . We build a poset positional game $\mathcal{G} = (X, \mathcal{F}, P)$ as follows:

- For any $1 \leq i \leq 2n$, we add two vertices v_i and \bar{v}_i in X .
- For any $1 \leq i \leq 2n - 1$, we also add a vertex u_i .
- For any clause $C_j = l_{i_1} \vee l_{i_2} \vee l_{i_3}$ in ϕ , we add a winning set $S_j \in \mathcal{F}$. For $1 \leq k \leq 3$, we have $v_{i_k} \in S_j$ if $l_{i_k} = x_{i_k}$ and $\bar{v}_{i_k} \in S_j$ if $l_{i_k} = \neg x_{i_k}$.
- For any $1 \leq i \leq 2n - 1$, we add in P the relations $v_i < u_i$, $\bar{v}_i < u_i$, $u_i < v_{i+1}$ and $u_i < \bar{v}_{i+1}$.

See Figure 1 for an example. It can be proved that ψ is True if and only if Breaker wins in \mathcal{G} . ◀

If the number of winning sets is also bounded (by m), we give an algorithm running in time $O(|X|^{w2^m w^4})$.

► **Theorem 8.** MB POSET POSITIONAL GAME can be decided in time $O(|X|^{w2^m w^4})$ for instances where the poset is of width at most w and there are at most m winning sets.

Sketch of the proof. The proof uses a dynamic approach. We characterize the games that can be reached from a game played on $\mathcal{G} = (X, \mathcal{F}, P)$ using an antichain Y of X , that represents the current available moves, and a boolean vector $B = (b_1, \dots, b_m) \in \{0, 1\}^m$ that represents, for each winning set, if Maker has claimed all its elements below Y (and thus can still hope to claim it entirely). ◀

4 Posets made of pairwise disjoint chains

In this section, motivated by the game Connect-4, we consider posets made of pairwise disjoint chains. In the rest of this section, we give positive results for poset positional games on pairwise disjoint chains with very small winning sets (but an unbounded number of them). We will often use parity arguments when discussing games on posets made of pairwise disjoint chains. Therefore, we introduce the following framework, which will be useful in this section. Consider a poset positional game $\mathcal{G} = (X, \mathcal{F}, P)$ where P is made of pairwise disjoint chains C_1, \dots, C_w . The elements of the chain C_i are denoted as $x_{i,1} > x_{i,2} > \dots > x_{i,\ell_i}$. This numbering of the vertices from top to bottom is best adapted to define the following coloring: a vertex $x_{i,j}$ is colored white if j has same parity as $|X|$, otherwise it is colored black.

4.1 Winning sets of size 1

The next two lemmas give sufficient conditions for Maker to win under particular conditions that are based on the above coloring. They can be proved with simple parity arguments. Combined together in the next theorem, they allow to have a complete characterization of the general case where all the winning sets are of size 1.

► **Lemma 9.** *If $\{u\} \in \mathcal{F}$ for some white vertex u , then Maker wins.*

► **Lemma 10.** *If $|X|$ is odd and $\{u\}, \{v\} \in \mathcal{F}$ for some black vertices u and v sitting on different chains, then Maker wins.*

► **Theorem 11.** *Let $\mathcal{G} = (X, \mathcal{F}, P)$ be a poset positional game where P is made of pairwise disjoint chains and all elements of \mathcal{F} are of size 1. Maker wins \mathcal{G} if and only if at least one of the following conditions hold:*

- *there is a winning set that is a minimal element; or*
- *there is a white winning set; or*
- *$|X|$ is odd and there are two black winning sets on different chains.*

In particular, MB POSET POSITIONAL GAME can be solved in linear time for such games.

Sketch of the proof. The “only if” direction is the one that remains to be proved. For even $|X|$, we show that Breaker can claim all the nonminimal black vertices. For odd $|X|$, we show that Breaker can claim all the nonminimal black vertices of any given chain. ◀

Using this result, we can solve a basic case of $w \times h$ *Connect- k* in Maker-Breaker convention, which is the poset positional game played on w chains of height h where the winning sets are the alignments of k consecutive vertices (horizontal, vertical or diagonal). Note that the game is easily solved if $k \leq 2$, or if $k > w$, or if $k > h$ and wh is even.

► **Corollary 12.** *Let k, w, h be integers with $3 \leq k \leq w$ and $h > 1$. If w and h are both odd, then Maker wins the $w \times h$ *Connect- k* game in Maker-Breaker convention.*

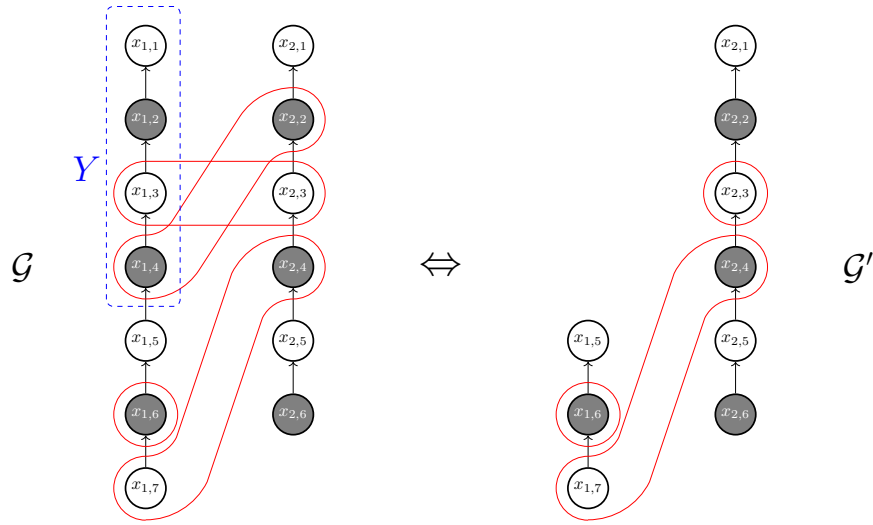
4.2 Winning sets of size at most 2

Things become more complicated when the winning sets are not all of size 1. In the presence of a black winning set of size 1, we can simplify the game. Using the same notations as before, assume that there exists a winning set $\{x_{i,j}\}$ for some nonminimal black vertex $x_{i,j}$ with $j \geq 3$. We define a reduced game $\mathcal{G}' = (X', \mathcal{F}', P')$ as follows.

If $|X|$ is even, then define $Y = \{x_{i,1}, \dots, x_{i,j-1}\}$, otherwise define $Y = \{x_{i,1}, \dots, x_{i,j-2}\}$. In both cases, let $X' = X \setminus Y$ and let P' be the poset induced by P on X' . Note that, since $x_{i,j}$ is black, we have removed an even number of vertices in both cases. In particular, the coloring of the vertices is the same for (X', P') as for (X, P) . We now define the winning sets of \mathcal{G}' as follows. Let $S \in \mathcal{F}$. If $S \subseteq X'$, then $S \in \mathcal{F}'$. If S only intersects Y on white vertices, then $S \cap X' \in \mathcal{F}'$. Otherwise, i.e. if S contains a black vertex greater than $x_{i,j}$, then we ignore S . See Figure 2 for an illustration in the case where $|X|$ is odd.

► **Lemma 13.** *Assume that \mathcal{G} has no winning set containing only white vertices that are greater than $x_{i,j}$. Then the two games \mathcal{G} and \mathcal{G}' have the same outcome when Maker starts.*

Sketch of the proof. We prove that, if a player has a winning strategy in \mathcal{G}' , then that player has also a winning strategy in \mathcal{G} . The idea is the same for both players: they follow their strategy in \mathcal{G}' , as long as possible. When $x_{i,j}$ is claimed (by Breaker, otherwise Maker wins immediately), Maker can ensure to claim all the white vertices above $x_{i,j}$, thus completing a



■ **Figure 2** Reduction of a game \mathcal{G} containing a black winning set of size 1 to a game \mathcal{G}' . The winning set $\{x_{1,4}, x_{2,2}\}$ disappears since it contains a black vertex in Y . The winning set $\{x_{1,3}, x_{2,3}\}$ becomes the winning set $\{x_{2,3}\}$ in \mathcal{G}' since $x_{1,3}$ is white. By Lemma 13, the two games have the same outcome. In this case, since \mathcal{G}' has a white winning set of size 1, both games are winning for Maker.

winning set she would have filled up in \mathcal{G}' . If she cannot fill up a winning set of \mathcal{G}' (i.e. if Breaker wins \mathcal{G}'), then Breaker can claim all the black vertices above $x_{i,j}$ and thus claim a vertex in all the winning sets of \mathcal{G} . ◀

Now equipped with this reduction, we can provide an algorithm when all the winning sets have size at most 2. Our algorithm is based on a dynamic programming approach. Thanks to Lemmas 9 and 13, we can reduce the number of sub-positions to consider. Indeed, we can assume that each chain contains at most one (black) winning set of size 1. Our algorithm dynamically computes the outcome of all these “useful” positions.

► **Theorem 14.** MB POSET POSITIONAL GAME can be solved in time $O(h^{2(w+1)})$ for instances where the poset consists of w pairwise disjoint chains of height at most h and all winning sets have size at most 2.

4.3 Winning sets of size at most 2, chains of height at most 2

The algorithm from Theorem 14 is not efficient for posets of unbounded width. However, when restricting the problem to posets of height at most 2, we do get a polynomial-time algorithm in this case also. Since the chains are of height at most 2, we will simply use the “top/bottom” terminology already adopted in the proof of Theorem 4, rather than the coloring previously used in this section.

► **Theorem 15.** MB POSET POSITIONAL GAME can be solved in time $O(|X|^4)$ for instances where the poset on X consists of pairwise disjoint chains of height at most 2 and all winning sets are of size at most 2.

Sketch of the proof. Let $x_1, \dots, x_t, y_1, \dots, y_t$ be the vertices of the chains of height 2, with $x_i < y_i$ for all $1 \leq i \leq t$. Let x_{t+1}, \dots, x_w be the other vertices if there are any. We say the x_i are the “bottom” vertices and the y_j are the “top” vertices. The proof distinguishes

two cases depending on the parity of $|X|$. When $|X|$ is odd, there are several possibilities, but each of them is a straightforward win for one of the two players. When $|X|$ is even, the situation is more tricky. In that case, the key argument is based on two reductions R1 and R2 that preserve the outcome:

- R1:** Assume there exists a winning set of the form $\{x_i, x_j\}$ that is disjoint from all the other winning sets. Then remove x_i and x_j .
- R2:** Assume there exists a nonempty set of indices $I \subseteq \{1, \dots, t\}$ such that all the winning sets that contain some x_i with $i \in I$ also contain some y_j with $j \in I$. Then remove all the vertices with indices in I . ◀

5 Conclusion and future work

The current work introduces a new framework that opens the door to a variety of perspectives. Our analysis with respect to the parameters of the poset led to a first classification of the complexity of MB POSET POSITIONAL GAME, which is a step towards a better understanding of the boundary between tractability and hardness. The results presented here directly induce a list of open problems that arise naturally in order to refine this boundary. Among them, we have identified the following two questions that seem the most relevant for us:

- When there is exactly one winning set of size 1, the height of the poset makes a difference between tractability ($h = 2$) and NP-hardness ($h = 3$). Therefore, when $h = 2$, it is worthwhile to examine how the conditions on the number m of winning sets and their size s impact the algorithmic complexity. A next study would be to investigate the case $m = 2$ and $s = 1$.
- In the case of disjoint chains, an analysis of the complexity according to their width (i.e. the number of chains) is a natural perspective. In particular, one may focus on the case $w = 2$, even when restricted to $s \leq 3$ (since we have a polynomial-time algorithm for $s = 2$). It would also be interesting to obtain a hardness results for disjoint chains.

Going back to our initial motivation of giving a general framework for Connect- k games, one could also examine the case of boards of even size for this game. For example, the famous case $(k, w, h) = (4, 7, 6)$, is known to be a Maker win in the Maker-Breaker convention since it a first player win in the Maker-Maker convention. However, a direct proof for the Maker-Breaker convention could be considered, that could also be extended to other sizes.

Finally, while we mostly studied the poset positional games in the Maker-Breaker convention, our definition can be transposed to all other conventions of positional games. It would therefore be interesting to perform a similar analysis for Maker-Maker or Avoider-Enforcer poset positional games.

References

- 1 James D. Allen. Expert play in Connect-Four, 1990. URL: <https://tromp.github.io/c4.html>.
- 2 James D. Allen. *The Complete Book of Connect 4: History, Strategy, Puzzles*. Puzzlewright Press, 2010.
- 3 Louis Victor Allis. *A Knowledge-Based Approach of Connect-Four*. Report IR-163. Vrije Universiteit Amsterdam, The Netherlands, 1988.
- 4 Pranav Avadhanam and Siddhartha G. Jena. Restricted positional games. *CoRR*, 2021. Preprint. [arXiv:2108.12839](https://arxiv.org/abs/2108.12839).

- 5 Guillaume Bagan, Éric Duchêne, Florian Galliot, Valentin Gledel, Mirjana Mikalački, Nacim Oijid, Aline Parreau, and Miloš Stojaković. Poset positional games. *CoRR*, 2024. Preprint. [arXiv:2404.07700](https://arxiv.org/abs/2404.07700).
- 6 József Beck. *Combinatorial Games: Tic-Tac-Toe Theory*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2008. doi:10.1017/CB09780511735202.
- 7 Jesper Makholm Byskov. Maker-Maker and Maker-Breaker games are PSPACE-complete. *BRICS Report Series*, 11(14), 2004.
- 8 Noam D. Elkies. On numbers and endgames: combinatorial game theory in chess endgames. In *Games of No Chance, Proceedings of 7/94 MSRI Conference on Combinatorial Games*, pages 135–150. Cambridge University Press, 1996. URL: <https://api.semanticscholar.org/CorpusID:17418896>.
- 9 Paul Erdős and John L. Selfridge. On a combinatorial game. *Journal of Combinatorial Theory*, 14:298–301, 1973. doi:10.1016/0097-3165(73)90005-8.
- 10 Florian Galliot, Sylvain Gravier, and Isabelle Sivignon. Maker-Breaker is solved in polynomial time on hypergraphs of rank 3. *CoRR*, 2022. Preprint. [arXiv:2209.12819](https://arxiv.org/abs/2209.12819).
- 11 Alfred W. Hales and Robert I. Jewett. Regularity and positional games. *Trans. Am. Math. Soc*, 106:222–229, 1963.
- 12 Dan Hefetz, Michael Krivelevich, Miloš Stojaković, and Tibor Szabó. *Positional Games*, volume 44 of *Oberwolfach Seminars*. Birkhäuser (Springer), 2014. doi:10.1007/978-3-0348-0825-5.
- 13 Md Lutfar Rahman and Thomas Watson. 6-uniform Maker-Breaker game is PSPACE-complete. In *Proceedings of the 38th International Symposium on Theoretical Aspects of Computer Science (STACS 2021)*, volume 187 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 57:1–15, 2021.
- 14 Thomas J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16:185–225, 1978.
- 15 Larry J Stockmeyer and Albert R Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 1–9, 1973.
- 16 John Tromp. John’s Connect Four Playground. URL: <http://tromp.github.io/c4/c4.html>.
- 17 John Tromp. Solving Connect-4 on medium board sizes. *ICGA Journal*, 31(2):110–112, 2008.

Snake in Optimal Space and Time

Philip Bille ✉ 

Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Martín Farach-Colton ✉ 

New York University, NY, USA

Inge Li Gørtz ✉ 

Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Ivor van der Hoog ✉ 

Department of Applied Mathematics and Computer Science, Technical University of Denmark, Lyngby, Denmark

Abstract

We revisit the classic game of Snake and ask the basic data structural question: how many bits does it take to represent the state of a snake game so that it can be updated in constant time? Our main result is a data structure that uses optimal space (within constant factors). To achieve our results, we introduce several interesting data structural techniques, including a decomposition technique for the problem, a tabulation scheme for encoding small subproblems, and a dynamic memory allocation scheme.

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

Keywords and phrases Data structure, Snake, Nokia, String Algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.3

Funding *Philip Bille*: Supported by the Independent Research Fund Denmark (DFR-9131-00069B and 10.46540/3105-00302B).

Inge Li Gørtz: Supported by the Independent Research Fund Denmark (DFR-9131-00069B and 10.46540/3105-00302B).

Ivor van der Hoog: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

1 Introduction

The classic game of *Snake* involves players navigating a grid with a growing line (themed as a snake) while avoiding collisions with itself and the boundary. As the line increases in length, the game gets progressively harder. The game originated in the 1976 arcade video game Blockade and later evolved into a single-player version (see the history overview of the game [11]). In 1988, the game largely became responsible for introducing mobile phone gaming to the world after being included with the Nokia 6110 cellular phone [12]. Subsequently, the Snake game has appeared in several different versions on Nokia phones and elsewhere. From a CS perspective, the Snake game has been studied in the context of motion-planning problems [9] and deep learning [1, 8, 10]. Surprisingly, the game has not been studied from a data structural perspective. In this paper, we ask the basic question: how many bits does it take to represent the state of a snake game so that it can be updated in constant time? Our main result uses asymptotically optimal bits of space and constant time per operation. To achieve our results, we introduce several interesting data structural techniques, including a decomposition technique for the problem, a tabulation scheme for encoding small subproblems, and a dynamic memory allocation scheme.



© Philip Bille, Martín Farach-Colton, Inge Li Gørtz, and Ivor van der Hoog; licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 3; pp. 3:1–3:15

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Setup and Results

Consider a $u \times u$ grid G . A *snake* S is a sequence of n distinct points s_1, \dots, s_n in G such that any two consecutive points are adjacent in either the vertical or horizontal direction. We call s_n and s_1 the *head* and the *tail* of S , respectively. Our goal is to maintain S dynamically and compactly while supporting the following operations:

- `extend(d)`: where d is a direction (up, down, left, or right). Add a new point to S adjacent to the head in the direction d . If the new point is already a point on the S or outside of G , we terminate and report a collision.
- `reduce()`: remove the tail from S .

There are two immediate solutions to this problem. The first solution is to explicitly store the grid as a bit string of length u^2 with 1's at all positions containing a point of S and 0's elsewhere. We additionally maintain the positions of the head and tail of S . To implement the operations, we use the bit string to check for collisions and update the bit string and head and tail pointers accordingly. This solution uses $O(u^2 + \log u) = O(u^2)$ bits of space and implements both operations in constant time. Alternatively, we can store the head and tail of S separately plus all points in S in a balanced search tree, where the points in S are sorted in lexicographical order. To implement the operations, we search the tree for the collision check and update the snake by insertions and deletions in the tree.

This solution uses $O(n \log u)$ bits and supports operations in $O(\log n)$ time. Using exponential search trees [2], we can improve the time of this solution to $O(\sqrt{\log n / \log \log n})$, or if we allow randomization, to constant time with high probability [3–7]. All of the above solutions work on a standard word RAM model of computation. Each word can store the location of a coordinate in the grid and hence the word length $w \geq \log u$. In the same model, we show the following result.

► **Theorem 1.** *We can represent a snake of length n in $O(n + \log u)$ bits and support `extend` and `reduce` operations in constant time.*

Note that this improves all of the above combinations of time and space bounds. Any solution must use at least $\Omega(n + \log u)$ bits.

Techniques. To obtain the results of Theorem 1, we introduce and combine several interesting data structural techniques. We first present a simple solution using $O(n \log u)$ bits of space and constant time for operation. This solution is based on a partitioning of the grid into square subgrids of size $\log u \times \log u$, called *tiles*. The main component of this structure is a standard balanced binary search tree that stores all of the non-empty tiles (i.e., the tiles intersected by the snake). For each such tile, we then store a bit string of length $\log^2 u$ that encodes the positions of the snake within the tile. This uses $O(n \log u)$ bits of space.

To support the operations in constant time, we show how to efficiently schedule and buffer operations. Intuitively, since the tiles have size $\log u \times \log u$ and the snake only moves a single position in each operation, we can schedule the needed traversals and updates of the balanced binary search tree using constant additional time at each operation.

Next, we extend our solution to add another level of tiles of doubly logarithmic size “nested” within the first level. The new level also maintains balanced binary search trees of non-empty tiles and an encoding of the snake within each such tile. We show how to maintain the structure as above in constant time per operation. Here, the key challenge is achieving and implementing the structure in optimal space. First, we cannot afford to explicitly encode a tile at level 2 as above. Instead, we show how efficiently encode the

snake within these tiles and tabulate the operations. Secondly, we cannot afford to explicitly store $\log u$ pointers between different components of the data structure (such as the balanced binary search trees at the lowest level). Instead, we present a new memory allocation scheme that efficiently packs together components of the data structure such that the entire structure fits on $O(n + \log u)$ bits of space.

2 Fast Snake with $O(n \log u)$ Bits of Space

As a warm-up, we first demonstrate an approach that uses $O(n \log u)$ bits of space and constant time for each operation. Our solution relies upon *tilings* of the grid G .

► **Definition 2.** *Given a parameter $\tau > 1$, we partition G into $\lfloor u^2/\tau \rfloor$ squares called tiles, of size $\tau \times \tau$ (the bottom row and rightmost column may be smaller). We call this a τ -tiling of G . A tile v is empty if it does not contain a point of S . Otherwise, it is non-empty. The τ -tile set for S , denoted by T_S^τ , is the set of non-empty tiles in a τ -tiling of G .*

Since S is a collection of consecutively adjacent points in G , we may immediately conclude:

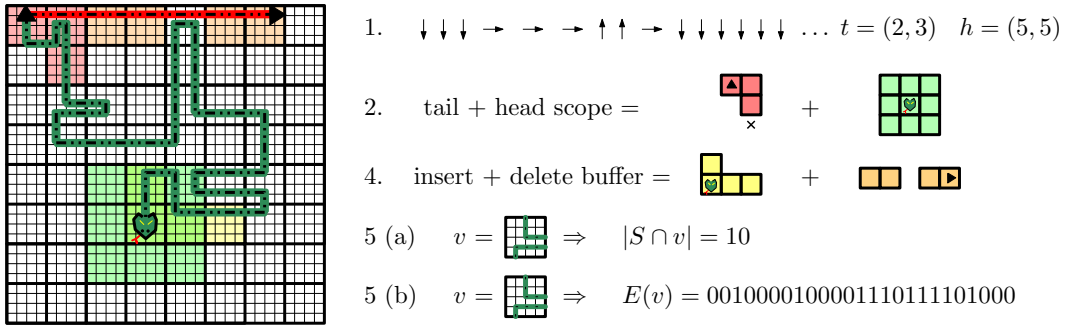
► **Lemma 3.** *For all snakes of length n , for all τ , $|T_S^\tau| \leq 4 \lceil \frac{n}{\tau} \rceil$.*

2.1 Data Structure

We construct a tiling for G with parameter $\tau = \log u$. Let S be a snake of length n . Our data structure consists of the following components (Figure 1):

1. A *direction string* D of length $n - 1$ that stores for each position in S , except the head h , the direction of its successor. It also stores the *relative coordinates* of h and t .
2. A *head and tail scope*. The head scope stores a pointer to the τ -tile $v \in T$ that contains h , the relative coordinates of the head h in v , and the up to eight non-empty τ -tiles incident to v . The tail scope stores a pointer to each of the τ -tiles intersected by the last $\tau/2$ points of S , the coordinates of the tail t , and the coordinates of the point $t_\tau = s_\tau$. If $n < \tau$ then $t_\tau = h$.
3. A balanced binary search tree B containing the tile set T_S^τ . The tiles in B are ordered by their top-left coordinates in lexicographical order.
4. An *insert and delete* buffer. The insert buffer stores up to the last four τ -tiles that were empty before the head entered them (these tiles are not yet in B). The delete buffer stores up to the first four τ -tiles that became empty after the tail left them (these tiles are not yet deleted from B).
5. For each τ -tile $v \in T_S^\tau$,
 - a. A counter recording $|S \cap v|$.
 - b. A bit string $E(v)$ that stores a single bit for each position in v , indicating if the position is empty or non-empty.

The direction string D uses $O(n)$ bits, as we encode each direction with two bits. The head and tail scope use $O(\log u)$ bits as they store $O(1)$ pointers. The counters use $O(n)$ bits in total, as the total number of counters is at most $O(n/\tau)$ and each counter is at most $2 \log \tau$ bits. The binary search tree uses $O(n)$ bits as by Lemma 3, $|T_S^\tau| \in O(n/\tau)$ and each pointer uses at most $\tau = \log u$ bits. A single bitstring $E(v)$ uses τ^2 bits as each tile v contains τ^2 points. Thus the total space for the bitstrings is $O(\tau^2(n/\tau)) = O(n \log u)$.



■ **Figure 1** On the left we show a grid G , tiled with τ -tiles for $\tau = 5$. There is a snake in green. The red indicates the previous positions of the snake, which are no longer occupied. On the left, we show the components of our data structure.

2.2 The extend Operation

We now explain how to implement the extend operation. Let h be the head of S before the operation, and let h' be the new position of the head. Also, let v and v' be the τ -tiles in the grid that contain h and h' , respectively. We proceed as follows:

Step 1: Check for Collisions. We check if there is a collision of h' with a point on S . Given h , D , and the head scope, we first identify v' and compute the relative coordinates of h' in constant time. We can test whether v' is empty using its pointer to the head scope in $O(1)$ time. There are two cases depending on whether v' is empty or not:

- (i) If v' is empty, there is no collision. We then construct $E(v')$ in $O(1)$ time. We set the corresponding counter to one. We then check if v' is in the delete buffer (in this case, we may be in the process of deleting v' from our data structure). If it is and it is not the first tile in there, we remove it from the delete buffer. Otherwise, we add v' to the back of the insert buffer.
- (ii) If v' is non-empty, we use $E(v')$ and the relative coordinates of h' in v' to check if there is a collision. If so, we stop and output this. Otherwise, we update $E(v')$ and the corresponding counter in $O(1)$ time.

Finally, we update D in constant time.

Step 2: Update Search Tree and Buffers. We move τ -tiles from the insert buffer into B . To do so, we do a constant amount of work towards inserting the first τ -tile x in the insert buffer into B . We do enough work to ensure that after $\tau/8$ extend and reduce operations, x is fully inserted into B . Since an insertion or a deletion in B takes $O(\tau)$ time, this can be obtained by doing $O(1)$ work for each operation. When the tile is fully inserted into B , we remove it from the insert buffer. Similarly, we do a constant amount of work towards deleting the first τ -tile x in the deletion buffer from B . By Lemma 3, it follows that there can be no more than 4 tiles in each of the buffers at any point in time.

Step 3: Update Scopes. The head scope is a set of 3×3 tiles centered at the τ -tile that contains h . To maintain the invariant that the scope is in memory at all times, we store slightly more than the scope. Consider the set of 7×7 tiles centered at the τ -tile that contains h and denote it by N_h . Each non-empty $x \in N_h$ is either in B (which supports lookups in $O(\tau)$ time) or in the insert/delete buffer (which have constant size). Each time

we perform `extend`, we do constant work towards a lookup for each tile in N_h . If we finish a lookup and find a τ -tile z , we add z to our head scope list. If a tile $z' \in N_h$ is in the head scope list and $z' \notin N_{h'}$, we delete z' from the list. This way, we maintain that the head scope is always in the head scope list whilst having $O(1)$ τ -tiles in the head scope list at all times. Note that this does not change the asymptotic space of the data structure. By Lemma 3, if $|S| > \tau$, the tail scope remains unchanged. Else, we add v' to the tail scope if $v \neq v'$ and update t_τ .

2.3 The reduce Operation

We show how to implement the `reduce` operation. Let t be the tail of S before the operation and let v be the τ -tile in the grid that contain t . We proceed as follows:

Step 1: Update Search Trees and Buffers. Given t and the tail scope, we identify v in $O(1)$ time. Given the relative coordinates of t in v , we update $E(v)$ and the corresponding counter in $O(1)$ time. If v is now empty, we first check if it is in the insert buffer. If v is in the insert buffer and it is not the first tile in there, then we remove it from the insert buffer. Otherwise, we add it to the delete buffer. Using t , t_τ and D we update the tail t and t_τ to their new positions. As in the `extend` operation, we then do a constant amount of work towards inserting the first tile from the insert buffer into B and deleting the first τ -tile x in the delete buffer from B .

Step 2: Update Tail Scope. Let the extended tail scope N_t be the set of tiles intersected by the last τ points of the snake. As with the head scope, we will maintain a set of tiles, including all the tiles in the tail scope and possibly some of the tiles in N_t that are not in the tail scope. Let v' be the τ -tile containing t_τ . If v' is already in our tail scope list, we do nothing. Otherwise, we do a constant amount of work towards the search for each of the tiles in the extended tail scope N_t that are not yet in the tail scope list. We do this fast enough to ensure that we have finished the search for v' when it belongs to the tail scope. When we have finished the search, we add v' to the tail scope list. By Lemma 3, there can be at most 4 tiles in the extended tail scope, and thus, we never search for more than 4 tiles at a time. Note that this does not change the asymptotic space of the data structure. Since a search in B takes $O(\tau)$ time, this can be done with a constant amount of work for each `reduce` operation.

2.4 Summary

In summary, the space of the data structure is $O(n \log u)$ bits, and each of the operations uses constant time. Hence, we have the following result.

► **Lemma 4.** *We can represent a snake S of length n in $O(n \log u)$ bits and support `extend` and `reduce` operations in constant time.*

In the following sections, we improve our solution to obtain the result of Theorem 1. We first show how to efficiently tabulate small subproblems for tiles of size $\log \log n$ in Section 3. In Section 4, we then extend the above solution to a two-level data structure with nested tilings of sizes $\log n$ and $\log \log n$ and apply the tabulation at the lowest level. Unfortunately, naively storing our data structure would use $\log u$ pointers between different components, which we can not afford within our space bound of $O(n + \log u)$ bits. In Section 5, we show how to dynamically allocate all components of the data structure compactly leading to Theorem 1.

3 Tabulation

We now show how to efficiently tabulate τ -tiles for $\tau = \log \log n$. This will be a key component in our multi-level data structure. Intuitively, we consider all possible ways a snake S can intersect a tile v of dimension $\tau \times \tau$, together with all placements of the (future) head of S , and store for each combination the outcome after executing the update. Formally, fix any snake S and let v be any $\tau \times \tau$ square. We define two concepts:

► **Definition 5.** We denote for any $\tau \times \tau$ square v by δv all vertices of the grid that are in v and incident to the boundary of v , in the clockwise direction.

► **Definition 6.** We define a marked grid as any $\tau \times \tau$ square X that marks each point in the tile as either $\{\text{empty, occupied, head, tail}\}$. For each snake S and τ -tile v we immediately have a corresponding marked grid $S \cap v$. Finally, for any pair of marked grids X, Y , we denote $X \simeq Y$ whenever the two marked grids are identical.

3.1 Tabulation Code

We define what we call our *tabulation code* $\text{ENC}_S(v)$ which is the concatenation of four strings $\alpha(v), \beta(v), \gamma(v), \epsilon(v)$. To this end, we note that S uniquely corresponds to a rectilinear curve that is obtained by connecting consecutive grid points in S by an edge. For any rectilinear curve S and any square v , we intuitively define the set of maximally connected subcurves in $S \cap v$. We define:

- The string $\alpha(v)$ is defined by traversing the vertices $x \in \delta v$ in order and denoting a 1 whenever $x \in S$ and a zero otherwise.
- The string $\beta(v)$ considers all maximal subcurves $S' \subset S \cap v$ that intersect δv sorted by their first point of intersection with δv . The string $\beta(v)$ concatenates for each maximal subcurve S' a string $\beta(S')$. The string $\beta(S')$ denotes for all $x \in S'$ the direction of its successor by using two bits, followed by a symbol that denotes the end of S' .
- There are at most two subcurves in $S' \subset S \cap v$ that do not intersect δv ; these must contain the head and/or tail of S and are considered at the end. If the head and tail of S are not in v then $\gamma(v)$ denotes a *null* symbol. Otherwise, $\gamma(v)$ uses $2 \log \tau$ bits to specify the relative position of h (and/or t) in v .
- If the string $\gamma(v)$ is not null, the string $\epsilon(v)$ considers the maximal subcurve S' of $S \cap v$ that contains the head (or tail) and denotes for all $x \in S'$ the direction of its successor by using two bits, followed by a symbol that denotes the end of S' .

► **Lemma 7.** For any snake S and any τ -tile v , $|\text{ENC}_S(v)| \leq 4\tau + 2|S \cap v| \leq 2\tau^2 + 4\tau$ and $\sum_{v \in T_S^\tau} |\text{ENC}_S(v)| \in O(n)$.

Proof. The strings $\alpha(v)$ and $\gamma(v)$ contain together fewer than 4τ bits. The strings $\beta(v)$ and $\epsilon(v)$ contain fewer than four times the number of points in $S \cap v$. It immediately follows that $\text{ENC}_S(v)$ has fewer than $\tau + 2|S \cap v| \leq 2\tau^2 + 4\tau$ bits. By Lemma 3, there are at most $O(n/\tau)$ non-empty τ -tiles and thus $\sum_{v \in T_S^\tau} |\text{ENC}_S(v)| \in O(n)$. ◀

► **Lemma 8.** For any pair of snakes S, S' and any pair of τ -tiles v, v' if $\text{ENC}_S(v) = \text{ENC}_{S'}(v')$ then $S \cap v \simeq S' \cap v'$.

Lemma 8 allows us to define the *inverse* of an encoding:

► **Definition 9.** For any string s of at most $4(\tau^2 + \tau)$ bits, we denote by $\text{ENC}^{-1}(s)$ the unique marked grid X such that for all snakes S and τ -tiles v with $\text{ENC}_S(v) = s$, $X \simeq S \cap v$. $\text{ENC}^{-1}(s)$ is NULL whenever there exist no snakes S and τ -tiles v with $\text{ENC}_S(v) = s$.

3.2 Constructing Tables

Given our encoding scheme, we define tables M_{coll} , M_{head} , and M_{tail} to check for collisions, update the head, and update the tail in constant time:

- Each entry in M_{coll} corresponds to an encoding string s and a position p and stores a single bit that indicates whether or there is a collision with p and $\text{ENC}^{-1}(s)$.
- Each entry in M_{head} corresponds to an encoding string s and a position p adjacent to the head of s and stores the encoding string s' resulting from moving the head to position p .
- Each entry in M_{tail} corresponds to an encoding string s and stores the encoding string s' resulting from reduce the tail.

If an entry to a table is not a valid input, i.e., the encoding string e does not encode a valid snake or the position p in M_{head} is not adjacent to the head of the snake, the entry stores an arbitrary value. We use the tables to simulate the **extend** and **reduce** operations in constant time. Initially, when we first enter a tile v , we fetch the corresponding initial encoding from the table and store it in the search tree for v . All subsequent new encoding strings for v are obtained from the above tables, and hence, it inductively follows that we only access entries corresponding to valid input in the tables.

The space for the tables is dominated by the space M_{head} that has $2^{4(\tau^2+\tau)+2\log\tau}$ entries each storing $O(\tau^2)$ bits. Thus, in total we use $O(\tau^2 \cdot 2^{4(\tau^2+\tau)}) = O((\log\log n)^2 \cdot 2^{4(\log\log n)^2 + \log\log n}) = O(n)$ bits of space. We can compute an entry in $O(\tau^2) = O((\log\log n)^2)$ time, and hence construct the tables in $O((\log\log n)^2 2^{4(\log\log n)^2 + \log\log n}) = O(n)$ time. We use a standard global rebuilding with deamortization to construct the tables for exponentially increasing values of n as the length of the snake changes. I.e., we assume that $n \in [\frac{1}{2}N, 2N]$ and set $\tau = \log\log N$. Since we can construct the tables in linear time, the overhead of rebuilding our tabulation before n leaves $[\frac{1}{2}N, 2N]$ is constant (see also Section 4.3 for a detailed description of the deamortization).

4 Fast Snake Using Two Levels

We now describe our two-level data structure.

4.1 Data Structure

Let S be a snake of length n . For now, we assume that $n \in [\frac{1}{2}N, 2N]$ for some given N (we show how to lift this assumption later). Let $\tau_1 = \log N$ and $\tau_2 = \log\log N$. We construct tilings $T_1 = T_S^{\tau_1}$ and $T_2 = T_S^{\tau_2}$ for G . Our data structure stores the following components for each T_i where $i \in \{1, 2\}$.

- (a) A *direction string* D of length $n - 1$ that stores for each position in S , except the head h , the direction of its successor. It also stores the *relative coordinates* of h and t in their τ_i -tiles.
- (b) A *head and tail scope*. The head scope stores a pointer to the τ_i -tile $v \in T$ that contains h , the relative coordinates of h in v , and the up to eight non-empty τ_i -tiles incident to v . The tail scope stores a pointer to each of the τ_i -tiles intersected by the last $\tau_i/2$ points of S , the coordinates of the tail t , and the coordinates of the point $t_{\tau_i} = s_{\tau_i}$. If $n < \tau_i$ then $t_{\tau_i} = h$.
- (c) An *insert and delete* buffer. The insert buffer stores up to the last four τ_i -tiles that were empty before the head entered them. The delete buffer stores up to the last four τ_i -tiles that became empty after the tail left them.

- (d) If $i = 1$, we store for all $v_j \in T_1$ a counter recording $k_j = |S \cap N(v_j)|$ where $N(v_j)$ is the tile v_j plus the at most four tiles immediately above, left, right or below v_j .
If $i = 2$, we store for all $w_l \in T_2$ a counter recording $k'_l = |S \cap w_l|$.
- (e) If $i = 1$ we store one balanced binary tree B on all τ_1 tiles in T_1 . If $i = 2$ we maintain for each $v \in T_1$ a balanced binary search tree B_v storing $v \cap T_2$ in lexicographical order.
- (f) If $i = 1$, we maintain for each $v \in T_1$ a pointer to the set $N(v)$: the tile v plus the at most four non-empty τ_1 -tiles immediately above, left, right or below v .
- (g) If $i = 2$, we store for each tile $v \in T_2$ the tabulation code $\text{ENC}_S(v)$.

4.2 Operations

We implement the extend operation as in Section 2.2. Instead of querying and updating an explicit bit string for a tile v on the lowest level we use the corresponding tables and $\text{ENC}_S(v)$. Note that we can query, update, and replace $\text{ENC}_S(v)$ in constant time using our tabulation. Since the scope at level 2 is fully included in the scope of level 1, we can maintain the scope at level 2 in constant time using the same technique as in Section 2.2. Similarly, we can implement the reduce operation in constant time.

4.3 Lifting the Assumption

Since our data structure has $O(1)$ update time, we may trivially lift the assumption that $n \in [\frac{1}{2}N, 2N]$ using standard deamortization techniques. Suppose that n starts out as N . If $n = \frac{3}{2}N$, we make a copy S' of the current S and start building a second copy of our data structure on S' with $N' = \frac{3}{2}N$. Whilst n remains greater than $\frac{3}{2}N$ but smaller than $2N$, we perform our updates on S as regular. In the meantime, we do eight extend operations per operation on S per update on S . Each update instruction on S gets additionally recorded in a queue. If we have our data structure on S' , we perform eight updates in the queue on S' per operation on S . This way, when $|S| = 2N$, it must be that $S' = S$. We make the copy of our data structure our primary data structure (setting $N \leftarrow N'$). By doing a symmetric procedure for when $|S| < \frac{2}{3}N$, we may always assume that $|S| \in [\frac{1}{2}N, 2N]$.

5 Achieving $O(n + \log u)$ Bits

Naively, pointers take up a word and thus use $O(\log u)$ bits. Even with more clever pointer management dependent on the input size, a collection of pointers that point to n arbitrary objects in memory require $\Omega(\log n)$ bits per pointer for $O(n \log n)$ bits in total. We show, through clever pointer management, that our data structure can be implemented using $\Theta(n + \log u)$ bits instead. This is asymptotically tight, since the input size is n , and all data requires at least one word.

Storing components (a),(b), and (c). For both T_1 and T_2 , we store components (a), (b), and (c) in an arbitrary contiguous set of memory. The direction string requires $O(n)$ bits. We require $O(1)$ pointers to this string (specifying its start, end, and the location required by the tail buffer) which take $O(\log u)$ bits. Components (b) and (c) each have constant size, storing these objects plus a pointer to their location thus requires $O(\log u)$ bits and so these components can trivially be stored using $O(n + \log u)$ bits.

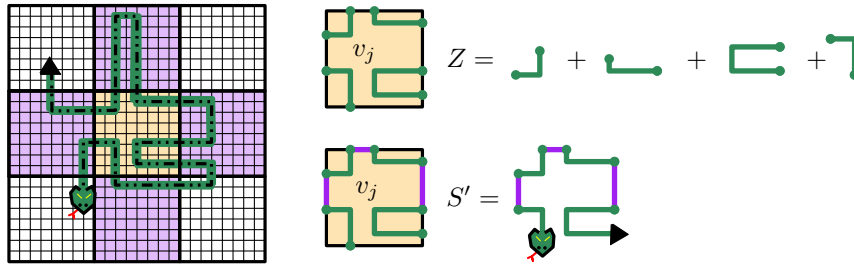
Storing components (d),(e), (f) and (g). We store components (d),(e), (f) and (g) through a two-level definition: where we first store the data structure of T_1 and then the data structure on T_2 within it. Our approach is dependent on the following geometric lemma:

► **Lemma 10.** *For any snake S , for any $v_j \in T_1$, the number of τ_2 -tiles in v_j is at most $4k_j/\tau_2$.*

Proof. We illustrate the proof by Figure 2. Recall that S uniquely corresponds to a rectilinear curve and consider the set Z of maximal subcurves of S in $S \cap v_j$. We can construct a snake S' by connecting all curves Z with a path along the boundary, so that S' intersects a minimal number of grid points. Note that the number of points $|S \cap v_j|$ is at least the number of points in S' since S is a connected curve. We may now apply Lemma 3 to note that:

$$|T_2 \cap v_j| \leq 4|S'|/\tau_2 \leq 4|S \cap N(v_j)|/\tau_2 = 4k_j/\tau_2 \quad \blacktriangleleft$$

► **Corollary 11.** *We may store for each v_j , for all $w_l \in T_2 \cap v_j$ a constant number of values of $O(\tau_2) = O(\log \log N)$ bits each, using at most $O(k_j) = O(|S \cap v_j|)$ bits.*



■ **Figure 2** Left we show a snake S intersecting a τ_1 -tile v_j in orange. We show $N(v)$ as the orange and purple τ_1 -tiles. The set Z are all maximally connected curves in $S \cap v_j$. We construct a snake S' by connecting Z using a minimal number of points.

5.1 Technical overview

Before we state our argument, we first provide a technical overview.

First, we show a static algorithm to store a snake S of size N in $O(N)$ space. We view memory as a contiguous interval of $1000N$ bits denoted by $[1, 1000N]$. We assume that $N > 1000$ (else, we have constant input size and may deploy a trivial polynomial dynamic solution). A consequence of this assumption is that any pointer to a location in $[1, 1000N]$ uses $2 \log N = 2\tau_1$ bits.

Each $v \in T_1$ requires two types of memory as it wants to store:

1. Constantly many pointers and counters of $O(\log N)$ bits each,
2. The set $T_2 \cap v$, which we store using $O(|S \cap N(v)|)$ bits.

By Lemma 3, we have at most N/τ tiles in T_1 and so storing these pointers takes $O(N)$ bits of space. However, storing $T_2 \cap v$ using $O(|S \cap N(v)|)$ bits is considerably more difficult:

Each $w \in T_2 \cap v$ requires two types of memory as it wants to store:

1. Constantly many counters, pointers, and the strings $\alpha_S(w), \gamma_S(w)$ of component (g).
2. The strings $\beta_S(w), \epsilon_S(w)$ of component (g) which have $O(|S \cap w|)$ bits.

If we manage to store $T_2 \cap v$ in at most $O(\log^2 N)$ bits of contiguous memory, pointers to locations within that memory require at most $O(\log \log N)$ bits. We can restrict our counters to have size $O(\log \log N)$ and the strings $\alpha_S(w), \gamma_S(w)$ have $O(\log \log N)$ bits each. It follows

by Corollary 11 that, if we can store $T_2 \cap v$ in a contiguous interval in memory, we can store $w \in T_2 \cap v$ using $O(|S \cap N(v)|)$ bits. Statically, guaranteeing that we store data in contiguous intervals is trivial and this leads to a static algorithm to store our data structure using $O(N)$ bits and time.

From static to dynamic. To dynamically maintain our data structure we use our deamortization rebuilding technique. We execute $\frac{N}{200}$ updates without ever deallocating memory. After $\frac{N}{100}$ updates, we apply the deamortization technique. We set $N' \leftarrow |S|$ and run our static algorithm at 200 operations per update to store our data structure using $O(N')$ bits, at which point we release our previous memory. This way, our analysis only has to show that during $\frac{N}{100}$ extend operations, our data structure still fits within $1000N$ bits. The reduce operation is thus for free. To illustrate this fact, consider for example $\frac{N}{200}$ consecutive reduce updates. Afterwards, $N' = N - \frac{N}{200}$ and so our rebuilt data structure will be considerably smaller.

To make sure that during the first $\frac{N}{100}$ extend operations all data structure components stay within their allotted memory, we recursively apply the deamortization technique. However, to apply this technique we must be very careful: Each tile has two types of objects which require a different order of space. Indeed consider a tile $v \in T_1$ where $O(|S \cap N(v)|)$ is constant. The tile v still requires pointers of size $O(\log N)$ each. If we then double $|S \cap N(v)|$ using $O(1)$ extend operations, we have neither the time nor the space to make a copy of these pointers. So, we split our data into these two types and store (and double) them separately. Similarly, the data corresponding to each $w \in T_2$ must be stored and treated by two separate categories, as they grow at different rates.

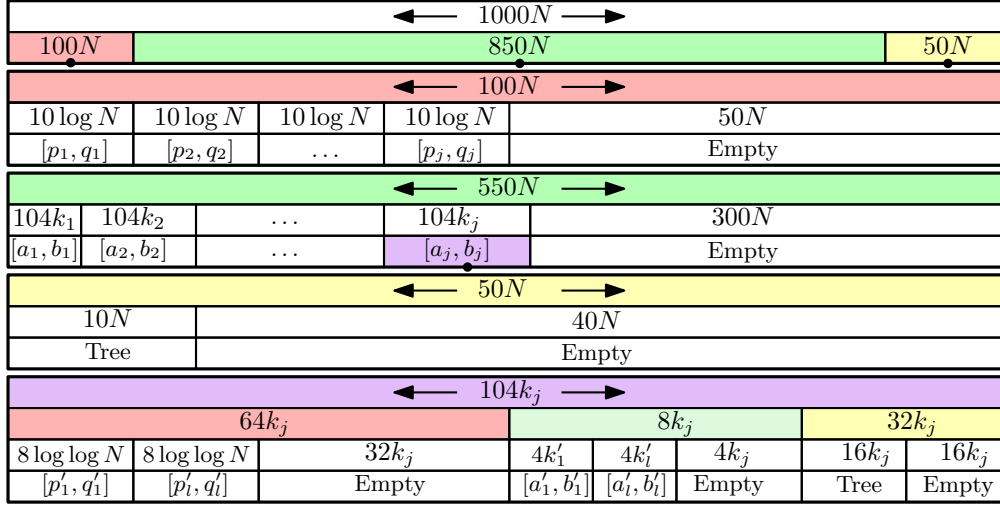
Our solution is to partition our memory into three types, indicated by the color red, green and yellow (see Figure 3). Red memory contains all pointers and counters. We only allocate red memory whenever tiles get added to either T_1 or T_2 which requires $\Omega(N/\log N)$ and $\Omega(N/\log \log N)$ extend operations, respectively. Green memory contains data (either the set $T_2 \cap v$ or the strings $\beta_S(w) + \epsilon_S(w)$). We allocate more green memory each extend operation, and must thus carefully to only allocate $O(1)$ data per update. Yellow intervals are search trees that for any tile $v \in T_1$ or $w \in T_2$ can return two pointers: indicating the location of its data in the red and green interval. These pointers again take up non-constant space; and are only allocated during tile inserts.

5.2 A static algorithm to allocate space

Given are $N = |S|$, $\tau_1 = \log N$, $\tau_2 = \log \log N$ and the set $T_1 = (v_1, v_2, \dots)$ (sorted lexicographical order by their top-left coordinate) where each $v_j \in T_1$ stores:

- A pointer to each τ_1 -tile in the set $N(v_j)$.
- An integer counter k_j recording $|S \cap N(v_j)|$,
- The set of τ_2 -tiles $v_j \cap T_2$ in lexicographical order where each $w_l \in T_2$ stores:
 - the integer $k_l = |S \cap w_l|$ and the string $\text{ENC}_S(w_l)$.

Space allocation. We allocate a contiguous set of $1000N$ bits in memory. For brevity, we consider memory as an interval in \mathbb{R}^1 and thus our memory is $[1, 1000N]$. For any integer a and interval $[b, c]$ we denote by $a + [b, c]$ the interval $[a + b, a + c]$. We assume that $N > 1000$. Therefore, pointers that point within $[1, 1000N]$ have a size of at most $2 \log N$ bits.



■ **Figure 3** An illustration for how we allocate memory.

We define our memory allocation and then prove that our data structure fits. We store:

1. In $[1, 100N]$ for all $v_j \in T_1$ an interval $[p_j, q_j]$. Specifically, $[p_1, q_1] = [1, 10 \log N]$ and $[p_j, q_j] = q_{j-1} + [1, 10 \log N]$. $[50N, 100N]$ remains empty. Each $[p_j, q_j]$ stores:
 - Components (d) and (f) of $v_j \in T_1$.
2. In $[100N, 950N]$ for all $v_j \in T_1$ an interval $[a_j, b_j]$. Specifically, $[a_1, b_1] = 100N + [1, 104k_1]$ and $[a_j, b_j] = b_{j-1} + [1, 104k_j]$. The interval $[650N, 950N]$ remains empty. $[a_j, b_j]$ stores:
 - i. In $a_j + [1k_j, 32k_j]$ for all $w_l \in T_2 \cap v_j$ an interval $[p'_l, q'_l]$. $[p'_1, q'_1] = a_j + [1, 8 \log \log N]$ and $[p'_l, q'_l] = q'_{l-1} + [1, 8 \log \log N]$. Each interval $[p'_l, q'_l]$ stores:
 - Component (d) of $w_l \in T_2 \cap v_j$ and from Component (g) the strings $\alpha_S(w_l) + \gamma_S(w_l)$. In $a_j + [32k_j, 64k_j]$ we keep empty space (for now).
 - ii. In $a_j + [64k_j, 68k_j]$ for all $w_l \in T_2 \cap v_j$ an interval $[a'_l, b'_l]$ of size $4k'_l$. Specifically, $[a'_1, b'_1] = a_j + 64k_j + [1, 4k_j]$ and $[a'_l, b'_l] = b'_{l-1} + [1, 4k_j]$. Each interval $[a'_l, b'_l]$ stores:
 - From component (g) the strings $\beta_S(w_l) + \epsilon_S(w_l)$ plus $2k'_l$ bits of empty space. In $a_j + [68k_j, 72k_j]$ we store empty space (for now).
 - iii. In $a_j + [72k_j, 88k_j]$ component (e): a balanced binary tree B_{v_j} over $T_1 \cap v_j$.
 - Each node storing w_l also stores a pointer to the location of p'_l and a'_l . In $a_j + [88k_j, 104k_j]$ we store empty space (for now).
3. In $[950N, 100N]$ component (e): a balanced binary tree B over T_1 . Each node storing v_j also stores pointers to the locations of p_j and a_j . $[450N, 500N]$ remains empty.

► **Lemma 12.** *Components (d), (e), (f) and (g) fit within their allocated memory. Moreover, given our input as specified we can allocate our data structure in $O(N)$ time.*

Proof. We prove the statement in order:

- We allocate intervals $[p_j, q_j]$ for $v_j \in T_1$ of $10 \log N$ bits. By Lemma 3, there are at most $4\lceil N/\tau_1 \rceil = 4\lceil N/\log N \rceil$ τ_1 -tiles v_j in $T_S^{T_1}$. So, $\sum_{v_j \in T_1} 4 \log N \leq 40N$ and $\bigcup_j [p_j, q_j] \subset [1, 40N]$.
- Each interval $[p_j, q_j]$ stores components (d) and (f) of v_j . The counter k_j is at most $\log((\tau_1)^2) \leq 2 \log N$ bits and four pointers in $[0, 1000N]$ use fewer than $8 \log N$ bits.
- We allocate intervals $[a_j, b_j]$ of width $104k_j$. $\forall s_i \in S$, there are at most 5 τ_1 -tiles with $s_i \in N(v_j)$. Thus, $\sum_j 104k_j = \sum_j 104|S \cap N(v_j)| \leq 5 \cdot \sum_{s_i \in S} 104 \leq 520N$. For each $[a_j, b_j]$:

- i. We allocate intervals $[p'_l, q'_l]$ for $w_l \in T_2 \cap v_j$ of $8 \log \log N$ bits. By Lemma 10, there are at most $4k_j/\tau_2 = 4k_j/\log \log N$ τ_2 -tiles in $T_2 \cap v_j$. So, $\sum_{w_l \in T_2 \cap v_j} 8 \log \log N \leq 32k_j$.

Thus $\bigcup [p'_l, q'_l]$ has a width of at most $32k_j$ and an interval of width $32k_j$ remains empty.

- Each interval $[p'_l, q'_l]$ stores component (d) of w'_l and of component (g) $\alpha(w_l) + \gamma(w_l)$. The counter k'_l is at most $\log((\tau_2)^2) \leq 2 \log \log N$ bits. By Lemma 7, these strings use $4 \log \log N$ bits. So the total takes $6 \log \log N < 8 \log \log N$ bits.

- ii. We allocate $[a'_l, b'_l]$ for $w_l \in T_2 \cap v_j$ of $4k'_l$ bits. $\sum_{w_l \in T_2 \cap v_j} 4k'_l = 4 \sum_{w_l \in T_2 \cap v_j} |S \cap w| \leq 4k_j$.

Thus, we may leave an interval of $4k_j$ bits empty.

- By the proof of Lemma 7, the strings $\beta_S(w_l)$ and $\epsilon_S(w_l)$ of $\text{ENC}_S(w_l)$ use $2k'_l$ bits and so $[a'_l, b'_l]$ has $2k'_l$ empty bits remaining.

- iii. By Lemma 10, $|T_2 \cap v_j| \leq 4k_j/\log \log N$. We store component (e), a balanced binary tree B_{v_j} where each node representing w_l stores a pointer to the location of p'_l and a'_l of at most $2 \log \log N$ bits. Moreover, $|T_2 \cap v_j| \leq (\log N)^2$. Thus, we may construct a balanced binary tree on $T_2 \cap v_j$ using $16k_j$ bits in total, leaving $16k_j$ bits empty.

By Lemma 3, there are at most $4\lceil N/\tau_1 \rceil = 4\lceil N/\log N \rceil$ τ_1 -tiles in T_1 . We store component (e) as a balanced binary tree where nodes store two pointers of at most $2 \log N$ bits. Thus, the balanced binary tree fits within an interval of width $10N$.

Since the input is already sorted, the whole allocation can trivially be done in $O(N)$ time. ◀

5.3 Dynamically maintaining our space allocation

We dynamically maintain our space allocation as follows. For $\frac{N}{100}$ updates, we never deallocate any space. Thus, the `reduce` operation has no effect on our storage. For any `extend` operation during this time, we allocate additional space: filling up the pre-allocated empty space in our memory. Whenever a pre-allocated empty interval is half full, we allocate a new interval of twice the size and start copying the data into the new interval. We prove that this way, we never store data in an interval that is empty. We first introduce some additional concepts:

- Denote for each τ_1 -tile $v_j \in T_1$ a counter $K_j^* = [a_j, b_j]$.
 - There are two ways this counter can be stored. Firstly this counter never exceeds $2 \log N$ bits and store it alongside k_j in $[p_i, q_j]$ by increasing our space by a small constant factor. If the reader is hesitant towards increasing space any further, this counter may always be computed from $[a_j, b_j]$ in $O(1)$ time.

We describe the compounded effect of `extend` and `reduce` operations through five *events*:

1. In the `MINI-SPAWN` event, we add a τ_2 -tile w_l to T_2 with $w_l \subset v_j \in T_1$.
 - Denote by $[p, q]$ and $[a, b]$ the empty set of bits in $a_j + [16k_j, 32k_j]$ and $a_j + [64k_j, 72k_j]$. We allocate $[p'_l, q'_l] = q + [1, 8 \log \log N]$ and $[a'_l, b'_l] = b + [1, 4k_j]$ in $[a, b]$ (shrinking $[p, q]$ and $[a, b]$ accordingly). Then we insert w_l into B_{v_j} .
2. In the `SPAWN` event, we add a τ_1 -tile v_j to T_1 .
 - Denote by $[p, q]$ the remaining space in $[1, 100N]$. We allocate $[p_j, q_j] = q + [1, 10 \log N]$ and shrink $[p, q]$ accordingly. Similarly let $[a, b]$ be the remaining space in $[650N, 950N]$. We allocate $[a_j, b_j] = b + [1, 104k_j]$ and shrink $[a, b]$ accordingly. Finally, we insert v_j into the balanced binary tree B , allocating $4 \log \log N$ bits in $[950N, 1000N]$.
3. In the `MINI-DOUBLE` event for $w_l \in T_2$, $[a'_l, b'_l]$ has three quarters of its bits allocated.
 - Conceptually, we set $[x'_l, y'_l] \leftarrow [a'_l, b'_l]$.
 - Let $[a, b]$ be the empty space in a space in $a_j + [68k_j, 72k_j]$. We allocate a new interval $[a'_l, b'_l] = a + [1, 2 \cdot |[x'_l, y'_l]|]$ to w_l and reduce $[a, b]$ accordingly.
 - We refer to $[x'_l, y'_l]$ as the *shadow* of $[a'_l, b'_l]$.

4. In the DOUBLE event for $v_j \in T_1$, the empty half of the red, green or yellow interval in $[a_j, b_j]$ has at least half of its bits allocated. I.e., either $a_j + [32K_j^*, 64K_j^*]$, or $a_j + [68K_j^*, 72K_j^*]$, or $a_j + [88K_j^*, 104K_j^*]$ has at least half of its bits allocated.
 - Conceptually, we set $[x_j, y_j] \leftarrow [a_j, b_j]$.
 - Let $[a, b]$ be the empty space in $[550N, 950N]$. We allocate a new interval $[a_j, b_j] = b + [1, 104k_j]$ to v_j and reduce $[a, b]$ accordingly.
 - We refer to $[x_j, y_j]$ as the *shadow* of $[a_j, b_j]$ and set $K_j^* \leftarrow 2K_j^*$.
5. In the MEGA-DOUBLE-event, the empty half of the red, green or yellow interval in $[1, 1000N]$ has at least half of its bits allocation. I.e., one of the tree intervals: $[50N, 100N]$, $[650N, 950N]$ or $[984N, 1000N]$ has at least half of its bits allocated.

► **Lemma 13.** *For any $w_l \in T_2$, it takes at least $\lceil |a'_l, b'_l| \rceil / 2$ extend operations (where the head is in w_l) before we trigger a MINI-DOUBLE event for w_l .*

Proof. By Lemma 7, the length of the substrings $\beta_S(w_l) + \epsilon_S(w_l)$ of $\text{ENC}_S(w_l)$ have at most $2k'_l$ bits. When $[a'_l, b'_l]$ was created let $|S \cap w_l| = X$. Per definition, the width of $[a'_l, b'_l]$ was $4X$ with $2X$ bits remaining empty and so the lemma follows. ◀

► **Lemma 14.** *For any $v_j \in T_2$, it takes at least $K_j^* / 4$ extend operations (with the head in $N(v_j)$) before we trigger a DOUBLE event for v_j .*

Proof. We do a case distinction on what triggered the event:

- Suppose that $a_j + [32K_j^*, 64K_j^*]$ has at least half of its bits allocated (this is the empty space of the red interval). We only allocate memory in these intervals during a MINI-SPAWN event in v_j . At a MINI-SPAWN event, we allocate $8 \log \log N$ bits in the red interval. Thus, by the time that we have filled $16K_j^*$ bits, we must have triggered $2K_j^* / \log \log N$ MINI-SPAWN events. By Lemma 3, it takes at least $\log \log N$ extend operations in $N(v_j)$ to trigger four MINI-SPAWN events in v_j and so the lemma follows.
- Suppose that $a_j + [88K_j^*, 104K_j^*]$ has at least half its bits allocated (this is the empty space of the yellow interval). We only allocate memory in these intervals during a MINI-SPAWN event. At a MINI-SPAWN event, we allocate $4 \log \log N$ bits. Thus, by the time that we have filled $8K_j^*$ bits we must have triggered $K_j^* / \log \log N$ MINI-SPAWN events. By Lemma 3, it takes at least $\log \log N$ extend operations in $N(v_j)$ to trigger four MINI-SPAWN events in v_j and so the lemma follows.
- Suppose that $a_j + [68K_j^*, 72K_j^*]$ has at least half of its bits allocated (this is the empty space of the green interval). If this memory was (largely) allocated through MINI-SPAWN events, then since each such event allocates $O(1)$ bits in this range and requires at least one extend operation. Thus, the lemma trivially follows. So suppose that this memory was allocated through MINI-DOUBLE events instead.

For any $w_l \in T_2 \cap v_j$ denote $[a'_l(1), b'_l(1)] = [a'_l, b'_l]$ whenever $[a'_l, b'_l] \subset a_j + [68K_j^*, 72K_j^*]$. I.e., consider the current interval associated to w_l if it is stored in the empty half of the green interval. We recursively define the intervals $[a'_l(t), b'_l(t)]$ as the *shadow* of $[a'_l(t-1), b'_l(t-1)]$. We note two facts: Firstly, our assumption implies that the intervals $[a'_l(t), b'_l(t)]$ use at least $2K_j^*$ bits in $a_j + [68K_j^*, 72K_j^*]$. Secondly, for all l , for all t , $\lceil |a'_l(t), b'_l(t)| \rceil = \frac{1}{2} \lceil |a'_l(t-1), b'_l(t-1)| \rceil$. It follows from the geometric series that:

$$\sum_{w_l \in T_2 \cap v_j} \sum_t \lceil |a'_l(t), b'_l(t)| \rceil \geq 2K_j^* \Rightarrow \sum_{w_l \in T_2 \cap v_j} \lceil |a'_l(1), b'_l(1)| \rceil \geq K_j^*.$$

The interval $[a'_l(1), b'_l(1)]$ can only have been created through a MINI-DOUBLE event on $[a'_l(2), b'_l(2)]$. By Lemma 13, it takes $\frac{1}{2} \lceil |a'_l(2), b'_l(2)| \rceil \geq \frac{1}{4} \lceil |a'_l(1), b'_l(1)| \rceil$ extend operations in w_l before this event is triggered and so the lemma follows. ◀

► **Lemma 15.** *It takes at least $N/100$ extend operations to trigger a MEGA-DOUBLE event.*

Proof. The proof is analogous to Lemma 14 as a case distinction on the event trigger:

- Suppose that $[50N, 100N]$ has at least half of its bits allocated (this is the empty space of the red interval). We only allocate memory in these intervals during a SPAWN event. At a SPAWN event, we allocate $10 \log \log N$ bits in the red interval. Before we filled $25N$ bits, we must have triggered $2N/\log \log N$ SPAWN events. By Lemma 3, it takes at least $\log \log N$ extend operations to trigger four SPAWN events and so the lemma follows.
- Suppose that $[984N, 1000N]$ has at least half its bits allocated (this is the empty space of the yellow interval). We only allocate memory in these intervals during a SPAWN event. At a SPAWN event, we allocate $4 \log \log N$ bits. Thus, by the time that we have filled $40N$ bits we must have triggered $10N/\log \log N$ SPAWN events. By Lemma 3, it takes at least $\log \log N$ extend operations to trigger four SPAWN events and so the lemma follows.
- Suppose that $[650N, 950N]$ has at least half of its bits allocated (this is the empty space of the green interval). The space required by $\frac{1}{10}N$ SPAWN events may be charged to all the other space in memory (since each SPAWN event allocates memory proportional to how much memory is already in space). Thus, we may suppose that this memory was allocated through DOUBLE events.

For any $v_j \in T_1$ denote $[a_j(1), b_j(1)] = [a_j, b_j]$ whenever $[a_j, b_j] \subset [650N, 950N]$. I.e., consider the current interval associated to v_j if it is stored in the empty half of the green interval. We recursively define $[a_j(t), b_j(t)]$ as the *shadow* of $[a_j(t-1), b_j(t-1)]$.

We note two facts: Firstly, our assumption implies that the intervals $[a_j(t), b_j(t)]$ use at least $150N$ bits in $[650N, 950N]$. Secondly, for all l , for all t , $|[a_j(t), b_j(t)]| = \frac{1}{2} |[a_j(t-1), b_j(t-1)]|$. It follows from the geometric series that:

$$\sum_{v_j \in T_1} \sum_t |[a_j(t), b_j(t)]| \geq 150N \Rightarrow \sum_{v_j \in T_1} |[a_j(1), b_j(1)]| \geq 75N.$$

$[a_j(1), b_j(1)]$ can only have been created through a DOUBLE event on $[a_j(2), b_j(2)]$. By Lemma 14, it takes $\frac{1}{8}K_j^* \geq \frac{1}{8 \cdot 104} |[a_j(1), b_j(1)]|$ extend operations in $N(v_j)$ before this event is triggered.¹ Each extend operation occurs in $N(v_j)$ for at most five tiles in T_1 . This implies that at least $\frac{75}{8 \cdot 104 \cdot 5} \geq \frac{N}{100}$ extend operations must have occurred. ◀

5.3.1 Proving our main theorem

We are now ready to prove:

► **Theorem 1.** *We can represent a snake of length n in $O(n + \log u)$ bits and support extend and reduce operations in constant time.*

Proof. We assume that $n \in [0.5N, 2N]$ and that we have built our data structure with $\tau_1 = \log N$ and $\tau_2 = \log \log N$. On a macro-level, we use the standard rebuilding deamortization technique. For the first $\frac{N}{200}$ updates, we update our data structure in $O(1)$ worst-case time using our update strategy of Section 4.2; allocating space until we trigger a MINI-DOUBLE or DOUBLE event.

Whenever we trigger a MINI-DOUBLE event for a tile $w_l \in T_2$, we reallocate the new memory $[a'_l, b'_l]$ in $O(1)$ time. Over the next $|[a'_l, b'_l]|/10$ updates that change $S \cap W_l$, we perform ten times the work to both execute the work in the shadow of $[a'_l, b'_l]$ and copy all

¹ Indeed, it took $\frac{1}{4}K_j^*$ extend operations to trigger a DOUBLE event, after which K_j^* was doubled. Moreover, per definition, $[a_j, b_j]$ is $104K_j^*$ bits wide.

info from its shadow into $[a'_i, b'_i]$ (whilst queueing updates not executed in $[a'_i, b'_i]$). When we have successfully copied the shadow of $[a'_i, b'_i]$ we dequeue updates at four times the pace we are queueing them so that our interval $[a'_i, b'_i]$ is up to date before its shadow is full.

Whenever we trigger a DOUBLE event for a tile $v_j \in T_1$, we reallocate the new memory $[a_i, b_i]$ in $O(1)$ time. Over the next $K_j^*/10$ updates that change $S \cap v$, we perform ten times the work to both execute the update in the shadow of $[a_j, b_j]$ and copy all info from its shadow into $[a_i, b_i]$ (whilst queueing updates not executed in $[a_j, b_j]$). When we have successfully copied the shadow of $[a_j, b_j]$ we dequeue updates at four times the pace we are queueing them so that our interval $[a_j, b_j]$ is up to date before its shadow is full.

This way, we maintain our data structure in $O(1)$ time for the first $\frac{N}{200}$ updates. By Lemma 15, we cannot trigger a MEGA-DOUBLE event in this time and thus everything fits well within memory. At this point, we set $N = n$ and we reallocate a new interval of $1000N$ bits. Over the next $\frac{N}{200}$ updates, we do four times the work: executing updates on our original interval whilst copying the info from our original interval into the new interval (queueing updates in the process). When we have successfully copied the original interval into the new interval, we dequeue the updates at four times the pace that we are queueing them so that the new copy is up to date before we reach $\frac{N}{200}$ updates in the first interval. Thus, we never trigger a MEGA-DOUBLE event and we always fit within $O(N)$ bits. ◀

References

- 1 Ali Jaber Almalki and Pawel Wocjan. Exploration of reinforcement learning to play Snake game. In *Proc. CSCI*, pages 377–381, 2019.
- 2 Arne Andersson and Mikkel Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3), 2007.
- 3 Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proc. 51st FOCS*, pages 787–796, 2010.
- 4 Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. All-purpose hashing. *arXiv preprint arXiv:2109.04548*, 2021.
- 5 Michael A Bender, Martín Farach-Colton, John Kuszmaul, and William Kuszmaul. Modern hashing made simple. In *Proc. 7th SOSA*, pages 363–373. SIAM, 2024.
- 6 Michael A Bender, Martín Farach-Colton, John Kuszmaul, William Kuszmaul, and Mingmou Liu. On the optimal time/space tradeoff for hash tables. In *STOC*, pages 1284–1297, 2022.
- 7 Ioana O Bercea and Guy Even. Dynamic dictionaries for multisets and counting filters with constant time operations. *Algorithmica*, 85(6):1786–1804, 2023.
- 8 Russell Sammut Bonnici, Chantelle Saliba, Giulia Elena Caligari, and Mark Bugeja. Exploring reinforcement learning: A case study applied to the popular Snake game. In *DTMAD*, 2022.
- 9 Marzio De Biasi and Tim Ophelders. The complexity of snake. In *FUN*, 2016.
- 10 Shubham Sharma, Saurabh Mishra, Nachiket Deodhar, Akshay Katageri, and Parth Sagar. Solving the classic Snake game using AI. In *Proc. PuneCon*, pages 1–4, 2019.
- 11 Wikipedia. URL: [https://en.wikipedia.org/wiki/Snake_\(video_game_genre\)](https://en.wikipedia.org/wiki/Snake_(video_game_genre)).
- 12 Wikipedia. URL: [https://en.wikipedia.org/wiki/Snake_\(video_game\)](https://en.wikipedia.org/wiki/Snake_(video_game)).

Uniform-Budget Solo Chess with Only Rooks or Only Knights Is Hard

Davide Bilò  

Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Luca Di Donato 

Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Luciano Gualà  

Department of Enterprise Engineering, University of Rome "Tor Vergata", Italy

Stefano Leucci  

Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Abstract

We study the Solo-Chess problem which has been introduced in [Aravind et al., FUN 2022]. This is a single-player variant of chess in which the player must clear all but one piece from the board via a sequence captures while ensuring that the number of captures performed by each piece does not exceed the piece's *budget*. The time complexity of finding a winning sequence of captures has already been pinpointed for several combination of piece types and initial budgets. We contribute to a better understanding of the computational landscape of Solo-Chess by closing two problems left open in [Aravind et al., FUN 2022]. Namely, we show that Solo-Chess is hard even when all pieces are restricted to be only rooks with budget exactly 2, or only knights with budget exactly 11.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Mathematics of computing → Combinatorics

Keywords and phrases solo chess, puzzle games, board games, NP-completeness

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.4

1 Introduction

Solo-Chess is a puzzle game available on `chess.com` [9]. The game is played by a single player on a 8×8 chessboard that initially contains an arrangement of chess pieces. All the pieces have the same color but they are otherwise allowed to capture each other following the standard capturing rules of chess. Each move performed by the player is required to be a capture and the goal is that of removing all but one piece from the board. Moreover, each piece has an associated budget that limits the number of captures it can make. More precisely, all the initial budgets are 2, and only pieces with positive budget are allowed to capture by spending one unit of their budget.¹

The problem of finding a winning sequence of captures has been first studied from the computational point of view in [1], where the authors generalize the chessboard to an arbitrary size and allow each piece to have an arbitrary non-negative initial budget. More precisely, given a set of piece types $P \subseteq \{\♙, ♚, ♜, ♝, ♞, ♟\}$ and a collection of allowed budgets $B \subseteq \mathbb{N}$, we denote by $\text{SOLO-CHESS}(P, B)$ the problem in which all pieces on the board have some type in P and an initial budget in B .²

¹ In the `chess.com` version of the game, there is at most one king on the chessboard and, if a king exists, it must be the last remaining piece.

² To lighten the notation, we sometimes write $\text{SOLO-CHESS}(t, B)$ instead of $\text{SOLO-CHESS}(\{t\}, B)$.



Aravind et al. [1] focus on instances containing pieces of a single type, and show that $\text{SOLO-CHESSES}(\text{♞}, \{0, 1, 2\})$, $\text{SOLO-CHESSES}(\text{♟}, \{0, 1, 2\})$, and $\text{SOLO-CHESSES}(\text{♚}, \{2\})$ are NP-hard, while $\text{SOLO-CHESSES}(\text{♜}, \{0, 1, 2\})$ and $\text{SOLO-CHESSES}(\{\text{♜}, \text{♞}, \text{♟}, \text{♚}, \text{♛}, \text{♙}\}, \{0, 1\})$ can be solved in polynomial time. They also consider the $\text{SOLO-CHESSES}(\text{♞}, \{0, 1, 2\})$ problem played 1D boards (i.e., boards with a single row/column) and provide a polynomial-time algorithm. Among others, [1] explicitly mentions the following two open problems:

- What is the computational complexity of $\text{SOLO-CHESSES}(\text{♞}, \{2\})$?
- What can be said about the complexity of Solo-Chess played by knights alone?

Our results. In this paper we answer these two questions by showing that both the above problems are NP-hard. In particular, Solo-Chess played by only knights remains NP-hard even when all knights have the same constant budget. Formally, we prove the NP-hardness of $\text{SOLO-CHESSES}(\text{♞}, \{2\})$ and $\text{SOLO-CHESSES}(\text{♞}, \{11\})$ by providing polynomial-time reductions from (suitable versions of) the vertex cover and Hamiltonian path problems.

An interactive demonstration of our reductions can be found at <https://www.isnphard.com/g/solo-chess/>.

Other related work. The Solo-Chess problem has also been studied in [4], where the special case in which the number of captures of each piece is unrestricted was considered, and it is shown that it is polynomial-time solvable whenever $|P| = 1$, while the problem becomes NP-hard for any choice of P with $|P| \geq 2$. The authors also consider the case in which exactly one of the pieces cannot be captured (and hence it must be the last piece of the board in any solution), and all other (capturable) pieces are of the same type (which might or might not coincide with type of the uncapturable piece). Almost all possible combinations of types are shown to be either NP-complete or polynomial-time solvable.

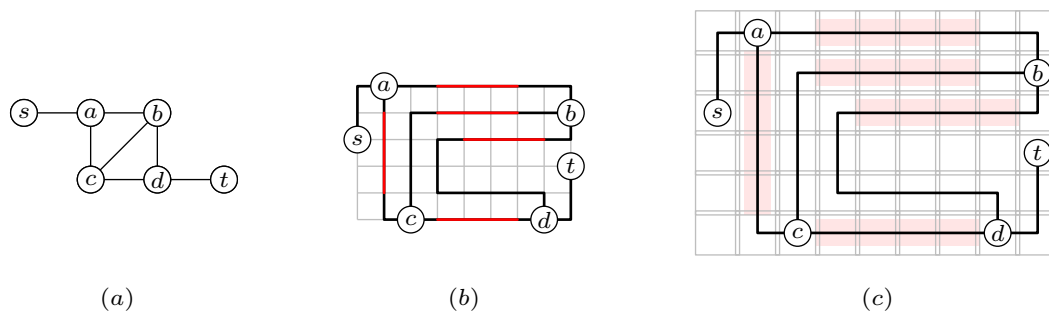
Aravind et al. [1], also study a variant of Solo-Chess that is played on a graph, where vertices represent pieces, and edges represent the available captures. Solo-Chess is also close in spirit to other problem that require capturing pieces in order to clear a board, such as peg solitaire and its variants [11, 8, 3].

Finally, we mention that the classical 2-player chess game is known to be EXPTIME-complete or PSPACE-complete depending on whether the number of allowed moves is upper-bounded by a polynomial [5, 10].

Structure of the paper and notation. The NP-Hardness of $\text{SOLO-CHESSES}(\text{♞}, \{11\})$ is discussed in Section 2, while $\text{SOLO-CHESSES}(\text{♞}, \{2\})$ is considered in Section 3. Throughout the paper, we use the notation $p \rightarrow p'$ to denote a move in which piece p captures piece p' . Sometimes, it will be more convenient to refer to the squares occupied by the pieces instead. If p and p' are on squares q and q' , respectively, all of the following will also denote move $p \rightarrow p'$: (i) $p \rightarrow q'$, (ii) $q \rightarrow p'$, and (iii) $q \rightarrow q'$. We also shorten a sequence of k consecutive captures of the form $p_1 \rightarrow p_2, p_2 \rightarrow p_3, \dots, p_{k-1} \rightarrow p_k$ by simply writing $p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow \dots p_{k-1} \rightarrow p_k$.

2 Solo-Chess with only knights

In this section we establish the NP-hardness of $\text{SOLO-CHESSES}(\text{♞}, \{11\})$. In order to do so, we first show that the more general $\text{SOLO-CHESSES}(\text{♞}, \{0, 2, 11\})$ problem is NP-hard, and then, in Section 2.3, we argue that knights with budgets 0 and 2 can be simulated by only using knights with budget 11. In the rest of this section, we refer to a knight with budget b as a b -knight.



■ **Figure 1** (a) An instance G of RHP. (b) The corresponding planar orthogonal grid drawing \mathcal{D} of G , where the segments $\ell_{u,v}$ are shown in red. (c) A sketch of the chessboard obtained from \mathcal{D} , where the tiles corresponding to the segments $\ell_{u,v}$ are highlighted with a red band along the direction of the segment.

Our reduction to $\text{SOLO-CHESSES}(\blacktriangle, \{0, 2, 11\})$ is from a restriction of the Hamiltonian path problem to a suitable class of input graphs, which we name RHP. The Hamiltonian path problem is a well-known NP-hard problem that asks to decide whether a given input graph $G = (V, E)$ contains a simple path that traverses all vertices in V . In RHP we restrict ourselves to instances in which G is planar, V contains exactly two vertices s, t with degree 1, and all other vertices have degree 3 (see Figure 1 (a)). We call s the *start vertex* and t the *end vertex* of G . Clearly, G contains a Hamiltonian path iff it contains a Hamiltonian path from its start to its end vertex.

Notice that the RHP problem is NP-hard. Indeed, [7] shows that the Hamiltonian cycle problem is NP-hard when the instances are restricted to planar cubic graphs (i.e., graphs in which every vertex has degree exactly three) by providing a reduction from the 3-SAT problem. A closer inspection of such a reduction shows that the resulting graphs G' contain some special edge $e = (u, v)$ such that if G' admits a Hamiltonian path P then P must traverse e .³ We can then obtain an instance G of RHP by deleting e from G' and replacing it with two new vertices s, t along with the edges (s, u) and (u, t) .

The high-level idea of our reduction is that of embedding the graph G on a chessboard. Since all vertices in $V \setminus \{s, t\}$ have degree 3, finding a Hamiltonian path in G can be thought of as the problem of deleting exactly one edge incident to each vertex of degree 3. To achieve this, each “edge” in our reduction will be equipped with a suitable edge deletion gadget. Once the selected edges have been deleted, the rest of the board encodes the sought Hamiltonian path, but it still contains some knights that need be cleared along the Hamiltonian path. This will be done by tracing the Hamiltonian path using a knight that is initially placed in s , while capturing all remaining knights along the way. Actually, in order to keep the budgets small, this traversal will not be performed by a single knight but rather by a collection of knights that use suitable gadgets as relay stations.

We now describe the technical details of our reduction to $\text{SOLO-CHESSES}(\blacktriangle, \{0, 2, 11\})$: we start by finding a planar orthogonal grid drawing \mathcal{D} of G , i.e., a mapping that associates each vertex $v \in V$ to a distinct point p_v having integer coordinates, and each edge $(u, v) \in E$ to a non self-intersecting polyline $\ell_{u,v}$ connecting p_u to p_v and consisting of the union of alternating (and non-empty) horizontal and vertical segments such that (i) all the segments' endpoints

³ Such an edge can be found in polynomial-time. In fact, the reduction of [7] even uses a special gadget for the exact purpose of forcing an edge to be in all Hamiltonian paths of G' .

are at integer coordinates, (ii) $\ell_{u,v}$ does not contain any point p_w with $w \in V \setminus \{u, v\}$, and (iii) the polylines of different edges do not intersect (except possibly at their endpoints). Suppose w.l.o.g. that the x -coordinates (resp. y -coordinates) used by the drawing range from 1 to w (resp. from 1 to h). A drawing \mathcal{D} with $w \cdot h = O(n^2)$ can be found in polynomial time w.r.t. $|V|$ (see [12] and the references in Section 5.3 of [2]). We further assume that each polyline $\ell_{u,v}$ between two distinct vertices $u, v \in V \setminus \{s, t\}$ contains at least one horizontal or vertical segment ς with a length of at least 5,⁴ and we choose a contiguous portion $\ell'_{u,v}$ of ς that has length 3, starts and ends at integer coordinates, and does not include the endpoints of ς (see Figure 1 (b)).

The chessboard of our instance of SOLO-CHESS(\blacktriangle , $\{0, 2, 11\}$) has size $(14h+1) \times (14w+1)$ and consists of $h \times w$ tiles, i.e., contiguous sub-chessboards of size 15×15 , such that each tile corresponds to a point at integer coordinates in \mathcal{D} and any two horizontally or vertically adjacent tiles share 15 squares along their common edge (see Figure 1 (c)). Each of these tiles is either *empty* or it hosts a (portion of) some *gadget*. Gadgets are arrangements of knights which span an integral number of connected tiles. We will make use of six *gadget types*, five of which span exactly one tile, while the remaining one spans 4 consecutive tiles (either horizontally or vertically). More precisely, we use a *start gadget* and an *end gadget* for the tiles corresponding to s and t , respectively; a *straight edge gadget* for each tile corresponding to a point that lies on some polyline $\ell_{u,v}$ but is neither in $\{p_u, p_v\}$ nor on $\ell'_{u,v}$ (if any); a *corner edge gadget* for each tile corresponding to an endpoint of a segment of some polyline $\ell_{u,v}$, except for the endpoints p_u, p_v of $\ell_{u,v}$ itself; and a *cubic vertex gadget* for each tile corresponding to a point p_v for $v \in V \setminus \{s, t\}$. The sixth and final gadget type is the *edge deletion gadget*. We use an edge deletion gadget for each edge $(u, v) \in E$ with $u, v \in V \setminus \{s, t\}$ and we place it on the four tiles corresponding to the points of integer coordinates in $\ell'_{u,v}$.

Our gadgets will interact with one another by sharing 0-knights placed on their perimeter. The squares hosting these knights are marked with a \times symbol in our figures and will be referred to as *input/output (I/O) squares*. An I/O square q of a gadget acts as an input if some $(b+1)$ -knight that is not in one of the gadget's squares captures the 0-knight originally placed on q . In this case, we say that the gadget *takes a b -knight as an input*. An I/O square q of a gadget acts as an output whenever some b -knight that is in one of the gadget's squares captures a knight on q . In such a case we say that the gadget *outputs a $(b-1)$ -knight*. The gadgets are designed to ensure that, in any winning sequence of moves, no I/O square can be the target of two distinct captures, and hence it cannot act as both an input and as an output (although it might play different roles in different winning sequences).

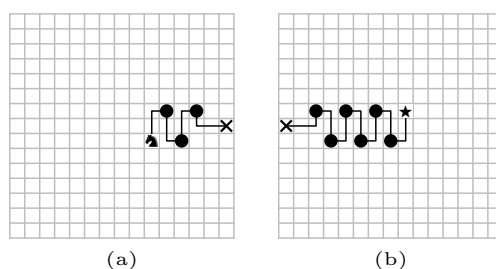
In the following, all squares marked with \bullet , \circ , \star , or \times contain 0-knights, the ones marked with \blacksquare contain 2-knights, and the ones marked with \blacktriangle contain 11-knights. We say that a b -knight is *lively* if $b \geq 6$ and *lazy* if $b \leq 5$. We now discuss our gadgets.

Start and end gadgets

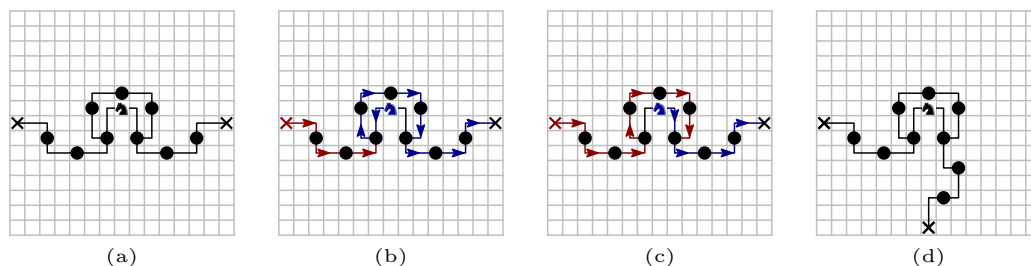
The start and end gadgets are shown in Figure 2 (a) and Figure 2 (b), respectively, and have a single I/O square each. The start gadget corresponds to vertex s and is meant to output a single 7-knight (and no b -knight with $b > 7$ can be output).

The end gadget corresponds to vertex t and is meant to be played at the end of any winning sequence. Since the knight initially at \star has budget 0 and can only be captured from exactly one of the \bullet squares, any winning sequence must necessarily place the last remaining knight on \star , which we name the *goal square*. It is possible to clear all but a single knight from the end gadget iff the gadgets takes a b -knight with $b \geq 7$ as input.

⁴ This can always be guaranteed, e.g., by “scaling up” the drawing by a factor of 5.



■ **Figure 2** (a) The arrangement of knights in the start gadget. (b) The arrangement of knights in the end gadget.



■ **Figure 3** (a) The arrangement of knights in a straight edge gadget. (b) A sequence of moves that allows the gadget to output a 3-knight from the right I/O square when a 3-knight is input in the left I/O square. Red moves are played before blue moves. (c) A sequence of moves that allows the gadget to output a 7-knight from the right I/O square when a 7-knight is input in the left I/O square. (d) The arrangement of knights in a corner edge gadget.

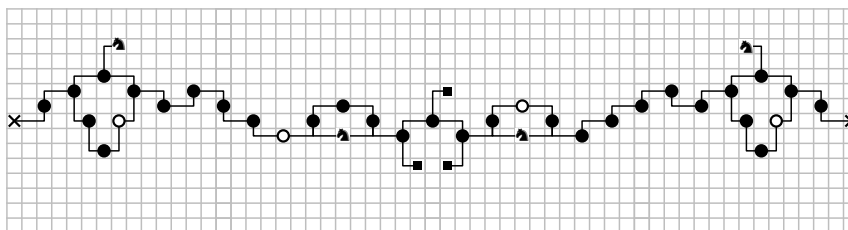
Straight edge gadgets

Each of these gadgets corresponds to a portion of either a horizontal or a vertical segment of some polyline $\ell_{u,v}$ connecting p_u to p_v in \mathcal{D} , as long as such portion does not lie in $\ell'_{u,v}$ (which is handled by the edge deletion gadget). In the following we discuss how knights are arranged in the case of a horizontal segment. The vertical case is obtained by rotating the discussed gadget by 90 degrees (either clockwise or counterclockwise).

The arrangement of knights in the gadget is shown in Figure 3 (a). If a b -knight with $b \geq 3$ is input in one of the I/O squares, then it is possible to output a 3-knight from the opposite I/O square (see Figure 3 (b)). Similarly, when a 7-knight is input in one of the I/O squares, it is possible to output a 7-knight from the opposite I/O square (see Figure 3 (c)) but it is not possible to output any b -knight with $b > 7$.

Moreover, if a lazy knight is input in one of the I/O squares, there exists no winning sequence of captures that allows the gadget to output a lively knight. Finally, it is impossible for any sequence of moves to use both I/O locations as outputs, or for any winning sequence of moves to use both I/O locations as inputs (since this would isolate some knight in the gadget from the goal square).

Essentially, this gadget allows to “teleport” either a 3-knight or a 7-knight from an I/O square to the opposite one, while clearing all but the latter square. By chaining together multiple straight edge gadgets it is possible to move a 3-knight or a 7-knight across any horizontal or vertical segment of a polyline.



■ **Figure 4** The arrangement of knights in an edge deletion gadget.

Corner edge gadgets

A corner edge gadget allows to connect an horizontal segment of a polyline to an adjacent vertical segment, or vice-versa, and is shown in Figure 3 (d). We only discuss one possible orientation of the gadget, as the others are obtained by a 90-, 180-, or 270-degree rotation. The gadget is almost identical to the straight edge gadget of Figure 3 (a), as the only differences are the positions of a square marked ●, and that of the rightmost I/O square which has been relocated to the bottom edge of the gadget. Neither of these changes affects the threat relationships between the pieces, hence our discussion of the straight edge gadgets also applies to corner edge gadgets.

By chaining together a combination of straight edge and corner edges gadgets it is possible to move a 3-knight or a 7-knight across any portion of a polyline.

Edge deletion gadgets

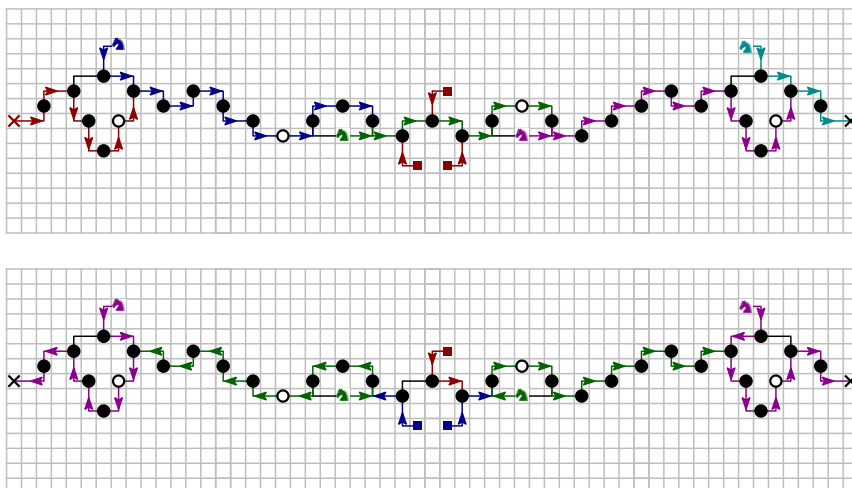
We use exactly one edge deletion gadget per edge $(u, v) \in E$ with $u, v \notin \{s, t\}$. Such a gadget spans 4 consecutive tiles either horizontally or vertically (i.e., a 15×57 or 57×15 sub-chessboard) and is placed in the tiles corresponding to the portion $\ell'_{u,v}$ of the polyline $\ell_{u,v}$.

As for the straight edge gadget, in the following we only discuss the horizontal version of the gadget, shown in Figure 4, since the vertical version can be obtained by a 90-degree rotation.

The gadget has two I/O squares on opposite sides and there are two intended ways to play the gadget, which are shown in Figure 5, and we name them *traversal mode* and *deletion mode*. In traversal mode a lively knight is input in one of the I/O squares and a 7-knight is output from the opposite I/O square. In deletion mode each of the two I/O squares outputs a 3-knight.

No winning sequence of moves can use both I/O squares as inputs (since this would isolate some knight in the gadget from the goal square). Moreover, if a lazy knight is input in one I/O square, then it is impossible for a winning sequence to output a lively knight from the opposite I/O square. To see this, let k_1, k_2, k_3, k_4 and q_1, q_2, q_3, q_4 respectively be the 11-knights and the squares marked with ○ in Figure 4, from left to right. Assume w.l.o.g.⁵ that a lively knight is output by the rightmost I/O square q^* , and notice that this implies that the 0-knight on q^* is captured by k_4 , which cannot clear q_4 . Then, q_4 must be cleared by k_3 and, in turn, q_3 must be cleared by k_2 , and q_2 must be cleared by k_1 . This means the input knight k on the leftmost I/O square must clear q_1 , i.e., k must have a budget of at least 6.

⁵ A symmetric argument holds when the I/O square used as an output is the leftmost one (once k_i is renamed in k_{5-i} and the appropriate symmetric squares for q_1, \dots, q_4 are chosen).



■ **Figure 5** Top: the sequence of moves played when an edge deletion gadget is used in traversal mode in order to output a 7-knight from the right I/O square when a lively knight is input in the left I/O square. Bottom: the sequence of moves played when an edge deletion gadget is used in deletion mode in order to output a 3-knight from each of the two I/O squares. In both figures the order of the moves is: red, blue, green, purple, and teal (if any).

Cubic vertex gadgets

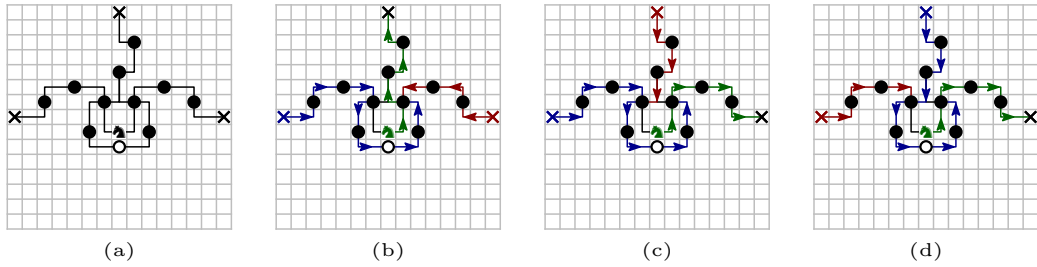
Each vertex $v \in V \setminus \{s, t\}$ corresponds to a tile (namely, the tile associated to coordinates p_v) which contains (a suitable rotation of) the cubic edge gadget of Figure 6 (a). This gadget has 3 I/O squares, each of which corresponds to a distinct edge incident to v .

In the intended operation of the gadget, any two I/O squares are used as inputs while the remaining one is an output. In details, if the gadget takes two knights with budgets at least 3 and 7 as input then it can be used to output a 7-knight. Figures 6 (b)–(d) show how this can be done for any combination of the intended input/outputs, up to symmetries.

No winning sequence of moves can use all the I/O squares as inputs. Moreover, if the gadget outputs a lively knight then it must take at least one lively knight as input. Indeed, the lively knight output from the gadget must necessarily be the only 11-knight k initially placed in the gadget itself (notice that any input knight placed on some I/O square must perform at least 6 captures to reach another I/O square), which implies that k cannot clear the square q marked with \circ . Hence, q must be cleared by some input knight k' , but this requires k' to perform at least 6 captures.

A similar reasoning shows that, in any winning sequence moves, the gadget cannot output two ore more lively knights. Indeed, for this to happen, there needs to be an output lively knight $k' \neq k$, which must necessarily be also an input knight. Then, (a) k must be the other lively output knight and, since k' cannot clear q (as this would require at least 6 captures, resulting in a budget of at most 5), (b) k must clear q . However, it is impossible for both (a) and (b) to happen.

Finally, we point out that it is possible to play the gadget in the following unintended way: whenever two knights with budget at least 3 as used as inputs, a 3-knight can be output form the remaining I/O square. However, as we argue in more details later, our edge deletion gadgets ensures that doing so always results in a losing configuration.



■ **Figure 6** (a) The arrangement of knights in a cubic vertex gadget. (b)–(d) each show a sequence of moves that outputs a 7-knight from one of the I/O squares when two knights having budgets at least 3 and 7 are input in the other two I/O squares. Red moves are performed first, followed by blue moves, and then by green moves.

2.1 One direction

Let P be a Hamiltonian path of $G = (V, E)$ that starts in s and ends in t . Let E_P be the edges in P and $\bar{E}_P = E \setminus E_P$, and recall that the edges connecting s and t to their sole neighbors must belong to E_P . The following sequence of moves wins the instance of SOLO-CHESS(\blacktriangle , $\{0, 2, 11\}$). For each edge $(u, v) \in \bar{E}_P$ we play the edge deletion gadget $g_{u,v}$ corresponding to (u, v) in deletion mode in order to output two 3-knights: one on the u -side and one on the v -side of (u, v) ; then, we play all the straight edge and corner edge gadgets that connect $g_{u,v}$ to u (resp. v), in this order, which has the effect of placing a 3-knight on the input of the cubic vertex gadget corresponding to u (resp. v) while clearing all other squares of the played gadgets. After this step, the only unplayed gadgets are (i) the start and end gadgets, (ii) the cubic vertex gadgets corresponding to the vertices in V , and (iii) the straight edge, corner edge, or edge deletion gadgets corresponding to the edges in E_P . We can play all these gadgets in the same order as vertices and edges are encountered in P : we start by outputting a 7-knight from the start gadget, then we play the sequence of straight/corner edge gadgets (possibly none) until we reach the cubic vertex gadget g_u corresponding to the vertex u that follows s in P . This brings a 7-knight to one of the I/O squares of g_u , while a 3-knight was already on some other I/O square. We use the 3-knight and the 7-knight as inputs for g_u in order to output a 7-knight on the only remaining I/O square of g_u , which corresponds to the edge (u, v) following u in P . We then play the gadgets associated with (u, v) so as to place a 7-knight on an I/O square of the cubic vertex gadget g_v of v . Notice that, in addition to straight/corner edge gadgets, playing the gadgets associated with (u, v) also involve playing a single edge deletion gadget in traversal mode. We repeat this process until a 7-knight is output from the cubic vertex gadget g_z corresponding to the vertex z immediately preceding t in P . Finally, we play the straight/corner edge gadgets associated with (z, t) to place a 7-knight on the input of the end gadget which, at this point, is the only gadget containing non-empty squares. To complete the winning sequence it suffices to play the end gadget using the input 7-knight.

2.2 The other direction

Fix a winning sequence σ and consider a gadget g that is not the start gadget. In order for g to output a lively knight from some I/O square in σ , it is necessary for g to receive a lively knight as input from another I/O square. Moreover, in σ , no gadget can output more than one lively knight and the end gadget must take a lively knight as input.

We define a directed graph H_σ whose vertex set is the set of gadgets in our instance of SOLO-CHESS(\blacktriangle , $\{0, 2, 11\}$) and such that H_σ contains a directed edge (g, g') iff there a move that causes a lively knight to be output from g and be input into g' . The previous observations imply that the out-degree of all vertices of H_σ is at most 1, that all vertices with out-degree 1 have in-degree at least 1 except for the start gadget, and that the in-degree of the end gadget is 1.

Then, H_σ must contain a path P from the start gadget to the end gadget. Moreover, P must traverse all cubic vertex gadgets. Indeed, suppose towards a contradiction that there is some cubic vertex gadget g , corresponding to a vertex $u \in V$, that is not in P .

If g does not take any lively knight as input then the only way to clear all squares of g results in g outputting a lazy knight from an I/O square associated with some edge $(u, v) \in E$, where $v \notin \{s, t\}$.⁶ Then, our instance of SOLO-CHESS(\blacktriangle , $\{0, 2, 11\}$) has an edge deletion gadget g' associated with (u, v) . Since the I/O square q on “ u ’s side” of g' cannot be used as input in traversal mode (as no lively knight can be input from q), g' must output a knight on q and this causes one or more knights placed on the (straight or corner) edge gadgets used to encode the portion of the polyline $\ell_{u,v}$ between p_u and $\ell'_{u,v}$ to become disconnected from the goal knight, a contradiction.

Otherwise g takes a lively knight as input, which means that there must exist a path P' from the start gadget to g in H_σ . Let g' be the last vertex of P' that is also in P , and notice that g' must have out-degree at least 2 in H_σ , i.e., it must output at least two lively knights, a contradiction.

If a graph obtained from G by performing edge subdivisions⁷ contains a simple path spanning V , then G contains a Hamiltonian path. Let G' be the graph obtained from G by subdividing each edge (u, v) into a path containing as many internal nodes as the number of gadgets placed in the tiles corresponding to the internal points of the polyline $\ell_{u,v}$ (that is, $|\ell_{u,v}| - 1$ if $u \in \{s, t\}$ or $v \in \{s, t\}$, and $|\ell_{u,v}| - 4$ if $u, v \notin \{s, t\}$). There is an injective homomorphism between the undirected version of H_σ and G' such that each start, end, and cubic vertex gadget in H_σ is mapped to the corresponding vertex in G' . Since P is a (simple) path that spans all start, end, and cubic vertex gadgets in H_σ , there is some (simple) path in G' that spans all vertices in V , hence G contains a Hamiltonian path.

2.3 Uniform budgets

Given a configuration c and a knight k , we denote by $\tau_c(k)$ the number of knights in c that are threatened by k .

We start by proving the following two lemmas which provide some “local” rules that allow us to perform some captures without compromising the solvability the configuration.

► **Lemma 1.** *Let C be a configuration containing a b -knight k that threatens only a single knight k' . Assume further that either (i) $b = 1$, or (ii) $b \geq 2$ and $\tau_c(k') = 2$. If C is solvable then it admits a winning sequence of moves that starts with $k \rightarrow k'$.*

Proof. Let q and q' be the squares containing k and k' in C , respectively, and let $\sigma = \langle m_1, m_2, \dots \rangle$ be any winning sequence of moves for C . We prove the claim by showing that σ can be transformed into a winning sequence that starts with $k \rightarrow k'$.

⁶ The only I/O square of the start gadget must be used as output, and the only I/O square of the end gadget must take lively knight as input.

⁷ The subdivision of an edge (u, v) into a path with $\ell \geq 1$ internal nodes consists in inserting of the new vertices w_1, w_2, \dots, w_ℓ , deleting (u, v) , and adding the edges in $\{(u, w_1), w(w_\ell, v)\} \cup \{(w_i, w_{i+1}) \mid i = 1, \dots, \ell - 1\}$.

We start by ensuring that σ contains a move of the form $k \rightarrow q'$. If this is not already the case, then it must contain $q' \rightarrow k$ and such a capture must necessarily be the last move of σ (since $q' \rightarrow k$ clears square q' and isolates the knight on q from any other knight in the resulting configuration). Then, replacing $q' \rightarrow k$ with $k \rightarrow q'$ also results in winning sequence of moves.

Let i be the index of move $k \rightarrow q'$ in σ . We now argue that, if m_i is not already the first move, then swapping it with its preceding move m_{i-1} still results into a winning sequence. The claim follows by iteratively performing such a swap until $k \rightarrow q'$ becomes the first move. If m does not involve square q nor square q' then performing either of $\langle m_1, \dots, m_{i-1}, m_i \rangle$ and $\langle m_1, \dots, m_{i-2}, m_i, m_{i-1} \rangle$ from c results in the same configuration. Otherwise, since m_{i-1} cannot clear q' , it must necessarily be of the form $k'' \rightarrow q'$ for some knight $k'' \neq k$. If (i) holds, then the configurations obtained from C by performing $\langle m_1, \dots, m_{i-1}, m_i \rangle$ and $\langle m_1, \dots, m_{i-2}, m_i, m_{i-1} \rangle$ are identical except, possibly, for the budget of the knight on q' which is *exactly* 0 in the former case and *at least* 0 in the latter. If (ii) holds then, after move m_{i-1} , the only remaining knights are on squares q and q' and m_i is the last move of σ . Then moving m_i immediately before m_{i-1} also results in a winning sequence. ◀

A repeated application of Lemma 1 shows that a knight k_0 that either has budget 0, or has $\tau_c(k_0) = 1$ and a budget in $\{1, \dots, 10\}$, can be simulated by setting the budget of k_0 to 11 and adding a *retinue* of $\eta = 11 - b$ additional 11-knights k_1, \dots, k_η such that each k_i with $i = 1, \dots, \eta - 1$ threatens only k_{i-1} and k_{i+1} , while k_η threatens only $k_{\eta-1}$.

To show that SOLO-CHESS(\blacktriangle , {11}) is NP-hard, we adapt our reduction of Section 2 as follows: instead of using tiles of size 15×15 we use *super-tiles* of size 57×57 and corresponding *super-gadgets*. Each super-tile can be thought of as a 5×5 grid of (regular) tiles. We first discuss how the start, end, straight edge, corner edge, and cubic vertex gadgets can be turned into super-gadgets. Each such super-gadget is obtained by first placing the corresponding regular gadget g in the center tile of the super-tile, and then using suitable straight edge gadgets to “connect” the I/O squares of g to the perimeter of the super-tile.⁸ Finally, we simulate each 0-knight by setting its budget to 11 and adding a retinue of eleven 11-knights as discussed above.⁹

The super gadget g^* corresponding to the edge-deletion gadget spans 4 super-tiles. To obtain the horizontal version of g , we arbitrarily choose one of these super-tiles t and we place knights in the other three super tiles as in the horizontal version of the super straight edge gadget.¹⁰ We arrange the knights in t as follows: first we place an edge deletion gadget g spanning 4 of the 5 tiles in the middle row of t , so that one of the I/O squares of g lies on one side of t ; then, we place a straight edge gadget in the missing tile of the middle row in order to “connect” the other I/O square of g to the opposite side of t . Finally, we replace the three 2-knights in g , along with all 0-knights in both g and the straight edge gadget, with 11-knights and we add the corresponding retinues, as discussed above.

All the 11-knights resulting from the above transformations will be entirely contained within the same super-gadget as the original knight. Moreover, none of the additional knights will introduce any inter-cluster threat. In fact, the reason for using super-tiles in place of

⁸ Two of the straight edge gadgets used in the super cubic vertex gadgets are rotated by 180 degrees.

⁹ This causes the knights placed on the I/O squares belonging to two super-gadgets to have two retinues each (one for each of the two gadget). Although one retinue would suffice, using one retinue per gadget simplifies the description of the reduction.

¹⁰ Defining the super edge deletion gadget in order to only span a single super-tile t would be sufficient to obtain a reduction to SOLO-CHESS(\blacktriangle , {11}). We employ four super-tiles in order to re-use the same construction described for the non-uniform case.

regular tiles is having enough free space around the gadgets to fit these new knights. Due to space limitations, the figures showing the resulting super-gadgets are omitted from this manuscript and can be found in the full version of the paper.

We have therefore proved the main results of this section, namely:

► **Theorem 2.** $\text{SOLO-CHESSE}(\blacktriangle, \{11\})$ is NP-hard.

3 Solo-Chess with only rooks

In this section we establish the NP-hardness of $\text{SOLO-CHESSE}(\blacktriangle, \{2\})$. Our construction is similar to the one employed by [1] to show the NP-hardness of $\text{SOLO-CHESSE}(\blacktriangle, \{0, 1, 2\})$. Roughly speaking, we would like to simply increase the budget of all rooks to 2, but this allows for some rook capture that were previously forbidden and breaks the reduction. To circumvent this, we are forced to place additional rooks with budget 2, which in turn can perform even more captures. The main technical challenge lies in showing that we force arbitrary winning strategies to follow the intended scheme of the reduction.

We start by proving the NP-hardness of an auxiliary problem named $\text{SOLO-CHESSE}^*(\blacktriangle, \{0, 2\})$, and then we show (see Section 3.3) how an instance of such a problem can be first transformed into an equivalent instance of $\text{SOLO-CHESSE}^*(\blacktriangle, \{2\})$ which, in turn, can be converted into another equivalent instance of $\text{SOLO-CHESSE}(\blacktriangle, \{2\})$. In the following we will refer to a rook with budget b as a b -rook.

The auxiliary problem $\text{SOLO-CHESSE}^*(\blacktriangle, \{0, 2\})$ is similar to $\text{SOLO-CHESSE}(\blacktriangle, \{0, 2\})$ except for the following two variations:

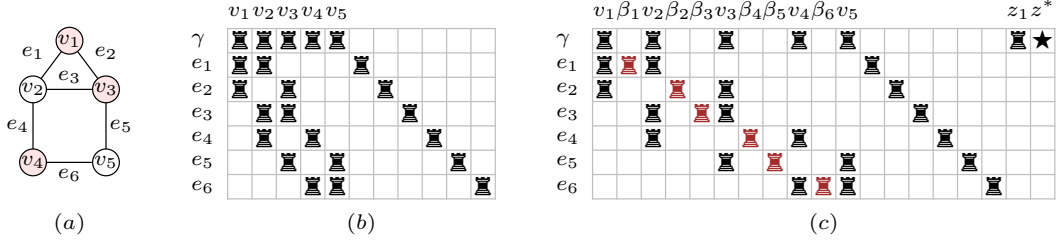
- we refer to the topmost row (resp. rightmost column) of an instance \mathcal{I} of $\text{SOLO-CHESSE}^*(\blacktriangle, \{0, 2\})$ as the *goal row* (resp. *goal column*), and to the square on their intersection of the goal row and the goal column as the *goal square*. The goal square must initially contain a special rook called the *goal rook*, and any winning sequence of moves for \mathcal{I} is required to leave the last remaining rook on the goal square;¹¹
- the goal column contains no rook other than the goal rook. Similarly, any column containing a 0-rook r , contains no rook other than r .

Our reduction for $\text{SOLO-CHESSE}^*(\blacktriangle, \{0, 2\})$ is from the (decision version of the) vertex cover problem (VC for short), which a well-known NP-hard problem [6]. The input of VC consists of a graph $G = (V, E)$ and of an integer k , and the goal is that of deciding whether there exists a set $S \subseteq V$ of size at most k such that at least one endvertex of each edge in E lies in S .

We build our instance of $\text{SOLO-CHESSE}^*(\blacktriangle, \{0, 2\})$ in two steps: first we construct a $(m + 1) \times (n + m)$ chessboard, and then we augment it by inserting some additional columns.

We start by describing the $(m + 1) \times (n + m)$ chessboard. Let $V = \{v_1, \dots, v_n\}$ and $E = \{e_1, \dots, e_m\}$. We associate the j -th of the first n columns with vertex v_j . Moreover, we associate the $(i + 1)$ -th row with edge e_i , and we refer to the topmost row as the *goal row* and we denote it with γ . To improve readability, we often refer to the squares of the chessboard using the row and column names instead of their integer coordinates, e.g., square (e_3, v_2) is at coordinates $(4, 2)$ and square (γ, v_5) is at coordinates $(1, 5)$. The goal row contains a 2-rook on each of the n columns v_1, \dots, v_n , and for each edge $e_i = (v_h, v_j)$ we place three 2-rooks: an *incidence rook* $r_{i,h}$ on square (e_i, v_h) , another *incidence rook* $r_{i,j}$

¹¹The budget of the goal rook is irrelevant since any winning sequence of moves cannot clear the goal square.



■ **Figure 7** (a) An instance of the (decision version of) vertex cover problem for $k = 3$. The vertices of a possible vertex cover are highlighted in red. (b) The corresponding $(m + 1) \times (n + m)$ chessboard constructed in the first step of our reduction. (c) The corresponding instance of $\text{SOLO-CHESS}^*(\blacksquare, \{0, 2\})$, where 0-rooks are in red and $\Delta = 1$.

on square (e_i, v_j) , and a final *collector rook* c_i on row e_i and column $n + i$. Essentially, the sub-chessboard consisting of rows e_1, \dots, e_m corresponds to the juxtaposition of the (transposed) incidence matrix of G with a $m \times m$ identity matrix, where each non-zero entry corresponds to a 2-rook. See Figure 7 (a) for an example instance of VC and Figure 7 (b) for the corresponding $(m + 1) \times (m + n)$ chessboard.

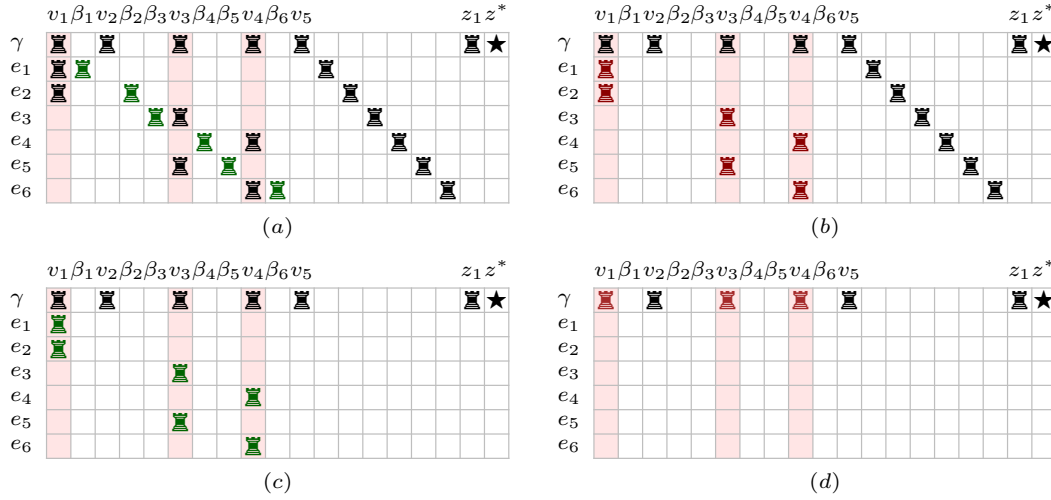
We now augment the above board. For each edge $e_i = (v_h, v_j)$ we add a new column β_i between v_h and v_j and a *blocker* 0-rook on square (e_i, β_i) . Next, let $\Delta = 2k - n$ and add $|\Delta| + 1$ new columns $z_1, \dots, z_{|\Delta|}, z^*$ to the right of the board. Each of the squares (γ, z_i) for $i = 1, \dots, |\Delta|$ contains a 2-rook if $\Delta > 0$ and a 0-rook if $\Delta < 0$. Finally, we place the goal rook on square (γ, z^*) . See Figure 7 (c) for an example.

3.1 One direction

Before discussing how a vertex cover of G can be turned into a winning sequence of moves for our instance of $\text{SOLO-CHESS}^*(\blacksquare, \{0, 2\})$, we restate the following characterization from [1] for the solvability of instances on one-dimensional boards. We define the potential of a rook with budget b to be $b - 1$ and the overall potential of a collection of rooks to be the sum of their potentials (where an empty collection of rooks has potential 0). We say that an instance of $\text{SOLO-CHESS}(\blacksquare, \mathbb{N})$ is (i, j) -solvable if it can be solved with the additional constraint that the last remaining rook must be placed on square at coordinates (i, j) .

► **Lemma 3** ([1], reformulated). *Consider an instance of $\text{SOLO-CHESS}(\blacksquare, \mathbb{N})$ on a board of size $1 \times n$, and let $\phi(j_1, j_2)$ denote the overall potential of the rooks on columns $j_1, j_1 + 1, \dots, j_2$. The instance is $(1, j)$ -solvable iff j contains a rook, $\phi(1, j - 1) \geq 0$, and $\phi(j + 1, n) \geq 0$.*

If S is a vertex cover of G of size at most k then the following is a winning strategy for our instance of $\text{SOLO-CHESS}^*(\blacksquare, \{0, 2\})$. For each edge e_i , we choose an endvertex v_h of e_i such that $v_h \in S$, we let v_j be the other endvertex (which might or might not be in S), and we perform the horizontal captures $r_{i,j} \rightarrow b_i \rightarrow r_{i,h}$ (see Figure 8 (a) and Figure 8 (b)), followed by $c_i \rightarrow (e_i, v_h)$ (see Figure 8 (c)). After these captures, each edge row contains only a single 1-rook on a column associated to a vertex in S . Then, for each column $v_i \in S$, we examine all the edge rows e_i in increasing order of i , and for each such row e_i containing a 1-rook in square (e_i, v_i) , we perform the vertical capture $(e_i, v_i) \rightarrow (\gamma, v_i)$. We are now left with a chessboard where the only non-empty row is the goal row γ (see Figure 8 (d)). In particular, the goal row contains one rook on each column v_1, \dots, v_n and at most k of these rooks have budget 0 (i.e., those resulting from the previous vertical captures), while the others (not involved in vertical captures) are 2-rooks. The remaining rooks are the $|\Delta|$ rooks on columns



■ **Figure 8** Notable configurations encountered in an winning sequences of moves for the instance of Figure 7 (c). Red columns corresponding to the vertices in a vertex cover of size 3. 2-rooks, 1-rooks, and 0-rooks are shown in black, greens, and red, respectively. (a) The configuration after all blocker rooks have been captured. (b) The configuration after all the squares initially containing the blocker rooks have been cleared. (c) The configuration after all collector rooks capture some rook on a red column. (d) The configuration after all squares in non-goal rows have been cleared. A winning sequence of moves from configuration (d) is $(\gamma, v_2) \rightarrow (\gamma, v_1) \rightarrow (\gamma, v_3), (\gamma, v_5) \rightarrow (\gamma, v_4) \rightarrow (\gamma, v_3), (\gamma, z_1) \rightarrow (\gamma, v_3) \rightarrow (\gamma, z^*)$.

$z_1, \dots, z_{|\Delta|}$, with overall potential $\Delta = 2k - n$, and the goal rook (on the rightmost column). Hence the sum of the potentials of all non-goal rooks is at least $0 \cdot k + (n - k) \cdot 2 - n + \Delta \geq 0$ and Lemma 3 implies that this configuration is (γ, z^*) -solvable.

3.2 The other direction

Here we show that a winning sequence of moves for our instance of $\text{SOLO-CHES}^*(\mathbb{R}, \{0, 2\})$ implies the existence of a vertex cover of G of size at most k .

We start by introducing the notions of *depleted row* and *disconnected configuration*, and we argue that any sequence of moves that results in a configuration that is either disconnected or creates a depleted row cannot be winning.

A row is *depleted* if it is not γ , it contains only a single 0-rook, and it contains no 1-rooks or 2-rooks. A configuration C is *disconnected* if the graph whose nodes are non-empty squares in C , and such that two distinct squares are linked by an edge iff they share the same row or the same column, is disconnected. It is immediate to verify that no disconnected configuration is solvable.

► **Lemma 4.** *Let $\sigma = \langle m_1, m_2, \dots \rangle$ be a winning sequence of moves. All configurations encountered during σ contain no depleted row.*

Proof. Let C_ℓ be the configuration obtained after performing the first ℓ moves of σ . Suppose towards a contradiction that some configuration C_h contains some depleted row e_i , and that all $C_{h'}$ with $h' > h$ contain no depleted rows. Let (e_i, v_j) be the square containing the unique 0-rook on row e_i in C_h .¹²

¹²The 0-rook in e_i must necessarily be on a column v_j where v_j is an endvertex v_j , since otherwise C_h would be disconnected.

Since (e_i, v_j) must eventually be cleared by σ , there is some configuration $C_{h'}$ with $h' \geq h$ and some 2-rook r in $C_{h'}$ such that move $m_{h'+1}$ is the capture $r \rightarrow (e_i, v_j)$. Since e_i contains no 2-rooks in $C_{h'}$, $r \rightarrow (e_i, v_j)$ must be a vertical capture. We split the proof depending on whether r is on the goal row or in some row $e_{i'}$ (with $e_{i'} \in E \setminus \{e_i\}$) in $C_{h'}$.

Suppose that r is in row $e_{i'}$ in $C_{h'}$ and focus on row $e_{i'}$ in $C_{h'+1}$ (whose square $(e_{i'}, v_j)$ is empty), which falls in one of the following three cases:

- If $(e_{i'}, \beta_{i'})$ contains a 0-rook in $C_{h'+1}$ then, among moves $m_{h'+2}, m_{h'+3}, \dots$, there is some 2-rook $r' \neq r$ on $e_{i'}$ that first captures $(e_{i'}, \beta_{i'})$ and then captures the only other remaining rook on $e_{i'}$. This results in a configuration $C_{h''}$ with $h'' > h' \geq h$ where $e_{i'}$ is depleted, which is a contradiction.
- If $(e_{i'}, \beta_{i'})$ contains a 1-rook in $C_{h'+1}$ then there is some 2-rook $r' \neq r$ on $e_{i'}$ such that some move among $m_1, \dots, m_{h'}$ is the capture $r' \rightarrow (e_{i'}, \beta_{i'})$ and some move among $m_{h'+2}, m_{h'+3}, \dots$ consists of r' capturing the only remaining rook on $e_{i'}$. Hence there is a configuration $C_{h''}$ with $h'' > h' \geq h$ where $e_{i'}$ is depleted, which is a contradiction.
- If $(e_{i'}, \beta_{i'})$ is empty in $C_{h'+1}$ then, in moves $m_1, \dots, m_{h'}$, the blocker $c_{i'}$ must have been captured by some 2-rook r' on $e_{i'}$ which then captured some other rook r'' on row $e_{i'}$ other than r' . This can only happen if r, r' , and r'' are the leftmost incidence rook, the rightmost incidence rook, and the collector of row $e_{i'}$, respectively. Then, $e_{i'}$ is depleted in $C_{h'+1}$ since its only rook is a 0-rook in square $(e_{i'}, \beta_{i'})$, and this provides the sought contradiction.

Suppose now that r is on the goal row in $C_{h'}$. Then, in $C_{h'+1}$, r is a 1-rook on (e_i, v_j) and either r captures a rook on column v_j , or there must be some 2-rook that first captures r and then captures some other rook on column v_j . In any case, at least one move among $m_{h'+1}, m_{h'+2}, \dots$ is a vertical capture performed by a 1-rook on column v_j . Let $m_{h''}$ be the last such move and let $(e_{i'}, v_j)$ be its target square, where $e_{i'} \in E$, so that $(e_{i'}, v_j)$ contains a 0-rook in $C_{h''}$.

We describe the state of row $e_{i'}$ with a 4-tuple $t \in \{0, 1, 2, \square, ?\}^4$ whose entries represent the contents of the left incidence square of $e_{i'}$, $(e_{i'}, \beta_{i'})$, the right incidence square of $e_{i'}$, and the collector square of $e_{i'}$, in this order. More precisely, 0, 1, and 2 respectively denote a 0-rook, a 1-rook, and a 2-rook, \square denotes an empty square, and $?$ denotes any of the above. Moreover, we underline the entry corresponding to the square on column v_j .

- If $(e_{i'}, \beta_{i'})$ is empty in $C_{h''}$, then the state of $e_{i'}$ must be $(\underline{0}, \square, \square, ?)$ or $(\square, \square, \underline{0}, ?)$. In any case, some move $m_{h'''}$ with $h''' > h''$ clears square $(e_{i'}, v_j)$. Since this cannot be a horizontal move (as it would result in $e_{i'}$ being depleted), it must be a vertical move (of a 1-rook), which contradicts our choice of h'' .
- If $(e_{i'}, \beta_{i'})$ contains a 0-rook in $C_{h''}$, then the state of $e_{i'}$ must be $(\underline{0}, 0, ?, ?)$ or $(?, 0, \underline{0}, ?)$. Since $(e_{i'}, \beta_{i'})$ must be cleared by a 2-rook (on row $e_{i'}$) that first captures $(e_{i'}, \beta_{i'})$ and then captures another rook on $e_{i'}$, the state of $e_{i'}$ resulting from this latter capture is one of (a) $(\underline{0}, \square, \square, ?)$, (b) $(\underline{?}, \square, \square, 0)$, (c) $(\square, \square, \underline{0}, ?)$, and (d) $(0, \square, \underline{\square}, \square)$. However (a) and (c) lead to a contradiction by using analogous arguments to the ones of the previous case, (d) implies that $e_{i'}$ is depleted in some configuration $C_{h'''}$ with $h''' > h'' > h' \geq h$, and in (b) row $e_{i'}$ cannot be cleared since capturing the 0-rook on the cleaner square results in a disconnected configuration.
- If $(e_{i'}, \beta_{i'})$ contains a 1-rook r' in $C_{h''}$, then the state of $e_{i'}$ must be either $(\underline{0}, 1, \square, ?)$ or $(\square, 1, \underline{0}, ?)$. Either r' captures some other rook in $e_{i'}$, in which case the resulting state of $e_{i'}$ is one of $(\underline{0}, \square, \square, ?)$, $(\underline{?}, \square, \square, 0)$, and $(\square, \square, \underline{0}, ?)$, thus the same arguments as above apply, or some 2-rook (on row $e_{i'}$) first captures on $(e_{i'}, \beta_{i'})$ and then captures another rook on $e_{i'}$, leaving $e_{i'}$ in state $(\underline{0}, \square, \square, \square)$ which corresponds to a depleted row. ◀

We now consider an arbitrary winning sequence of moves and we perform two consecutive transformations, each of which will result in another winning sequence that follows some desirable pattern of moves and is easier to analyze.

First transformation

To perform our first transformation, we observe that each square (e_i, β_i) that initially contains a 0-rook and must be cleared, which means that there must be some 2-rook r that performs the capture $r \rightarrow (e_i, \beta_i)$. Moreover, r must be on row e_i and cannot be the collector c_i , since, after capture $c_i \rightarrow (e_i, \beta_i)$, either the only non-empty square on row e_i is (e_i, β_i) , or there are two non-empty squares (e_i, β_i) and (e_i, v_j) where v_j is an endvertex of e_i . In the former case (e_i, β_i) is disconnected from the goal square, while in the latter case neither $(e_i, v_j) \rightarrow (e_i, \beta_i)$ nor $(e_i, \beta_i) \rightarrow (e_i, v_j)$ are possible since they would yield either a disconnected configuration or single 0-rook on row e_i (which is not solvable by Lemma 4). We conclude that r is one of the two incidence rooks on row e_i , which is still in its original square.

We now argue that rearranging the moves so that $r \rightarrow (e_i, \beta_i)$ becomes the first capture still results in a winning sequence. Indeed, we can iteratively swap $r \rightarrow (e_i, \beta_i)$ with its preceding move m since, if m involved square (e_i, β_i) then the configuration obtained after performing all moves up to $r \rightarrow (e_i, \beta_i)$ would be disconnected.

Performing the above rearrangement for each row e_i with $i = 1, \dots, m$, and executing the first m moves yields a solvable configuration C' in which the goal row is identical to that of the initial configuration, and each row e_i contains its collector c_i , exactly one incidence rook, and a 1-rook on square (e_i, β_i) . See Figure 8 (a) for an example.

Second transformation

For the second transformation, consider a generic edge $e_i \in E$ and recall that row e_i contains a single incidence rook $r_{i,j}$ on some square (e_i, v_j) (where v_j is an endvertex of e_i).

We first argue that no winning sequence of moves from C' contains a capture that targets square (e_i, β_i) . Indeed, if that were the case, there would also be some 2-rook r that performs the capture $r \rightarrow (e_i, \beta_i)$ (since (e_i, β_i) must eventually be cleared). The rook r must be either $r_{i,j}$ or the collector c_i . In the former case, the configuration resulting from the move $r_{i,j} \rightarrow (e_i, \beta_i)$ is disconnected. In the latter case, immediately after $c_i \rightarrow (e_i, \beta_i)$, row e_i contains a 1-rook on (e_i, β_i) and possibly another rook on (e_i, v_j) , hence the only possible moves result in either a disconnected configuration or in a single 0-rook on row e_i .

Since (e_i, β_i) must be cleared and the rook r on (e_i, β_i) is never captured, the sequence must include the move $r \rightarrow (e_i, v_j)$ (the only other option is $r \rightarrow c_i$ which results in a configuration where (e_i, β_i) cannot be cleared). Similarly to the previous transformation, we now argue that $r \rightarrow (e_i, v_j)$ can be performed as the first move of a winning sequence by iteratively swapping it with the previous move m . Indeed, if m targets (e_i, v_i) then the configurations obtained by (i) performing all the moves up to $r \rightarrow (e_i, v_j)$ and (ii) swapping $r \rightarrow (e_i, v_j)$ with m and then performing all the moves up to m , are identical except possibly for the budget of the rook in (e_i, v_j) which is 0 in the former case and *at least* 0 in the latter.

Performing the above rearrangement for each row e_i with $i = 1, \dots, m$, and executing the first m moves yields a solvable configuration C'' in which the goal row is identical to that of initial configuration, and each row e_i contains its collector c_i and exactly one 0-rook on some square (e_i, v_j) where v_j is an endvertex of e_i . As a consequence the set S containing all vertices $v_j \in V$ such that there exists at least one row e_i for which (e_i, v_j) is non-empty is a vertex cover of G . See Figure 8 (b) for an example.

Concluding the proof

We are now ready to conclude the proof, by showing that S has size at most k via a potential argument. More precisely, given a configuration C , we assign a potential $\psi_j(C)$ to each column $j \neq z^*$ as follows:

- $\psi_j(C) = 1$ if (γ, j) contains a 2-rook and there is no other rook on column j ;
- $\psi_j(C) = -1$ if (γ, j) is non-empty and either it contains a 0-rook, or there exists some $e_i \in E$ such that (e_i, j) is non-empty in C (possibly both);
- $\psi_j(C) = 0$ in the remaining cases.

We also let $\psi_{z^*}(C) = 0$ and we define the potential $\psi(C)$ of C as the sum of $\psi_j(C)$ over all columns j .

Fix any winning sequence of moves that starts from configuration C'' and let C_ℓ be the configuration resulting from performing the first ℓ moves of the sequence.

► **Lemma 5.** $\psi(C_\ell)$ is non-increasing w.r.t. ℓ .

Proof. First of all, notice that no rook on row γ ever performing any vertical capture, since this would result in a disconnected configuration, and the same holds for horizontal captures that target a column containing a collector. Of the remaining captures, only those that target a square on the goal row can affect $\psi(\cdot)$. We consider these captures separately depending on whether they are vertical or horizontal and we study how the potential of the affected column(s) changes as a result of the move.

Any vertical capture from some configuration C_ℓ that targets a square (γ, j) results in a 0-rook on square (γ, j) in configuration $C_{\ell+1}$. Therefore $\psi_j(C_\ell) = \psi_j(C_{\ell+1}) = -1$.

Consider now an horizontal capture $(\gamma, j) \rightarrow (\gamma, j')$ performed by a b -rook from some configuration C_ℓ and observe that (γ, j) must be the only non-empty square in column j . If $b = 2$ then $\psi_j(C_\ell) = 1$, $\psi_j(C_{\ell+1}) = 0$, $\psi_{j'}(C_\ell) \geq -1$, and $\psi_{j'}(C_{\ell+1}) \leq 0$. If $b = 1$ and $j' \neq z^*$ then $\psi_j(C_\ell) = \psi_j(C_{\ell+1}) = 0$, $\psi_{j'}(C_\ell) \geq -1$ and $\psi_{j'}(C_{\ell+1}) = -1$. Finally, if $b = 1$ and $j' = z^*$ then $\psi_j(C_\ell) = \psi_j(C_{\ell+1}) = \psi_{j'}(C_\ell) = \psi_{j'}(C_{\ell+1}) = 0$. ◀

For the configuration C'' we have $n - |S| + \max\{0, \Delta\}$ columns $j \neq j^*$ with $\psi_j(C'') = 1$ and $|S| + \max\{0, -\Delta\}$ columns $j \neq j^*$ with $\psi_j(C'') = -1$, hence $\psi(C'') = n - 2|S| + \Delta = 2k - 2|S|$. For the final configuration C^* (which contains a single rook in column j^*) we have $\psi(C^*) = 0$. Using Lemma 5 we have $2k - 2|S| = \psi(C'') \geq \psi(C^*) = 0$, which implies $|S| \leq k$.

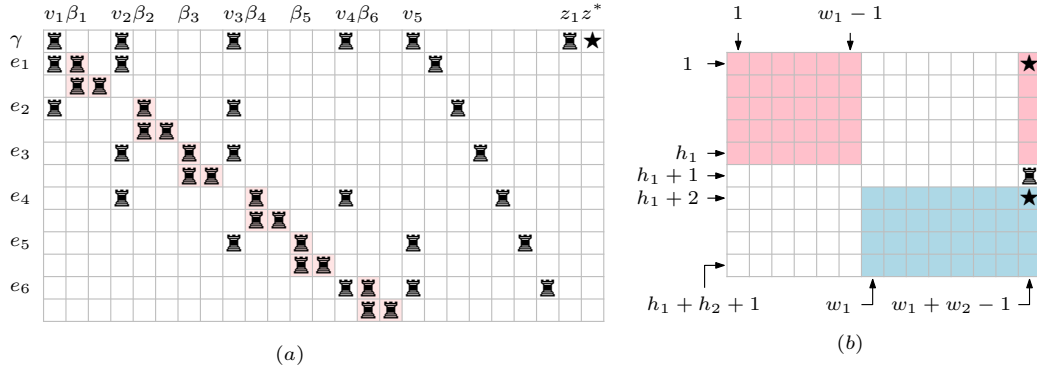
3.3 Uniform budgets

Here we show that the 0-rooks in the instances of SOLO-CHESS^{*}(\mathbb{N} , $\{0, 2\}$) resulting from the previous reduction can be simulated with 2-rooks, thus showing that SOLO-CHESS^{*}(\mathbb{N} , $\{2\}$) is NP-hard, and then we reduce SOLO-CHESS^{*}(\mathbb{N} , $\{2\}$) to SOLO-CHESS(\mathbb{N} , $\{2\}$).

Given a configuration C and a rook r , we denote with $\tau_C(r)$ the number of rooks $r' \neq r$ that are on the same row or on the same column as r .

► **Lemma 6.** Let C be a configuration containing a b -rook r such that $\tau_C(r) = 1$ and let r' be the only rook threatened by r . Assume further that neither r nor r' are on the goal square, and that either (i) $b = 1$, or (ii) $b = 2$ and $\tau_C(r') = 2$. If C is solvable then it admits a winning sequence of moves that starts with the capture $r \rightarrow r'$.

Proof. Let q and q' be the squares containing r and r' in C , respectively. Any winning sequence of moves $\sigma = \langle m_1, m_2, \dots \rangle$ for C cannot contain $q' \rightarrow r$, since q is not the goal square and hence such a move would result in a disconnected configuration. Then, some



■ **Figure 9** (a) The instance of $\text{SOLO-CHESS}^*(\clubsuit, \{2\})$ by applying our transformation to the instance of $\text{SOLO-CHESS}^*(\clubsuit, \{0, 2\})$ of Figure 7 (c). Red squares contains the 2-rooks replacing the original 0-rooks. (b) A sketch of the instance $\mathcal{I}_{1,2}$ of $\text{SOLO-CHESS}(\clubsuit, \{2\})$ obtained from the instances $\mathcal{I}_1, \mathcal{I}_2$ of $\text{SOLO-CHESS}^*(\clubsuit, \{2\})$.

move m_i of σ is the capture $r \rightarrow q'$. We show that, if $i \neq 1$, swapping m_i with m_{i-1} still results in a winning sequence. Indeed, m_{i-1} cannot involve q and it cannot clear q' (as m_i would then be illegal).

If (ii) holds, then m_{i-1} cannot involve q' (if m_{i-1} targeted q' then the resulting configuration would be disconnected) and $\langle m_1, \dots, m_i \rangle$ and $\langle m_1, \dots, m_{i-2}, m_i, m_{i-1} \rangle$ result in the same configuration.

If (i) holds, then either m_{i-1} does not involve q' , or m_{i-1} targets q' . In any case, the configurations obtained by performing $\langle m_1, \dots, m_i \rangle$ and $\langle m_1, \dots, m_{i-2}, m_i, m_{i-1} \rangle$ are identical except possibly for the budget of the rook in q' , which is 0 in the former configuration and *at least* 0 in the latter. ◀

Notice that all the (non-goal) 0-rooks used in our reduction are on columns that do not contain any other rook. Then, to simulate a (non-goal) 0-rook on square (i, j) we first sets its budget to 2, then we insert new row $i + 1$ immediately below i and a new column $j + 1$ immediately to the right of j , and finally we add two 2-rooks in squares $(i + 1, j)$ and $(i + 1, j + 1)$. See Figure 9 (a) for an example instance of $\text{SOLO-CHESS}^*(\clubsuit, \{2\})$ resulting from the above process. Clearly, if the original instance is solvable so is the one obtained after these 0-rooks have been replaced using the above strategy (since it is always possible to “recover” the original configuration, except for some additional empty rows and column, by performing two captures for each 0-rook that has been replaced), and a repeated application of Lemma 6 shows that the converse also holds.

We now reduce $\text{SOLO-CHESS}^*(\clubsuit, \{2\})$ to $\text{SOLO-CHESS}(\clubsuit, \{2\})$.

Let $\mathcal{I}_1, \mathcal{I}_2$ be two instances of $\text{SOLO-CHESS}^*(\clubsuit, \{2\})$ whose chessboards have sizes $h_1 \times w_1$ and $h_2 \times w_2$, respectively. We construct a new instance $\mathcal{I}_{1,2}$ of $\text{SOLO-CHESS}(\clubsuit, \{2\})$, whose chessboard has size $(h_1 + h_2 + 1) \times (w_1 + w_2 - 1)$, as follows (see Figure 9 (b)):

- the sub-chessboard of size $h_1 \times w_1$ consisting of the intersection of rows $1, 2, \dots, h_1$ and columns $1, 2, \dots, w_1 - 1, w_1 + w_2 - 1$ of $\mathcal{I}_{1,2}$ is a copy of chessboard of \mathcal{I}_1 in which the goal-rook is replaced with a 2-rook;
- the sub-chessboard of size $h_2 \times w_2$ consisting of the intersection of rows $h_1 + 2, h_1 + 3, \dots, h_1 + h_2 + 1$ and columns $w_1, w_2, \dots, w_1 + w_2 - 1$ of $\mathcal{I}_{1,2}$ is a copy of the chessboard of \mathcal{I}_2 in which the goal-rook is replaced with a 2-rook;
- square $(h_1 + 1, w_1 + w_2 - 1)$ contains a 2-rook.

► **Lemma 7.** *If both \mathcal{I}_1 and \mathcal{I}_2 are solvable then $\mathcal{I}_{1,2}$ is solvable. If $\mathcal{I}_{1,2}$ is solvable then at least one of \mathcal{I}_1 and \mathcal{I}_2 is solvable.*

Proof. Let R_1 be the set of rows $1, 2, \dots, h_1$, and R_2 be the set of rows $h_1 + 2, \dots, h_1 + h_2 + 1$. Let $q_1 = (1, w_1 + w_2 - 1)$, $q_2 = (h_1 + 2, w_1 + w_2 - 1)$, and $q^* = (h_1 + 1, w_1 + w_2 - 1)$. Moreover, let r_1 , r_2 , and r^* be the rooks initially on q_1 , q_2 , and q^* in $\mathcal{I}_{1,2}$, respectively.

If both \mathcal{I}_1 and \mathcal{I}_2 are solvable then a winning sequence of moves for $\mathcal{I}_{1,2}$ is obtained by: (i) performing all moves in any winning sequence for \mathcal{I}_1 , where any capture $r \rightarrow (1, w_1)$ targeting $(1, w_1)$ in \mathcal{I}_1 is replaced with the capture $r \rightarrow q_1$ in $\mathcal{I}_{1,2}$; (ii) performing all moves in any winning sequence for \mathcal{I}_2 , where any capture $(i, j) \rightarrow (i', j')$ in \mathcal{I}_2 is replaced with the capture $(h_1 + 1 + i, w_1 - 1 + j) \rightarrow (h_1 + 1 + i', w_1 - 1 + j')$ in $\mathcal{I}_{1,2}$; (iii) performing the captures by $r^* \rightarrow q_1 \rightarrow q_2$.

To show that if $\mathcal{I}_{1,2}$ is solvable then at least one of \mathcal{I}_1 and \mathcal{I}_2 is solvable, let $\sigma = \langle m_1, m_2, \dots \rangle$ be a winning sequence of moves $\mathcal{I}_{1,2}$. Moreover, let ℓ be the smallest index such that m_ℓ is a vertical capture on column $w_1 + w_2 - 1$ (such an index must exist) and notice that there is some $h \in \{1, 2\}$ such that m_ℓ is either the capture $q_h \rightarrow q^*$ or the capture $r^* \rightarrow q_h$. We consider these two cases separately.

If m_ℓ is the capture $q_h \rightarrow q^*$, then r_h performs no capture in moves $m_1, \dots, m_{\ell-1}$. Moreover, after move m_ℓ , all squares belonging to the rows in R_h must be empty (since otherwise the configuration would be disconnected). Since no move among $m_1, \dots, m_{\ell-1}$ can simultaneously involve both a square of a row in R_1 and a square of a row in R_2 , the sub-sequence of moves obtained from $\langle m_1, \dots, m_{\ell-1} \rangle$ by selecting all moves that involve a square in R_h is a winning sequence of moves for \mathcal{I}_h .

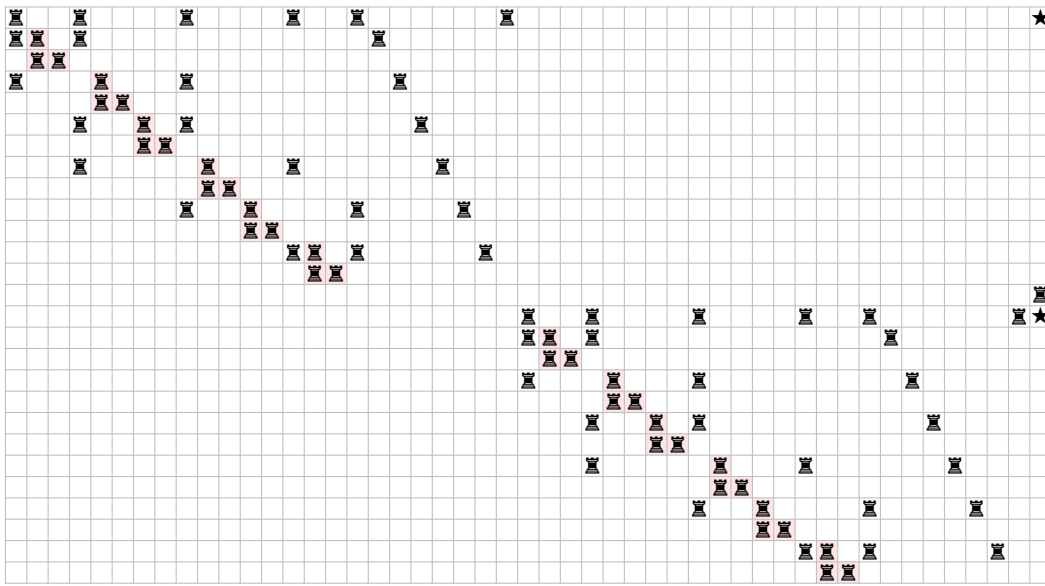
If m_ℓ is the capture $r^* \rightarrow q_h$, then let $\ell' > \ell$ the only other index such that $m_{\ell'}$ is a vertical capture on column $w_1 + w_2 - 1$. Such a capture is either $q_h \rightarrow q_{3-h}$ or $q_{3-h} \rightarrow q_h$. In the former case, the capture $q_h \rightarrow q_{3-h}$ has the effect of clearing square q_h and replacing the b -rook in q_{3-h} (where $b \in \{0, 1, 2\}$) with a 0-rook. Since no move m_t with $t \notin \{\ell, \ell'\}$ can simultaneously involve both a square of a row in R_1 and a square of a row in R_2 , the sub-sequence of moves of σ that involve a squares of a row in R_{3-h} is a winning sequence of moves for \mathcal{I}_{3-h} . In the latter case, after capture $q_{3-h} \rightarrow q_h$, all squares belonging to rows in R_{3-h} must be empty, and the sub-sequence of moves obtained from $\langle m_1, \dots, m_{\ell-1}, m_{\ell+1}, \dots, m_{\ell'-1} \rangle$ by selecting all moves that involve a square in R_{3-h} is a winning sequence of moves for \mathcal{I}_h . ◀

Then, if \mathcal{I} is an instance of $\text{SOLO-CHESS}^*(\mathbb{X}, \{2\})$, we can perform the above transformation with $\mathcal{I}_1 = \mathcal{I}_2 = \mathcal{I}$ to obtain an instance $\mathcal{I}_{1,2}$ of $\text{SOLO-CHESS}(\mathbb{X}, \{2\})$ that is solvable iff \mathcal{I} is solvable, as ensured by Lemma 7 (see Figure 10). We have thus proved the following:

► **Theorem 8.** *$\text{SOLO-CHESS}(\mathbb{X}, \{2\})$ is NP-hard.*

References

- 1 N. R. Aravind, Neeldhara Misra, and Harshil Mittal. Chess is hard even for a single player. In Pierre Fraigniaud and Yushi Uno, editors, *11th International Conference on Fun with Algorithms, FUN 2022, May 30 to June 3, 2022, Island of Favignana, Sicily, Italy*, volume 226 of *LIPICs*, pages 5:1–5:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICs.FUN.2022.5.
- 2 Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom.*, 4:235–282, 1994. doi:10.1016/0925-7721(94)00014-X.




■ **Figure 10** The final instance $\text{SOLO-CHESS}(\text{♚}, \{2\})$ corresponding to the vertex cover instance of Figure 7 (a).

- 3 Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti, and Mirko Rossi. On the pspace-completeness of peg duotaire and other peg-jumping games. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms, FUN 2018, June 13-15, 2018, La Maddalena, Italy*, volume 100 of *LIPICs*, pages 8:1–8:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPICs.FUN.2018.8.
- 4 Josh Brunner, Lily Chung, Michael J. Coulombe, Erik D. Demaine, Timothy Gomez, and Jayson Lynch. Complexity of solo chess with unlimited moves. *CoRR*, abs/2302.01405, 2023. doi:10.48550/ARXIV.2302.01405.
- 5 Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n . *J. Comb. Theory, Ser. A*, 31(2):199–214, 1981. doi:10.1016/0097-3165(81)90016-9.
- 6 M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
- 7 M. R. Garey, David S. Johnson, and Robert Endre Tarjan. The planar hamiltonian circuit problem is np-complete. *SIAM J. Comput.*, 5(4):704–714, 1976. doi:10.1137/0205049.
- 8 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 für Informatik, 2016. doi:10.4230/LIPICs.FUN.2016.18.
- 9 Chess.com LLC. Solo Chess - Capture All the Pieces on the Board. <https://www.chess.com/solo-chess>. Accessed: 2024-02-25.
- 10 James A. Storer. On the complexity of chess. *J. Comput. Syst. Sci.*, 27(1):77–100, 1983. doi:10.1016/0022-0000(83)90030-2.
- 11 Ryuhei Uehara and Shigeki Iwata. Generalized hi-q is np-complete. *IEICE TRANSACTIONS (1976-1990)*, 73(2):270–273, 1990.
- 12 Leslie G. Valiant. Universality considerations in VLSI circuits. *IEEE Trans. Computers*, 30(2):135–140, 1981. doi:10.1109/TC.1981.6312176.

Swapping Mixed-Up Beers to Keep Them Cool

Davide Bilò  

Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Maurizio Fiusco 

Department of Enterprise Engineering, University of Rome “Tor Vergata”, Italy

Luciano Gualà  

Department of Enterprise Engineering, University of Rome “Tor Vergata”, Italy

Stefano Leucci  

Department of Information Engineering, Computer Science and Mathematics, University of L'Aquila, Italy

Abstract

There was a mix-up in Escher’s bar and n customers sitting at the same table have each received a beer ordered by somebody else in the party. The drinks can be rearranged by swapping them in pairs, but the eccentric table shape only allows drinks to be exchanged between people sitting on opposite sides of the table. We study the problem of finding the minimum number of swaps needed so that each customer receives its desired beer before it gets warm.

Formally, we consider the COLORED TOKEN SWAPPING problem on complete bipartite graphs. This problem is known to be solvable in polynomial time when all ordered drinks are different [Yamanaka et al., FUN 2014], but no results are known for the more general case in which multiple people in the party can order the same beer. We prove that COLORED TOKEN SWAPPING on complete bipartite graphs is NP-hard and that it is fixed-parameter tractable when parameterized by the number of distinct types of beer served by the bar.

2012 ACM Subject Classification Theory of computation → Fixed parameter tractability; Mathematics of computing → Combinatoric problems; Theory of computation → Problems, reductions and completeness

Keywords and phrases Colored Token Swapping, Complete Bipartite Graphs, Labeled Token Swapping, FPT Algorithms, NP-Hardness

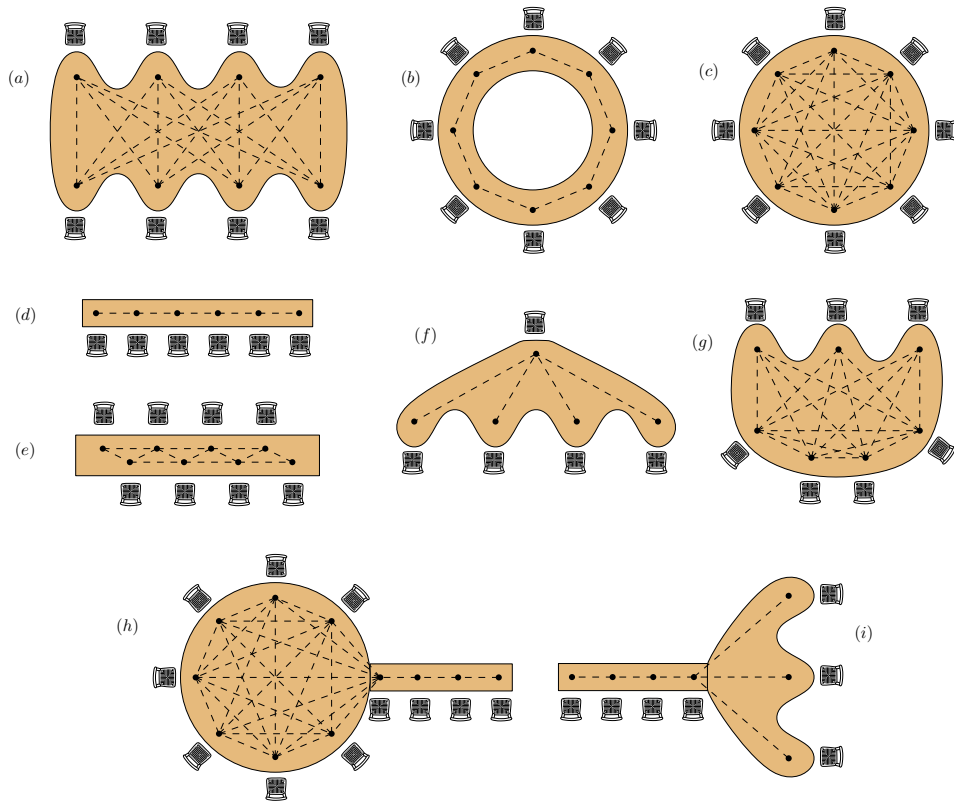
Digital Object Identifier 10.4230/LIPIcs.FUN.2024.5

1 Introduction

A party of n theoretical computer scientists walks into Escher’s bar, which is renowned for its tasty beers and its tables with eccentric geometric shapes. One of their papers just got accepted to an important conference¹ and they want to celebrate with a toast. They sit at a table shaped like the one in Figure 1 (a), and each of them orders one of the k beers from the bar’s selection. The waiter brings the order to the table but, unfortunately, it delivers the drinks to the patrons in a mixed order. The scientists decide to rectify the situation by swapping pairs of drinks, so that everybody eventually ends up with their beer of choice. However the table’s shape prevents some of these swaps: two people sitting on the same side of the table cannot easily swap their drinks, while people sitting on opposite sides of the table can do that by sliding their beers across. To avoid the disastrous waste of beer that would

¹ The reader might have already guessed the name of the conference. It suffices to say that it is held in a beautiful island, and that it is known for its entertaining talks.





■ **Figure 1** The interior of Escher’s bar with its eccentric tables. (a) a complete bipartite table; (b) a cycle table; (c) a clique table; (d) a path table; (e) a square of a path table; (f) a star table; (g) a complete split table; (h) a lollipop table; (i) a broom table. Notice that (h) and (i) are actually obtained by joining two different tables, as it might happen when large parties need to be accommodated.

result if two pints were to crash, they adopt the safe strategy of only performing one such swap at a time. As nobody likes their beer warm (see, e.g., [5]), the above rearrangement process should be completed as quickly as possible.

This can be formalized as an instance of the COLORED TOKEN SWAPPING (CTS) problem on complete bipartite graphs. In this problem we are given an integer $k \in \{1, \dots, n\}$, and an n -vertex graph $G = (V, E)$ (whose vertices represent scientists, and whose edges represent the swaps allowed by the table shape), in which each vertex v has an associated *color* $c(v) \in \{1, \dots, k\}$, and hosts a token of color $t(v) \in \{1, \dots, k\}$ (the colors represent the beers in the bar’s selection). A move (or swap) consists in selecting an edge $(u, v) \in E$ and swapping the tokens placed on u and v . The goal is that of finding a shortest sequence of swaps needed to place each token on a vertex of the same color.

The CTS problem is known to be NP-Hard for any $k \geq 3$ even for planar (non-complete) bipartite graphs with maximum degree 3, while it is solvable in polynomial time when $k = 2$ [15]. If one considers special classes of graphs, the problem has been shown to be polynomial time solvable on stars and paths [3]. On cliques, CTS remains NP-hard and, assuming the exponential time hypothesis (ETH) [7], it does not admit any $2^{o(n)}$ -time algorithm [3]. On the positive side, it is fixed-parameter tractable when parameterized by k [15].

The CTS problem is a generalization of the LABELED TOKEN SWAPPING (TS) problem, which corresponds to the case in which $k = n$ and there is exactly one vertex and one token of each color (i.e., scientists order distinct drinks). This special case has received extensive

■ **Table 1** State of the art of the CTS problem and of the TS problem (which corresponds to the case $k = n$ when there is exactly one vertex and one token of each color). References to results in this paper are in bold. Δ denotes the maximum degree of the input graph while results marked with “(ETH)” hold unless the exponential time hypothesis fails.

Graph class	# of colors (k)	Status	Ref.
General	$k = 2$	Solvable in polynomial time	[15]
General	Part of the input	Solvable in time $2^{O(n \log n)}$	[11]
General	Part of the input	4-approximable	[11]
Trees	Part of the input	2-approximable	[11]
Stars	Part of the input	Solvable in polynomial time	[3]
Paths	Part of the input	Solvable in polynomial time	[3]
Planar bipartite, $\Delta = 3$	Any fixed $k \geq 3$	NP-Hard	[15]
Cliques	Part of the input	Solvable in time $O(f(k) \cdot \text{poly}(n))$	[15]
Cliques	Part of the input	NP-Hard	[3]
Cliques	Part of the input	No $2^{o(n)}$ -time algorithm (ETH)	[3]
Complete bipartite	Part of the input	Solvable in time $O(f(k) + n)$	Thm. 14
Complete bipartite	Part of the input	NP-Hard	Thm. 17
Complete bipartite	Part of the input	No $2^{o(n)}$ -time algorithm (ETH)	Thm. 18
$\Delta = O(1)$	$k = n$ (TS)	APX-Hard	[11]
Trees	$k = n$ (TS)	NP-Hard	[1]
Cliques	$k = n$ (TS)	Solvable in polynomial time	[4]
Cycles	$k = n$ (TS)	Solvable in polynomial time	[8]
Brooms	$k = n$ (TS)	Solvable in polynomial time	[13, 10]
Lollipops	$k = n$ (TS)	Solvable in polynomial time	[10]
Square of paths	$k = n$ (TS)	Solvable in polynomial time	[6]
Complete split	$k = n$ (TS)	Solvable in polynomial time	[17]
Complete bipartite	$k = n$ (TS)	Solvable in polynomial time	[16]

attention in the literature and it is known to be APX-Hard on bounded-degree graphs [11] and NP-Hard on trees [1]. Similarly to the colored case, TS has been considered on special classes of graphs and, besides those mentioned above, polynomial-time algorithms are also known for cliques [4], cycles [8], brooms [13, 10], lollipop graphs [10], squares of paths [6], and complete split graphs [17] (see Figure 1 for the corresponding table shapes, and Table 1 for a summary).

TS is also known to be polynomial-time solvable on complete bipartite graphs [16], where the complexity status of the more general CTS is still unknown. This is exactly the focus of this work, where we show that CTS is NP-Hard for general k , while it can be solved in time $O(\varphi(k) + n)$ for a suitable function $\varphi(\cdot)$ depending only on k , i.e., it is fixed-parameter tractable w.r.t. k .² Moreover, we show that no $2^{o(n)}$ -time algorithm exists unless the ETH fails [7].

Other related work. The approximation of the CTS problem has been also studied, and a 4- and 2-approximation algorithms have been designed for general graphs and trees, respectively [11]. The same paper also shows that the problem can be solved in time $2^{O(n \log n)}$. Stars

² We assume that the algorithm does not have to check the validity of the input instance which, for CTS instances, can be done in time linear in the size of G .

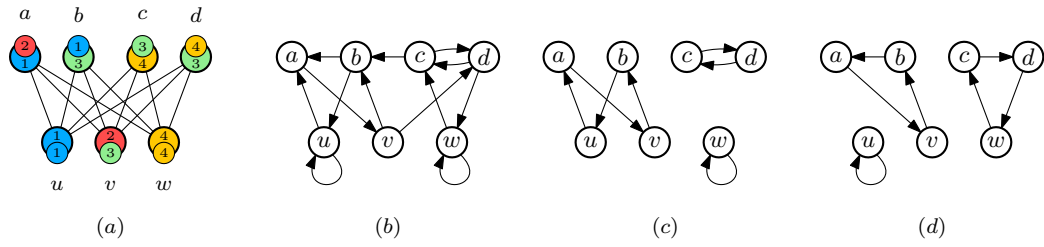


Figure 2 An instance of CTS on a complete bipartite graph with $X = \{a, b, c, d\}$ and $Y = \{u, v, w\}$ is shown in (a). The corresponding moves graph is shown in (b) while (c) shows a possible swapping plan \mathcal{P} with one self-loop, one X -cycle, no Y -cycles, and one XY -cycle, hence $f(\mathcal{P}) = 1$. A better swapping plan \mathcal{P}' with $f(\mathcal{P}') = 3$ is shown in (d). \mathcal{P}' consists of one self-loop, no X - or Y -cycles, and two XY -cycles.

and paths can be solved in polynomial time even for the weighted version of CTS, where each color has an associated weight and a the cost of a swap is given by the sum of the color-weights of the two tokens involved [2]. A generalization of the CTS problem, called *subset* token swapping, where each token has a subset of destination vertices it can be placed on, has been studied in [3]. Finally, a *parallel* version of both CTS and TS, where tokens can be simultaneously swapped over a matching in a single round, and the objective is to minimize the number of rounds (see for example [1] and the references therein).

Structure of the paper. In Section 2 we discuss a useful connection between solutions of the CTS problem on complete bipartite graphs and cycle-covers of a suitable auxiliary graph. As a warm-up, Section 3 is devoted to the special case $k = 3$, where we show how to solve the problem in polynomial time. We do not concern ourselves with finer grained complexity considerations since a $O(n)$ -time algorithm for this case follows from our more general $O(\varphi(k) + n)$ -time algorithm for the general case, which is described in Section 4. Finally, in Section 5 we establish the NP-hardness of CTS on complete bipartite graphs and show that no $2^{o(n)}$ -time algorithm exists unless the ETH fails.

2 Swapping plans and optimal solutions

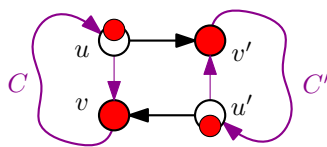
In this section we argue that the problem of finding an optimal sequence of swaps can be rethought as the problem of finding a suitable (vertex-disjoint) cycle cover of an auxiliary moves graph.

The *moves graph* associated with an instance of swapping colored tokens on bipartite graphs is a directed graph M with vertex set V that contains edge (u, v) iff $t(u) = c(v)$, where u and v are not necessarily distinct (see Figure 2 (a) and Figure 2 (b) for an example).

A *swapping plan* is a feasible assignment of tokens to vertices of the same color. Formally, a swapping plan is a collection $\mathcal{P} = \{C_1, \dots, C_h\}$ of vertex-disjoint cycles in M such that each vertex is part of exactly one cycle. A *partial swapping plan* is a swapping plan for a vertex-induced subgraph of M . We can show the following useful lemma:

► **Lemma 1.** *Let \mathcal{P}' be a partial swapping plan that spans a subset of vertices U of the moves graph M . Then, the subgraph of M induced by all the vertices that are not in U admits a swapping plan \mathcal{P}'' . Furthermore, $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$ is a swapping plan for M .*

Proof. Both M and every cycle in \mathcal{P}' contain as many vertices as tokens of the same color, for every color. Hence, this also holds for the subgraph M' of M induced by the vertices that are not in U . We arbitrarily match each token in M' with a vertex of the same color. If



■ **Figure 3** An example showing the merge operation of Lemma 3. The cycle C contains the edge (u, v) while the cycle C' contains the edge (u', v') . The two vertices v and v' have the same color. We can merge C and C' into a single cycle by substituting the edge (u, v) with the edge (u, v') and the edge (u', v') with the edge (u', v) .

a token on vertex u is matched with vertex v then $t(u) = c(v)$ and (u, v) is an edge of M' . Since each vertex of M' has exactly one incoming and one outgoing edge in the matching, such a matching induces a swapping plan \mathcal{P}'' per M' . Clearly, $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$ is a swapping plan for M . ◀

The CTS problem on bipartite graph is essentially that of finding a good swapping plan. Indeed, once a swapping plan \mathcal{P} is fixed, the problem becomes an instance of labelled TS, where a generic token on vertex u needs to be placed on the unique vertex v such that (u, v) is an edge of some cycle in \mathcal{P} . The labelled version on complete bipartite graphs can be solved optimally in polynomial time, and it has been shown that the optimal number of swaps is $n - f(\mathcal{P})$, where $f(\mathcal{P})$ is a function that depends on the topologies of the cycles in \mathcal{P} [14]. In the rest of the paper we use X and Y to denote the two sides of the bipartition of G , and we classify each cycle in \mathcal{P} as either a *self-loop*, as an X -*cycle* (resp. a Y -*cycle*) which has length at least 2 and contains only vertices in X (resp. Y), or as an XY -*cycle* which contains at least one vertex in X and one vertex in Y . Defining $\eta_0(\mathcal{P})$, $\eta_X(\mathcal{P})$, $\eta_Y(\mathcal{P})$, $\eta_{XY}(\mathcal{P})$ as the number of self-loops, X -cycles, Y -cycles, and XY -cycles in \mathcal{P} , respectively, we have:

$$\begin{aligned} f(\mathcal{P}) &= \eta_0(\mathcal{P}) + \eta_{XY}(\mathcal{P}) + \eta_X(\mathcal{P}) + \eta_Y(\mathcal{P}) - 2 \max\{\eta_X(\mathcal{P}), \eta_Y(\mathcal{P})\} \\ &= \eta_0(\mathcal{P}) + \eta_{XY}(\mathcal{P}) - |\eta_X(\mathcal{P}) - \eta_Y(\mathcal{P})|. \end{aligned} \quad (1)$$

As a consequence, the CTS problem on complete bipartite graphs can be equivalently thought of as the problem of finding a swapping plan maximizing $f(\cdot)$. Figure 2 (c) and Figure 2 (d) show two possible swapping plans with different values of $f(\cdot)$ for the instance in Figure 2 (a). We say that a cycle that is either a self-loop or an XY -cycle is *happy*, while X -cycles and Y -cycles are *unhappy*. Roughly speaking, one seeks to maximize the number of happy cycles while keeping the number of unhappy X - and Y -cycles as balanced as possible.

It turns out that, once an optimal swapping plan for the problem has been computed, the corresponding optimal sequence of swaps can be computed in $O(n)$ time, as stated in the following lemma, whose proof is given in Appendix A.

► **Lemma 2.** *A swapping plan \mathcal{P} for an instance \mathcal{I} of CTS on complete bipartite graphs can be converted, in time $O(n)$, into a solution for \mathcal{I} consisting of $n - f(\mathcal{P})$ swaps.*

In the following we provide two lemmas that allow us to rearrange cycles of a swapping plan. We start with a *merge* operation which combines two cycles of a swapping plan that share some color into a single cycle. We say that a color c appears in a cycle C if there exists at least one vertex with color c in C . Equivalently, we can say that c appears in C if and only if C contains some token of color c .

► **Lemma 3** (Merge operation). *Let \mathcal{P} be a (partial) swapping plan, let C, C' be two distinct cycles in \mathcal{P} and let c be a color that appears in both C and C' . Then, there exists a cycle C^* spanning all and only the vertices in C and C' such that $(\mathcal{P} \setminus \{C, C'\}) \cup \{C^*\}$ is a (partial) swapping plan.*

Proof. Let v and v' be two vertices of color c belonging to C and C' , respectively. And let u (resp. u') be the vertex that immediately precedes v (resp. v') in C (resp. C'). Notice that it might be $u' = u$ and/or $v' = v$. The cycle C^* is obtained from C and C' by removing the edges (u, v) and (u', v') and by adding the edges (u, v') and (u', v) (see Figure 3). ◀

Next lemma shows that the converse also holds: a cycle contains two vertices/tokens of the same color, it can be split into two shorter cycles.

► **Lemma 4** (Split operation). *Let C be a cycle and let u, v be two distinct vertices of C such that $c(u) = c(v)$ or $t(u) = t(v)$. There exist two cycles C_u, C_v whose vertex-sets partition the set of vertices in C and such that u is a vertex of C_u and v is a vertex of C_v .*

Proof. Let $C = \langle u = w_1, w_2, \dots, w_{j-1}, v = w_j, w_{j+1}, \dots, w_\ell, u \rangle$.

In the case $c(u) = c(v)$, we must have $t(w_{j-1}) = c(v) = c(u)$ and $t(w_\ell) = c(u) = c(v)$. Then, we choose $C_u = \langle u = w_1, \dots, w_{j-1}, u \rangle$ and $C_v = \langle v = w_j, w_{j+1}, \dots, w_\ell, v \rangle$. Notice that it might be $j = 1$, in which case C_u is a self-loop and/or $\ell = j$, in which case C_v is a self loop.

In the case $t(u) = t(v)$, we must have $c(w_2) = t(u) = t(v)$ and $c(w_{j+1}) = t(v) = t(u)$, hence we can choose $C_u = \langle u, w_{j+1}, \dots, w_\ell, u \rangle$ and $C_v = \langle v, w_2, \dots, w_{j+1}, v \rangle$. ◀

3 Swapping tokens of 3 colors

In this section, as a warm-up, we focus on the special case of CTS on complete bipartite graphs with $k = 3$ possible colors for the tokens/vertices. We introduce the main ideas that will be used also in the next section to solve the more general case of unbounded number of colors.

An instance is *lopsided* if there is some side $Z \in \{X, Y\}$ and some color c such that all the vertices in Z are *monochromatic*, i.e., they induce self-loops in the moves graph, and have all the same color c . The following lemma shows that lopsided instances can be easily solved, hence in the rest of this section we only consider instances that are not lopsided.

► **Lemma 5.** *A lopsided instance can be solved in polynomial time.*

Proof. W.l.o.g., we can assume that at least one token is misplaced, which means that not all vertices in M induce self-loops.

Let U be the set of vertices in M that form self-loops. Let $\mathcal{P}' = \{\langle u, u \rangle \mid u \in U\}$ be a partial swapping plan that contains all self-loops of M . As Z is monochromatic, we have $Z \subseteq U$. Let $\mathcal{P}'' = \{C_1, \dots, C_h\}$ be a swapping plan for the subgraph of M induced by all the vertices of M but those of U . The existence of this swapping plan \mathcal{P}'' is guaranteed by Lemma 1. Moreover, \mathcal{P}'' can be computed in polynomial time as it is a cycle cover of the vertex-induced subgraph of M .

We argue that all cycles in \mathcal{P}'' can be merged into a single cycle. By construction, no cycle in \mathcal{P}'' can be a self-loop. As a consequence, given any two cycles $C, C' \in \mathcal{P}''$, by the pigeonhole principle there is a color that appears in both C and C' . This implies that we can merge the two cycles into a single cycle C^* as proved in Lemma 3. Therefore, we can assume that \mathcal{P}'' contains a single cycle.

We divide the proof into two cases according to whether the color c of vertices in Z also appears in the unique cycle of \mathcal{P}'' or not.

The first case is when c also appears in the unique cycle of \mathcal{P}'' , say C' . Let C be a self-loop in \mathcal{P}' such that c appears in C . We define \mathcal{P} as the swapping plan obtained by the union of $\mathcal{P}' \setminus \{C\}$ plus the cycle C^* obtained by merging C with C' (Lemma 3). We argue that \mathcal{P} maximizes the function f defined in Equation (1). By construction, $f(\mathcal{P}) = |U|$. Let \mathcal{P}^* be an optimal swapping plan. Observe that $\eta_0(\mathcal{P}^*) + \eta_{XY}(\mathcal{P}^*) \leq |U|$ as each XY -cycle must contain at least one vertex of Z and thus of U . Therefore, $f(\mathcal{P}^*) \leq |U| - |\eta_X(\mathcal{P}^*) - \eta_Y(\mathcal{P}^*)| \leq |U| = f(\mathcal{P})$.

The second case is when c does not appear in the unique cycle of \mathcal{P}'' . W.l.o.g., we assume that $Z = X$, as the proof for the case $Z = Y$ is similar. We argue that $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$ maximizes the function f defined in Equation (1). By construction $f(\mathcal{P}) = |U| - 1$ as the unique cycle in \mathcal{P}'' is unhappy because it spans a subset of vertices in Y . Let \mathcal{P}^* be an optimal swapping plan. As each XY -cycle must contain at least one vertex of Z and thus of U , we have that $\eta_0(\mathcal{P}^*) + \eta_{XY}(\mathcal{P}^*) + \eta_X(\mathcal{P}^*) \leq |U|$, from which we derive $\eta_0(\mathcal{P}^*) + \eta_{XY}(\mathcal{P}^*) \leq |U| - \eta_X(\mathcal{P}^*)$. Furthermore, as no vertex spanned by the cycle in \mathcal{P}'' has the same color c of the vertices of Z , we have that \mathcal{P}^* contains unhappy Y -cycles (i.e., those that span vertices of colors different from c). This implies that $\eta_Y(\mathcal{P}^*) \geq 1$. Therefore, $f(\mathcal{P}^*) \leq \eta_0^* - \eta_X(\mathcal{P}^*) - |\eta_X(\mathcal{P}^*) - \eta_Y(\mathcal{P}^*)| \leq \eta_0^* - 1 = f(\mathcal{P})$. ◀

Section 3 allows us to rule out the case in which an instance is lopsided. As the following lemma shows, all the remaining instances have the nice properties that optimal swapping plans consist of happy cycles only.

► **Lemma 6.** *Let \mathcal{I} be an instance that is not lopsided. All optimal swapping plans for \mathcal{I} contain only happy cycles.*

Proof. Assume towards a contradiction that some optimal swapping plan \mathcal{P}^* for \mathcal{I} contains an unhappy cycle, and assume further that such a cycle is an X -cycle (w.l.o.g.).

Observe that no unhappy cycle C in \mathcal{P}^* contains any monochromatic vertex v , since otherwise we could increase the number of happy cycles without affecting the number of unhappy (thus increasing $f(\cdot)$) by replacing C with two cycles consisting of (i) a self-loop on v , and (ii) the cycle obtained from C by adding a directed edge (u, w) from the vertex immediately before v to the vertex immediately after v in C , and then deleting v (notice that $t(u) = c(v) = t(v) = c(w)$ and that u and v might coincide).

Then, every unhappy cycle involves at least two colors and any two distinct unhappy cycles can always be merged into a single cycle. This implies that it is impossible for \mathcal{P}^* to contain both an unhappy X -cycle and an unhappy Y -cycle, since they could be merged (see Lemma 3) into a single happy XY -cycle, increasing $f(\cdot)$. Therefore \mathcal{P} contains no unhappy Y -cycle and exactly one unhappy X -cycle C (otherwise all unhappy X -cycles could be merged into a single unhappy cycle, increasing $f(\cdot)$).

We now argue that C can be merged with some happy cycle containing a vertex in Y , thus decreasing the number of unhappy cycles to 0 without affecting the number of happy cycles, which is a contradiction. Since the vertices in C have at least two distinct colors and \mathcal{I} is not lopsided, we can always find some vertex $v \in Y$ such that at least one of $c(v)$ and $t(v)$ coincides with the color c of a vertex u in C . Then color c appears both in C and in the happy cycle C' of \mathcal{P}^* that contains v , which implies that C and C' can be merged (see Lemma 3). ◀

As a consequence of Lemma 6, we can restrict our attention to finding swapping plans with only happy cycles. However, such cycles could potentially be too long. However, it turns out that one can instead consider a relaxed version of the problem where the goal is

that of finding a partial swapping plan that maximizes the number of happy cycles, and such version results in short happy cycles. Clearly, the value of an optimal partial swapping plan, i.e., number of its happy cycles, is an upper bound to the value $f(\mathcal{P}^*)$ of an optimal swapping plan \mathcal{P}^* . We prove that the converse also holds, and we show how to convert an optimal partial swapping plan into an optimal swapping plan.

We now formalize the above relation between the two problems. With a slight abuse of notation we let $f(\tilde{\mathcal{P}})$ be the number of happy cycles in a partial swapping plan $\tilde{\mathcal{P}}$.

► **Lemma 7.** *Let $\tilde{\mathcal{P}}$ be an optimal partial swapping plan for a non-lopsided instance \mathcal{I} . We can compute an optimal swapping plan \mathcal{P} for \mathcal{I} with $f(\mathcal{P}) = f(\tilde{\mathcal{P}})$ in polynomial time.*

Proof. We say that a vertex is uncovered if it does not belong to any of the cycles in $\tilde{\mathcal{P}}$. We consider the case in which there exists at least one uncovered vertex, since otherwise the claim is trivial. We assume w.l.o.g., that such uncovered vertex is in X .

Observe that $\tilde{\mathcal{P}}$ contains no monochromatic uncovered vertex, since otherwise we could add a self-loop to $\tilde{\mathcal{P}}$ increasing $f(\cdot)$. Complete $\tilde{\mathcal{P}}$ into a swapping plan \mathcal{P}' by partitioning the uncovered vertices in an arbitrary set of additional cycles (Lemma 1 ensures that this is always possible), and notice that the optimality of $\tilde{\mathcal{P}}$ implies that each such cycle is unhappy.

Since each unhappy cycle contains vertices with at least two different colors any two such cycles can always be merged (see Lemma 3). In particular, any unhappy X -cycle can always be merged with any unhappy Y -cycle in \mathcal{P}' resulting in an happy XY -cycle. However, any such merge would contradict the optimality of $\tilde{\mathcal{P}}$, which implies that \mathcal{P}' contains no unhappy Y -cycles.

We can then merge all unhappy X -cycles in \mathcal{P}' into a single cycle C . Since \mathcal{I} is not lopsided, C can be further merged with some (happy) cycle containing a suitable vertex from Y by using analogous arguments to the ones employed in the proof of Lemma 6. This results in a swapping plan \mathcal{P} with $f(\mathcal{P}) \geq f(\tilde{\mathcal{P}})$, which implies $f(\mathcal{P}) = f(\tilde{\mathcal{P}})$. ◀

The following lemma provides another important key ingredient that allows us to find optimal solutions of our problem instance. In particular, the lemma states that we can focus on partial swapping plans containing only very short happy cycles.

► **Lemma 8.** *There exists an optimal partial swapping plan containing only happy cycles having a length of at most 4.*

Proof. In the rest of this proof we name the three distinct colors **red**, **green**, and **blue**. W.l.o.g., we can assume that the optimal partial swapping plan contains only happy cycles as unhappy cycles can be discarded.

We argue that, given any happy cycle C of length 5 or more, there exists another happy cycle C' of length at most 4 that contains only a subset of the vertices of C . The existence of an optimal partial swapping plan with cycles of length at most 4 follows by starting from any optimal partial swapping plan and iteratively replacing any long happy cycle C with a short happy cycle C' that spans only (a subset of) vertices of C , while discarding the vertices that are in C but not in C' .

Given two vertices u, v , we say that they are complementary if $c(u) = t(v)$ and $t(u) = c(v)$. In the rest of the proof we assume that C contains no monochromatic vertices, since otherwise we can choose C' as the self-loop consisting of a single monochromatic vertex from C . We can also assume that there are no two complementary vertices u, v of C on different sides of the bipartition, otherwise we choose $C' = \langle u, v, u \rangle$.

Since C has length of at least 5, there exists one side of the bipartition, say X , that contains 3 vertices of C . If any two of these three vertices have the same color, or host tokens of the same color then, by Lemma 4, we can split C into two cycles such that at least one the two cycles must include a vertex on the opposite side of the bipartition, i.e., it is happy.

As a consequence, we consider the case in which all vertices in X have different colors and host tokens of different colors as well. Let $u \in X$ and $z \in Y$ such that $c(z) = c(u)$, and let v, w two other vertices in X . W.l.o.g. (i.e., up to renaming of the colors), $c(u) = c(z)$ is **red**, $t(u) = c(v)$ is **blue**, and $c(w)$ is **green**. Notice that $t(v)$ cannot be **blue** (since v would be monochromatic) and cannot be **red** (since w would be monochromatic), hence $t(v)$ is **green**, and $t(w)$ is **red**. Then $\bar{C} = \langle u, v, w, u \rangle$ is a cycle. We have that $t(z)$ cannot be **red** (since z would be monochromatic) and it cannot be **green** (since z and w would be complementary). Then $t(z)$ is **blue**, and we can choose $C' = \langle z, v, w, z \rangle$, which is happy. ◀

We now show how to find an optimal partial swapping plan fulfilling the conditions of Lemma 8 in polynomial time. The idea is that of encoding the problem as an integer linear program (ILP) with a constant number of variables and a constant number of constraints, and then using Lenstra's algorithm [9] (whose running time is super-exponential in the number of variables and polynomial in the number of constraints) to solve such ILP.³

We define the type τ_v of a vertex v as the tuple $(c(v), t(v), \mathbb{1}_{x \in X})$ where $\mathbb{1}_{x \in X} = 1$ if $x \in X$ and 0 otherwise, and we let T_{vertex} be the set of all possible vertex types. Notice that two vertices of the same type are copies of one-another, hence we can describe any happy cycle by counting the number of nodes of each type. Given a cycle C , we define the type γ_C of C as a tuple that has one entry $\gamma_C(\tau)$ for each vertex type $\tau \in T_{\text{vertex}}$. The value of $\gamma_C(\tau)$ is the number of occurrences of vertices of type τ in C . Notice that there are only a constant number of distinct types γ that can be associated to happy cycles of length at most 4 (see Lemma 8), and we denote the set of all such types with T_{happy} .

Given a cycle type γ we denote by $\gamma(\tau)$ and $n_\tau(\mathcal{I})$ the number of nodes of type τ in γ and in the input instance \mathcal{I} , respectively. Our ILP has one variable $h_\gamma \in \mathbb{N}$ (where \mathbb{N} denotes the set of all non-negative integers) for each type $\gamma \in T_{\text{happy}}$ that represents the number of occurrences of happy cycles of type γ that are part of a partial swapping plan. The constraint associated to a type $\tau \in T_{\text{vertex}}$ ensures that the partial swapping plan spans at most $n_\tau(\mathcal{I})$ vertices of type τ .

$$\begin{aligned} \max \quad & \sum_{\gamma \in T_{\text{happy}}} h_\gamma \\ \text{s.t.} \quad & \sum_{\gamma \in T_{\text{happy}}} \gamma(\tau) h_\gamma \leq n_\tau(\mathcal{I}) \quad \forall \tau \in T_{\text{vertex}}, \\ & h_\gamma \in \mathbb{N} \quad \forall \gamma \in T_{\text{happy}}. \end{aligned}$$

Once an optimal solution for the above ILP has been found, we can convert it into an optimal partial swapping plan $\tilde{\mathcal{P}}$ in polynomial time, which, using Lemma 7 can be further converted into an optimal \mathcal{P} for \mathcal{I} , and then into optimal sequence of swaps (see Lemma 2). We have therefore shown the following:

► **Theorem 9.** *The CTS problem on complete bipartite graphs and $k = 3$ colors can be solved in polynomial time.*

4 Arbitrary number of colors

We extend the ideas used in Section 3 for instances with $k = 3$ colors to solve CTS problem on complete bipartite graphs with k colors in time $O(\varphi(k) + n)$, for some function φ that depends only on k .⁴

³ See [12] for a recent improvement over Lenstra's algorithm.

⁴ We assume that the input graph G is a complete bipartite graph and that the sides X and Y of the bipartition can be found in time $O(n)$. This is the case, e.g., when G is represented using adjacency lists.

5:10 Swapping Mixed-Up Beers to Keep Them Cool

We will solve the problem instance via an ILP formulation with a number of variables and constraints that depends only on k . Unfortunately, some of the nice properties we have proved for the 3-color case, as the existence of optimal swapping plans containing only happy cycles (Lemma 6) and the existence of partial swapping plans maximizing the number of happy cycles and containing only short happy cycles (Lemma 8), are no longer true. Therefore, we need to find alternative structural properties of some optimal solutions for our problem instances.

We say that a cycle is *long* if its length is at least $2k + 1$, and *short* otherwise. We first show the existence of optimal swapping plans with few long happy cycles.

► **Lemma 10.** *There exists an optimal swapping plan in which the number of happy cycles that are long is at most k .*

Proof. Given a generic swapping plan \mathcal{P} , we define $\Phi(\mathcal{P})$ as a vector, sorted in non-decreasing order, that contains one entry with value equal to the length $|C|$ of C for each long cycle C in \mathcal{P} .

We show that, if \mathcal{P} has $\ell \geq k + 1$ happy long cycles, then we can transform it into another swapping plan \mathcal{P}' that has the same number of self-loops, X -cycles, Y -cycles, and XY -cycles and either has $\ell - 1$ happy long cycles, or is such that $\Phi(\mathcal{P}')$ is greater than $\Phi(\mathcal{P})$ in lexicographic order. Since there are only finitely many possible vectors $\Phi(\cdot)$, a repeated application of the above transformation eventually results in a swapping plan \mathcal{P}^* with $f(\mathcal{P}^*) = f(\mathcal{P})$ and at most k happy long cycles.

Assume then that \mathcal{P} has $\ell \geq k + 1$ happy long cycles. For each such cycle C , the pigeonhole principle guarantees that we can find a pair $\{u, v\}$ of vertices in C that are both on the same side of the bipartition and such that $c(u) = c(v)$. We label C with the color of $c(u) = c(v)$ of these vertices.⁵ Since there are $\ell \geq k + 1$ cycles but at most k distinct labels, another invocation of the pigeonhole principle ensures that we can find two cycles C', C'' with the same label c . Assume w.l.o.g. that $|C'| \leq |C''|$ and let $\{u', v'\}$ and $\{u'', v''\}$ be the pairs of vertices chosen for C' and C'' , respectively.

By Lemma 4 we can partition the vertices of C' into two cycles $C_{u'}, C_{v'}$, where $C_{u'}$ contains u' and $C_{v'}$ contains v' . Since u' and v' are on the same side and C is happy, at least one cycle $C^* \in \{C_{u'}, C_{v'}\}$ must also be happy. Let \bar{C} the unique cycle in $\{C_{u'}, C_{v'}\} \setminus \{C^*\}$.

We choose \mathcal{P}' as the swapping plan obtained from \mathcal{P} by deleting C and C' and replacing them with C^* and the cycle obtained by merging \bar{C} with C'' (see Lemma 3, and notice that vertex u'' in C'' has the same color c of some vertex in \bar{C} , which is either u' or v').

To relate $\Phi(\mathcal{P})$ to $\Phi(\mathcal{P}')$, we observe that the entry with value $|C''|$ corresponding to C in $\Phi(\mathcal{P})$ is replaced with an entry of value $|C''| + |\bar{C}| > |C''|$ in $\Phi(\mathcal{P}')$, the entry corresponding to C' decreases, and all other entries are unaffected. ◀

The following lemma considers optimal swapping plans with a few long happy cycles and shows the existence of such a swapping plan that additionally contains few unhappy cycles.

► **Lemma 11.** *There exists an optimal swapping plan \mathcal{P} with at most k happy long cycles, $\eta_X(\mathcal{P}) \leq k/2$, and $\eta_Y(\mathcal{P}) \leq k/2$.*

This is needed since we obtain an algorithm with a running time of $O(n)$ for $k = O(1)$, but complete bipartiteness cannot be tested in time $o(n^2)$.

⁵ Notice that there might be multiple such pairs of vertices for C , and that different choices can result in different labels for C . In this case, we arbitrarily choose one of the pairs.

Proof. Among all optimal swapping plans having most k happy long cycles, let \mathcal{P} be one that minimizes $\max\{\eta_X(\mathcal{P}), \eta_Y(\mathcal{P})\}$ (the existence of \mathcal{P} is guaranteed by Lemma 10) and assume towards a contradiction that $\max\{\eta_X(\mathcal{P}), \eta_Y(\mathcal{P})\} > k$.

The optimality of \mathcal{P} ensures that no unhappy cycle $C \in \mathcal{P}$ contains any monochromatic vertex v , otherwise we could replace C with a self-loop on v (which is a short happy cycle), and an unhappy-cycle that spans all other vertices of C . This either increases the number of happy cycles by 1 without affecting the number of unhappy X - and Y -cycles (when $|C| > 2$) thus increasing $f(\cdot)$ by 1, or it increases the number of happy short cycles by 2 and removes exactly one unhappy cycle (when $|C| = 2$), thus increasing $f(\cdot)$ by at least 1.

In the rest of the proof we focus on the case $\eta_X(\mathcal{P}) \geq \eta_Y(\mathcal{P})$ since the complementary case is symmetric.

Since each X -cycle C in \mathcal{P} contains vertices of at least two different colors, the pigeonhole principle ensures that we can find two vertices of the same color that belong to two distinct X -cycles C'_X and C''_X . Then C'_X and C''_X can be merged into a single cycle C^*_X (see Lemma 3).

If $\eta_X(\mathcal{P}) > \eta_Y(\mathcal{P})$, the swapping plan $(\mathcal{P} \setminus \{C'_X, C''_X\}) \cup \{C^*_X\}$ has the same number of self-loops, XY -cycles, and Y -cycles as \mathcal{P} but one less X -cycle, which contradicts the optimality of \mathcal{P} .

Consider then $\eta_X(\mathcal{P}) = \eta_Y(\mathcal{P})$. Using analogous counting arguments as the ones used for X -cycles, we can find two Y -cycles C'_Y, C''_Y that can be merged into a single cycle C^*_Y . Then $\mathcal{P}^* = (\mathcal{P} \setminus \{C'_X, C''_X, C'_Y, C''_Y\}) \cup \{C^*_X, C^*_Y\}$ has the same set of self-loops and XY -cycles as \mathcal{P} , one less X -cycle, and one less Y -cycle. Hence we simultaneously have that (i) \mathcal{P}^* has at most k happy long cycles, (ii) $f(\mathcal{P}) = f(\mathcal{P}^*)$ which implies that \mathcal{P}^* is optimal, and (iii) $\max\{\eta_X(\mathcal{P}^*), \eta_Y(\mathcal{P}^*)\} = \eta_X(\mathcal{P}^*) = \eta_X(\mathcal{P}) - 1 = \max\{\eta_X(\mathcal{P}), \eta_Y(\mathcal{P})\} - 1$. This contradicts our choice of \mathcal{P} . \blacktriangleleft

As we will see, Lemma 11 allows us to model long happy cycles and unhappy cycles of an optimal swapping plan using a few additional variables in our ILP. However, as some of these cycles might be long, it is not clear how to guess the types of such long cycles. To deal with this issue, we introduce the concept of base cycles that will allow us to re-think of these long cycles as if they were short.

We say that a cycle of C is a *base cycle* if it does not contain two distinct vertices u, v of same type, i.e., $\tau_u = \tau_v$. Given a collection B of base cycles, we say that B is connected if the directed graph $H(B)$ that has one vertex for each vertex type that appears in some cycle in B , and an edge (τ, τ') iff the token color of type τ is the same as the vertex color of type τ' , is strongly connected. We now provide two useful lemmas that show the connection between cycles and base cycles of the moves graphs.

► **Lemma 12.** *Given a cycle C , there exists a connected collection B of base cycles spanning the vertices in C such that each vertex in C is part of exactly one cycle in B .*

Proof. We build B by starting with $B = \{C\}$ and then iteratively replacing any cycle \bar{C} in B containing two distinct vertices with the same color, with two (shorter) cycles that together span all and only the vertices in \bar{C} exactly once (see Lemma 4). Notice that $\{C\}$ is connected and that the above operation preserves the connectedness property. \blacktriangleleft

► **Lemma 13.** *Given a connected collection of base cycles B , there exists a cycle C that spans all and only the vertices in B .*

Proof. We maintain a collection of cycles B' which initially coincides with B and we prove the claim by iteratively merging pairs of cycles in B' until B' contains a single cycle C .

5:12 Swapping Mixed-Up Beers to Keep Them Cool

We now show that, as long as $|B'| \geq 2$, there always exist two distinct cycles $C', C'' \in B'$ that can be merged. Let C' be an arbitrary cycle in B' and call $T_{C'}$ and T_B the set of all types of the vertices in C' and in (the cycles of) B , respectively. If $T_{C'} = T_B$ then C' can be merged with any other cycle in B' (see Lemma 3) and we are done. Otherwise, let (τ, τ') be some edge of $H(B)$ such that $\tau \in T_B \setminus T_{C'}$ and $\tau' \in T_{C'}$. Such an edge always exist since $H(B)$ is strongly connected. Let $C'' \neq C'$ be any cycle in B' that contains a vertex u of type τ , let v be the vertex that immediately follows u in C'' (possibly $v = u$), and let w be a vertex of type τ' in C' . Our choice of vertices ensures that $t(u) = c(v)$ and that $t(u) = c(w)$, hence $c(v) = c(w)$, and Lemma 3 implies that C' and C'' can be merged. ◀

Let $T_{\text{s-happy}}$ to be the set of all possible types of short happy cycles, and let T_{base} be the set of all possible types of base cycles. The cardinalities of s-happy and T_{base} depend only on k since they contain only cycles of length at most $2k$ and $2k^2$, respectively.

Given a cycle type $\gamma \in T_{\text{base}}$, we denote by $\beta_\gamma(B)$ the number of base cycles of type γ in B . Given a collection B of base cycles, the *signature* $\sigma(B)$ of B is the set containing all types $\gamma \in T_{\text{base}}$ such that there is at least one cycle of type γ in B . Notice that since $|T_{\text{base}}|$ only depends on k , the same also holds for the number of the possible distinct signatures $\sigma(B)$.

We guess the values ℓ^* , η_X^* and η_Y^* corresponding to number of long happy cycles, X -cycles $\eta_X(\mathcal{P}^*)$ and Y -cycles $\eta_Y(\mathcal{P}^*)$ of an optimal swapping plan \mathcal{P}^* . Thanks to Lemmas 10 and 11, we can restrict ourselves to $\ell^* \in \{0, \dots, k\}$ and $\eta_X^*, \eta_Y^* \in \{0, \dots, \lfloor k/2 \rfloor\}$.

Then, we look for a swapping plan that, (i) has exactly ℓ^* happy long cycles C_1, C_2, \dots , (ii) has exactly η_X^* X -cycles C_1^X, C_2^X, \dots , (iii) has exactly η_Y^* Y -cycles C_1^Y, C_2^Y, \dots , and (iv) maximizes the number of short happy cycles.

Instead of searching for a generic long happy cycle C_i , X -cycle C_i^X , or Y -cycle C_i^Y , Lemmas 12 and 13 together allow us to look for connected collections B_i, B_i^X , or B_i^Y of base cycles, respectively.

To this aim we further guess the signatures σ_i, σ_i^X , and σ_i^Y of each collection B_i, B_i^X, B_i^Y , respectively. In particular, σ_i must be some connected signature that involves at least one vertex type for each side of the bipartition, while σ_i^X and σ_i^Y must be connected signatures in which all types are on side X and Y , respectively. We can now write an integer linear program that has:

- one variable $h_\gamma \in \mathbb{N}$ for each type $\gamma \in T_{\text{s-happy}}$;
- one variable $h_{i,\gamma} \in \mathbb{N}^+$ associated to each type $\gamma \in \sigma_i$ which counts the number of base cycles of type γ in B_i (here \mathbb{N}^+ denotes the set of all positive integers);
- one variable $x_{i,\gamma} \in \mathbb{N}^+$ associated to each type $\gamma \in \sigma_i^X$ which counts the number of base cycles of type γ in B_i^X ; and
- one variable $y_{i,\gamma} \in \mathbb{N}^+$ for each type $\gamma \in \sigma_i^Y$ which counts the number of base cycles of type γ in B_i^Y .

$$\begin{aligned}
\max \quad & \ell^* + |\eta_X^* - \eta_Y^*| + \sum_{\gamma \in T_{\text{happy}}} h_\gamma \\
\text{s.t.} \quad & \sum_{i=1}^{\ell^*} \sum_{\gamma \in \sigma_i} \gamma(\tau) h_{i,\gamma} + \sum_{i=1}^{\eta_X^*} \sum_{\gamma \in \sigma_i^X} \gamma(\tau) x_{i,\gamma} \\
& + \sum_{i=1}^{\eta_Y^*} \sum_{\gamma \in \sigma_i^Y} \gamma(\tau) y_{i,\gamma} + \sum_{\gamma \in T_{\text{happy}}} \gamma(\tau) h_\gamma = n_\tau(\mathcal{I}) \quad \forall \tau \in T_{\text{vertex}}, \\
& h_\gamma \in \mathbb{N} \quad \forall \gamma \in T_{\text{s-happy}}, \\
& h_{i,\gamma} \in \mathbb{N}^+ \quad \forall i \in \{1, \dots, \ell^*\} \quad \forall \gamma \in \sigma_i, \\
& x_{i,\gamma} \in \mathbb{N}^+ \quad \forall i \in \{1, \dots, \eta_X^*\} \quad \forall \gamma \in \sigma_i^X, \\
& y_{i,\gamma} \in \mathbb{N}^+ \quad \forall i \in \{1, \dots, \eta_Y^*\} \quad \forall \gamma \in \sigma_i^Y.
\end{aligned}$$

Similarly to Section 3, we once again solve the above ILP using Lenstra's algorithm [9], whose running time is upper bounded by a function of the number and variables and of the number of constraints of the ILP, both of which depend only on k .

After an optimal solution for the above ILP has been found, it needs to be converted into a corresponding swapping plan. We will now argue that this can be done in time $O(\varphi(k) + n)$, for some function $\varphi(\cdot)$ that depends only on k .

We first create a collection of buckets β_τ , with $\tau \in T_{\text{vertex}}$, where β_τ contains all vertices of G of type τ . We now assign the vertices of G to each of the $\sum_{\gamma \in T_{\text{happy}}} h_\gamma$ short happy cycles, and to each base cycle in the collections B_i, B_i^X, B_i^Y corresponding to the ℓ^* long happy cycles C_i, η_X^* unhappy X -cycles C_i^X , and η_Y^* unhappy Y -cycles C_i^Y . This can be done by drawing the vertices of the needed types from the corresponding bucket. Once this assignment is complete each of the above cycles can be built by brute-force from the assigned vertices in a time that depends only on k .

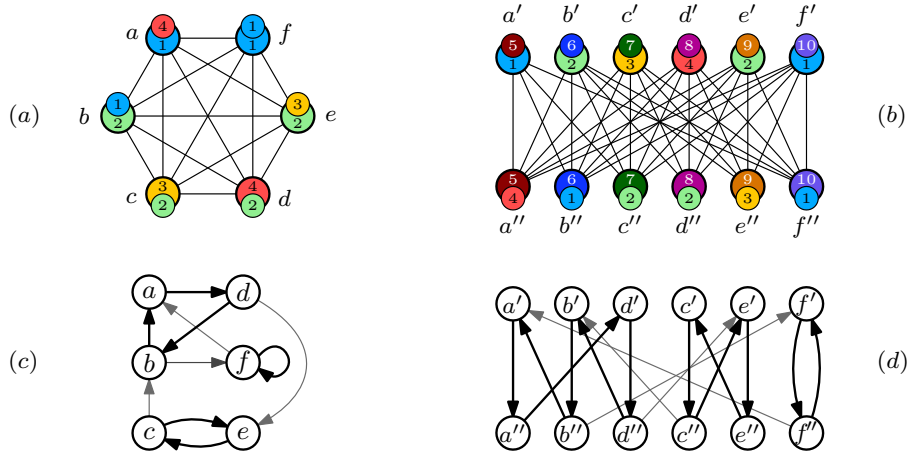
It remains to transform each (connected) collection of base cycles into a single cycle the spans the same vertices (whose existence is guaranteed by Lemma 13). It is not hard to come up with an algorithm that performs this task in time $O(k + n_B)$ for a collection spanning n_B vertices (see, e.g., Appendix B). Since there are $\ell^* + \eta_X^* + \eta_Y^* \leq 2k$ such collections, and each vertex of G is spanned by at most one base cycle, this takes time $O(\varphi(k) + n)$. By Lemma 2, the resulting optimal swapping plan can be converted into an optimal sequence of swaps in time $O(n)$, thus we have:

► **Theorem 14.** *The CTS problem on complete bipartite graphs with k colors can be solved in time $O(\varphi(k) + n)$ for a suitable function $\varphi(\cdot)$ that depends only on k .*

5 NP-hardness of CTS on complete bipartite graphs

We show that CTS on bipartite graphs is NP-Hard by reducing from CTS on cliques, which has been shown to be NP-Hard in [3].

Given an instance \mathcal{I}_H of CTS on a clique H with n_H vertices, we create an instance \mathcal{I}_G of CTS on a complete bipartite graph $G = (V, E)$ with $n = 2n_H$ vertices, where V is partitioned into two sets X and Y which correspond to the two sides of the bipartition.



■ **Figure 4** An example of the CTS instance \mathcal{I}_H on a clique graph with $n_H = 6$ vertices is shown in (a). The corresponding instance \mathcal{I}_G on a complete bipartite graph with $2n_H$ vertices defined by our reduction is shown in (b). In (c) and (d) we have the moves graphs of instances \mathcal{I}_H and \mathcal{I}_G , respectively. Notice that any directed edge (u, v) in the moves graph of \mathcal{I}_H is modelled by the path consisting of the two directed edges (u', u'') and (u'', v') in the moves graph of \mathcal{I}_G , the first one going from a vertex of X to a vertex of Y and the second one going from a vertex of Y to a vertex of X . Thus any swapping plan for \mathcal{I}_H corresponds to a swapping plan for \mathcal{I}_G , and vice-versa. The bold edges depict an optimal swapping plan for \mathcal{I}_H and the corresponding swapping plan for \mathcal{I}_G .

The graph G of \mathcal{I}_G is the complete bipartite graph obtained by creating two copies v', v'' of each vertex v in H and placing v' in X and v'' in Y . The set of colors of \mathcal{I}_G consists of all the colors in \mathcal{I}_H plus one new color c_v for each vertex v of H . We set $c(v') = c(v)$, $t(v') = c(v'') = c_v$, and $t(v'') = t(v)$. Let M be the moves graph of \mathcal{I}_G .⁶ See Figure 4 for an example.

► **Lemma 15.** *The sets X and Y are a bipartition of M . Moreover, each vertex $v' \in X$ has a single outgoing edge incident to it, which is the edge (v', v'') .*

Proof. Let $v' \in X$. Our construction of \mathcal{I}_G ensures that $t(v') = c_v$ and that the only vertex of color c_v is v'' . This implies that (v', v'') is in M as is the only outgoing edge of v' .

We now show that the moves graph M is bipartite by proving that there cannot be any edge (u'', v'') between two vertices $u'', v'' \in Y$. Indeed, as each vertex $v' \in X$ has a single outgoing edge incident to it which enters a vertex of Y , it cannot be the case that M contains an edge between two vertices of X . Consider now a generic vertex $u'' \in V$. Since $t(u'')$ is one of the colors of \mathcal{I}_H , while each $v'' \in Y$ has color $c(v'') = c_v$, which is one of the colors that has been introduced in \mathcal{I}_G but was not in \mathcal{I}_H , we conclude that (u'', v'') is not in M . ◀

► **Lemma 16.** *There exists a swapping plan \mathcal{P}_H for \mathcal{I}_H if and only if there exists a swapping plan \mathcal{P}_G for \mathcal{I}_G with $|\mathcal{P}_G| = |\mathcal{P}_H|$.*

Proof. Let \mathcal{P}_H be a swapping plan for \mathcal{I}_H . The swapping plan \mathcal{P}_G for \mathcal{I}_G contains one cycle C' for each cycle $C \in \mathcal{P}_H$. The cycle C' is obtained by renaming each vertex v of C into v' (i.e., the copy of v on side X of G) and then splitting each resulting edge (u', v') into the two

⁶ With a little abuse of notation, we use the same functions c and t to denote the colors of the vertices and of the tokens of both instances, respectively.

edges (u', u'') and (u'', v') via the inclusion of the intermediate vertex u'' (see Figure 4 for an example). As each vertex v of H is spanned by a unique cycle in \mathcal{P}_H , say C , by construction of \mathcal{P}_G , the two vertices v' and v'' are spanned by the unique cycle C' that corresponds to C . This implies that \mathcal{P}_G is a swapping plan for \mathcal{I}_G .

Consider now a swapping plan \mathcal{P}_G for \mathcal{I}_G . The swapping plan \mathcal{P}_H for \mathcal{I}_H contains one cycle C for each cycle $C' \in \mathcal{P}_G$. Lemma 15 implies that a generic cycle $C' \in \mathcal{P}_G$ consists of an alternation of vertices from X and Y , where each vertex u'' of C' that is in Y is immediately preceded by vertex $u' \in X$. The cycle C is obtained from C' by considering each vertex u'' of C' that is in Y , deleting it, and replacing its two incident edges in C' , namely the incoming edge (u', u'') and an outgoing edge (u'', v') for some $u' \in X$, with the edge (u, v) . As each vertex v' of G is spanned by a unique cycle in \mathcal{P}_G , say C' , by construction of \mathcal{P}_H , v is spanned by the corresponding cycle C . Therefore \mathcal{P}_H is a swapping plan for \mathcal{I}_H . ◀

Lemma 15 implies that all cycles in M are happy. Hence, all swapping plans \mathcal{P}_G for \mathcal{I}_G have $\eta_X(\mathcal{P}_G) = \eta_Y(\mathcal{P}_G) = 0$ and, from Equation (1), we have $f(\mathcal{P}_G) = |\mathcal{P}_G|$. Therefore the minimum number of swaps needed to solve \mathcal{I}_G is $n - f(\mathcal{P}_G^*)$, where \mathcal{P}_G^* is a swapping plan for \mathcal{I}_G that maximizes $|\mathcal{P}_G^*|$.

As shown in [4], the minimum number of swaps needed to solve \mathcal{I}_H is $n_H - |\mathcal{P}_H^*|$, where \mathcal{P}_H^* is a swapping plan of maximum cardinality for \mathcal{I}_H . Lemma 16 and the above discussion imply that $|\mathcal{P}_H^*| = |\mathcal{P}_G^*|$, and hence \mathcal{I}_G admits a solution with at most $n - x$ swaps if and only if \mathcal{I}_H admits as solution with at most $n_H - x$ swaps. We thus have the following:

► **Theorem 17.** *The COLORED TOKEN SWAPPING problem on complete bipartite graphs is NP-hard.*

Moreover the CTS problem on a clique of n_H vertices cannot be solved in time $2^{o(n_H)}$ unless the exponential time hypothesis fails [3, 7], and our reduction ensures that $n = \Theta(n_H)$, a similar result also holds for complete bipartite graphs:

► **Theorem 18.** *The COLORED TOKEN SWAPPING problem on complete bipartite graphs cannot be solved in time $2^{o(n)}$, unless the exponential time hypothesis fails.*

References

- 1 Oswin Aichholzer, Erik D. Demaine, Matias Korman, Anna Lubiw, Jayson Lynch, Zuzana Masárová, Mikhail Rudoy, Virginia Vassilevska Williams, and Nicole Wein. Hardness of token swapping on trees. In Shiri Chechik, Gonzalo Navarro, Eva Rotenberg, and Grzegorz Herman, editors, *30th Annual European Symposium on Algorithms, ESA 2022, September 5-9, 2022, Berlin/Potsdam, Germany*, volume 244 of *LIPIcs*, pages 3:1–3:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.ESA.2022.3.
- 2 Ahmad Biniiaz, Kshitij Jain, Anna Lubiw, Zuzana Masárová, Tillmann Miltzow, Debajyoti Mondal, Anurag Murty Naredla, Josef Tkadlec, and Alexi Turcotte. Token swapping on trees. *Discret. Math. Theor. Comput. Sci.*, 24(2), 2022. doi:10.46298/DMTCS.8383.
- 3 Édouard Bonnet, Tillmann Miltzow, and Pawel Rzazewski. Complexity of token swapping and its variants. *Algorithmica*, 80(9):2656–2682, 2018. doi:10.1007/S00453-017-0387-0.
- 4 Arthur Cayley. Lxxvii. note on the theory of permutations. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 34(232):527–529, 1849.
- 5 Matt Groening, James L. Brooks, Sam Simon, Dan Castellana, and Deb Lacusta. The Simpsons. Fox Broadcasting Company, 1999. Season 11, Episode 18 (Days of Wine and D’oh’ses), Timestamp: 18:30. URL: <https://www.imdb.com/title/tt0701222>.
- 6 Lenwood S. Heath and John Paul C. Vergara. Sorting by short swaps. *J. Comput. Biol.*, 10(5):775–789, 2003. doi:10.1089/106652703322539097.

- 7 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k -sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/JCSS.2000.1727.
- 8 Mark Jerrum. The complexity of finding minimum-length generator sequences. *Theor. Comput. Sci.*, 36:265–289, 1985. doi:10.1016/0304-3975(85)90047-7.
- 9 Hendrik W. Lenstra Jr. Integer programming with a fixed number of variables. *Math. Oper. Res.*, 8(4):538–548, 1983. doi:10.1287/MOOR.8.4.538.
- 10 Jun Kawahara, Toshiki Saitoh, and Ryo Yoshinaka. The time complexity of permutation routing via matching, token swapping and a variant. *J. Graph Algorithms Appl.*, 23(1):29–70, 2019. doi:10.7155/JGAA.00483.
- 11 Tillmann Miltzow, Lothar Narins, Yoshio Okamoto, Günter Rote, Antonis Thomas, and Takeaki Uno. Approximation and hardness of token swapping. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPICs*, pages 66:1–66:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPICs.ESA.2016.66.
- 12 Victor Reis and Thomas Rothvoss. The subspace flatness conjecture and faster integer programming. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023, Santa Cruz, CA, USA, November 6-9, 2023*, pages 974–988. IEEE, 2023. doi:10.1109/FOCS57990.2023.00060.
- 13 Theresa P. Vaughan. Factoring a permutation on a broom. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 30:129–148, 1999.
- 14 Katsuhisa Yamanaka, Erik D. Demaine, Takehiro Ito, Jun Kawahara, Masashi Kiyomi, Yoshio Okamoto, Toshiki Saitoh, Akira Suzuki, Kei Uchizawa, and Takeaki Uno. Swapping labeled tokens on graphs. *Theor. Comput. Sci.*, 586:81–94, 2015. doi:10.1016/J.TCS.2015.01.052.
- 15 Katsuhisa Yamanaka, Takashi Horiyama, J. Mark Keil, David G. Kirkpatrick, Yota Otachi, Toshiki Saitoh, Ryuhei Uehara, and Yushi Uno. Swapping colored tokens on graphs. *Theor. Comput. Sci.*, 729:1–10, 2018. doi:10.1016/J.TCS.2018.03.016.
- 16 Katsuhisa Yamanaka, Takashi Horiyama, David G. Kirkpatrick, Yota Otachi, Toshiki Saitoh, Ryuhei Uehara, and Yushi Uno. Swapping colored tokens on graphs. In Frank Dehne, Jörg-Rüdiger Sack, and Ulrike Stege, editors, *Algorithms and Data Structures - 14th International Symposium, WADS 2015, Victoria, BC, Canada, August 5-7, 2015. Proceedings*, volume 9214 of *Lecture Notes in Computer Science*, pages 619–628. Springer, 2015. doi:10.1007/978-3-319-21840-3_51.
- 17 Gaku Yasui, Kouta Abe, Katsuhisa Yamanaka, and Takashi Hirayama. Swapping labeled tokens on complete split graphs. *Inf. Process. Soc. Japan. SIG Tech. Rep.*, 2015(14):1–4, 2015.

A Converting a swapping plan to a solution

In this section we show how a swapping plan \mathcal{P} for an instance of CTS can be transformed into a solution consisting of $n - f(\mathcal{P})$ swaps in time $O(n)$. This is essentially an algorithmic rendition of the inductive proof of [14], which is constructive, and also implies the correctness of our algorithm.

We say that a vertex v of some cycle C of a swapping plan is *special* if the two vertices immediately following v in C are both on the opposite side of the bipartition compared to v .

As long as there is a cycle in \mathcal{P} that is not a self-loop, the algorithm iteratively finds two nodes on opposite sides of the bipartition, swaps their tokens, and updates \mathcal{P} to be a swapping plan for the resulting configuration. In particular, each iteration of the algorithm executes the first of applicable rule among the following three:

1. If \mathcal{P} contains some XY -cycle C , then choose three vertices u, v, w of C as follows:
 - If C contains no special vertices (this is always the case when $|C| = 2$), let (u, v) be an arbitrary edge of C , and let w be the vertex that appears immediately after v in C (possibly $w = u$).

- Otherwise (i.e., $|C| \geq 3$ and C contains some special vertex), let u be a special vertex, and let v, w be the two vertices that appear immediately after u in C , in this order. Then swap the tokens on u and v . To update \mathcal{P} replace (u, v) and (v, w) with the edges (u, w) and (v, v) . This has the effect of removing v from C (possibly resulting in a self-loop) and creating a new self-loop on v .
- 2. If \mathcal{P} contains an X -cycle C and an Y -cycle C' , let u (resp. v) be an arbitrary vertex in C (resp. C'). Swap the tokens on u and v . To update \mathcal{P} remove the two edges (u, u') and (v, v') outgoing from u and v , respectively, and add the two new edges (u, v') and (v, u') . This causes C and C' to merge into a single XY -cycle.
- 3. Otherwise let C be any X -cycle or Y -cycle in \mathcal{P} . Choose an arbitrary vertex u from C and an arbitrary (singleton) vertex v from the opposite side of the bipartition. Swap the tokens on u and v . To update \mathcal{P} delete the edge (u, u') outgoing from u and the self-loop (v, v) , and replace them with the edges (u, v) and (v, u') . This has the effect of merging C and (v, v) into a single cycle.

We now argue that the above algorithm can be implemented to run in time $O(n)$. We keep track of four lists of cycles that store self-loops, X -cycles, Y -cycles, and XY -cycles, respectively. Each C is represented by storing a doubly-linked list L_{vertices} of its vertices (in order), the number of vertices of C that are in X and in Y (respectively), and a list L_{special} of the special nodes of C . The generic element that stores a node v in L_{special} also has a pointer to the element that stores v in L_{vertices} , and vice-versa. Clearly, given \mathcal{P} , we can initialize the above data structures in time $O(n)$. Moreover, selecting the next rule to apply, and updating the data structure following the changes to \mathcal{P} dictated by such rule can be done time $O(1)$. Hence the overall running time is $O(n + |\mathcal{P}|) = O(n)$.

B Merging a connected collection of base cycles in time $O(n)$

In this section we argue that the cycles of a connected collection B of base cycles can be merged into a single cycle C^* in time $O(k + n_B)$, where n_B denotes the overall number of vertices spanned by the base cycles in B .

We start by building an auxiliary undirected bipartite graph H' in which the *color-vertices* on one side of the bipartition are the distinct colors of the vertices spanned by the cycles in B , and the *cycle-vertices* on the other side of the bipartition are the base cycles in B . H contains an edge (c, C) iff there is some vertex in cycle C that has color c , and this edge is labelled with the name of any such vertex. The above auxiliary graph can be built in time $O(n)$.⁷

► **Lemma 19.** *The graph H' is connected.*

Proof. Since each color-vertex in G has at least one cycle-vertex as a neighbor, it suffices to show that there exists a walk W in H' between any two distinct cycles $C, C' \in B$.

⁷ For each $C \in B$, we can find a collection L_C containing exactly one vertex v with $c(v) = c$ for each distinct color c that appears in C . We start with $L_C = \emptyset$ and we examine the vertices of C one at a time while updating a *global* array of k flags. The c -th flag is set to true iff some vertex of color c has already been encountered in C . Whenever a vertex v of a new color c is encountered, the corresponding flag is set to true and v is added to L_C . After all the vertices of C have been processed, the flags of the colors in L_C are reset to false. The overall running time is $O(k + n_B)$.

5:18 Swapping Mixed-Up Beers to Keep Them Cool

Let τ and τ' denote the types of two arbitrary vertices from C and C' , respectively. Since $H(B)$ is connected, there exists some (directed) path $\pi = \langle \tau = \tau_1, \tau_2, \dots, \tau_\ell = \tau' \rangle$ between τ and τ' in $H(B)$. Let $C'_1 = C, C'_\ell = C'$ and, for $i = 2, \dots, \ell - 1$, choose C'_i as an arbitrary cycle from B that contains some vertex of type τ_i . Moreover, let c_i and t_i be the colors of the vertex and the token of type τ_i , respectively.

For $i = 1, \dots, \ell - 1$, C_i contains a vertex of color $t_i = c_{i+1}$ and so does C_{i+1} . Hence the two edges (C_i, c_{i+1}) and (c_{i+1}, C_{i+1}) form a walk W_i between C_i and C_{i+1} in H' . We choose W as the composition of all walks $W_1, \dots, W_{\ell-1}$, in this order. \blacktriangleleft

Next, we compute a spanning tree T of H rooted in an arbitrary cycle-vertex (T exists by Lemma 19) and we iteratively (i) locate the deepest color vertex c ; (ii) merge all its neighboring cycles (i.e., the parent and all the children of C) in T into a single cycle C' by repeatedly performing merge operations; and (iii) delete c and all its children, and replace the former parent of c with a new cycle-vertex corresponding to C' . We stop this process when T contains a single cycle C^* as its root, and we return C^* .

Notice that the time spent to process T is proportional to the number n_T of vertices in T . Indeed the color vertices c can be listed in order of depth in time $O(n_T)$ by a BFS visit of T , and step (ii) can be performed in time proportional to the number of neighbors of c by exploiting the fact that a generic edge (c, C) incident to c in T is labelled with a vertex of C having color (see also Lemma 3). The overall time spent is therefore $O(k + n_T) = O(k + n_B)$ since H (and hence T) contains at most k color-vertices and $n_B/2$ cycle-vertices.

Bottom-Up Rebalancing Binary Search Trees by Flipping a Coin

Gerth Stølting Brodal  

Department of Computer Science, Aarhus University, Denmark

Abstract

Rebalancing schemes for dynamic binary search trees are numerous in the literature, where the goal is to maintain trees of low height, either in the worst-case or expected sense. In this paper we study randomized rebalancing schemes for sequences of n insertions into an initially empty binary search tree, under the assumption that a tree only stores the elements and the tree structure without any additional balance information. Seidel (2009) presented a top-down randomized insertion algorithm, where insertions take expected $O(\lg^2 n)$ time, and the resulting trees have the same distribution as inserting a uniform random permutation into a binary search tree without rebalancing. Seidel states as an open problem if a similar result can be achieved with bottom-up insertions. In this paper we fail to answer this question.

We consider two simple canonical randomized bottom-up insertion algorithms on binary search trees, assuming that an insertion is given the position where to insert the next element. The subsequent rebalancing is performed bottom-up in expected $O(1)$ time, uses expected $O(1)$ random bits, performs at most two rotations, and the rotations appear with geometrically decreasing probability in the distance from the leaf. For some insertion sequences the expected depth of each node is proved to be $O(\lg n)$. On the negative side, we prove for both algorithms that there exist simple insertion sequences where the expected depth is $\Omega(n)$, i.e., the studied rebalancing schemes are *not* competitive with (most) other rebalancing schemes in the literature.

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

Keywords and phrases Binary search tree, insertions, random rebalancing

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.6

Related Version *Full paper:* <https://doi.org/10.48550/arXiv.2404.08287> [12]

Funding *Gerth Stølting Brodal:* Supported by Independent Research Fund Denmark, grant 9131-00113B.

1 Introduction

Binary search trees is one of the most fundamental data structures in computer science, dating back to the early 1960s, see, e.g., Windley (1960) [28] for an early description of binary search trees and Hibbard (1962) [20] for an analysis of random insertions and deletions. Knuth [21, page 453] gives a detailed overview of the early history of binary search trees, and Andersson *et al.* [7] an overview of later developments on balanced binary search trees.

When inserting new elements into the leaves of an unbalanced binary search tree the height of the tree might deteriorate, in the sense that it becomes super-logarithmic in the number of elements stored (see Figure 1). In the literature numerous rebalancing schemes have been presented guaranteeing logarithmic height: Some are deterministic with worst-case update bounds, like AVL-trees [1], red-black trees [19]; some deterministic with amortized bounds, like splay-trees [26] and scapegoat trees [5, 18]; and others are randomized, like treaps [25] and randomized binary search trees [22], just to mention a few.

In this paper we study simple randomized rebalancing schemes for sequences of insertions into an initially empty binary search tree. The goal of this paper is to study randomized rebalancing schemes under a set of constraints, and to study how good rebalancing schemes



© Gerth Stølting Brodal;

licensed under Creative Commons License CC-BY 4.0

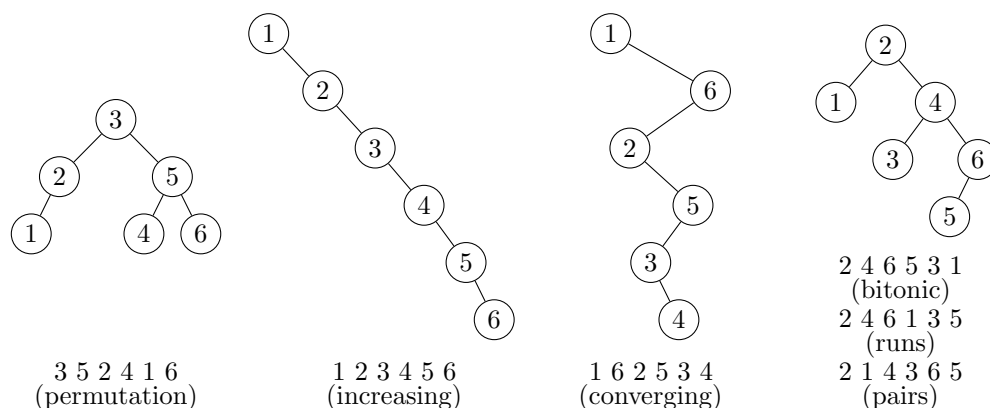
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 6; pp. 6:1–6:15

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Unbalanced binary search trees resulting from inserting permutations of $\{1, \dots, 6\}$. The insertion order is shown below the trees. The types of permutations are defined in Table 2.

can be achieved within these constraints. In general the proposed rebalancing schemes in Section 2 and Section 3 are *not* competitive with existing rebalancing schemes in the literature. The constraints we consider are the following:

1. The search tree should not store any balancing information, only the tree and the elements should be stored.
2. Insertions should perform limited restructuring, say, worst-case $O(1)$ rotations.
3. Most rotations should happen near the inserted elements.
4. Rebalancing should be based on local information (tree structure) at the insertion point only (e.g., without knowledge of n nor the current height of the tree).
5. Rebalancing should be performed in expected $O(1)$ time.
6. Rebalancing should use expected few random bits per insertion, say, expected $O(1)$ bits.
7. Each node should have low expected depth, ideally $O(\lg n)$.

The constraints are motivated by the properties of the randomized treaps [25] (but treaps need to store random priorities as balance information); that the random distribution of tree structures achieved by treaps can be achieved without storing balancing information [24] (but with slower insertions); and treaps can be the basis for efficient concurrent search trees [3].

1.1 Deterministic Previous Work

Red-black trees [19] are deterministic dynamic balanced binary search, with good amortized performance. They violate constraint (1), since each node is required to store a single bit of balance information, indicating if the node is red-black. But otherwise, red-black trees are guaranteed to have height $O(\lg n)$, insertions at a leaf can be performed in amortized $O(1)$ time and perform at most two rotations, i.e., red-black trees essentially satisfy constraints (2)–(7), if expected bounds are substituted by amortized bounds.

Brown [13, 14] showed how to encode a single bit of information in the internal nodes of a binary tree by considering “supernodes” consisting of pairs of consecutive elements arranged as parent-child pairs together with a pointer to an empty leaf between the two elements. Depending of the relative placement of the two elements the encoded bit can be decoded from the placement of the pointer to the empty leaf. Brown showed how to encode 2-3 trees [2, Chapter 4] using this technique, achieving balanced binary search trees storing no balance information and supporting insertions in worst-case $O(\lg n)$ time.

Splay trees [26] are the canonical deterministic amortized efficient dynamic binary search trees satisfying constraint (1), i.e., they do not store any additional information than the binary tree and the elements. Splay trees support insertions in amortized $O(\lg n)$ time, i.e., insertions are amortized efficient. The drawback of splay trees is that they do a significant amount of restructuring (memory updates) per insertion, since they rotate a constant fraction of the nodes on the path from the inserted element to the root. The number of rotations depends on what variant of splaying is applied, see [11, 26]. Albers and Karpinski [4] and Fürer [17] considered randomized variants of splaying to reduce the restructuring cost.

Scapegoat trees are another deterministic dynamic binary search tree with good amortized performance, independently discovered by Anderson [5, 6] and Galperin and Rivest [18]. Scapegoat trees achieve amortized $O(\lg n)$ insertions by maintaining the invariant that the height of a tree containing n elements is $O(\lg n)$. If an insertion causes the invariant to be violated, a local subtree is rebuilt into a perfectly balanced binary tree (in the worst-case this is the root and the full tree is rebuilt). A scapegoat tree only needs to store the number of elements n as a global integer value in addition to the binary tree and its elements.

1.2 Randomized Previous Work

Randomization and binary search trees can be addressed in two directions in the context of insertions: Either insertions are random (e.g., the insertion sequence is a random permutation) and we analyze the expected performance for a binary search tree with respect to the insertion distribution; or insertions can be arbitrary but the rebalancing of the binary search tree exploits random bits and we analyze the performance with respect to the random bits.

We call a search tree containing n elements inserted in random order without rebalancing a *random binary search tree*. A classic result on random binary search trees is that each element in the resulting tree has expected depth at most $2 \ln n + O(1)$ [10, 20, 28]. The important property is that the root equals each of the n elements with probability exactly $1/n$, and this property again holds recursively for the left and right subtrees. A consequence is that all valid search trees with $n \geq 3$ elements do not have the same probability. See Panny [23] for a history on deletions in random binary search trees.

The structure of random binary search trees has been used as guideline to construct different dynamic binary search trees where at each point of time the probability of a given tree equals that of a random binary search tree [22, 24, 25].

Aragon and Seidel [25] introduced the *treap*, that with each element stores an independently uniformly assigned random priority in the range $[0, 1]$, and organizes the search tree such that priorities satisfy heap order [27], i.e., the root stores the element with minimum priority. Since each element has probability $1/n$ to have the smallest priority, all elements have probability $1/n$ to be at the root, the property required to be random search trees. Insertions into treaps can be done by bottom-up rotations in expected $O(1)$ time and $O(1)$ rotations using $O(1)$ random bits. After n insertions into a treap the expected shape of the treap equals a random binary search tree. Blelloch and Reid-Miller [9] considered parallel algorithms for the set operations union, intersection and difference on treaps. Alapati *et al.* [3] considered concurrent insertions and deletions into treaps.

Martínez and Roura [22] presented a different approach denoted *randomized binary search trees* to achieve the structure of a random binary search tree after n insertions. Their approach stores at each node the size of the subtree rooted at the node, and insertions are performed top-down in expected $O(\lg n)$ time, where the inserted element is inserted in a node with probability $\frac{1}{k+1}$, where k is the size of the current subtree rooted at the node (see [22] for details). Each insertion requires expected $O(\lg n)$ random integers in the range $1, \dots, n + 1$.

■ **Table 1** Rebalancing cost of selected binary search trees. Space refers to the space required for additional balance information. O_A , O_E and O denote amortized, expected and worst-case bounds, respectively. *Our results do not guarantee (expected) logarithmic depth.

	Time	Rotations	Random bits	Space (bits)
Red-black tree [19]	$O_A(1)$	$O(1)$	0	1
Encoded 2-3 trees [13, 14]	$O(\lg n)$	$O(\lg n)$	0	0
Splay trees	$O_A(\lg n)$	$O_A(\lg n)$	0	0
Treaps [8]	$O_E(1)$	$O_E(1)$	$O_E(1)$	$O_E(1)$
Randomized BST [22]	$O_E(\lg n)$	$O_E(1)$	$O_E(\lg^2 n)$	$O(\lg n)$
Seidel [24]	$O_E(\lg^2 n)$	$O_E(1)$	$O_E(\lg^3 n)$	0
<i>Algorithms in this paper*</i>	$O_E(1)$	$O(1)$	$O_E(1)$	0

■ **Table 2** Different sequences of length n (assuming n is even) considered in this paper.

permutation	random permutation of $1, \dots, n$
increasing	$1, 2, 3, \dots, n$
decreasing	$n, n-1, n-2, \dots, 1$
converging	$1, n, 2, n-1, 3, n-2, \dots, \frac{n}{2}, \frac{n}{2}+1$
pairs	$2, 1, 4, 3, 6, 5, \dots, n, n-1$
bitonic	$2, 4, 6, \dots, n-2, n, n-1, n-3, \dots, 5, 3, 1$
runs	$2, 4, 6, \dots, n-2, n, 1, 3, 5, \dots, n-3, n-1$

Seidel [24] gave a unified presentation of [25] and [22], emphasizing the similarity of the two approaches, and describes a variation of [22] that avoids storing subtree sizes, but the insertion time increases to expected $O(\lg^2 n)$ and uses $O(\lg^3 n)$ random bits. Seidel states it as an open problem if there exists a bottom-up rebalancing algorithm that without storing any balancing information can obtain the structure of random binary search trees.

1.3 Results

We consider two very simple algorithms to rebalance a binary search tree after a new element has been inserted at a leaf. Our aim is to try to meet the requirements (1)–(7), and in particular not the ambitious goal of having the same distribution as random binary search trees. Both our algorithms repeatedly flip a coin until it comes out head. Whenever the coin shows tail (with probability p) we move to the parent of the current node (starting at the new leaf, and if we reach the root, the rebalancing terminates without modifying the tree). When the coin shows head, the first algorithm (REBALANCEZIG in Algorithm 1) rotates the current node up, and the rebalancing terminates. The second algorithm (REBALANCEZIGZAG in Algorithm 2) does one or two rotations, depending on if it is a zig-zag or zig-zig case (inspired by the rebalancing rules of splay trees).

Ignoring the depths of the nodes of the resulting trees, we immediately have the following fact, since the coin tosses are independent Bernoulli trials, with an expected $O(1)$ coin tosses necessary (assuming a coin with constant non-zero probability for head). It follows that both algorithms satisfy our constraints (1)–(6).

► **Fact 1.** *The rebalancing done by REBALANCEZIG with $0 \leq p < 1$ takes expected $O(1)$ time, uses expected $O(1)$ random bits, and performs at most one rotation. REBALANCEZIGZAG performs at most two rotations, but otherwise with identical performance.*

To study to what extent the proposed algorithms achieve logarithmic depth of the nodes, constraint (7), we study the behavior of the algorithms on the insertion sequences listed in Table 2.

In Section 2 we study REBALANCEZIG. Our first result is that REBALANCEZIG is sufficient to achieve a balanced binary search tree when inserting elements in increasing order (and symmetrically decreasing order). The below theorem follows from Lemma 9.

► **Theorem 1.** *Executing REBALANCEZIG with $0 < p < 1$ on an increasing and decreasing sequence of n insertions results in a binary search tree, where each node has expected depth $O(\lg n)$.*

We then show that there are very simple insertion sequences where REBALANCEZIG fails to achieve a balanced tree. We denote the sequence $1, n, 2, n-1, \dots, n/2, n/2+1$ the *converging* sequence. The below theorem follows from Lemma 11.

► **Theorem 2.** *Executing REBALANCEZIG with $0 \leq p \leq 1$ on a converging sequence of n insertions results in a binary search tree with expected average node depth $\Theta(n)$.*

Another sequence where REBALANCEZIG fails to achieve logarithmic depth is on the *pairs* sequence $2, 1, 4, 3, 6, 5, \dots, n, n-1$, provided $p \neq \frac{1}{2}$. The following theorem restates Lemma 12.

► **Theorem 3.** *Executing REBALANCEZIG with $0 \leq p < \frac{1}{2}$ or $\frac{1}{2} < p \leq 1$ on a pairs sequence of n insertions results in a binary search tree with expected average node depth $\Theta(n)$.*

For $p = \frac{1}{2}$ algorithm REBALANCEZIG behaves significantly better on pairs sequences. We conjecture the expected average node depth to be $O(\sqrt{n})$.

In Section 3 we study the second algorithm REBALANCEZIGZAG. For increasing (decreasing) sequences, where the new leaf is always the rightmost (leftmost) node in the tree, REBALANCEZIGZAG is essentially identical to REBALANCEZIG, i.e., our result for increasing and decreasing sequences for REBALANCEZIG immediately carries over to REBALANCEZIGZAG. The following theorem restates Corollary 14.

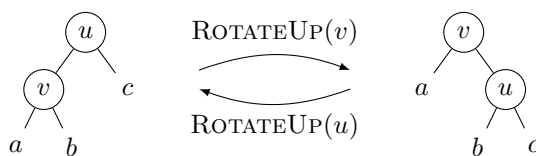
► **Theorem 4.** *Executing REBALANCEZIGZAG with $0 < p < 1$ on an increasing or decreasing sequence of n insertions results in a binary search tree where each nodes has expected depth $O(\lg n)$.*

In Section 3.2 we generalize the proof to also hold for the convergent sequence for REBALANCEZIGZAG (where REBALANCEZIG failed to achieve logarithmic depth), and more generally finger sequences, where the next insertion always becomes the successor or predecessor of the last insertion. The following theorem restates Lemma 15.

► **Theorem 5.** *Executing REBALANCEZIGZAG with $\frac{1}{2}(\sqrt{5}-1) < p < 1$ on a convergent or finger sequence of n insertions results in a binary search tree where each nodes has expected depth $O(\lg n)$.*

On the negative side, we prove that REBALANCEZIGZAG fails to achieve balanced trees for pairs sequences, for all $0 \leq p \leq 1$. The following theorem restates Lemma 16.

► **Theorem 6.** *Executing REBALANCEZIGZAG with $0 \leq p \leq 1$ on a pairs sequence of n insertions results in a binary search tree with expected average node depth $\Theta(n)$.*



■ **Figure 2** (Left-to-right) The right rotation of u rotates v up; note that a is moved one level up in the tree, b remains at the same level, and c is moved down one level. (Right-to-left) the left rotation of v rotates u up.

We complement our theoretical findings by an experimental evaluation of REBALANCEZIG and REBALANCEZIGZAG in Section 5, supporting our theoretical findings. We briefly discuss random permutations in Section 4, but otherwise only have an experimental evaluation of the rebalancing algorithms on inserting random permutations.

If the insights from our results can lead to an improved bottom-up randomized rebalancing scheme for binary search trees remains open.

1.4 Notation and Terminology

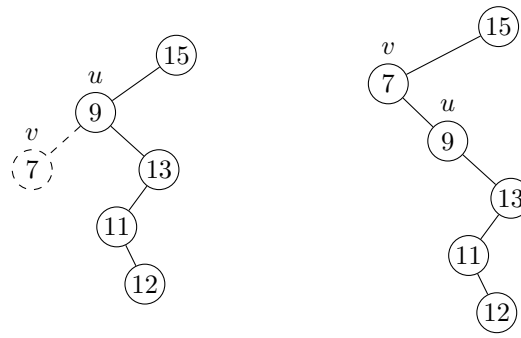
Throughout this paper n denotes the number nodes in a binary search tree, i.e., the number of insertions performed. The *depth* of a node is the number of edges from the node to the root, i.e., the root has depth zero. The height of a tree is the maximum depth of a node. Rebalancing will be done by the standard primitives of left and right *rotations*, see Figure 2. Both rotate up a node one level in the tree. Since our updates are performed bottom-up, we assume that each node v stores an element and pointers to its left child $v.l$, right child $v.r$, and parent $v.p$ (possibly equal to NIL if no such node exists). We let $\lg n$ and $\ln n$ denote the binary and natural logarithm of n , respectively.

2 Algorithm REBALANCEZIG

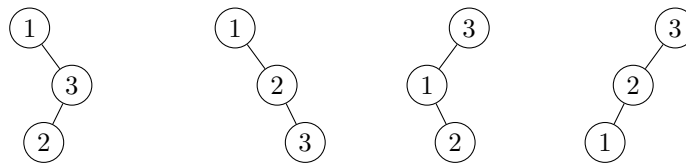
In this section we show that on increasing and decreasing sequences applying algorithm REBALANCEZIG results in binary search trees where each node has expected depth $O(\lg n)$. We also show that on the converging and pairs sequences ($p \neq \frac{1}{2}$) the expected average node depth is linear.

Assume a new element has been inserted into a binary search tree as a new leaf v (before rebalancing the tree). Algorithm REBALANCEZIG rebalances the tree as follows: After inserting the new node v , we flip a coin, that with probability p is tail and $1 - p$ is head, for a constant $0 \leq p \leq 1$. If the coin is head, we rotate v up, and the insertion terminates. Otherwise, we recursively move to the parent, i.e., set $v \leftarrow v.p$, flip a coin, and rotate the parent up if the coin is head, or continue recursively at the grandparent if the coin is tail. The rebalancing terminates when the first rotation has been performed or when we reach the root. See the pseudo-code in Algorithm 1.

Note that $p = 1$ is the special case where we always move up and never rotate, i.e., identical to insertions without rebalancing. When $p = 0$ the new node is always the node rotated up. In this case the tree is a single path containing all n nodes, since inserting a node v as a child of u on the path, rotating up v causes v to be inserted into the path as the parent u . See Figure 3. In the following we assume $0 < p < 1$. That REBALANCEZIG can not achieve the same tree distribution as random binary search trees (like treaps and randomized binary search trees do) follows by the example in Figure 4.



■ **Figure 3** REBALANCEZIG with $p = 0$ always rotates up the inserted node, and maintains the invariant that the tree is a single path. (left) insertion point of 7; (right) 7 is rotated up onto the path.



■ **Figure 4** Binary search trees resulting from inserting the sequence 1, 3, 2 using REBALANCEZIG. Each of the four search trees has probability $1/4$ when $p = 1/2$. Note that the perfect balanced binary search tree on three nodes cannot be achieved by REBALANCEZIG on this insertion sequence.

2.1 Increasing Sequences

We let the *right height* of a tree denote the depth of the rightmost node in the tree. When inserting elements in increasing order the new element will always be inserted as a rightmost node in the tree, i.e., the right height increases by one before any rebalancing is performed. If REBALANCEZIG performs a (left) rotation on the rightmost path, the right height is reduced by one again, and the right height does not change by the insertion.

► **Lemma 7.** *If the right height is d before inserting a new rightmost node and applying REBALANCEZIG, then afterwards the right height is d or $d + 1$ with probability $1 - p^{d+1}$ or p^{d+1} , respectively.*

Proof. The right height increases if and only if no rotation is performed, i.e., we reach the root because the coin $d+1$ times in a row shows tail, which happens with probability p^{d+1} . ◀

► **Lemma 8.** *After inserting n elements in increasing order using REBALANCEZIG, the right height is at most $\lceil (c + 1) \cdot \log_{1/p} n \rceil$ with probability $1 - 1/n^c$, for any constant $c > 0$.*

Proof. Assume that the right height at some point of time during the insertions is $d = c' \cdot \lg n$. By Lemma 7, the probability that the next insertion increases the right height is p^{d+1} . The probability of any of the at most n remaining insertions increases the right height is at most $np^{d+1} = np^{1+c' \lg n} \leq np^{c' \lg n} = n^{1+c' \lg p} \leq n^{-c}$ for $c' \geq -(c + 1)/\lg p$. It follows that the right height after n insertions is at most $\lceil -(c + 1)/\lg p \cdot \lg n \rceil = \lceil (c + 1) \log_{1/p} n \rceil$ with probability $1 - 1/n^c$. ◀

Lemma 8 gives a high probability guarantee on the expected depth of the nodes on the rightmost path. We now prove an expected depth for all nodes in the tree.

■ **Algorithm 1** REBALANCEZIG(v).

```

while  $v.p \neq \text{NIL}$  and coin flip is tail do
   $v \leftarrow v.p$ 
if  $v.p \neq \text{NIL}$  then
  ROTATEUP( $v$ )

```

► **Lemma 9.** *After inserting n elements in increasing order using REBALANCEZIG, with $0 < p < 1$, each node has expected depth $O(1/p \cdot \log_{1/p} n)$.*

Proof. Consider an element inserted in a node v during the sequence of insertions. The element goes through the following five phases:

1. The element is not yet inserted. The less than n elements inserted before v create a tree with right height $O(\log_{1/p} n)$ with high probability (Lemma 8).
2. v is created as the rightmost node with depth $O(\log_{1/p} n)$ with high probability.
3. v remains on the rightmost path for the subsequent insertions until v 's right child u becomes the target for being rotated up. An insertion that rotates up v or an ancestor of v will decrease the depth of v . Rotations below u do not change the depth of v .
4. v is moved out of the rightmost path by rotating up the right child u of v , making v the left child of u . This increases the depth of v by one.
5. v is in the left subtree of a node u on the rightmost path (u can change over the subsequent insertions, but the depth of the branching node u can never increase). Each insertion can affect the position of v by the rotation performed:
 - a. No rotation is performed and the path from the root through u to v is unchanged. The right height increases by one.
 - b. An ancestor of u is rotated up, where u remains the branching node to v , the depths of both u and v decrease by one.
 - c. u is rotated up, where u remains the branching node to v , the depth of u decreases by one, and the depth of v stays unchanged.
 - d. Rotating up the right child w of u increases the depth of v by one and w replaces u as the branching node to v on the rightmost path (with the same depth as u had before the rotation).
 - e. Rotations below the right child of u do not change the path from the root through u and v .

From Lemma 8 it follows that the depth of v after phases 1–4 is $O(\log_{1/p} n)$, with high probability. Cases 5a, 5b, 5c and 5e do not increase the depth of v . What remains is to bound the expected number of times case 5d occurs and increases the depth of v by one. For case 5d to happen, a coin must have been flipped at w showing head. Over all insertions in phase 5, a subsequence of the insertions flips a coin at the child w of the current branching node u . If an insertion flips a coin at w , there are two cases: The coin shows head with probability $1 - p$ and case 5d happens; or the coin shows tail with probability p , and case 5a, 5b or 5c happens. Since cases 5a, 5b and 5c at most happens $O(\log_{1/p} n)$ times with high probability (case 5a increases the right height; cases 5b and 5c decrease the depth of the branching node u to v), i.e., the coin shows tail at w at most $O(\log_{1/p} n)$ times with high probability. Since the expected number of times we need to flip a coin to get a tail is $1/p$, the expected number of times we flip a coin at the right child w of the branching node u to v is $O(1/p \cdot \log_{1/p} n)$, with high probability. This is then also an upper bound on the expected number of times the depth of v can increase by case 5d. It follows that with high probability, the expected depth of v is $O(\log_{1/p} n + 1/p \cdot \log_{1/p} n) = O(1/p \cdot \log_{1/p} n)$. Since the depth of v is at most $n - 1$, the expected depth of v is $O(1/p \cdot \log_{1/p} n)$ after all insertions is (without the high probability assumption). ◀

The following lemma states that the node rotated up is expected to be close to the inserted leaf, and states the number of coin flips as a function of the tail probability p .

► **Lemma 10.** *The distance from the inserted node to the node rotated up by REBALANCEZIG is expected at most $\frac{p}{1-p}$. The number of coin flips is at most $\frac{1}{1-p}$.*

Proof. The expected distance to the node rotated up is at most

$$\sum_{d=0}^{\infty} d(1-p)p^d = (1-p) \sum_{d=0}^{\infty} dp^d = (1-p) \frac{p}{(1-p)^2} = \frac{p}{1-p},$$

since the new node inserted (at distance 0) is rotated up with probability $1-p$, its parent with probability $p(1-p)$, etc. The d 'th ancestor is rotated up with probability $(1-p)p^d$, provided a d 'th ancestor exists. The number of coin flips is one plus the distance, i.e., at most $1 + \frac{p}{1-p} = \frac{1}{1-p}$. ◀

Note that inserting n elements in decreasing order is the symmetric case to the increasing order where the new node is inserted as the leftmost node, and at most one right rotation is performed on the leftmost path. It follows that Lemmas 8 and 10 also apply to decreasing sequences, by replacing the rightmost path by the leftmost path in the arguments,

2.2 Converging Sequences

Assume we have a *finger* into the sorted inserted sequence of elements pointing to the most recently inserted element, and whenever a new element is inserted it must be the new predecessor or successor of the element at the finger, i.e., we can only insert elements that are in the interval defined by the current predecessor and successor of the element at the finger. We call sequences satisfying this property for *finger insertions*. Increasing and decreasing sequences are examples of finger insertions.

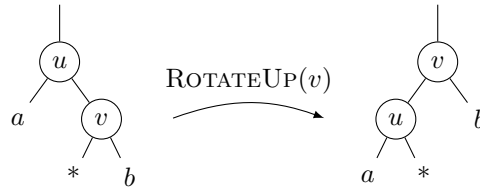
The following sequence of insertions also consists of finger insertions (for simplicity, we assume n is even). We denote this insertion sequence the *converging sequence*. See Figure 1 for an illustration of $n = 6$.

$$1, n, 2, n-1, 3, n-2, 4, n-3, \dots, n/2, n/2+1$$

As can be seen in the experimental evaluation in Figure 7(a,b), the average depth appears to be linear for the nodes in a binary search tree resulting from applying REBALANCEZIG to the converging sequence. The below lemma confirms this.

► **Lemma 11.** *Executing REBALANCEZIG with insertions $1, n, 2, n-1, \dots, n/2, n/2+1$, assuming n even, results in a binary search tree with an (external) leaf with expected depth at least $\frac{p(1-p)}{2}n$, for $0 < p < 1$. For $p = 0$ and $p = 1$ the resulting tree is a single path.*

Proof. For $p = 1$ we do no rebalancing, and the converging sequence results in a single path (see Figure 1). For $p = 0$, the new node is always rotated up onto an existing single path. We let the *insertion point* denote the (external) leaf, where the next insertion is going to create a node v . Since the converging sequence is a sequence of finger insertions, the next insertion point is always a child of the created node v (before rebalancing), i.e., each insertion increases the depth of insertion point by one (before rebalancing). Unfortunately, the rebalancing done by REBALANCEZIG does not always decrease the depth one. Consider inserting i , where $1 \leq i \leq n/2$, that creates a node u followed by inserting $n+1-i$ that creates a node v . Assume that the rebalancing after inserting i does not rotate up u (but possibly an ancestor



■ **Figure 5** Bad zag-zig case for $\text{REBALANCEZIG}(v)$, where rotating up v does not decrease the depth of the insertion point $*$.

of u has been rotated up, and possibly decreasing the depth of the insertion point by one again), and the insertion of $n + 1 - i$ causes v to be rotated up. This case is shown in Figure 5. We borrow the terminology from splay trees that if a path branches left, we say it is a zig, and if it branches right it is a zag. If a left branch is followed by a right branch it is a zig-zag. We denote the case in Figure 5 the zag-zig case. In this case the insertion point moves from being a child of v to being a child of u , but retains the same depth, i.e., the insertions and REBALANCEZIG increased the depth of the insertion point by one. The probability that u was not rotated up is p and the probability that v is rotated up is $1 - p$, i.e., the insertion of i and $n + 1 - i$ causes the depth of the insertion point to increase by one with probability at least $p(1 - p)$. It follows that after the insertion of all n elements, the expected depth of the insertion point is at least $\frac{1}{2}np(1 - p)$. ◀

If an external leaf has depth d , then the sum of the depths of the d internal nodes on the path to the external leaf is $\sum_{i=0}^{d-1} i = \frac{1}{2}d(d - 1)$. The average depth of all nodes in the tree is then at least $\frac{1}{2}d(d - 1)/n$, and Theorem 2 follows from Lemma 11.

It should be noted that the rotation after an insertion can happen higher in the tree, where there is also a zag-zig case or symmetric zig-zag case, where REBALANCEZIG also fails to decrease the depth of the insertion point after the insertion. This explains the gap between the experimental constant observed in Figure 7 and the theoretical analysis.

2.3 Pairs Sequences

The pairs sequence consists of $2, 1, 4, 3, 6, 5, \dots, n, n - 1$. It is essentially an increasing sequence, with pairs $2i - 1$ and $2i$ swapped. Pair sequences are not finger sequences. Interestingly, experiments show that REBALANCEZIG is challenged by this sequence. In our experimental evaluation, Figure 7(a), it appears that $p = \frac{1}{2}$ is a local minima for the average node depth when rebalancing pairs sequences using REBALANCEZIG , with increased average node depth for both p smaller than and larger than $\frac{1}{2}$. In [12] we prove the following lemma.

► **Lemma 12.** *Applying REBALANCEZIG to the pairs sequence with n elements, for n even and constant $p \neq \frac{1}{2}$, $0 \leq p \leq 1$, the resulting tree has expected average node depth $\Theta(n)$.*

The last inserted element has expected depth $\Theta(n)$ for $0 < p < \frac{1}{2}$ and $O(1)$ for $\frac{1}{2} < p < 1$, so Lemma 12 does not give any bounds on the expected depth of specific elements. Lemma 12 addresses pairs sequences for $p \neq \frac{1}{2}$, where the expected average node depth is linear. For $p = \frac{1}{2}$ we give the following conjecture, stating that the complexity is significantly different. See [12] for an experimental and theoretical motivation of the conjecture.

► **Conjecture 13.** *Applying REBALANCEZIG to the pairs sequence with n elements, for n even and $p = \frac{1}{2}$, the resulting tree has expected average node depth $O(\sqrt{n})$.*

3 Algorithm REBALANCEZIGZAG

To address the shortcomings of algorithm REBALANCEZIG in the case where v is in a zig-zag or zag-zig state, we borrow terminology from splay trees [26], and apply the zig-zag transformation to the tree (see Figure 6(right) and [26, Figure 3]) by rotating up v twice. In the zig-zig case, instead of rotating up v , we rotate up the parent of v (see Figure 6(left)). These two transformations ensure that everything in the subtree of v is moved one level up in the tree when applying the transformation at node v . The pseudo-code for algorithm REBALANCEZIGZAG is shown in Algorithm 2.

Algorithm 2 REBALANCEZIGZAG(v).

```

while  $v.p \neq \text{NIL}$  and coin flip is tail do
   $v \leftarrow v.p$ 
if  $v.p \neq \text{NIL}$  and  $v.p.p \neq \text{NIL}$  then
  if ( $v = v.p.l$  and  $v.p = v.p.p.l$ ) or ( $v = v.p.r$  and  $v.p = v.p.p.r$ ) then
    ROTATEUP( $v.p$ ) ▷ zig-zig or zag-zag case
  else
    ROTATEUP( $v$ ) ▷ zig-zag or zag-zig case
    ROTATEUP( $v$ )

```

3.1 Increasing Sequences

REBALANCEZIGZAG handles increasing and decreasing sequences identical to REBALANCEZIG, except that rotations happen one level higher, i.e., the trees are identical if ignoring the rightmost inserted node. Equivalently, this corresponds to inserting the next element without rebalancing, and first performing the rebalancing just before inserting the next element. From Theorem 1 we have the following corollary.

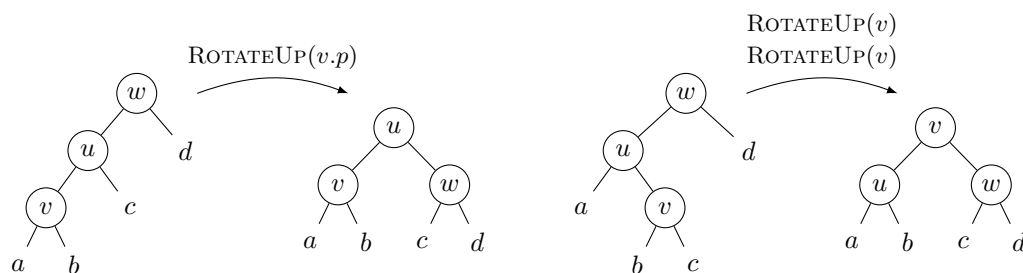
► **Corollary 14.** *For $0 < p < 1$, after inserting n elements in increasing or decreasing order using REBALANCEZIGZAG, each node has expected depth $O(1/p \cdot \log_{1/p} n)$.*

3.2 Finger Sequences

We will prove that using REBALANCEZIGZAG to rebalance finger sequences (like increasing, decreasing and converging sequences), ensures that the resulting tree is expected to be balanced, for p sufficiently large. Recall that a finger sequence is defined such that the next element is always the immediate predecessor or successor of the most recently inserted element among all elements inserted so far. In an unbalanced search tree, this means that the next node will be a (left or right) child of the most recently inserted node, i.e., the resulting tree is always a path. We denote the external leaf where to create the next node the *insertion point*. The crucial property of the restructuring done by REBALANCEZIGZAG in Figure 6 is that if the insertion point is at an external leaf in the subtree rooted at v before the rotation, then the depth of this external leaf is reduced by exactly one in both cases.

► **Lemma 15.** *After inserting a finger sequence with n elements using REBALANCEZIGZAG, each node has expected depth $O(\lg n)$, for $\frac{1}{2}(\sqrt{5} - 1) < p < 1$.*

Proof. The proof follows the same idea as in Section 2.1 for the analysis of REBALANCEZIG on increasing sequences. Instead of right height we consider *insertion depth*, i.e., the depth d of the parent node of the insertion point. See [12] for proof details. ◀



■ **Figure 6** The rebalancing performed by algorithm $\text{REBALANCEZIGZAG}(v)$ in (left) zig-zig case and (right) zig-zag case.

In our experiments, see Figure 7(c), it shows up that REBALANCEZIGZAG performs better for $p > \frac{1}{2}$ than $p < \frac{1}{2}$ on the converging sequence, that is an example of a finger sequence. We leave open the question what the dependency on p is for REBALANCEZIGZAG for $0 < p \leq \frac{1}{2}(\sqrt{5} - 1)$.

3.3 Pairs Sequences

While REBALANCEZIGZAG achieves better average node depth on converging sequences compared to REBALANCEZIG , it fails on pairs sequences, where the expected average node depth is linear for all values $0 \leq p \leq 1$ (for $p = \frac{1}{2}$ this is worse than REBALANCEZIG , if our conjecture turns out to be true). In [12] we prove the following lemma.

► **Lemma 16.** *Applying REBALANCEZIGZAG to the pairs sequence with n elements, for n even and $0 \leq p \leq 1$, the resulting tree has expected average node depth $\Theta(n)$.*

4 Random Permutations

We do not prove anything for random permutations, for neither REBALANCEZIG nor REBALANCEZIGZAG . If $p = 1$ no rebalancing is performed, and it is known that the expected depth of a node is $O(\lg n)$ [10, 20, 28], whereas for $p = 0$, a rotation is always performed at the inserted leaf, and the tree will always be a single path. How exactly the average node depth depends on p is an open problem. In the experiments, see [12], it appears that REBALANCEZIGZAG is “about” logarithmic for $p \geq 0.7$.

5 Experimental Evaluation

In this section we present an experimental evaluation of our algorithms REBALANCEZIG and REBALANCEZIGZAG . See [12] for more experimental results.

We implemented the algorithms in Python 3.12, and ran the algorithms with different choices of p on the types of insertion sequences listed in Table 2 with sequence lengths being powers of two. Each data point in Figure 7 is the average over 25 runs. The increasing, decreasing and converging sequences are examples of finger insertions. Inserting the pairs, bitonic and runs sequences into a search tree without rebalancing result in identical search trees (see Figure 1). Note that the first half of the bitonic sequence is an increasing sequence, whereas the second part evenly distributes the remaining elements into the created leaves right-to-left. The runs sequences is identical to the bitonic sequences, except that the second part is performed left-to-right.

Figure 7 shows our main experimental findings. It shows the resulting average node depths of running the algorithms on the different types of insertion sequences from Table 2 with insertion sequences of length between two and 1024 and various values of p in the range zero to one. Note that for a path with n nodes, the root has depth 0 and the bottommost node depth $n - 1$, i.e., the average depth is $\frac{1}{n} \sum_{d=0}^{n-1} d = \frac{1}{n} \cdot \frac{n(n-1)}{2} = \frac{n-1}{2}$. For $n = 1024$, an average node depth of 511.5 implies that the tree is a path. In Figure 7(a) this explains why all curves share the top-left point, since REBALANCEZIG always generates a path when $p = 0$, independent of the insertion sequence, as discussed in Section 2. In Figure 7(left) the rightmost data point ($p = 1$) for random permutations corresponds to the average node depth in unbalanced binary search trees. Note that the pairs, bitonic and runs insertion sequences end up with different average node depth characteristics for each of the three algorithms (dashed curves in Figure 7), even that they would generate the same trees without rebalancing.

Figure 7(a, b) clearly shows that REBALANCEZIG has problems with the converging sequences (consistent with Theorem 2); Figure 7(c, d) that REBALANCEZIGZAG has problems with the pairs sequences (consistent with Theorem 6).

6 Conclusion and Open Problems

This paper leaves more open problems than it solves. None of the considered randomized rebalancing algorithms meets all conditions (1)–(7) introduced in Section 1. Inspired by a question raised by Seidel [24], we considered bottom-up randomized rebalancing schemes for binary search trees without storing any balance information. We studied randomized rebalancing strategies, inspired by the rebalancing primitives from splay trees [26]. They meet conditions (1)–(6), but fail to achieve logarithmic depth on all insertion sequences. In the experiments REBALANCEZIGZAG appears often to have the best performance, although it provably does not achieve expected logarithmic average depth for all insertion sequences. It remains an open problem if a randomized bottom-up rebalancing scheme exists that can guarantee expected logarithmic average node depths for all insertion sequences and satisfies requirements (1)–(6), or what the best depth guarantee can be given requirements (1)–(6), or how much these requirements need to be relaxed to enable expected logarithmic average node depths.

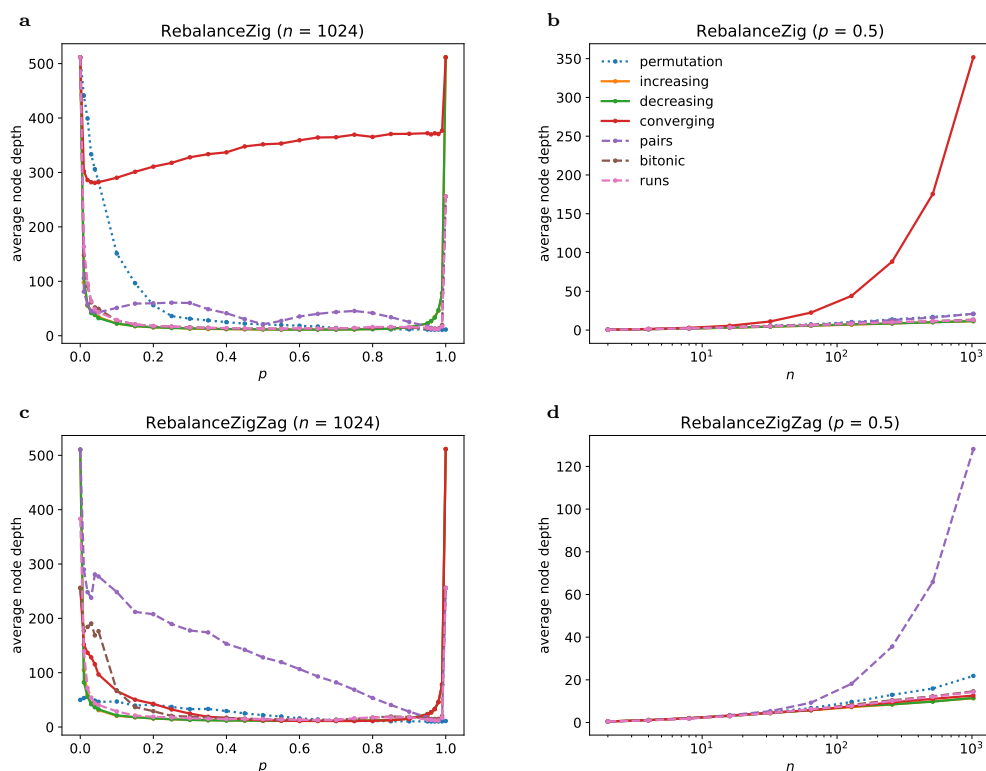
We did not consider deletions at all in this paper (see [15, 16, 23] for challenges on performing deletions in random binary search trees).

References

- 1 Georgy Adelson-Velsky and Evgenii Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences (in Russian)*, 146:263–266, 1962.
- 2 Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- 3 Praveen Alapati, Swamy Saranam, and Madhu Mutyam. Concurrent treaps. In Shadi Ibrahim, Kim-Kwang Raymond Choo, Zheng Yan, and Witold Pedrycz, editors, *Algorithms and Architectures for Parallel Processing - 17th International Conference, ICA3PP 2017, Helsinki, Finland, August 21-23, 2017, Proceedings*, volume 10393 of *Lecture Notes in Computer Science*, pages 776–790. Springer, 2017. doi:10.1007/978-3-319-65482-9_63.
- 4 Susanne Albers and Marek Karpinski. Randomized splay trees: Theoretical and experimental results. *Inf. Process. Lett.*, 81(4):213–221, 2002. doi:10.1016/S0020-0190(01)00230-7.
- 5 Arne Andersson. Improving partial rebuilding by using simple balance criteria. In Frank K. H. A. Dehne, Jörg-Rüdiger Sack, and Nicola Santoro, editors, *Algorithms and Data*

- Structures, Workshop WADS '89, Ottawa, Canada, August 17-19, 1989, Proceedings*, volume 382 of *Lecture Notes in Computer Science*, pages 393–402. Springer, 1989. doi:10.1007/3-540-51542-9_33.
- 6 Arne Andersson. General balanced trees. *J. Algorithms*, 30(1):1–18, 1999. doi:10.1006/JAGM.1998.0967.
 - 7 Arne Andersson, Rolf Fagerberg, and Kim S. Larsen. Balanced binary search trees. In Dinesh P. Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*, chapter 10. Chapman and Hall/CRC, 2004. doi:10.1201/9781420035179.CH10.
 - 8 Cecilia R. Aragon and Raimund Seidel. Randomized search trees. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 540–545. IEEE Computer Society, 1989. doi:10.1109/SFCS.1989.63531.
 - 9 Guy E. Blelloch and Margaret Reid-Miller. Fast set operations using treaps. In Gary L. Miller and Phillip B. Gibbons, editors, *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, pages 16–26. ACM, 1998. doi:10.1145/277651.277660.
 - 10 Andrew Donald Booth and Andrew J. T. Colin. On the efficiency of a new method of dictionary construction. *Information and Control*, 3(4):327–334, 1960. doi:10.1016/S0019-9958(60)90901-3.
 - 11 Gunnar Brinkmann and Jan Degraer and Karel De Loof. Rehabilitation of an unloved child: semi-splaying. *Software: Practice and Experience*, 39(1):33–45, 2009. doi:10.1002/SPE.886.
 - 12 Gerth Stølting Brodal. Bottom-up rebalancing binary search trees by flipping a coin. *CoRR*, abs/2404.xxxxx, 2024. doi:10.48550/ARXIV.2404.xxxxx.
 - 13 Mark R. Brown. A storage scheme for height-balanced trees. *Information Processing Letters*, 7(5):231–232, 1978. doi:10.1016/0020-0190(78)90005-4.
 - 14 Mark R. Brown. Addendum to “a storage scheme for height-balanced trees”. *Information Processing Letters*, 8(3):154–156, 1979. doi:10.1016/0020-0190(79)90009-7.
 - 15 Joseph C. Culberson and J. Ian Munro. Explaining the behaviour of binary search trees under prolonged updates: A model and simulations. *Comput. J.*, 32(1):68–75, 1989. doi:10.1093/COMJNL/32.1.68.
 - 16 Joseph C. Culberson and J. Ian Munro. Analysis of the standard deletion algorithms in exact fit domain binary search trees. *Algorithmica*, 5(3):295–311, 1990. doi:10.1007/BF01840390.
 - 17 Martin Fürer. Randomized splay trees. In Robert Endre Tarjan and Tandy J. Warnow, editors, *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms, 17-19 January 1999, Baltimore, Maryland, USA*, pages 903–904. ACM/SIAM, 1999. URL: <https://dl.acm.org/doi/10.5555/314500.315079>.
 - 18 Igal Galperin and Ronald L. Rivest. Scapegoat trees. In Vijaya Ramachandran, editor, *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, 25-27 January 1993, Austin, Texas, USA*, pages 165–174. ACM/SIAM, 1993. URL: <https://dl.acm.org/doi/10.5555/313559.313676>.
 - 19 Leonidas J. Guibas and Robert Sedgwick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*, pages 8–21. IEEE Computer Society, 1978. doi:10.1109/SFCS.1978.3.
 - 20 Thomas N. Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM*, 9(1):13–28, 1962. doi:10.1145/321105.321108.
 - 21 Donald Ervin Knuth. *The art of computer programming, Volume III, 2nd Edition*. Addison-Wesley, 1998.
 - 22 Conrado Martínez and Salvador Roura. Randomized binary search trees. *J. ACM*, 45(2):288–323, 1998. doi:10.1145/274787.274812.
 - 23 Wolfgang Panny. Deletions in random binary search trees: A story of errors. *Journal of Statistical Planning and Inference*, 140(8):2335–2345, 2010. doi:10.1016/j.jspi.2010.01.028.

- 24 Raimund Seidel. Maintaining ideally distributed random search trees without extra space. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *Lecture Notes in Computer Science*, pages 134–142. Springer, 2009. doi:10.1007/978-3-642-03456-5_9.
- 25 Raimund Seidel and Cecilia R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996. doi:10.1007/BF01940876.
- 26 Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi:10.1145/3828.3835.
- 27 J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964. doi:10.1145/512274.512284.
- 28 P. F. Windley. Trees, forests and rearranging. *The Computer Journal*, 3(2):84–88, 1960. doi:10.1093/COMJNL/3.2.84.



■ **Figure 7** (left) The average node depths in binary search trees created by REBALANCEZIG and REBALANCEZIGZAG, respectively, for various types of choices of p for insertion sequences of length 1024; (right) similar results but for fixed coin probability $p = \frac{1}{2}$, and sequence lengths in the range 1 to 1024. For $p = 0$ all rotations are performed at the inserted node (and REBALANCEZIG always creates a path), whereas no rotations are performed when $p = 1$, i.e., unbalanced binary search trees.

Physical Ring Signature

Xavier Bultel 

INSA Centre Val de Loire, Laboratoire d'informatique fondamentale d'Orléans, Bourges, France

Abstract

Ring signatures allow members of a group (called *ring*) to sign a message anonymously within the group, which is chosen ad hoc at the time of signing (the members do not need to have interacted before). In this paper, we propose a physical version of ring signatures. Our signature is based on one-out-of-many signatures, a method used in many real cryptographic ring signatures. It consists of boxes containing coins locked with padlocks that can only be opened by a particular group member. To sign a message, a group member shakes the boxes of the other members of the group so that the coins are in a random state (“heads” or “tails”, corresponding to bits 0 and 1), and opens their box to arrange the coins so that the exclusive “or” of the coins corresponds to the bits of the message they wish to sign. We present a prototype that can be used with coins, or with dice for messages encoded in larger (non-binary) alphabets. We suggest that this system can be used to explain ring signatures to the general public in a fun way. Finally, we propose a *semi-formal* analysis of the security of our signature based on real cryptographic security proofs.

2012 ACM Subject Classification Security and privacy → Public key (asymmetric) techniques

Keywords and phrases Physical Cryptography, Ring Signature, Anonymity

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.7

Funding ANR France 2030 “CyberINSA” (ANR-23-CMAS-0019).

1 Introduction

The signature is a fundamental primitive of public key cryptography that allows the owner of a secret key to sign messages in such a way that anyone can verify the signature using a public key. In some cases, it may be useful to allow group members to sign on behalf of the group without revealing their personal identity. A simple solution is to agree a priori on a public and secret key pair within the group, but this solution does not allow the signer to dynamically choose group members without consulting them when signing the message.

In 2001, Rivest, Shamir, and Tauman introduced ring signatures in their seminal paper “How to Leak a Secret” [13]. In this primitive, a user generates a signature on behalf of the group using their own secret key and the public keys of the group members (which the user can select when signing). The term “ring” refers to the method of signing used in this pioneering paper: the signer generates a chain of values depending on the message, using successively the public keys of all the group members except their own to encrypt the values. Then, using their own secret key, the signer decrypts the last value in the chain to append it to the beginning of the chain, thus closing the chain in a ring. To verify the signature, a user verifies that the ring is correct by reproducing the successive encryptions with the group members public keys, but cannot guess with which key the ring has been closed.

Ring signatures have always been of great interest because of their relevance to real-world problems, both technical and societal. The first motivation for ring signatures is to protect whistleblowers [13]. For instance, an employee of a company with illegal practices could, if each employee had a public key, expose those practices by signing as a member of the company, but without revealing their exact identity. Less directly, ring signatures have been used in many protocols, such as e-voting and e-cash [17], to guarantee anonymous membership. More recently, they have been used to anonymize certain actions on the blockchain [15], and to prevent transactions in the Monero cryptocurrency from being traced [16].



© Xavier Bultel;
licensed under Creative Commons License CC-BY 4.0
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 7; pp. 7:1–7:18



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

7:2 Physical Ring Signature

This primitive is therefore fundamental to many tools designed for a wide audience without advanced computer or mathematical skills. We believe that the use of security technology is only possible if users trust it, which is only possible if they feel that they have understood how their data is being processed and protected. It is thus necessary to find straightforward and convincing ways to explain the mechanisms used in ring signatures in order to inform and reassure the people who use its applications.

In this paper we propose a physical ring signature construction based on everyday objects such as boxes, padlocks, coins, dice, and glass. The actions to be performed are simple and consist of opening and closing the padlocks on the boxes, shaking the boxes, or looking through the glass to see the value of the coins and dice. Some basic calculations are also required, such as adding small integers, which can be done with a calculator. The overall look of the device is intriguing, and it is fun and easy enough for children to use. We believe that this playful aspect makes it an accessible tool for popularising the concept of ring signatures to the general public.

Technical Overview of our Contributions

Our signature mechanism is based on one-out-of-many signatures [1], which are themselves inspired by proofs of partial knowledge [6]. This general paradigm has been widely used to construct ring signatures, so the mechanism of our physical ring signature gives an accurate idea of how cryptographic ring signatures are actually designed. In a nutshell, this paradigm is based on message-randomizable signatures, *i.e.*, signature schemes where it is possible to construct signatures on random messages even without knowing the signer's secret key (on the other hand, it is impossible to construct a signature on a fixed message). To sign a message within a group, the signer creates signatures on random values for the public key of all the other members of the group. The signer then computes the bitwise exclusive "or" of all the random values and the message, and signs the result with their secret key. The ring signature is the set of signed values. To verify it, the verifier checks the signature of each value with the public key of the corresponding group member and verifies that the bitwise exclusive "or" of all the values is equal to the message. This method is based on the indistinguishability of the signature made with the secret key from those generated for random values, and on the impossibility of stumbling upon random values that will give the message if no secret key is known.

The first building block in our construction is a physical message-randomizable signature. To do this, we use compartmentalized boxes with a transparent top and place a coin with two different sides in each compartment. The signer manually signs padlocks to which they have the key and locks the boxes with these padlocks. These locked boxes are their public keys, and they distribute them to everyone. To sign a binary message, the signer takes one of their boxes, opens it with their key, and arranges the coins so that they correspond to the bits in the message (we assign "heads" to 0 and "tails" to 1). Note that this operation requires their key. They then close the box. To verify the signature, the verifier checks the manual signature on the padlock and checks that the value of the coins matches the message by looking through the transparent top (without opening the box). To obtain a signature on a random message, anyone can take a box from the signer and shake it so that the coins are disturbed and end up in random "heads" or "tails" state.

The general construction of the physical ring signature results from using the one-out-of-many signatures method applied to our physical message-randomisable signature. As it stands, this construction allows a user to generate ring signatures for random messages, which can be problematic in some cases. We propose a countermeasure where the message

must start with a given binary string. To sign longer messages, it is possible to replace the coins with dice. Our prototype uses 30-sided dice, which can encode the letters of the alphabet and certain punctuation marks. Finally, we highlight the security assumptions that our material must verify for our signature to be secure, and we prove the security of our signature in a *semi-formal* way using a sequence of games [14]. By semi-formal, we mean that we try to get close to a real computational security proof. This is not entirely possible because our assumptions are physical and not computational, whereas the ring signature security model considers an adversary modelled by a polynomial time Turing machine.

Related Works

Many cryptographic tools have been adapted in physical form. For example, several physical zero-knowledge proofs, mainly using cards and envelopes, have been proposed for many logic puzzles [9, 4]. The use of cards has also been exploited to build secure multi-party computation protocols [12]. Other works use different tools, such as coin-based secure computation [10] or light cryptography [11], which uses light and shadows for specific secure computation protocols. Physical secure auction protocols have also been proposed [8], one using envelopes and the other a more complex construction using wooden boxes and padlocks. Another example is the construction of threshold access control using padlocks and latches placed in certain configurations [7]. However, to the best of our knowledge, no physical ring signature has ever been proposed, none of the existing physical primitives can be easily adapted to obtain a physical ring signature, and no physical construction uses a mechanism similar to ours (shaking transparent boxes containing coins/dice to randomise data).

2 Technical Background

In this section we first define our notations, then recall the definition of ring signatures and their security properties, and finally recall a property about modular additions.

► **Notations.** $(x_i)_{i=0}^{n-1}$ (resp. $\{x_i\}_{i=0}^{n-1}$) denotes the vector (resp. set) containing the indexed elements x_0, x_1, \dots , and x_{n-1} , and \mathbb{Z}_n denotes the set of integers modulo n (i.e., $\{i\}_{i=0}^{n-1}$). The expression $y \leftarrow x$ denotes the affectation of the value of the variable x to the variable y , the expression $y \leftarrow \text{Algo}(x)$ denotes the affectation of the output of the algorithm Algo on input x to the variable y , and the expression $y \xleftarrow{\$} S$ denotes the affectation of a value chosen in the uniform distribution on a set S to the variable y . The acronym p.p.t. in λ means *probabilistic polynomial time in λ* (when the context is clear, we omit the parameter λ).

► **Definition 1** (Ring Signature [2]). *Let λ be a security parameter. A ring signature is a tuple of p.p.t. algorithms $(\text{Gen}, \text{Sig}, \text{Ver})$ defined as follows:*

Gen(λ): *on input λ , returns a pair of public/secret keys (pk, sk) .*

Sig(sk, R, m): *on input a secret key sk , a set of public keys R (containing the public key corresponding to sk), and a message m , returns a signature σ .*

Ver(R, m, σ): *on input a set of public keys R , a message m , and a signature σ , returns a bit $b \in \{\text{accept}, \text{reject}\}$.*

Moreover, for any integers s and j such that $j < s$, any message m , any $(\text{pk}_i, \text{sk}_i)$ outputted by $\text{Gen}(\lambda)$ for all $i \in \mathbb{Z}_s$, and any σ outputted by $\text{Sig}(\text{sk}_j, \{\text{pk}_i\}_{i=0}^{s-1}, m)$, the condition $\text{Ver}(\{\text{pk}_i\}_{i=0}^{s-1}, m, \sigma) = \text{accept}$ is required to hold.

A secure ring signature is required to satisfy two security properties: unforgeability and anonymity [2]. These properties are modelled by experiments that simulate the use of a ring signature and where a p.p.t. adversary tries to perform an attack.

7:4 Physical Ring Signature

Unforgeability ensures that a user who is not a member of the group cannot generate a valid signature for a fresh message, even if they can access ring signatures for other messages. In the unforgeability experiment, the adversary is given public keys (corresponding to secret keys they do not know), and can query an oracle for signatures on selected messages using these keys. Their goal is to generate a fresh valid ring signature that has not been generated by the oracle. A ring signature is considered to be unforgeable if no adversary can succeed in this attack with a significant (non-negligible¹) probability.

► **Definition 2** (Unforgeability [2]). *Let λ be a security parameter, let $RS = (\text{Gen}, \text{Sig}, \text{Ver})$ be a ring signature, and let \mathcal{A} be a p.p.t. algorithm. For any integer s , we define the s -unforgeability experiment on RS for \mathcal{A} as follows:*

- *The experiment generates s key pairs $\{(\text{pk}_i, \text{sk}_i)\}_{i=0}^{s-1}$ and sends $\{\text{pk}_i\}_{i=0}^{s-1}$ to \mathcal{A} .*
- *\mathcal{A} has access to an oracle $\text{Sig}(\cdot, \cdot, \cdot)$ that returns a signature generated by $\text{Sig}(\text{sk}_j, R, m)$ on query (j, R, m) .*
- *\mathcal{A} returns (R_*, m_*, σ_*) . The experiment returns 1 if and only if $\text{Ver}(R_*, m_*, \sigma_*) = 1$, $R_* \subseteq \{\text{pk}_i\}_{i=0}^{s-1}$, and no query (j, R, m) satisfies $(R, m) = (R_*, m_*)$.*

RS is said to be unforgeable if for all s and all p.p.t. algorithms \mathcal{A} , the probability that the s -unforgeability experiment returns 1 is negligible in λ .

Anonymity ensures that it is not possible to guess which member of the group is the author of a given signature. In the anonymity experiment, the adversary is given public/secret keys, chooses two of them, and is given a ring signature generated from one of these two secret keys (and whose ring contains the two public keys). The adversary tries to distinguish which of the two keys was used with a non-negligible advantage.

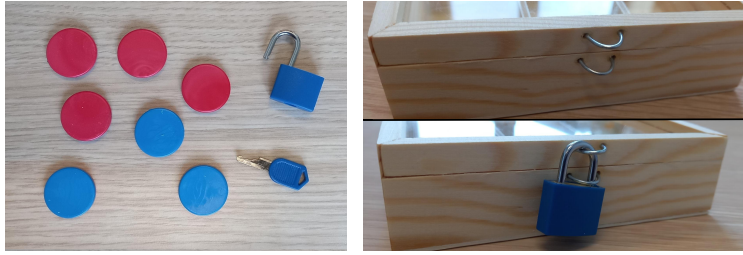
► **Definition 3** (Anonymity [2]). *Let λ be a security parameter, let $RS = (\text{Gen}, \text{Sig}, \text{Ver})$ be a ring signature, and let \mathcal{A} be a p.p.t. algorithm. For any integer s and any bit b , we define the (s, b) -anonymity experiment on RS for \mathcal{A} as follows:*

- *The experiment generates s key pairs $\{(\text{pk}_i, \text{sk}_i)\}_{i=0}^{s-1}$ and sends $\{(\text{pk}_i, \text{sk}_i)\}_{i=0}^{s-1}$ to \mathcal{A} .*
- *\mathcal{A} sends (R, m, i_0, i_1) to the experiment. If $R \subseteq \{\text{pk}_i\}_{i=0}^{s-1}$ and $(\text{pk}_{i_0}, \text{pk}_{i_1}) \in R^2$, then the experiment computes $\sigma \leftarrow \text{Sig}(\text{sk}_{i_b}, R, m)$, and sends σ to \mathcal{A} .*
- *\mathcal{A} returns a bit b_* .*

RS is said to be anonymous if for all s and all p.p.t. algorithms \mathcal{A} , the probability that \mathcal{A} returns 1 on the $(s, 0)$ -anonymity experiment is negligibly close (in λ) to the probability that \mathcal{A} returns 1 on the $(s, 1)$ -anonymity experiment.

The one-out-of-many signatures paradigm [1] presented in Section 1 uses the following result: for any $m \in \mathbb{Z}_n$, if we randomly generate s integers $(x_i)_{i=0}^{s-1}$ whose sum modulo n is m by choosing $j \in \mathbb{Z}_s$, by randomly drawing $x_i \xleftarrow{\$} \mathbb{Z}_n$ for all $i \neq j$, and by completing with the only possible x_j , the integers $(x_i)_{i=0}^{s-1}$ and m do not reveal any information about j . For instance, for $m = 0$, $n = 2$ and $s = 2$, if we randomly draw x_0 then we should set $x_1 = x_0$ to get $x_0 + x_1 \bmod 2 = 0$, and if we randomly draw x_1 then we should set $x_0 = x_1$; both cases return $(x_0, x_1) = (0, 0)$ and $(x_0, x_1) = (1, 1)$ with the same probability. On the other hand, for $m = 1$, the two cases $(0, 1)$ and $(1, 0)$ have the same probability, no matter which element was randomly generated. This result is generalised for vectors of integers in the following theorem. A proof of this theorem is given in Appendix A.

¹ A function f is negligible in x if for any positive polynomial p , there exists an integer x_0 such that for all $x > x_0$, $|f(x)| \leq 1/p(x)$



■ **Figure 1** On the left, some coins with red “heads” and blue “tails” associated with the values 0 and 1 and a padlock with its key, and on the right, a latch on a box, with and without the padlock.

► **Theorem 4.** *Let N, n , and s be three integers. For any $m \in \mathbb{Z}_n^N$, any pair $(i_0, i_1) \in \mathbb{Z}_s^2$, and any distinguisher \mathcal{D} , we have:*

$$\Pr \left[\begin{array}{l} \forall i \in \mathbb{Z}_s \setminus \{i_0\}, x_i \xleftarrow{\$} \mathbb{Z}_n^N; \\ \forall j \in \mathbb{Z}_N, x_{i_0, j} \leftarrow \left(m_j - \sum_{i=0; i \neq i_0}^{s-1} x_{i, j} \right) \bmod n; \end{array} : 1 \leftarrow \mathcal{D}((x_i)_{i=0}^{s-1}) \right] =$$

$$\Pr \left[\begin{array}{l} \forall i \in \mathbb{Z}_s \setminus \{i_1\}, x_i \xleftarrow{\$} \mathbb{Z}_n^N; \\ \forall j \in \mathbb{Z}_N, x_{i_1, j} \leftarrow \left(m_j - \sum_{i=0; i \neq i_1}^{s-1} x_{i, j} \right) \bmod n; \end{array} : 1 \leftarrow \mathcal{D}((x_i)_{i=0}^{s-1}) \right]$$

An example of unforgeable and anonymous cryptographic ring signature based on the BLS [3] signature that follows the one-out-of-many signatures paradigm [1] is given in [5].

3 Our Physical Ring Signature

In this section we present our physical ring signature scheme. We first introduce the material required, then explain how to use it to design a physical message-randomisable signature, and finally explain how a user can anonymously sign within a group using it. We illustrate the steps involved with the help of a physical prototype that we have built.

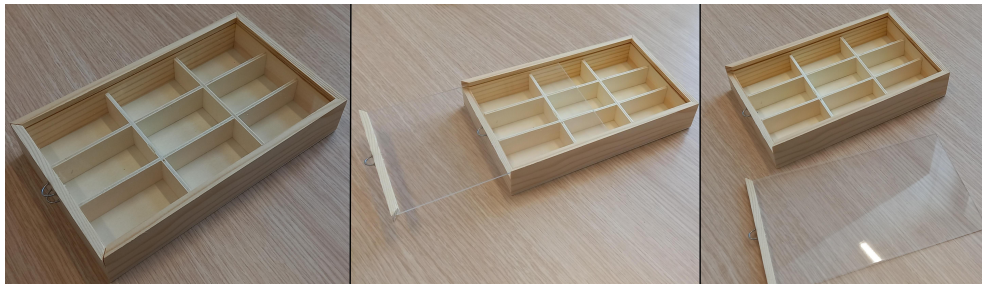
3.1 Material

Each member of the ring/group must be provided with indelible felt-tip pens to enable them to make indelible manual signatures on any surface. Each user must also have an unlimited number of padlocks. Padlocks belonging to the same user must be identical and have a single key that can be used to open them. We assume that users have access to an unlimited number of coins whose two sides (heads and tails) are easily distinguishable. We associate “heads” with the binary value 0 and “tails” with the binary value 1. The coins and the padlocks we use for our prototype are shown in Figure 1.

An unlimited number of compartmentalised boxes are also available (see Figure 2), with the following features:

- The box has a lid that can be opened by a mechanism.
- Each box is divided into N compartments indexed from 0 to $N - 1$ so that it is not possible to move an object from one compartment to another when the box is closed.
- When the box is open, a user can freely place or remove objects in each compartment.
- Each compartment has the shape of a parallelepiped whose edges are larger than the diameter of the coins. A coin in a compartment can therefore move freely within the space of the compartment.

7:6 Physical Ring Signature



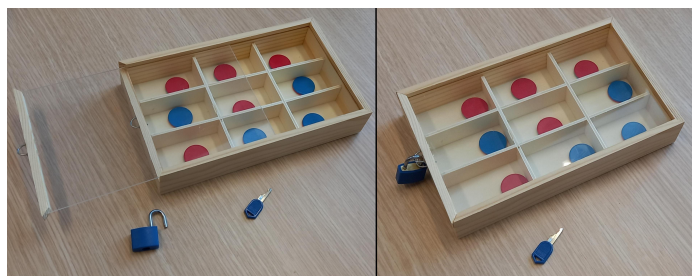
■ **Figure 2** A box with 9 compartments and with a slip-on lid.

- The lid of the box is transparent so that the top of an object in a compartment can be clearly seen.
- The box has a latch that prevents it from being opened (see Figure 1). This latch must be perfectly lockable with a padlock, preventing anyone from opening the box without the padlock key.

As explained in Section 1, our ring signature is based on the one-out-of-many signatures paradigm, which uses message-randomisable signatures. Therefore, we first present how to generate physical message-randomisable signatures, before showing how to use them to generate our physical ring signature.

3.2 Key Generation

We have assumed that a user has an unlimited number of identical padlocks and a single key that can be used to open these padlocks. The user manually signs their padlocks with an indelible felt-tip pen, so that the signature is visible and can be verified by anyone. The user then fills the compartments of several boxes by placing a coin in each compartment. The visible side of the coin (“heads” or “tails”) in each compartment is randomly chosen. The user closes each of these boxes with one of their signed padlocks. These boxes containing coins and closed by signed padlocks are their public key pk (see Figure 3) and they distribute them to the other users. If necessary, they can create new ones at any time. Their secret key sk is the key that opens their padlocks.



■ **Figure 3** A padlocked box (opened then closed) corresponding to a user’s public key, and the padlock key corresponding to their secret key.

3.3 Message-randomisable Signature Generation and Verification

To **sign** a binary message $(m_i)_{i=0}^{N-1}$ of N bits, the signer takes one of their public keys (*i.e.*, a box divided into compartments numbered from 0 to $N - 1$ containing coins and closed by a signed padlock for which they have the key), uses their key to open the padlock and the box to arrange the coins so that the state of the coin in the i -th compartment (“heads” or “tails”) corresponds to the i -th bit m_i of the message, and then closes the padlock on the box.

To **verify** this signature, the verifier checks that the padlock in the box has been manually signed by the signer (by comparing the manual signature with the manual signature on the padlock of one of the signer’s public keys), and looks through the transparent lid to check that the state of the coins corresponds to the bits of the message $(m_i)_{i=0}^{N-1}$.

Without knowing the secret key (and therefore without being able to open the box), any user can **generate the signature for a random message** by shaking the box: since the diameter of the coins is smaller than the dimensions of the compartments, the coins can turn on themselves and randomly land on one of their sides, forming a random binary message.

3.4 Ring Signature Generation

We now show how to use the physical message-randomizable signatures to create physical ring signatures. Let s be the size of the ring/group. The signer has all the (indexed) public keys $R = \{\text{pk}_i\}_{i=0}^{s-1}$ of the members of the group (including their own), their secret key sk , and wishes to sign a message $m = (m_i)_{i=0}^{N-1}$ of N bits. The index corresponding to their public key is denoted j . Recall that public keys are closed boxes containing coins whose visible sides are random. For each pk_i such that $i \neq j$, the signer shakes the box so that the coins in the compartments are randomly flipped (this is the mechanism used to generate signatures for random messages without knowing the key, as described above).

We set $c_{i,k}$ to the binary value associated with the state of the coin in the k -th compartment of the box of the i -th public key pk_i . Using their secret padlock key, the signer opens the box corresponding to their public key pk_j and manually arranges the coins (this is the mechanism used to sign a given message by knowing the key described above) so that $\sum_{i=0}^{s-1} c_{i,k} \bmod 2 = m_k$.

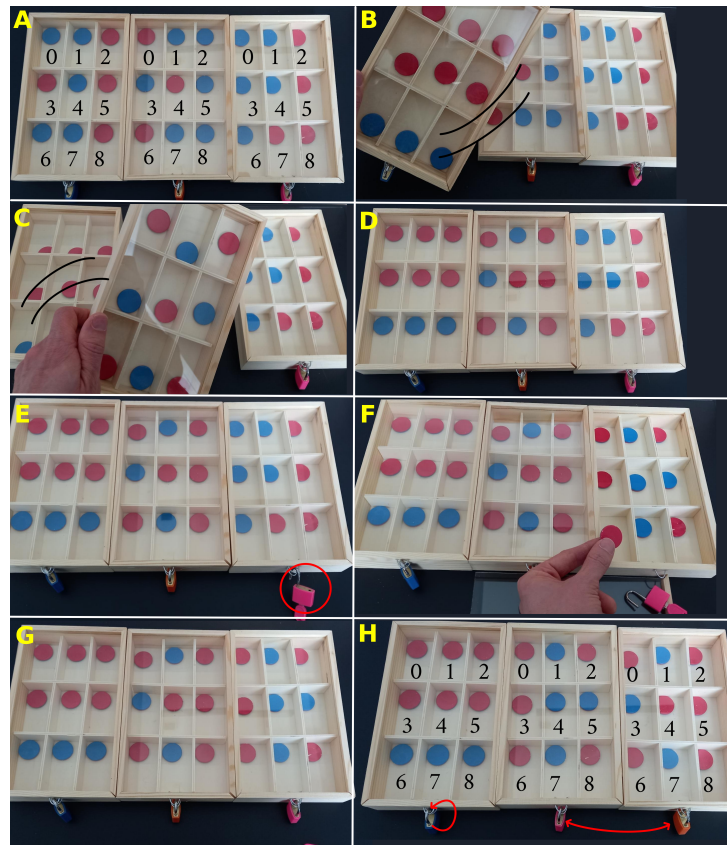
To put it in simple terms, this operation is equivalent to the following: for each bit m_i of the message, if $m_i = 0$ (resp. $m_i = 1$), then the signer places the coin in the i -th compartment of their own box, so that there is an even (resp. odd) number of coins on “tails” (corresponding to the bit 1) in the i -th compartments of all the boxes of all members.

The signer then closes the padlock on their box and arranges the boxes in random order. The signature is the set of all the boxes of the members after these operations.

3.5 Signature Verification

The verifier receives the signature that consists on s padlocked boxes signed by the s members of the group, and the binary message $(m_i)_{i=0}^{N-1}$. They first check that the padlocks have all been manually signed by a different member of the group, and that these signatures are valid (by comparing it with the manual signatures on the padlocks of the group member’s public keys). The verifier sets $c_{i,k}$ to the binary value associated with the state of the coin in the k -th compartment of the box signed by the i -th member of the group. They check that $\sum_{i=0}^{N-1} c_{i,k} \bmod 2 = m_k$ for each k .

To put it in simple terms, this operation is equivalent to the following: for each bit m_i of the message, if $m_i = 0$ (resp. $m_i = 1$), then the verifier checks that there is an even (resp. odd) number of coins on “tails” (corresponding to the bit 1) in the i -th compartments of all the boxes of all members. If this is the case, they accept the signature, if not, they refuse it.



■ **Figure 4** Signature of $M = 111111$ in a ring of $s = 3$ users with security parameter $\lambda = 3$ and boxes with $N = 9$ compartments.

3.6 Preventing the Signing of Random Messages

As it stands, it is possible to generate signatures on random messages without having any padlock key: all you have to do is take the group member boxes and shake them. The result is a signed message whose bits are the modulo two additions of the random values associated to the state of the coins in each compartment. To prevent this, the following countermeasure can be applied: given a security parameter λ , the first λ bits of the signed message must be 0, otherwise the signature is not valid. Thus, the probability of obtaining a valid signature by shaking the boxes is $1/2^\lambda$, which is negligible in λ . On the other hand, the size of the actual signed message becomes $N - \lambda$ bits.

3.7 Example Using a Prototype

We have built a prototype of our signature using boxes with $N = 9$ compartments. In Figure 4, we show all the steps to generate a ring signature of a message $M = 111111$ with $\lambda = 3$ and $s = 3$ group members. Each group member has a different coloured padlock (blue, orange, and pink) and we have omitted the manual signatures on the padlocks. The signer is the owner of the key for the pink padlock. We set $m = (m_k)_{k=0}^{N-1} = (0, 0, 0, 1, 1, 1, 1, 1, 1)$ the message that starts with $\lambda = 3$ times 0 and ends with the bits of M .

Step A: The signer places the three public key boxes side by side. The state of the coins in each of these boxes is associated with the respective values:

$$c_B = (c_{B,k})_{k=0}^{N-1} = (1, 1, 0, 0, 1, 0, 1, 1, 0); \quad c_O = (c_{O,k})_{k=0}^{N-1} = (0, 1, 1, 1, 0, 1, 0, 1, 1);$$

$$c_P = (c_{P,k})_{k=0}^{N-1} = (1, 1, 0, 1, 1, 0, 1, 0, 0);$$

Step B: The signer shakes the blue member box.

Step C: The signer shakes the orange member box.

Step D: At this step, the coins in each box are associated with the values:

$$c_B = (c_{B,k})_{k=0}^{N-1} = (0, 0, 0, 0, 0, 0, 1, 1, 1); \quad c_O = (c_{O,k})_{k=0}^{N-1} = (0, 1, 0, 1, 0, 0, 0, 1, 0);$$

$$c_P = (c_{P,k})_{k=0}^{N-1} = (1, 1, 0, 1, 1, 0, 1, 0, 0);$$

Step E: The signer opens the pink paddlock using their key, then opens the box.

Step F: The signer rearranges the coins in their open box in such a way that $c_{P,k} = m_k \oplus c_{B,k} \oplus c_{O,k}$ for $0 \leq k < N$ (exclusive “or” \oplus is equivalent to addition/substraction modulo 2). In other words, for each index k they make the number of coins on the blue side (corresponding to 1) even if the bit of the message m_k is 0, and odd if the bit of the message m_k is 1. This results in the following configuration:

$$c_B = (c_{B,k})_{k=0}^{N-1} = (0, 0, 0, 0, 0, 0, 1, 1, 1); \quad c_O = (c_{O,k})_{k=0}^{N-1} = (0, 1, 0, 1, 0, 0, 0, 1, 0);$$

$$c_P = (c_{P,k})_{k=0}^{N-1} = (0, 1, 0, 0, 1, 1, 0, 1, 0);$$

Step G: The signer closes their box with their padlock.

Step H: The signer shuffles the boxes: the blue member’s box stays first, and the orange and pink members’ boxes are swapped.

At the end of the signature, the coins are associated with the following binary values:

$$c_B = (c_{B,k})_{k=0}^{N-1} = (0, 0, 0, 0, 0, 0, 1, 1, 1); \quad c_P = (c_{P,k})_{k=0}^{N-1} = (0, 1, 0, 0, 1, 1, 0, 1, 0);$$

$$c_O = (c_{O,k})_{k=0}^{N-1} = (0, 1, 0, 1, 0, 0, 0, 1, 0);$$

Any user can compute: $(c_{B,k} \oplus c_{P,k} \oplus c_{O,k})_{k=0}^{N-1} = (0 \oplus 0 \oplus 0, 0 \oplus 1 \oplus 1, 0 \oplus 0 \oplus 0, 0 \oplus 0 \oplus 1, 0 \oplus 1 \oplus 0, 0 \oplus 1 \oplus 0, 1 \oplus 0 \oplus 0, 1 \oplus 1 \oplus 1, 1 \oplus 0 \oplus 0) = (0, 0, 0, 1, 1, 1, 1, 1, 1)$, and thus verify that $(c_{B,k} \oplus c_{P,k} \oplus c_{O,k})_{k=0}^{\lambda-1} = (0, 0, 0)$ and $(c_{B,k} \oplus c_{P,k} \oplus c_{O,k})_{k=\lambda}^{N-1} = M$. This is equivalent to verifying that for each index k , the number of blue coins in the k -th compartments of the 3 boxes is even if $m_k = 0$ (*i.e.*, for $0 \leq k \leq 2$), and odd if $m_k = 1$ (*i.e.*, for $3 \leq k \leq 8$). Note that the probability of producing a valid signature without the keys (by shaking the boxes only) is $1/2^\lambda = 1/8$ (this is the probability that the first three sums of bits give 0). Of course, to have a more realistic probability of preventing the generation of random message signatures, we would need to use boxes with more compartments.

3.8 Generalising on Larger Alphabets with Dice

A coin can be thought of as a two-sided die. By generalising the principle of our signature, we could sign messages on alphabets of n symbols using n -sided dice (numbered from 1 to n). For example, using 30-sided dice, any integer i between 1 and 26 can be associated with the i -th letter of the Latin alphabet, and 27, 28, 29, and 30 can be respectively associated with space, comma, dot, and a special character ‘*’. Figure 5 shows our prototype used with 30-sided dice (whose diameter is small enough for the dice to roll freely through the compartments).

7:10 Physical Ring Signature



■ **Figure 5** Two public keys of our prototype used with 30-sided dice.

The idea remains the same: the i -th character of the signed message corresponds to the sum modulo n of the values at the top of the dice in the i -th compartment of the boxes of the members of the group. The first λ characters must correspond to the special character '*', so the probability of generating a signature for a valid random message is $1/n^\lambda$. On the other hand, a signer who has the key to one of the padlocks will always be able to arrange the dice to obtain a valid signature for a given message.

In Figure 6, we show all the steps to generate a ring signature of a message $M = \text{"hello."}$ with our prototype using 30-sided dice and with $\lambda = 3$ and $s = 3$. Each user has a different coloured padlock (green, red, and yellow) and we have omitted the manual signatures on the padlocks. The signer is the owner of the key for the yellow padlock. We set $m = (m_k)_{k=0}^{N-1} = (0, 0, 0, 8, 5, 12, 12, 15, 29)$ the message that starts with $\lambda = 3$ times 0 (that corresponds to the special character '*' since $30 \bmod 30 = 0$) and ends with the integers that correspond to the characters 'h', 'e', 'l', 'l', 'o', and '.'.

Step A: The signer places the three public key boxes side by side. The dice in each of these boxes indicate the respective values:

$$c_G = (c_{G,k})_{k=0}^{N-1} = (28, 11, 29, 13, 25, 28, 30, 15, 11)$$

$$c_R = (c_{R,k})_{k=0}^{N-1} = (9, 11, 9, 16, 27, 30, 13, 1, 8)$$

$$c_Y = (c_{Y,k})_{k=0}^{N-1} = (21, 8, 21, 15, 27, 1, 16, 15, 20)$$

Step B: The signer shakes the green group member box.

Step C: The signer shakes the red group member box.

Step D: At this step, the dice in each of these boxes indicate the respective values:

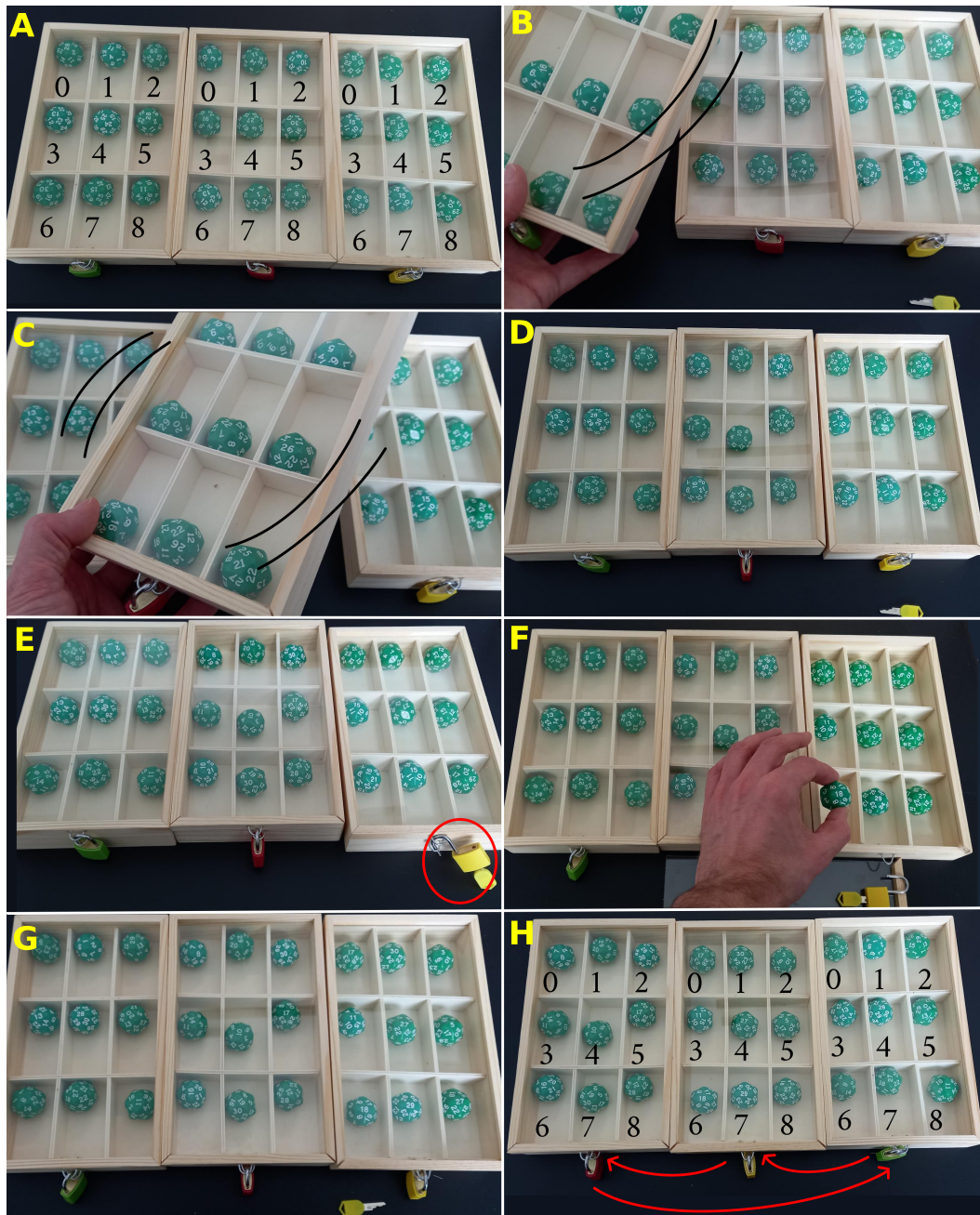
$$c_G = (c_{G,k})_{k=0}^{N-1} = (6, 1, 29, 13, 28, 20, 8, 27, 11)$$

$$c_R = (c_{R,k})_{k=0}^{N-1} = (7, 29, 28, 14, 10, 17, 16, 19, 21)$$

$$c_Y = (c_{Y,k})_{k=0}^{N-1} = (21, 8, 21, 15, 27, 1, 16, 15, 20)$$

Step E: The signer opens the yellow padlock using their key, then opens the box.

Step F: The signer rearranges the dice in their open box in such a way that $c_{Y,k} = m_k - c_{G,k} - c_{R,k} \bmod 30$ for $0 \leq k < N$. This results in the following configuration (where 0 is encoded by 30 on the dice, since dice are numbered from 1 to 30):



■ **Figure 6** Signature of $M = \text{"hello."}$ in a ring of $s = 3$ users with security parameter $\lambda = 3$ and boxes with $N = 9$ compartments.

$$\begin{aligned}
 c_G &= (c_{G,k})_{k=0}^{N-1} = (6, 1, 29, 13, 28, 20, 8, 27, 11) \\
 c_R &= (c_{R,k})_{k=0}^{N-1} = (7, 29, 28, 14, 10, 17, 16, 19, 21) \\
 c_Y &= (c_{Y,k})_{k=0}^{N-1} = (-6 - 7 \bmod 30, -1 - 29 \bmod 30, -29 - 28 \bmod 30, \\
 &\quad 8 - 13 - 14 \bmod 30, 5 - 28 - 10 \bmod 30, 12 - 20 - 17 \bmod 30, \\
 &\quad 12 - 8 - 16 \bmod 30, 15 - 27 - 19 \bmod 30, 29 - 11 - 21 \bmod 30) \\
 &= (17, 0, 3, 11, 27, 5, 18, 29, 27)
 \end{aligned}$$

7:12 Physical Ring Signature

Step G: The signer closes their box with their padlock.

Step H: The signer shuffles the boxes: the order of the boxes changes from green then red then yellow to red then yellow then green.

At the end of the signature, the dice are in the following configuration:

$$c_R = (c_{R,k})_{k=0}^{N-1} = (7, 29, 28, 14, 10, 17, 16, 19, 21)$$

$$c_Y = (c_{Y,k})_{k=0}^{N-1} = (17, 0, 3, 11, 27, 5, 18, 29, 27)$$

$$c_G = (c_{G,k})_{k=0}^{N-1} = (6, 1, 29, 13, 28, 20, 8, 27, 11)$$

Any user can verify the signature by computing:

$$\begin{aligned} & (c_{R,k} + c_{Y,k} + c_{G,k} \bmod 30)_{k=0}^{N-1} \\ &= (7 + 17 + 6 \bmod 30, 29 + 0 + 1 \bmod 30, 28 + 3 + 29 \bmod 30, \\ & 14 + 11 + 13 \bmod 30, 10 + 27 + 28 \bmod 30, 17 + 5 + 20 \bmod 30, \\ & 16 + 18 + 8 \bmod 30, 10 + 29 + 27 \bmod 30, 21 + 27 + 11 \bmod 30) \\ &= (0, 0, 0, 8, 5, 12, 12, 15, 29), \end{aligned}$$

and thus verify that $(c_{R,k} + c_{Y,k} + c_{G,k} \bmod 30)_{k=0}^{\lambda-1} = (0, 0, 0)$ and that $(c_{R,k} + c_{Y,k} + c_{G,k} \bmod 30)_{k=\lambda}^{N-1} = M$.

Note that the probability of producing a valid signature without the keys (by shaking the boxes only) is $1/n^\lambda = 1/27000$ (this is the probability that the first three sums give 0).

4 Security Analysis

In this section, we identify the assumptions required to ensure the security of our prototype.

► **Assumptions.** The following properties are assumed to be true:

1. It is not possible to break or force open a compartmentalised box, *i.e.*, a box can only be opened by its lid using the mechanism provided for this purpose.
2. It is impossible to forge a user's manual signature without being that user.
3. Padlocks are unbreakable and cannot be opened without a key. In particular, it is impossible to forge a key for a padlock.
4. A padlock attached to the latch of a box prevents the lid of the box from being opened by the mechanism provided for this purpose.
5. An object cannot be moved to another compartment when a box is locked.
6. The only action that can be performed on a closed box to move the objects contained in its compartments is to shake it.
7. Shaking the box when it contains dice is equivalent to rolling the dice. More precisely, given a box with dice, and knowing the previous position of the dice, it is impossible to determine whether the box has been shaken, or opened, the dice rearranged to show a random value chosen from the uniform distribution, and the box closed again.

These assumptions allow us to claim the following two theorems. For each of them we give an intuitive explanation of the results, then we give a security proof in a *semi-formal* style: we show a sequence of games [14] that reduce to each other by hops, eliminating events whose probability of occurrence is at most negligible under our assumptions. In our security proofs, we consider that the physical actions of closing a padlock, opening a padlock, closing a box, opening a box, moving an object, and shaking a box are achievable by an adversary

(modelled by a p.p.t. algorithm) in constant time. In general, no basic physical operation depends on the security parameter λ . We also consider that any physical action performed by the adversary can be observed by the challenger who simulates the security experiment for them.

► **Theorem 5.** *Our physical ring signature is unforgeable under Assumption 1, 2, 3, 4, 5, 6, and 7. More precisely, for any integer s , any security parameter λ , any p.p.t. algorithm \mathcal{A} , and any polynomial function q , the probability that the s -unforgeability experiment on our physical ring signature using n -sided dice for \mathcal{A} returns 1 is bounded by $\frac{q(\lambda)}{n^\lambda}$, where $q(\lambda)$ is the number of times that a box is shaken during the experiment.*

According to Assumptions 1, 2, 3, and 4, an adversary cannot manually sign a box in place of a member of the group, and cannot break/force the box and its mechanism if it is locked by a padlock. Nor can they open a box to manually change the value of the dice under Assumption 5 and 6. Furthermore, according to Assumption 6 and 7, they cannot bias the roll of the dice so that it is not uniform when they decide to shake a box. Their only possible strategy is to hope that shaking the boxes will produce a valid signature. To do this, the first λ dice in the boxes must match the special character '*', which happens with a negligible probability of at most $\frac{q(\lambda)}{n^\lambda}$ where $q(\lambda)$ is the number of times a box is shaken.

Proof (Theorem 5). Let s be an integer, and \mathcal{A} be a p.p.t. algorithm. We consider the following sequence of games.

Game G_0 : In this game, a challenger simulates the s -unforgeability experiment on our physical ring signature for \mathcal{A} . The event “ \mathcal{A} wins G_0 ” denotes that the unforgeability experiment returns 1.

Game G_1 : Same as G_0 , except that if \mathcal{A} breaks or opens a box on an other way that by its lid using the mechanism provided for this purpose, forge a group member's manual signature, or breaks or opens a padlock without its key, then the challenger returns 0. According with Assumption 1, 2, and 3, we have that $\Pr[\mathcal{A} \text{ wins } G_0] = \Pr[\mathcal{A} \text{ wins } G_1]$.

Game G_2 : Same as G_1 , except that if \mathcal{A} opens the box in one of the public keys pk_i , then the challenger returns 0. Note that at this step the public key boxes cannot be opened except by their normal opening mechanism, and each box is locked with a padlock that cannot be broken. According with assumption 4, we have that $\Pr[\mathcal{A} \text{ wins } G_1] = \Pr[\mathcal{A} \text{ wins } G_2]$.

Game G_3 : Same as G_2 , except that if \mathcal{A} moves one die to another compartment in the box of a public key, then the challenger returns 0. Note that at this step the public key boxes cannot be opened. According with assumption 5, we have that $\Pr[\mathcal{A} \text{ wins } G_2] = \Pr[\mathcal{A} \text{ wins } G_3]$.

At this step, the only way to change the position of the dice in the boxes corresponding to the public keys is to shake them, according to Assumption 6. In addition, according to Assumption 7, shaking a box is equivalent to giving random values to the dice inside.

Game G_4 : Same as G_3 , except that if \mathcal{A} returns a valid signature beginning with λ times the special character '*', then the challenger returns 0. We claim that $|\Pr[\mathcal{A} \text{ wins } G_3] - \Pr[\mathcal{A} \text{ wins } G_4]| \leq \frac{q(\lambda)}{n^\lambda}$.

In order for the k -th symbol of the message to be '*', we must have $\sum_{i=0}^{s-1} c_{i,k} \bmod n = 0$, where $c_{i,k}$ is the value indicated by the die in the k -th compartment of the box of the i -th member of the group. Since the $c_{i,k}$ are all drawn in a uniform distribution (when the box

7:14 Physical Ring Signature

is shaken), evaluating the probability of having $\sum_{i=0}^{s-1} c_{i,k} \bmod n = 0$ is equivalent to fixing the value $c_{i,k}$ of the last $s - 1$ dice and evaluating the probability of drawing the value $c_{0,k}$ verifying $\sum_{i=0}^{s-1} c_{i,k} \bmod n = 0$. This probability is $1/n$. Moreover, the adversary must succeed this for all $0 \leq k \leq \lambda - 1$, knowing that all the $c_{i,k}$ are randomly refreshed when a box is shaken. Thus, the probability of drawing the values $c_{0,k}$ verifying $\sum_{i=0}^{s-1} c_{i,k} \bmod n = 0$ for all $0 \leq k \leq \lambda - 1$ is $1/n^\lambda$. Since the number of times a box is shaken during the experiment is $q(\lambda)$, the adversary has at most $q(\lambda)$ tries to draw the correct $c_{0,k}$ (we assume that the box of each public key was shaken before being distributed). So \mathcal{A} has a probability bounded by $\frac{q(\lambda)}{n^\lambda}$ to draw all the correct $c_{0,k}$ together for $0 \leq k \leq \lambda - 1$ during the experiment, and so having λ times the symbol '*' at the beginning of its returned signing message, which concludes the proof of the claim.

Note that in G_4 , \mathcal{A} can no longer win, since a signature must begin with λ times the special character '*' to be valid, so its probability of winning the game is 0. Since $\Pr[\mathcal{A} \text{ wins } G_0] = \Pr[\mathcal{A} \text{ wins } G_3]$, we have $|\Pr[\mathcal{A} \text{ wins } G_0] - \Pr[\mathcal{A} \text{ wins } G_4]| \leq \frac{q(\lambda)}{n^\lambda}$, so $\Pr[\mathcal{A} \text{ wins } G_0] \leq \frac{q(\lambda)}{n^\lambda}$. Finally, the probability that the unforgeability experiment returns 1 is bounded by the negligible function $\frac{q(\lambda)}{n^\lambda}$, which concludes the proof. \blacktriangleleft

► **Theorem 6.** *Our physical ring signature is anonymous under Assumption 7. More precisely, for any integer s , any security parameter λ , and any p.p.t. algorithm \mathcal{A} , the probability that \mathcal{A} returns 1 on the $(s, 0)$ -anonymity experiment is equals to the probability that \mathcal{A} returns 1 on the $(s, 1)$ -anonymity experiment on our physical ring signature.*

Assumption 7 ensures that it is not possible to distinguish from a physical point of view whether a box has been shaken or whether the things in it have been moved manually, and Theorem 4 ensures that it is not possible to distinguish from a computational point of view which dice value have been drawn randomly and which have been chosen to complete the sum in order to obtain the message. Thus, an adversary has no way of distinguishing which box has been opened, and therefore the identity of the signer.

Proof (Theorem 6). Let s be an integer, and \mathcal{A} be a p.p.t. algorithm. We consider the following sequence of games.

Game G_0 : In this game, a challenger simulates the $(s, 0)$ -anonymity experiment on our physical ring signature for \mathcal{A} . The event " $b_* = 1$ in G_0 " denotes that \mathcal{A} returns 1 at the end of the experiment.

Game G_1 : Same as G_0 , except that when the challenger signs the message m chosen by \mathcal{A} , each time they are supposed to shake a box, they instead open it, manually arrange the dice to give them a random value, and close the box again. Under Assumption 7, we have that $\Pr[b_* = 1 \text{ in } G_0] = \Pr[b_* = 1 \text{ in } G_1]$.

Game G_2 : Same as G_1 , except that the challenger signs with sk_{i_1} instead of sk_{i_0} . Assuming that $\Pr[b_* = 1 \text{ in } G_1] \neq \Pr[b_* = 1 \text{ in } G_2]$, we will show that there exists a distinguisher \mathcal{D} that contradicts Theorem 4.

We build \mathcal{D} as follows: \mathcal{D} simulates G_2 to \mathcal{A} , receives (R, m, i_0, i_1) from \mathcal{A} , sets $|R| = s'$, and receives the input $(x_i)_{i=0}^{s'-1}$. We parse R as $\{\text{pk}'_i\}_{i=0}^{s'-1}$, m as $(m_k)_{k=0}^{N-1}$, and x_i as $(x_{i,k})_{k=0}^{N-1}$ for $0 \leq i < s'$. According to the definition of \mathcal{D} in Theorem 4 on the values N, n, s', m , and (i_0, i_1) , the input $(x_i)_{i=0}^{s'-1}$ verifies $m_k = \sum_{i=0}^{s'-1} x_{i,k} \bmod n$ for all $0 \leq k < N$. To forge the signature, \mathcal{D} opens the box corresponding to each key pk'_i and arranges the dice in such a

way that the die in the k -th compartment of the i -th box indicates the value $x_{i,k}$. Finally, \mathcal{D} sends the resulting signature to \mathcal{A} , receives b_* from \mathcal{A} , and returns it. We have:

$$\Pr \left[\begin{array}{l} \forall i \in \mathbb{Z}_{s'} \setminus \{i_0\}, x_i \xleftarrow{\$} \mathbb{Z}_n^N; \\ \forall k \in \mathbb{Z}_N, x_{i_0,k} \leftarrow \left(m_k - \sum_{i=0; i \neq i_0}^{s-1} x_{i,k} \right) \bmod n; \end{array} : 1 \leftarrow \mathcal{D}((x_i)_{i=0}^{s'-1}) \right] \\ = \Pr[b_* = 1 \text{ in } G_1];$$

$$\Pr \left[\begin{array}{l} \forall i \in \mathbb{Z}_{s'} \setminus \{i_1\}, x_i \xleftarrow{\$} \mathbb{Z}_n^N; \\ \forall k \in \mathbb{Z}_N, x_{i_1,k} \leftarrow \left(m_k - \sum_{i=0; i \neq i_1}^{s-1} x_{i,k} \right) \bmod n; \end{array} : 1 \leftarrow \mathcal{D}((x_i)_{i=0}^{s'-1}) \right] \\ = \Pr[b_* = 1 \text{ in } G_2].$$

This contradicts Theorem 4 on the values N, n, s', m , and (i_0, i_1) for the distinguisher \mathcal{D} , because $\Pr[b_* = 1 \text{ in } G_1] \neq \Pr[b_* = 1 \text{ in } G_2]$. Finally, we deduce that $\Pr[b_* = 1 \text{ in } G_1] = \Pr[b_* = 1 \text{ in } G_2]$.

Game G_3 : Same as G_2 , except that when the challenger signs the message m chosen by \mathcal{A} , each time they are supposed to open a box, manually arrange the dice to give them a random value, and close the box again, they instead shake the box. Under Assumption 7, we have that $\Pr[b_* = 1 \text{ in } G_2] = \Pr[b_* = 1 \text{ in } G_3]$.

We observe that in G_3 the challenger simulates the $(s, 1)$ -anonymity experiment on our physical ring signature for \mathcal{A} . Moreover, we have shown that $\Pr[b_* = 1 \text{ in } G_0] = \Pr[b_* = 1 \text{ in } G_3]$. Finally, we deduce that the probability that \mathcal{A} returns 1 on the $(s, 0)$ -anonymity experiment is equals to the probability that \mathcal{A} returns 1 on the $(s, 1)$ -anonymity experiment on our physical ring signature, which concludes the proof. ◀

5 Conclusion

In this paper we have described a physical ring signature that is easy to set up and that uses everyday objects. We have built a prototype, and we believe that it can be used to explain in a playful way how a ring signature works to a public not familiar with cryptography. Some ring signatures have additional properties, such as linkability (any user can link two signatures produced by the same member) and traceability (an authority can lift anonymity in some cases). In future work, we would like to find ways to adapt our physical ring signature, or propose new ones, to achieve these properties.

References

- 1 Masayuki Abe, Miyako Ohkubo, and Koutarou Suzuki. 1-out-of-n signatures from a variety of keys. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 415–432. Springer, 2002. doi:10.1007/3-540-36178-2_26.
- 2 Adam Bender, Jonathan Katz, and Ruggero Morselli. Ring signatures: Stronger definitions, and constructions without random oracles. *J. Cryptol.*, 22(1):114–138, 2009. doi:10.1007/S00145-007-9011-9.

- 3 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. *J. Cryptol.*, 17(4):297–319, 2004. doi:10.1007/S00145-004-0314-9.
- 4 Xavier Bultel, Jannik Dreier, Jean-Guillaume Dumas, Pascal Lafourcade, Daiki Miyahara, Takaaki Mizuki, Atsuki Nagao, Tatsuya Sasaki, Kazumasa Shinagawa, and Hideaki Sone. Physical zero-knowledge proof for makaro. In Taisuke Izumi and Petr Kuznetsov, editors, *Stabilization, Safety, and Security of Distributed Systems - 20th International Symposium, SSS 2018, Tokyo, Japan, November 4-7, 2018, Proceedings*, volume 11201 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2018. doi:10.1007/978-3-030-03232-6_8.
- 5 Ran Canetti and Ronald L. Rivest. Lecture 26. *Selected Topics in Cryptography*, 2004. URL: <https://courses.csail.mit.edu/6.897/spring04/L26.pdf>.
- 6 Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In Yvo Desmedt, editor, *Advances in Cryptology - CRYPTO '94, 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994, Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 1994. doi:10.1007/3-540-48658-5_19.
- 7 Jannik Dreier, Jean-Guillaume Dumas, Pascal Lafourcade, and Léo Robert. Optimal threshold padlock systems. *J. Comput. Secur.*, 30(5):655–688, 2022. doi:10.3233/JCS-210065.
- 8 Jannik Dreier, Hugo Jonker, and Pascal Lafourcade. Secure auctions without 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 158–170. Springer, 2014. doi:10.1007/978-3-319-07890-8_14.
- 9 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.
- 10 Yuichi Komano and Takaaki Mizuki. Coin-based secure computations. *Int. J. Inf. Sec.*, 21(4):833–846, 2022. doi:10.1007/S10207-022-00585-8.
- 11 Pascal Lafourcade, Takaaki Mizuki, Atsuki Nagao, and Kazumasa Shinagawa. Light cryptography. In Lynette Drevin and Marianthi Theocharidou, editors, *Information Security Education. Education in Proactive Information Security - 12th IFIP WG 11.8 World Conference WISE 12, Lisbon, Portugal, June 25-27, 2019, Proceedings*, volume 557 of *IFIP Advances in Information and Communication Technology*, pages 89–101. Springer, 2019. doi:10.1007/978-3-030-23451-5_7.
- 12 Takaaki Mizuki. Efficient and secure multiparty computations using a standard deck of playing cards. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 484–499, 2016. doi:10.1007/978-3-319-48965-0_29.
- 13 Ronald L. Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 552–565. Springer, 2001. doi:10.1007/3-540-45682-1_32.
- 14 Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Paper 2004/332, 2004. URL: <https://eprint.iacr.org/2004/332>.
- 15 Anh The Ta, Thanh Xuan Khuc, Tuong Ngoc Nguyen, Huy Quoc Le, Dung Hoang Duong, Willy Susilo, Kazuhide Fukushima, and Shinsaku Kiyomoto. Efficient unique ring signature for blockchain privacy protection. In Joonsang Baek and Sushmita Ruj, editors, *Information Security and Privacy - 26th Australasian Conference, ACISP 2021, Virtual Event, December 1-3, 2021, Proceedings*, volume 13083 of *Lecture Notes in Computer Science*, pages 391–407. Springer, 2021. doi:10.1007/978-3-030-90567-5_20.

- 16 Sri Aravinda Krishnan Thyagarajan, Giulio Malavolta, Fritz Schmid, and Dominique Schröder. Verifiable timed linkable ring signatures for scalable payments for monero. In Vijayalakshmi Atluri, Roberto Di Pietro, Christian Damsgaard Jensen, and Weizhi Meng, editors, *Computer Security - ESORICS 2022 - 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26-30, 2022, Proceedings, Part II*, volume 13555 of *Lecture Notes in Computer Science*, pages 467–486. Springer, 2022. doi:10.1007/978-3-031-17146-8_23.
- 17 Patrick P. Tsang and Victor K. Wei. Short linkable ring signatures for e-voting, e-cash and attestation. In Robert H. Deng, Feng Bao, HweeHwa Pang, and Jianying Zhou, editors, *Information Security Practice and Experience, First International Conference, ISPEC 2005, Singapore, April 11-14, 2005, Proceedings*, volume 3439 of *Lecture Notes in Computer Science*, pages 48–60. Springer, 2005. doi:10.1007/978-3-540-31979-5_5.

A Proof of Theorem 4

To prove Theorem 4, we prove the following two lemmas.

► **Lemma 7.** *Let n be an integer. For any $m \in \mathbb{Z}_n$ and any distinguisher \mathcal{D} , we have:*

$$\Pr[x_1 \xleftarrow{\$} \mathbb{Z}_n; x_0 \leftarrow m - x_1 \bmod n; : 1 \leftarrow \mathcal{D}(x_0, x_1)] = \\ \Pr[x_0 \xleftarrow{\$} \mathbb{Z}_n; x_1 \leftarrow m - x_0 \bmod n; : 1 \leftarrow \mathcal{D}(x_0, x_1)]$$

Proof. In the first case, since $x_1 \xleftarrow{\$} \mathbb{Z}_n$ and $x_0 \leftarrow m - x_1 \bmod n$, each pair $(x_0, x_1) \in \mathbb{Z}_n$ such that $x_0 + x_1 = m \bmod n$ is generated with probability $1/n$. We remark that each of the n pairs contains a different x_0 , so each $x_0 \in \mathbb{Z}_n$ appears with probability $1/n$. Similarly, in the second case, if $x_0 \xleftarrow{\$} \mathbb{Z}_n$ and $x_1 \leftarrow m - x_0 \bmod n$, then each pair $(x_0, x_1) \in \mathbb{Z}_n$ such that $x_0 + x_1 = m \bmod n$ with a different x_1 is generated with probability $1/n$. We deduce that the two cases are indistinguishable, which concludes the proof. ◀

► **Lemma 8.** *Let n and s be two integers. For any $m \in \mathbb{Z}_n$, any pair $(i_0, i_1) \in \mathbb{Z}_s^2$ such that $i_0 \neq i_1$, and any distinguisher \mathcal{D} , we have:*

$$\Pr \left[\begin{array}{l} \forall i \in \mathbb{Z}_s \setminus \{i_0\}, x_i \xleftarrow{\$} \mathbb{Z}_n; \\ x_{i_0} \leftarrow \left(m - \sum_{i=0; i \neq i_0}^{s-1} x_i \right) \bmod n; \end{array} : 1 \leftarrow \mathcal{D}((x_i)_{i=0}^{s-1}) \right] = \\ \Pr \left[\begin{array}{l} \forall i \in \mathbb{Z}_s \setminus \{i_1\}, x_i \xleftarrow{\$} \mathbb{Z}_n; \\ x_{i_1} \leftarrow \left(m - \sum_{i=0; i \neq i_1}^{s-1} x_i \right) \bmod n; \end{array} : 1 \leftarrow \mathcal{D}((x_i)_{i=0}^{s-1}) \right]$$

Proof. If $i_0 = i_1$, the result is trivial because the two expressions are the same. Else, by setting:

$$m' = \left(m - \sum_{i=0; i \notin \{i_0, i_1\}}^{s-1} x_i \right) \bmod n,$$

we have:

$$x_{i_0} = \left(m - \sum_{i=0; i \neq i_0}^{s-1} x_i \right) \bmod n \Leftrightarrow x_{i_0} = m' - x_{i_1} \bmod n \quad (1)$$

$$\Leftrightarrow x_{i_1} = m' - x_{i_0} \bmod n. \quad (2)$$

7:18 Physical Ring Signature

We recall that in the two cases, each x_i such that $i \notin \{i_0, i_1\}$ is generated at random. Therefore, these values cannot be used to distinguish between the two cases. If x_{i_1} is chosen at random, then Equation 1 corresponds to the expression in the first probability in Lemma 7. Similarly, if x_{i_0} is chosen at random, then Equation 2 corresponds to the expression in the second probability in Lemma 7. The values generated by these two expressions are therefore indistinguishable according to Lemma 7. Finally, the proof of Lemma 8 follows from Lemma 7. ◀

Lemma 8 can easily be generalized to the case where the x_i are vectors of integers, which leads directly to Theorem 4.

A Tractability Gap Beyond Nim-Sums: It’s Hard to Tell Whether a Bunch of Superstars Are Losers

Kyle Burke   




Florida Southern College, Lakeland, FL, USA

Matthew Ferland   

University of Southern California, Los Angeles, CA, USA

Svenja Huntemann   

Mount Saint Vincent University, Halifax, Canada

Shanghai Teng   

University of Southern California, Los Angeles, CA, USA

Abstract

In this paper, we address a natural question at the intersection of combinatorial game theory and computational complexity: “Can a sum of simple *tepid games* in canonical form be intractable?” To resolve this fundamental question, we consider *superstars*, positions first introduced in *Winning Ways* where all options are *nimbers*. Extending Morris’ classic result with hot games to tepid games, we prove that disjunctive sums of superstars are intractable to solve. This is striking as sums of nimbers can be computed in linear time. Our analyses also lead to a family of elegant board games with intriguing complexity, for which we present web-playable versions of the rulesets described within.

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

Keywords and phrases Combinatorial Game Theory, NP-hardness, Superstars

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.8

Funding *Shanghai Teng*: Supported in part by NSF Grant CCF-2308744 and the Simons Investigator Award from the Simons Foundation.

1 Introduction

“The whole is greater than the sum of their parts” is an ancient phrase that particularly exemplifies combinatorial game theory. As an area of mathematics dedicated to analyzing what happens when several games are combined, the field is rich with results both in isolation and with interdisciplinary connections. Indeed, even casually, games are often combined for enjoyment, such as Bughouse (2 simultaneous games of Chess) and Ultimate Tic Tac Toe (9 simultaneous games of Tic Tac Toe).

While several different ways to combine games are studied, the predominant one is the *disjunctive sum of games* following the *normal play* convention. In a disjunctive sum, players alternate turns choosing a single game component, making a move on it, and passing the turn over to their opponent, leaving the other components unchanged. Under normal play, when a player is unable to make a move (because there are no moves remaining for them in any game), then that player loses. In other words, the last player to make a move wins.

The modern forms of both combinatorial game theory and computational complexity theory were born around half a century ago, and there are several results of the latter about the former. Most relevant to this work, in 1981, Morris demonstrated that a sum of (individually polynomial-time solvable games) is PSPACE-complete [10]. The hard game sums in that reduction have several key requirements. First, they involve deeply asymmetric games (i.e., games where the moves available to the two players are very different). Second,



© Kyle Burke, Matthew Ferland, Svenja Huntemann, and Shanghai Teng;
licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 8; pp. 8:1–8:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

the games have exponential length (the number of turns). Third, a polynomial number of games are included in the sums. Finally, most components of the game are *hot games*, meaning that players are incentivized to play first on most games in the sum.

Later results by Yedwab [16], Mowes [9], and eventually Wolfe [15] improved the reduction by eliminating the exponential length and reducing the branching factor size (to a smaller constant). However, the other two limitations remain.

A more recent result in 2021 [4] demonstrated that the sum of two tractable symmetric polynomial-length impartial games¹, when combined, are PSPACE-complete. This was accomplished using a pair of natural games known as UNDIRECTED GEOGRAPHY. Although UNDIRECTED GEOGRAPHY positions can be solved in polynomial time [6], it is PSPACE-complete to determine their values, as shown in [4].

Therefore, the hardness comes from finding that value rather than describing the difficulty of performing the mathematical operation. In fact, these impartial games have a simple polynomial algorithm to identify a winner in a sum if the game is already in its simplest form.

This paper continues the chain of results that Morris started, finding intractible summands with even more shallow game trees than were previously known. First, instead of hot games, we sum components from a family of *tepid* games. Tepid is a term based in temperature theory, where the temperature of a game is the number that approximates the incentive to move in a position. Hotter games have a higher value of potentially-earned moves in the favor of the first player to play on them. This can be by supplying moves to use later or denying your opponent later free plays.² Cold games use up these moves when a player plays on them; their temperature is negative. Tepid games all have a temperature of zero: playing on them doesn't earn either player any moves but can influence the parity of the current situation.

The second reason our work continues the chain of intractible sums is that, instead of the more deeply asymmetric games that Morris used, we use a family of *nearly symmetric* games, which become symmetric after a single move. Third, unlike the result with UNDIRECTED GEOGRAPHY, we use a family of games that are already in *canonical form*, which is to say, directly in the form of their values. And finally, this family of games is deeply related to one whose existence traces back to the birth of the modern theory.

The main family of games we consider are called *superstars*³. These values naturally occur in the game PAINT CAN[13], which we discuss in Section 2. We show that for sums of superstars, it is computationally-intractable to determine which player has a winning strategy.

► **Theorem 1.** *A sum of superstars is NP-hard.*

The paper is structured as follows: In Section 2 we introduce the necessary concepts from combinatorial game theory. The proof of the main theorem is given in Section 3. The reduction used to prove our main theorem also leads to a nice new ruleset which we call BLACKOUT, introduced in Section 4.

¹ positions in which both players have the same options

² For a more thorough description of temperature theory in CGT, we recommend [12].

³ There are some inconsistencies with this choice of name, which we discuss in full in Section 2.2.

2 Superstars: Theory and Paint Can

2.1 Rising Stars: From Stars to Superstars

In this section we will give a brief introduction to concepts from combinatorial game theory (CGT) required for this paper. For more information and a rigorous treatment of these topics, see [2], [1], and [12].

In the game NIM, played on piles of tokens, the two players take turns choosing a pile and removing any nonzero number of tokens. Under *normal play* the player to pick the last token(s) wins.

NIM is an *impartial game*, meaning one in which the two players have the same possible moves. An impartial game is denoted $G = \{G_1, \dots, G_n\}$, where G_1, \dots, G_n are the options the players can move to.

When only a single pile remains in NIM, the current player will simply remove all tokens. But when several piles remain, the optimal move is often to only take some of the tokens. Thus all possible moves on a pile need to be considered when in a sum. To do so, we assign a *value* to each pile which represents the possible moves. An empty pile, thus one in which there are no moves, is given the value 0. The value of a pile with n tokens is the *number* $*n$. For a consistent recursive definition, we think of 0 as the number $*0$. The shorthand $*1 = *$ is also generally used.

► **Definition 2.** *The number $*n$ is recursively defined by its options as*

- $*0 = 0 = \emptyset$ (no available moves);
- $*1 = * = \{0\}$
- $*n = \{0, *, *2, \dots, *(n-1)\}$

The (*disjunctive*) sum $G_1 + \dots + G_n$ of games G_1, \dots, G_n is the game in which the players chose a summand (or component), then make a move in it. A Nim position is naturally the disjunctive sum of its separate piles. Many other games also naturally break into components, but we can consider sums of any games in general.

We say that two games are *equal* to each other when one can be replaced with the other in *any* disjunctive sum without changing the winnability. I.e., $A = B$ whenever who wins in $A + X$ is the same as in $B + X$ for any game X .

A sum of numbers is always equal to a single number. Finding which number it is requires only an XOR sum (also known as *nim sum* in CGT) and therefore is in P (solvable in polynomial time).

In NIM, both players have the same available moves. For a game where the two players, which we call *Left* and *Right*, have differing moves, we use the notation

$$\{\text{Left's options} \mid \text{Right's options}\}.$$

Such games are called *partizan games*, and they have four *outcome (winnability) classes*. A game in

- \mathcal{N} is won by the player that moves first, no matter whether they are Left or Right;
- \mathcal{P} is won by the player that moves second, no matter whether they are Left or Right;
- \mathcal{L} is won by Left no matter who goes first; and
- \mathcal{R} is won by Right no matter who goes first.

► **Definition 3.** *A superstar is a game in which all options for Left and Right are numbers, possibly not all the same.*

8:4 A Tractability Gap Beyond Nim-Sums

A superstar in which the options for both players are the same is a nimber. Even more in general we have the following:

► **Proposition 4** ([5]). *The superstar*

$$\{0, *, \dots, *(n-1), *x_1, \dots, *x_k \mid 0, *, \dots, *(n-1), *y_1, \dots, *y_l\},$$

where $x_i, y_j > n$ for all i and j , is equal to the nimber $*n$.

When a sum consists of only superstars of this form, the sum is reduced to a sum of nimbers, and is thus solvable in polynomial time. As we show in the main result of our paper in section 3, solving a sum of superstars is NP-hard.

2.2 Naming Superstars

There is some historical overloading of the term “superstar” in two foundational CGT texts, which share an author. In *Winning Ways*, first published in 1982, superstars are defined as we use them here[2]. In *On Numbers and Games*, first published in 1976, the same term is used to describe specific sums of (Winning Ways) superstars. In 2023, Silva et. al.[13] used another term, *quasi-nimbers*, because they were aware of the On Numbers and Games definition, but not the one from Winning Ways.

We do not make this choice lightly. The terminology collision was not known until parts of this paper was presented in the Virtual Combinatorial Games Seminar⁴ in 2023. (No one at the seminar was aware of both definitions beforehand.) We solicited informal advice from the greater CGT community. Based on that, we chose to use the term “superstars”, as in Winning Ways. Although this deviates from the first-published choice in On Numbers and Games, we are comfortable going forward with this because:

- We are still using historical terminology.
- As pointed out by Neil McKay, “superstar” is nice because these games are one move above stars (nimbers) in a game tree.
- Only one published paper uses the term superstar in either context since the two books have been published.⁵

This only solves the issue with superstars from Winning Ways. In order to handle the objects described as superstars in On Numbers and Games, we propose a new term, *comets*, and use that throughout. We like this term because comets are bright celestial objects like (super)stars, but have very little mass in comparison⁶. Additionally, the alliteration of “Conway Comets” works nicely.

We hope that our chosen terminology will continue to be used going forward.

2.3 The Game of Paint Can

PAINT CAN⁷ is a pleasant combinatorial game ruleset that models superstars [13].

⁴ <https://sites.google.com/view/virtual-cgt/seminar>

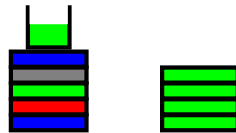
⁵ Combinatorial Game Theory [12], a text published in 2013, references the Winning Ways version in an exercise.

⁶ Although we do not provide the details of this property here, these comet positions have zero atomic weight.

⁷ A playable version of PAINT CAN is available online at: <http://kyleburke.info/DB/combGames/paintCan.html>.

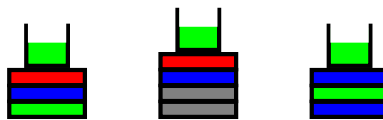
► **Definition 5** (Paint Can). *Paint Can* is a combinatorial game played on stacks of bricks, each colored Red, Blue, Green, or Gray. Each turn a player chooses a brick in one stack either of their own color or Green. (No player can choose Gray bricks.) The chosen brick and all bricks above it are then removed from the stack. On top of each stack that has any non-Green bricks sits a can of green paint. When any brick is taken from that stack, the can of paint spills, coloring all the remaining bricks in the stack Green.

Under Normal Play, the last person to move on a PAINT CAN position wins. Every superstar is equivalent to a PAINT CAN position with a single stack of bricks. Starting with an index of zero for the bottom brick: if brick i has color Blue, then $*i$ is only a Left option; if Red, $*i$ is only a Right option; if Green, $*i$ is both a Left and Right option; and if Gray, $*i$ is not an option for either player. For example, the game $\{ 0, *2, *4 \mid *, *2 \}$ is equal to the stack with bricks colored (from bottom to top) Blue, Red, Green, Gray, and Blue. See Figure 1 for an example of a position with that same stack. The entire position in the figure is equal to $\{ 0, *2, *4 \mid *, *2 \} + *4$.



■ **Figure 1** A PAINT CAN position consisting of two stacks of bricks with value $\{ 0, *2, *4 \mid *, *2 \} + *4$. In the leftmost stack, the Blue player may choose to remove either of the blue bricks or the green brick. The Red player may choose to remove either the red or green brick. Neither player may choose to remove the gray brick. In the rightmost stack, all bricks are already green, so no can of paint is necessary. If the Blue player removes the top brick from the left stack, the result will be a stack of four green bricks, as is in the right stack.

Any sum of superstars can be represented as an instance of PAINT CAN with each term equivalent to a single stack of bricks. For example, to create a position equivalent to $\{ 0, * \mid 0, *2 \} + \{ *2 \mid *3 \} + \{ 0, *, *2 \mid * \}$, we color bricks in each of three stacks corresponding to which player has the number option that matches the index of the brick (starting at the bottom with index 0). If both players have an option to $*i$, then brick i is green. If only Left has an option to $*i$, then brick i is blue. If only Right has an option to $*i$, then brick i is red. If neither player has an option to $*i$, then brick i is gray, though we do not include gray boxes for i higher than the numbers in either option. See Figure 2 for a position equal to the prior sum of superstars. Thus, PAINT CAN is a ruleset where all superstars and sums of superstars occur; players need to evaluate them in order to determine which player can win.



■ **Figure 2** A PAINT CAN position equal to $\{ 0, *1 \mid 0, *2 \} + \{ *2 \mid *3 \} + \{ 0, *, *2 \mid * \}$.

3 From Bits to Superstars: Hardness Reduction

In order to show that sums of superstars (and PAINT CAN) are NP-hard, we need to introduce some additional computational problems.

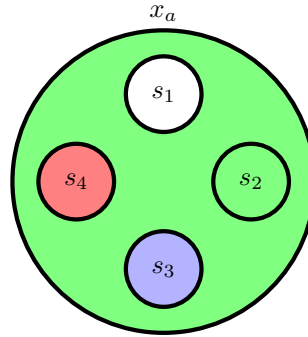
8:6 A Tractability Gap Beyond Nim-Sums

XOR-SAT [11] is a classical logical satisfiability problem consisting of a conjunction of clauses of the XOR of boolean literals. That is to say, it takes this form: $(x_i \oplus x_j \oplus \dots \oplus x_k) \wedge (x_l \oplus \dots \oplus x_p) \wedge \dots \wedge (x_q \oplus \dots \oplus x_r)$. It is known that XOR-SAT is polynomial-time solvable [11].

Our next problem, which is NP-hard, is motivated by XOR-SAT. It uses *multi-state variables*, which can be assigned to one of many states instead of just True and False. Each literal of a variable is labelled with one of those states (e.g. x_{a,s_i}) and is only true if the variable is assigned to that state. More formally, let x_a be a *multi-state* variable with possible states $s_1, s_2, \dots, s_i, \dots, s_k$, then for all states s_i of x_a we have

$$x_{a,s_i} = \begin{cases} \text{True} & \text{if } x_a \text{ is set to } s_i, \\ \text{False} & \text{if } x_a \text{ is set to } s_j \text{ and } j \neq i. \end{cases}$$

Figure 3 displays a multi-state variable.



■ **Figure 3** Multi-state variable x_a with four possible states: s_1, s_2, s_3, s_4 . The overall color indicates that the chosen state is s_2 .

► **Definition 6.** *Multistate XOR-SAT is a ruleset where a position is a conjunction of clauses consisting of the XOR of multi-state literals instead of boolean literals. In other words, the clauses are of the form $(x_{i,s_i} \oplus \dots \oplus x_{j,s_j})$. Variables are divided between the two players, X , and Y , and clauses may contain variables from both players, e.g. $(x_{i,s_i} \oplus y_{j,s_j} \oplus \dots)$. On their turn, the current player selects one of their unassigned variables and picks a state to assign it to. Once both players have assigned all variables, X wins if the formula is true, and Y wins if the formula is false. If both players have the same number of variables, then we call it *Equal-Partitioned Multistate-XORSAT*, or *EPMX*.*

We will show that EPMX is NP-hard after we show the reduction from EPMX to a sum of super stars (AKA PAINT CAN).

To reduce from EPMX, we must first discuss elementary strategies in a sum of superstars. Do aid in this, we partition superstars into six classes:

- numbers,
- *no-0*: neither player has 0 as one of their options,
- *left-0*: only Left has a move to 0,
- *right-0*: only Right has a move to 0,
- *both-0*: both players have moves to 0, and
- *one-sided*: one player has no options while the other player does.

First, an observation that follows directly from Proposition 4:

► **Corollary 7** (No-0 games). *A No-0 game has value 0.*

Then, we will prove the following lemma:

► **Lemma 8** (0 game win). *Consider a sum of superstars with no both-0 games nor one-sided games. If at the start of the Left player's turn there are more left-0 games than right-0 games, then Left wins. Similarly, if at the start of the Right player's turn there are more right-0 games than left-0 games, then Right wins.*

It is possible to prove Lemma 8 using atomic weights and the two-ahead rule, as shown in [13]. We provide the following proof that avoids use of atomic weights.

Proof. We will call the winning player A and the losing player B , so there are more A -0 games than B -0. We will prescribe the following algorithm for A to win: they can “eliminate” 0s in the B -0 games by making a move on the game (by choosing one of their number options arbitrarily). They repeatedly do this until no B -0 games remain. Now, after B 's following turn, the remaining games include at least one A -0 game, some numbers (maybe none), and some no-0 games (maybe none). At this point, A should avoid playing A -0 games until there are only A -0 games remaining, or there is exactly one A -0 game left (along with the other types of games), whatever comes first.

If there are only A -0 games remaining, then for the first of those games, A can just bring a game to 0, and then if there are any games left, B has to make one into a number, which A can just bring to 0. This will repeat until the last A -0 game is taken this way, in which case A wins.

If there is exactly one A -0 game (along with potential no-0 games and numbers), then A can identify the value of the sum of everything but the single game by XORing the numbers (by observation 7, the others have value 0). If the nim-sum is 0, then A may take the move to 0 and thus wins the game. Otherwise, they can bring the nim-sum to 0 and inductively keep it so until B plays on A -0, bringing the game to a non-zero nim sum, which A can then win from. ◀

With this lemma, we can prove the following theorem:

► **Theorem 9.** *There exists a polynomial-time reduction from Equal-Partitioned Multistate XORSAT (EPMX) to Sum of superstars, such that if True wins going first on EPMX, then the outcome class of Sum of superstars is \mathcal{L} or \mathcal{P} (i.e. Left wins going second).*

Proof. Let X be the player whose goal is to make the formula true in EPMX, Y be the player whose goal is to make it false, and m be the number of clauses. We assume that the EPMX formula contains literals for each state of each variable. (If it doesn't, we can create a dummy clause that will always be true for each missing variable missing a state. That clause contains one copy of each state that variable can have.)

We will use the following construction: First, we will assign the t^{th} clause an identity $z_t = 2^t$. In other words, we use power of twos $\{1, 2, 4, \dots, 2^{m-1}\}$ to identify clauses. For the i^{th} variable assigned to X , we will create a Right-0 game which we will call x_i , and for the i^{th} variable assigned to Y , we will create a Left-0 game we will call y_i . Right's options for each x_i contain only a single option of 0, and similarly, Left's options for y_i contain only 0. Then, for each possible state of the i^{th} variable for X , there will be a Left option in x_i to a number whose value is, for each clause that contains the variable at that state, the sum of their corresponding identity values (i.e., if the t^{th} clause is involved, then z_t is included in

8:8 A Tractability Gap Beyond Nim-Sums

the sum). Similarly, for each state of variable y_i for Y , there will be a Right option in y_i to a number with value equal to the sum of the corresponding identity values of the involved clauses. In addition, for each game x_j , there will be an additional option of $*2^{m+j}$. (The y_i positions do not have this extra option.)

The game we consider is then

$$G = x_0 + \dots + x_k + y_0 + \dots + y_\ell + *(2^m - 1).$$

For example, the position $(x_{0,a} \oplus x_{1,a} \oplus y_{0,a} \oplus y_{1,c}) \wedge (x_{0,b} \oplus x_{1,a} \oplus y_{0,b} \oplus y_{1,a}) \wedge (x_{1,a} \oplus x_{1,b}) \wedge (y_{1,a} \oplus y_{1,b} \oplus y_{1,c})$ with states $x_0 : \{a, b\}$, $x_1 : \{a, b\}$, $y_0 : \{a, b\}$, $y_1 : \{a, b, c\}$ reduces to $x_0 + x_1 + y_0 + y_1 + *15$, where

$$\begin{aligned} \text{--- } x_0 &= \left\{ \begin{array}{c|c} \underbrace{*}_a, \underbrace{*2}_b, \underbrace{*16}_{2^{m+0}} & 0 \end{array} \right\} \\ \text{--- } x_1 &= \left\{ \begin{array}{c|c} \underbrace{*7}_a, \underbrace{*4}_b, \underbrace{*32}_{2^{m+1}} & 0 \end{array} \right\} \\ \text{--- } y_0 &= \left\{ \begin{array}{c|c} 0 & \underbrace{*}_a, \underbrace{*2}_b \end{array} \right\} \\ \text{--- } y_1 &= \left\{ \begin{array}{c|c} 0 & \underbrace{*10}_a, \underbrace{*8}_b, \underbrace{*9}_c \end{array} \right\} \end{aligned}$$

In the example above, the identity of these four clauses are respectively, 1, 2, 4, and 8.

Now we demonstrate the correctness of the reduction. As mentioned in the theorem statement, the union of \mathcal{L} and \mathcal{P} is equivalent to proving that Left wins going second. We will show that the game should progress by alternating moves of Right playing on left-0s and Left playing on right-0s. Since the options of those components are all numbers, each of these plays changes the whole game by removing that component and modifying the number term. If this pattern is followed, then after Left plays on the final right-0, they win if and only if the number term has been reduced to zero. Otherwise, Right can bring the nim-sum to 0, and then win through following the nim strategy.

If both continue to hold to that pattern, at the beginning of each of Left's turns, there is one more right-0 component than left-0. Thus, Left must play on a right-0 component or they will lose by Lemma 8. Right starts each turn with balanced left-0s and right-0s, but they still need to follow the pattern. If Right deviates by playing on an x game, then they will lose by Lemma 8. If Right plays on the number term instead, this switches the roles of the players with respect to their ending conditions; now Right will win if and only if the number term is reduced to zero when they make their final move. Left, however, can avoid this by playing on one of the large number (with value at least 2^m) options included in any x . Right doesn't have any options that can cancel out that large number, so the final number term will always be non-zero and Right can no longer win.

Following the prescribed sequence of play, Left wins if the number term equals zero. Since it started at $*2^m - 1$, the sum of all number options chosen must also equal $*2^m - 1$. If Left has a winning strategy in EPMX, they can play on the nim values corresponding to each variable state in their winning strategy, which must then result in a final nimsum of 0. If they do not have a winning strategy, then note that playing on the 2^{m+j} values can't give them a chance to win, since Right can stick with their EPMX strategy and the nimsum can't equal 0. Thus, Right can follow Y 's winning strategy of assignments to result in a non-zero value. \blacktriangleleft

Now we will show NP-hardness for EPMX.

► **Theorem 10.** *There exists a polynomial-time reduction from 3SAT to EPMX.*

Proof. Let X be the player whose goal is to make the formula true in EPMX, Y be the player whose goal is to make it false. (In our reduction, Y will not make meaningful decisions in the course of the game.) Let n be the number of variables and m will be the number of clauses from the 3SAT instance.

Note that 3SAT is hard even if every variable appears only at most 3 times, at least once negated and at least once unnegated. It is also hard adding on a further restriction that there are an odd number of variables in the formula. We will assume both of these are true in this reduction.

For each clause in 3SAT, we will create a clause in EPMX. We will also have two separate clauses c_x and c_y . For each variable (x_i) in 3SAT, we will create an variable (x_i) in EPMX with five states: $\{x_{i_a}, x_{i_b}, x_{i_c}, x_{i_d}, x_{i_e}\}$. WLOG, we assume that x_i appears once unnegated and twice negated, in clauses r, s , and t in 3SAT respectively. Every state appears in c_x . In the EPMX formula, the first state (x_{i_a}) appears in no other clauses. The second state, x_{i_b} , corresponds to the unnegated appearance, and appears in clause r . The third state, x_{i_c} , corresponds to both negated 3SAT literals, and appears in clauses s and t . The fourth state, x_{i_d} , corresponds to only the first negated 3SAT literal, and appears only in clause s . Finally, the fifth state, x_{i_e} , corresponds to only the second negated literal, and only appears in clause t .

In order to be an EPMX position, we need to include the same number of y variables as x . We can do this by creating n dummy variables with two states each: a and b . We can include all of the states of each of the variables into c_y . Note that since there are an odd number of y variables, c_y will always be true (the same is true of c_x).

If there is a solution to the 3SAT formula, then a solution to EPMX can be constructed by iterating over the variables. For each x_i , if the 3SAT assignment is true, then we choose x_{i_b} , unless the EPMX clause r has already been satisfied, in which case we choose x_{i_a} . If x_i is assigned to false, then we select the correct choice of a, c, d , and e , depending on which of clauses r and t have already been satisfied.

The inverse direction is simpler: an assignment of x_{i_a} means it doesn't matter what we pick. x_{i_b} means x_i must be true to satisfy the 3SAT formula, and any of the others means it must be assigned to false. ◀

As explained in Section 2.2, we introduce the term comet to refer to the objects called superstars in *On Numbers And Games*[5]. Each superstar has an associated comet. We do not provide a full explanation of all cases of comets here. However, if the superstar, S , is a left-0, then the comet will be $\downarrow + * + S$, where $\downarrow = \{ * \mid 0 \}$. If S is a right-0, then it will be $\uparrow + * + S$ where $\uparrow = \{ 0 \mid * \}$. Finally if S is a number, then S is its own comet [5, 13]. Note that $\downarrow + \uparrow = 0$.

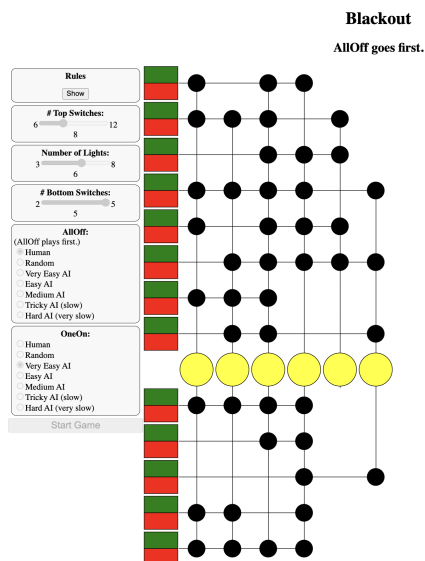
► **Corollary 11.** *A sum of comets is NP-hard.*

Proof. We can express our sums of superstars and a number resulting from the reduction from EPMX as a sum of comets. If we replace each x_i with the comet $\uparrow + * + x_i$ and each y_i with the comet $\downarrow + * + y_i$, then sum all of those comets, all the \uparrow and \downarrow components will cancel out and all of the individual $*$ will cancel each other out. Thus the game is the same sum of superstars. ◀

A minor note is that comets have something known as atomic weight 0, which also gives the result that sums of atomic weight 0 games can be NP-hard.

4 From Logic to Board Game: Blackout

In this section, we use a simplified version of the logical game that appeared in our earlier analysis to design a two-player board game, which we call BLACKOUT. The board of this game contains an array of light bulbs and two sets of switches, one above the light bulbs and one below. Two players, denoted by AllOff (Left) and OneOn (Right), each control one set of switches.



■ **Figure 4** An Example Game of BLACKOUT.

The AllOff player wins if all lights are off at the end of the game. They control the switches at the top of the board. The OneOn player wins if at least one light is on in the end. They use the switches at the bottom of the board. On their turn, a player turns one of their unchosen switches off (red) or on (green). If they turn the switch on, then all of the lights that switch is connected to get toggled (off becomes on and on becomes off). The OneOn player has an easier objective, so they may have fewer switches. Once all bottom switches are played, they can pass as long as all the lights are not out. During these final turns, the AllOff player is searching for configuration of their remaining switches to turn all lights off in order to win.

BLACKOUT is in the Maker-Breaker style, but it differs from traditional Maker-Breaker games in one crucial aspect: Even if AllOff turns all lights off at some point during the game, if OneOn still has some unflicked switches, then the game will continue until all switches for both players have been selected. I.e., BLACKOUT could potentially continue even after all light bulbs have been turned off, while Maker-Breaker games end as soon as the desired structure has been formed (or continue until there are no moves).

Figure 4 is an illustration from our two-dimensional board layout in our Web implementation, which can be found at the following link: <http://kyleburke.info/DB/combGames/blackout.html>.

4.1 Blackout Ruleset Formalities

In this subsection, we discuss the mathematical representation of positions and rules in BLACKOUT to set up our computational analysis. For BLACKOUT games with p lights, a position has three components $P = (L, S^{\text{AllOff}}, S^{\text{OneOn}})$:

- (a) L is a Boolean vector in $\{0, 1\}^p$, indicating whether each of the lights is off or on. That is, $L(i) = 1$ denotes that the i^{th} light is on.
- (b) S^{AllOff} is a Boolean matrix with p columns in which every row has at least one 1. Each row represents a switch that AllOff can use. If the switch controls the i^{th} bulb, then its i^{th} entry equals to 1.
- (c) S^{OneOn} is a Boolean matrix with p columns, in which every row has at least one 1. Each row represents a switch that player OneOn can use. If the switch controls the i^{th} bulb, then its i^{th} entry equals to 1.

In this position, we define the options as follows:

- AllOff has two options for each row in S^{AllOff} . (If the matrix is empty, then they have no options remaining.) For their turn, AllOff selects a row r and a binary action $\alpha \in \{0, 1\}$. If the action is $\alpha = 0$, then the position is moved to $(L, S_{-r}^{\text{AllOff}}, S^{\text{OneOn}})$, where S_{-r}^{AllOff} denotes the Boolean matrix obtained from S^{AllOff} by removing its r^{th} row. If the action is $\alpha = 1$, then let L' be the entry-wise exclusive-or of L and the r^{th} row of S^{AllOff} , and the position is moved to $(L', S_{-r}^{\text{AllOff}}, S^{\text{OneOn}})$. Note L' represents the result when player AllOff activates its r^{th} switch.
- OneOn's options depend on two cases: (1) If S^{OneOn} is not an empty matrix, then OneOn can select a row s and a binary action $\beta \in \{0, 1\}$. If the action is $\beta = 0$, then the position is moved to $(L, S^{\text{AllOff}}, S_{-s}^{\text{OneOn}})$, where S_{-s}^{OneOn} denotes the Boolean matrix obtained from S^{OneOn} by removing its s^{th} row. If the action is $\beta = 1$, then let L' be the entry-wise exclusive-or of L and the s^{th} row of S^{OneOn} , and position is moved to $(L', S^{\text{AllOff}}, S_{-s}^{\text{OneOn}})$. Note L' represents the result when player OneOn flicks on its s^{th} switch. (2) If S^{OneOn} is an empty matrix (i.e., player OneOn has no more switches to flick), then OneOn can make a pass move so long as there is at least one light on in L . If the lights are all out, then they have no options available.⁸)

4.2 The Intractability of Blackout

We now analyze the complexity of BLACKOUT and prove the following intractability result.

► **Theorem 12** (Intractability of Blackout). *Deciding whether or not player ALLOFF has a winning strategy at a given Blackout position is NP-hard.*

Proof. We begin by defining two decision problems, SET COVER and EXACT COVER.

In SET COVER, there is a collection V containing n sets S_1, S_2, \dots, S_n which each contain some subset of a ground set $E = \{e_1, \dots, e_m\}$ of m elements. There is also a given integer k , indicating a target number of sets to choose.

A SET COVER instance is feasible if there exists a selection of k sets in V such that every element in E is in at least one of the selected k sets. We call such selection a *cover*. A cover is *exact* if for each element $e \in E$, e appears exactly once in the selected sets. EXACT COVER determines whether the input has an exact cover.

⁸ In order to prevent OneOn from making unbounded passes in the context of a game sum, they should have a maximum number of passes at the beginning equal to the difference in heights of the matrices, $\text{height}(S^{\text{AllOff}}) - \text{height}(S^{\text{OneOn}})$.

8:12 A Tractability Gap Beyond Nim-Sums

In our desired problem of PURE SET COVER, we want to have a promise that if there is a SET COVER of size k , then there is also a EXACT COVER of size k .⁹

Next, we note that SET COVER is NP-complete even if there are only three elements in each set in V [7].

Now we can reduce from SET COVER with three elements in each set. For our reduction, we will enrich the input by adding new sets for each subset of those sets. E.g., if $S_1 = \{e_1, e_2, e_3\}$, then we include the six sets $\{e_1\}$, $\{e_2\}$, $\{e_3\}$, $\{e_1, e_2\}$, $\{e_1, e_3\}$, and $\{e_2, e_3\}$ in our new collection V as well. This enforces the promise, because if there is a set cover of size k , for each overlap, for one of the overlapping sets, one can instead choose to select a subset that doesn't overlap. We can repeat this for all overlapping sets without changing k .

In other words, with this enrichment, we have proved that PURE SET COVER is also NP-complete.

To reduce from PURE SET COVER to BLACKOUT, we create a light switch for the Alloff player for each set in the PURE SET COVER and $n - k$ light switches for the OneOn-player. We create a light for each element in E .

The switches are set as the following:

- **(Alloff)**: For the n light switches controlled by the Alloff player, we connect them to the lights corresponding to the elements contained in the sets, one for each set.
- **(OneOn)**: We connect $n - k - 1$ of the light switches controlled by the OneOn-player only to the lights corresponding to the elements in S_1 , which the Alloff player also has a switch for. Finally, the OneOn-player's last light switch is connected to all of the lights.

All lights begin in the on state and OneOn goes first.

We claim that the Alloff-player has a winning strategy if and only if there exists a Set Cover in the PURE SET COVER.

The OneOn player has $n - k - 1$ switches for S_1 , $n - k - 2$ of which are redundant, so they should start by playing all of these, without the setting changing the outcome of the game.

If there is a working k -sized cover, Alloff will spend their first $n - k - 2$ turns choosing to turn off switches that are not in their pure cover and not S_1 . OneOn, seeing that Alloff has a switch to negate theirs for S_1 , should play that again. If S_1 is part of the cover, then Alloff can choose the setting that shuts them off. Otherwise, they should choose to turn them on. OneOn will then choose to leave all lights on (otherwise Alloff can win immediately). Now Alloff has k turns to flip all switches in their cover of size k to turn all lights off. If OneOn decides to activate the switch connected to all lights earlier, they should choose to leave them all on, in which case Alloff just has extra turns to put their cover to work. No matter what, Alloff will win.

Then, if there exists no Exact Cover of size k , then there exists no set cover of size k . OneOn can win by saving their all-lights-switch for their last move. Since there is no set cover of size k , there must be at least one light not covered by the Alloff player's remaining switches. The OneOn player can either flip or not flip the final switch to make sure that light is on and win the game. ◀

⁹ We thank Neal Young for this idea.

5 Conclusion

Going beyond winnability, Sprague-Grundy Theory [14, 8] introduced the first notion of *game values* and *game algebra* for combinatorial games, which was later expanded to partizan games [2, 5]. It demonstrated that every impartial game can be mathematically reduced to a single-pile of NIM in the algebra of disjunctive sums. Because the nim sum can be computed in linear-time in the size of the binary representations of the summands [3], Sprague-Grundy Theory further captures the computational benefit of knowing game values of impartial games, rather than just their game rules or winnability [4].

By showing that the winnability of the sum of superstars and comets are intractable, our result highlights the fundamental subtlety of tepid partizan game values. It takes a significant step beyond Morris' [10] classical intractability result by demonstrating that intractability happens just one step above numbers (stars). This hardness result implies that the Bouton-like result for NIM [3] is unlikely for superstars and comets.

Part of our proof has also inspired the design of a board game, BLACKOUT, which enjoys intriguing complexity based on the shape of game boards: On the one hand, if both players have the same number of switches, which we refer to as the balanced case, then the game can be solved in polynomial time¹⁰. On the other hand, if players have a different number of switches, then the game is intractable in general.

Whereas the sum of superstars has been shown to be NP-hard to solve, its precise complexity remains open. It can be shown that the outcome class of a sum of superstars can be computed in polynomial space in the number of bits representing the sum.¹¹ Therefore, as part of the next step in our future research, we would like to settle the following conjecture.

► **Conjecture 13.** *A sum of superstars, and hence Paint Can, is PSPACE-complete.*

It is worth noting that it seems challenging to naturally extend the current setup of our intractability proof for this conjecture. Reducing directly from QSAT to EPMX seems fruitless since there is no clear way to “punish” player Y from covering a variable multiple times. For any reduction, the fundamental difficulty lies in the strong asymmetry between players X and Y in that Y is just too powerful with options. In other words, as soon as EPMX allows Y to be a “real” decision maker – as opposed to the construction in Theorem 10 – the game shifts dramatically in Y’s favor, and indeed, it is difficult to even find complicated positions where X wins with a non-trivial Y player. As it stands, either a different approach is needed, or a very clever reduction to EPMX is required.

References

- 1 M. H. Albert, R. J. Nowakowski, and D. Wolfe. *Lessons in Play: An Introduction to Combinatorial Game Theory*. A. K. Peters, Wellesley, Massachusetts, 2007.
- 2 Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for your Mathematical Plays*, volume 1. A K Peters, Wellesley, Massachusetts, 2001.
- 3 Charles L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1/4):pp. 35–39, 1901. URL: <http://www.jstor.org/stable/1967631>.
- 4 Kyle Burke, Matthew Ferland, and Shang-Hua Teng. Winning the war by (strategically) losing battles: Settling the complexity of Grundy-values in undirected geography. In *Proceedings of the 62nd Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2021.

¹⁰This statement follows from the basic idea in [11].

¹¹One of the proofs was given by Aaron Siegel.

8:14 A Tractability Gap Beyond Nim-Sums

- 5 J.H. Conway. *On Numbers and Games*. A.K. Peters. A.K. Peters, 2000. URL: <https://books.google.com/books?id=tXiVo8qA5PQC>.
- 6 Aviezri S. Fraenkel, Edward R. Scheinerman, and Daniel Ullman. Undirected edge geography. *Theor. Comput. Sci.*, 112(2):371–381, 1993.
- 7 M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63, New York, NY, USA, 1974. Association for Computing Machinery. doi:10.1145/800119.803884.
- 8 P. M. Grundy. Mathematics and games. *Eureka*, 2:198—211, 1939.
- 9 David Moews. *On some combinatorial games connected with Go*. PhD thesis, Citeseer, 1993.
- 10 FL Morris. Playing disjunctive sums is polynomial space complete. *International Journal of Game Theory*, 10(3-4):195–205, 1981.
- 11 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 216–226, New York, NY, USA, 1978. Association for Computing Machinery. doi:10.1145/800133.804350.
- 12 A.N. Siegel. *Combinatorial Game Theory*. Graduate Studies in Mathematics. American Mathematical Society, 2013. URL: <https://books.google.com/books?id=VUVrAAAAQBAJ>.
- 13 Alexandre Silva, Carlos Pereira dos Santos, João Pedro Neto, and Richard J. Nowakowski. Disjunctive sums of quasi-nimbers. *Theoretical Computer Science*, 945:113665, 2023. doi:10.1016/j.tcs.2022.12.015.
- 14 R. P. Sprague. Über mathematische Kampfspiele. *Tôhoku Mathematical Journal*, 41:438—444, 1935-36.
- 15 David Wolfe. Go endgames are PSPACE-hard. In Richard J. Nowakowski, editor, *More Games of No Chance*, volume 42 of *Mathematical Sciences Research Institute Publications*, pages 125–136. Cambridge University Press, 2002.
- 16 Laura Jo Yedwab. *On playing well in a sum of games*. PhD thesis, Massachusetts Institute of Technology, 1985.

The Steady-States of Splitter Networks

Basile Couëtoux ✉

Aix-Marseille Université, CNRS, LIS, Marseille, France

Bastien Gastaldi ✉

Télécom SudParis, Institut Polytechnique de Paris, Evry, France

Guyslain Naves ✉ 

Aix-Marseille Université, CNRS, LIS, Marseille, France

Abstract

We introduce splitter networks, which abstract the behavior of conveyor belts found in the video game Factorio. Based on this definition, we show how to compute the steady-state of a splitter network. Then, leveraging insights from the players community, we provide multiple designs of splitter networks capable of load-balancing among several conveyor belts, and prove that any load-balancing network on n belts must have $\Omega(n \log n)$ nodes. Incidentally, we establish connections between splitter networks and various concepts including flow algorithms, flows with equality constraints, Markov chains and the Knuth-Yao theorem about sampling over rational distributions using a fair coin.

2012 ACM Subject Classification Theory of computation → Network flows; Mathematics of computing → Network flows; Mathematics of computing → Graph algorithms; Theory of computation → Random walks and Markov chains

Keywords and phrases Factorio, splitter networks, flow, balancer, steady-state

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.9

Related Version *Extended Version*: <https://arxiv.org/abs/2404.05472>

1 Introduction

The transportation of materials or data within various networks represents an inexhaustible source of mathematical problems, which has led to almost as many solutions, theories and algorithms. These advancements have brought about significant improvements across diverse fields including supply chain management, logistics, network optimization. Transportation also serves as a central component in numerous games, as evidenced by the transportation category on BoardGameGeek which lists almost two thousand games [3]. In Factorio [24], a video game published in 2020 by Wube Software, players must mine natural resources to feed a rocket-building factory on an hostile planet. A major part of the gameplay involves the movement of resources within the factory, employing various mechanism: robotic arms, conveyor belts, drones or trains.

In this work, we study the conveyor belts of Factorio. An item placed on a belt will move at a constant speed toward the end of the belt, until it reaches that end, or is blocked by an item preceding it. Belts in Factorio can be combined using a splitter, connecting one or two incoming belts to one or two outgoing belts. A splitter takes items from the incoming belts and places them on the outgoing belts, trying to split the flow as fairly as possible between the incident belts, while maximizing the throughput. Given the scale of a typical Factorio game, players frequently encounter the need to balance the loads across multiple belts, and the community has devised numerous efficient networks to address this load-balancing problem.

An intriguing aspect of Factorio is its encouragement for players to construct vast systems of automation, requiring intensive planning and optimization. Ultimately, the limiting factor arises from the CPU load generated by game state updates. Consequently, players are



© Basile Couëtoux, Bastien Gastaldi, and Guyslain Naves;
licensed under Creative Commons License CC-BY 4.0

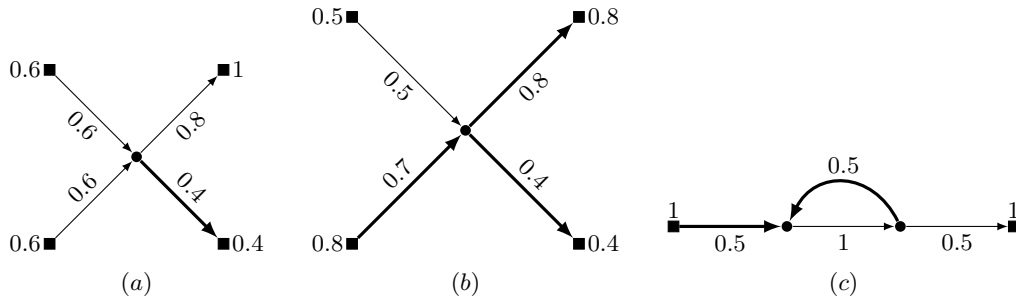
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 9; pp. 9:1–9:14

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Three splitter networks given with capacities and associated steady-states. Splitter will be represented by circle vertices, terminals by square vertices. Each terminal is tagged by its capacity, and each arc by its throughput. Saturated arcs are bolder than fluid arcs.

incentivized to prioritize resource efficiency, particularly concerning gameplay elements that entail frequent computations such as splitters. This motivates the minimization of the number of splitters in load-balancing networks.

Our goal is two-fold: first we model the steady-state of a network of splitters. The network of conveyor belts is abstracted as a directed graph, with nodes corresponding to splitters and arcs to belts. A steady-state is a throughput function on the arcs; a circulation with additional constraints to capture the fact that splitters are fair and locally optimizing. We present two polynomial-time algorithms for computing a steady-state in a splitter network. An analogy is made with two classical maximum-flow algorithms: the blocking-flow algorithm [8] and the push-relabel algorithm [10]. In contrast to maximum flows, the primary challenge arises when a belt reaches full capacity, as its supplying splitter may no longer stay both fair and maximizing. In that case, the splitter is allowed to become unfair, but that decision changes the constraints applied to the flow, making the problem fundamentally non-convex. In a second part, we showcase various load-balancing network designs sourced from the Internet, formalizing concepts defined by the players community. Furthermore, we prove that those designs approach optimality. Specifically, we prove that any balancing network on n belts must have $\Omega(n \log n)$ splitters, by exhibiting a relation with the problem of sampling the uniform distribution over a set of n elements using only a fair coin. The core design is the Beneš network, a circuit-switching network well-known in the field of telecommunication [1, 2].

The blocking-flow-like algorithm relies on finding circulations with equality constraints. A *circulation* on a directed graph is a flow without any excess at any vertex. Given a directed graph (G, A) , we denote $\delta^+(v)$ and $\delta^-(v)$ the sets of outgoing and incoming arcs incident to a vertex v . Let $\mathcal{C}^=$ be a partition of A such that for each part $C \in \mathcal{C}^=$, there is some vertex v with $C \subseteq \delta^+(v)$. The $\mathcal{C}^=$ -circulation problem is to decide whether there is a non-zero circulation f that is constant within each part. While this problem can easily be solved using linear programming, we require a good characterization of graphs admitting a $\mathcal{C}^=$ -circulation. Additionally a polynomial-time algorithm is needed to either construct a $\mathcal{C}^=$ -circulation or identify an obstacle that prevents its existence. The algorithm relies on the computation of a stationary distribution of an auxiliary graph. In contrast, solving maximum integral flow problems with additional equality constraints is known to be NP-hard [17], even when the partition is exactly the sets of leaving arcs of each vertex [23, 18].

Sorting networks [12] and Beneš networks have topologies similar to splitter networks, with nodes of in-degree and out-degree 2. In microfluidics, mixing graphs are used to produce droplets of specific concentration, using devices that produces two identical droplets from two droplets of any concentration [7]. The concentration values on the arcs are subject to equality

constraints similar to those of splitter networks, but without a maximizing constraint. The topology of splitter networks is nonetheless more general than these examples, as splitter networks may have directed cycles, those being necessary in particular to achieve load-balancing with an arbitrary number of outputs.

In an answer to a question on the mathematics section of `stackexchange`, David Ketcheson attempted to model and compute the throughputs of splitter networks [11]. Rather than binary categorizing each belt as full or not, each arc is assigned a density and a velocity. The density will be monotonically increasing, and the velocity monotonically decreasing during the run of the algorithm, until a steady-state is reached. In fact the velocity increases only after the density reaches its maximum at one. Therefore this description is equivalent to our solution, which involves a throughput function and a set of full belts. Unfortunately his algorithm does not always terminate, and its solutions do not satisfy that splitters use their incoming belts fairly. Ketcheson also gave a procedure, albeit non-polynomial, to determine whether a network (not necessarily load-balancing) may limit throughput. However, this procedure is applicable only to networks without directed cycle. In [15], Leue modeled splitter networks using Petri nets, and uses model checking to check the load-balancing properties of some small networks.

The Factorio community is very active and creative. Players have designed load-balancing networks of various sizes, with efficient embeddings into the grid while respecting the constraints of the game. Additionally, they have developed general methods for constructing arbitrary large load-balancing networks. They introduced the concept of balancing networks, along with the more robust properties of being throughput unlimited or universal, and subsequently designed networks that exhibit these characteristics. A notable example is the universal balancer presented by *pocarski* [20], although it uses non-fair splitters too; our universal balancer only uses fair splitters. They also discovered the relationship with Beneš networks. Factorio-SAT [21] is a project that uses a SAT-solver to find optimal embeddings of splitter networks in the grid. The project *VeriFactory* uses a SAT-solver to check various load-balancing properties of splitter networks [14]. Factorio belts are actually sufficiently complex to be Turing-complete [16]. There are many implementations of various devices inside Factorio, ranging from raytracers to programming language interpreters, using the diverse set of available gameplay mechanisms. Factorio has been the inspiration for several other academic works [22, 19, 4, 6, 9].

The rest of this paper presents an overview of the main concepts and results of this work. An extended version [5] will contain more details, proofs and additional results. Splitter networks and their steady states are defined in Section 2. Section 3 describes two algorithms to compute the steady-state of a splitter network. The concept of balancer is defined in Section 4, which also contains a presentation of some balancer designs. Section 5 describes how to derive a lower bound on the number of splitters in a balancer network. Finally in Section 6 we will present some perspectives.

2 Splitter networks and their steady-states

We start by modeling networks of conveyor belts and splitters by directed graphs, where each single belt is an arc, and each splitter is a node (thus abstracting the length of the belts).

► **Definition 1.** A splitter network is a directed graph G (with possible loops or parallel arcs) whose vertex set can be partitioned into three sets $V(G) = I \uplus S \uplus O$ where

- (i) I is the set of inputs, and $d^+(i) = 1$, $d^-(i) = 0$ for any input i ;
- (ii) O is the set of outputs, and $d^-(o) = 1$, $d^+(o) = 0$ for any output o ;
- (iii) S is the set of splitters, and $d^-(s) = d^+(s) = 2$ for any splitter s .

9:4 The Steady-States of Splitter Networks

We will use the word *flow* to informally describe the material transported by the network, and *throughput* for the amount of flow going through the arcs. Our work aims to understand the throughputs inside a splitter network at steady state, when some maximum throughputs are forced on its inputs and its outputs, which are respectively the sources and sinks of the flow passing through the network. To this end we will consider capacity functions on the input and output. A capacity c on an input means that the input has an incoming flow of throughput c . The input will try to push that much into the network, but no more. A capacity c on an output means that the output will accept a maximum throughput of c . We consider that the maximum throughput of any arc is 1, with all belts being identical.

A splitter can be described using two operational rules. The first rule, which takes precedence, is to maximize the amount of flow that goes through it. The secondary rule is to be fair. A splitter is fair relatively to its outgoing arcs: it tries to push as much flow onto each of them. It is also fair relatively to its incoming arcs: it tries to pull as much flow from each of them. As the maximization rule takes precedence, it will not be fair when being unfair leads to higher throughput. For instance, consider the network in Figure 1 (a), depicting a network with a single splitter. As one of the output has a lower capacity, it pushes more flow toward the other output, thereby maximizing the total throughput, while still being as fair as possible as it minimizes the difference of throughputs on its outgoing arcs.

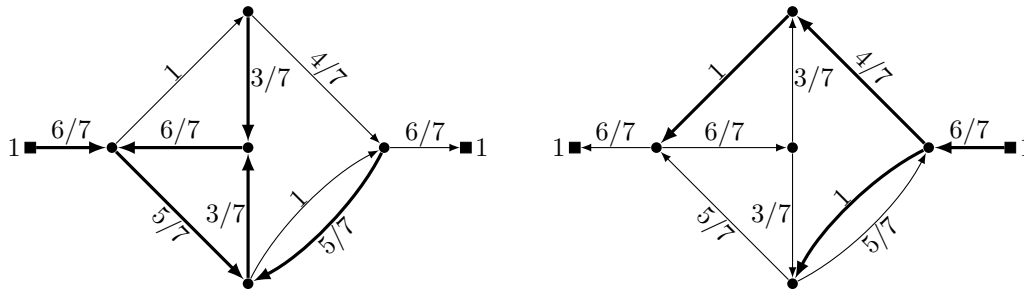
The throughput of an arc may reach a limit when its head is an output with a low capacity. For example in Figure 1 (a), an output of capacity 0.4 acts as a bottleneck. In other cases the head of an arc is a splitter, which itself is limited by what its outgoing arcs can accept. For example in Figure 1 (b), as all the outputs have reached their capacities, the splitter cannot accept more flow, even if the bottom input could provide even more flow. In terms of conveyor belts, some belts will initially receive more items that they can deliver, causing them to fill up. Once full, they can only accept from upstream as much as they deliver downstream, which may in turn limit throughputs upstream. We say that such belts are *saturated*.

The output capacities are not the only factor that limit the total throughput and create bottlenecks. This can be observed in Figure 1 (c). There, the rightmost splitter tries to be fair and send some of the flow back to the left. The leftmost splitter also tries to be fair, thus accept the flow coming from the right. This results in the stabilization into the given throughputs. This example illustrates that the throughput is not globally maximized, contrary to the expectation of a total throughput of 1 for this network. Instead, it is only 0.5.

The following definition formalizes the notions of capacity, throughput and saturations, as well as the behaviour of splitters related to the flow going through the network in a steady state.

► **Definition 2.** Let $G = (I \uplus S \uplus O, E)$ be a splitter network, and let $c : I \cup O \rightarrow [0, 1]$ be the maximal capacities of each input and output node. A steady-state for (G, c) is a pair (t, F) where

- R1** $t : E \rightarrow [0, 1]$ is the throughput function;
- R2** $F \subseteq E$ is the set of fluid arcs, $E \setminus F$ is the set of saturated arcs;
- R3** for each $i \in I$ with $\delta^+(i) = \{e\}$, $t(e) \leq c(i)$ and moreover if $e \in F$ then $t(e) = c(i)$;
- R4** for each $o \in O$ with $\delta^-(o) = \{e\}$, $t(e) \leq c(o)$ and moreover if $e \notin F$ then $t(e) = c(o)$;
- R5** for each $s \in S$, with $\delta^-(s) = \{e_1, e_2\}$ and $\delta^+(s) = \{e_3, e_4\}$, $t(e_1) + t(e_2) = t(e_3) + t(e_4)$;
- R6** for any $e_1, e_2 \in E$ with $\{e_1, e_2\} = \delta^-(s)$ and $e_1 \notin F$, $t(e_1) \geq t(e_2)$;
- R7** for any $e_1, e_2 \in E$ with $\{e_1, e_2\} = \delta^+(s)$ and $e_1 \in F$, $t(e_1) \geq t(e_2)$;
- R8** for any $uv \in E \setminus F$ and $vw \in F$, $t(uv) = 1$ or $t(vw) = 1$.



■ **Figure 2** An example of steady-state in a moderately small network, and the reverse network with its steady-state obtained by reversal. Notice that the reversed steady-state satisfies rule 8 but not rule 9.

Rules 3 and 4 say that the throughputs are limited at each input and each output, and moreover, an input pushes as much flow as allowed by its capacity on a fluid arc. Similarly an output absorbs as much flow as allowed by its capacity from a saturated arc. Rule 5 imposes the conservation of flow. Rules 6 and 7 enforce the fairness constraints: a splitter consumes no less flow from a saturated arc than from another incoming arc. A saturated arc represents a belt that is full. Therefore, the splitter is not limited in how much flow it can pull from that arc, and thus cannot pull less than from the other incoming arc. Similarly it produces no less flow in a fluid outgoing arc than in another outgoing arc. In particular, if both incoming arcs are saturated, or if both outgoing arcs are fluid, they must have equal throughput, suggesting the following definition.

- **Definition 3.** Given a splitter network $G = (I \uplus S \uplus O, E)$, and a set $F \subseteq E$ of fluid arcs, we say that two arcs $e, e' \in E$ are
- in-coupled if $e, e' \notin F$ and there is a splitter vertex $v \in S$ with $\delta^-(v) = \{e, e'\}$,
 - out-coupled if $e, e' \in F$ and there is a splitter vertex $v \in S$ with $\delta^+(v) = \{e, e'\}$,
 - coupled if they are in-coupled or out-coupled.

Finally rule 8 imposes the maximization of the throughput by each splitter. Indeed, a saturated arc can provide more flow, while a fluid arc can absorb more flow. Thus, a steady-state cannot contain a saturated arc followed by a fluid arc. The only exception is when one of them already has a throughput of 1. We will sometimes replace rule 8 by a stronger maximization rule :

R9 for any arcs $uv \in E \setminus F$ and $vw \in F$, $t(vw) = 1$.

Rule 9 implies rule 8, and although the converse is not true, any steady-state can be modified into a steady-state for which rule 9 also holds.

The definitions of splitter networks and steady-states exhibit a remarkable symmetry. By reversing each arc, exchanging the role of inputs and outputs, and complementing the set of fluid arcs, a steady-state is transformed into a steady-state of the reverse graph, as seen in Figure 2.

For convenience, when defining or representing splitter networks, we will allow splitters with in-degree one or out-degree one (see Figure 2 for instance). This is justified by the fact that if a splitter s has in-degree one, we can add a dummy input node i with capacity $c(i) = 0$. An arc from i to s can then be added, that will always remain fluid. Similarly if s has out-degree one, we can add a dummy output node o with capacity $c(o) = 0$, and an always-saturated arc from s to o . The throughputs on those arcs are forced to be 0. Therefore it does not induce any new constraint on the non-dummy arcs as rules 6 and 7 are clearly true for those arcs.

Additionally, for convenience, for any input $i \in I$ with outgoing arc e , we note $t(i) := t(e)$, and similarly for any output $o \in O$ with incoming arc e , $t(o) := t(e)$. We also extend the capacities to arcs by setting $c(e)$ to be either $c(i)$ if $e \in \delta^+(i)$, $i \in I$, or $c(o)$ if $e \in \delta^-(o)$, $o \in O$, or 1 otherwise.

3 Existence and computation of steady-states

Let F be a fixed set of fluid arcs. Then the set of possible throughput functions t of a steady-state (t, F) can be described as a polyhedron. Indeed, each of the rules 1, 3, 4, 5, 6, 7 can be encoded by linear inequations. Rule 8 is non-convex, but we will later introduce its slight strengthening, rule 9. That stronger rule admits an encoding as a family of linear inequations. Thanks to linear programming, finding a steady-state thus reduces to finding a set of fluid arcs that admits a steady-state. Nevertheless, we still need to find F . We propose two algorithms to compute a steady-state, which relates to two families of maximum flow algorithm:

- a push-relabel-like algorithm, where we relax the conservation rule 5, thus defining a *pre-steady-state* by analogy with *pre-flows*. Given a set F , we use a linear program to compute an optimal pre-steady-state (t, F) (for some well-chosen objective), and prove that either (t, F) is a steady-state, or there is an arc $e \in F$ such that $(t, F \setminus e)$ is also a (non-optimal) pre-steady-state. Then after at most $|E|$ steps we get a steady-state;
- a blocking-flow-like algorithm, where we relax the rule 3 on input capacities, removing the requirement that an input whose throughput is less than its capacity must have a saturated outgoing arc. This defines the notion of *sub-steady-state*. Given a set F , we solve a linear system to find a sub-steady-state t , and prove once again that either (t, F) is a steady-state or there is an arc $e \in F$ such that $(t, F \setminus e)$ is a sub-steady-state.

The pre-steady-state algorithm is technically simpler but requires an LP-solver. The sub-steady-state only requires an algorithm to compute stationary distributions in directed graphs. We defer a complete presentation and proof of these algorithms to the extended version of this paper, and focus here on explaining the sub-steady-state algorithm.

► **Definition 4.** Given $G = (I \uplus S \uplus O, E)$ a splitter network with capacities $c : I \uplus O \rightarrow [0, 1]$, a sub-steady-state for (G, c) is a pair (t, F) satisfying rules 1, 2, 4, 5, 6, 7 and the strong maximization rule 9, and for any $i \in I$ and $e \in \delta^+(i)$, $t(e) \leq c(i)$.

The algorithm starts with the trivial sub-steady-state $(t : e \rightarrow 0, E)$, and will improve it iteratively until reaching a steady-state. At each iteration of the algorithm, we will be trying to increase the throughputs of the arcs without violating any rule. Unlike in maximum flows, we do not have the choice of which leaving arc to increase the flow on. Furthermore, rule 8 forces each splitter to send as much flow forward as possible. A non-obvious consequence is that, when increasing the input capacities, throughputs can only increase on fluid arcs, and can only decrease on saturated arcs. This suggests a definition of the residual graph for the sub-steady-state (t, F) . Its vertex set is $\{z\} \cup S$, where z is obtained by identifying all the inputs and outputs into a single node. Its edge set contains some fluid arcs and the reverses of some saturated arcs.

Consider the splitters in Figure 3. We examine what happens when we increase the throughput on edge e_1 by $+\varepsilon$, or in case (d) when we decrease $t(e_3)$ by ε . In case (a), by rule 7, the throughputs on the two leaving arcs must stay equal, hence both increases by $\varepsilon/2$. In case (b), only the throughput of the fluid leaving arc e_4 can increase. In case (c), both leaving arc are saturated, the splitter cannot push more flow downward, hence it is

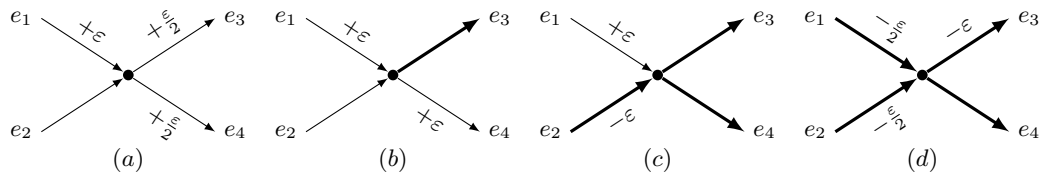


Figure 3 Four examples of throughput changes at a single splitter, depending on which arcs are fluid.

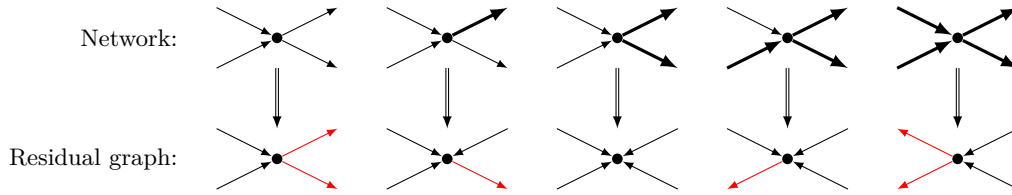


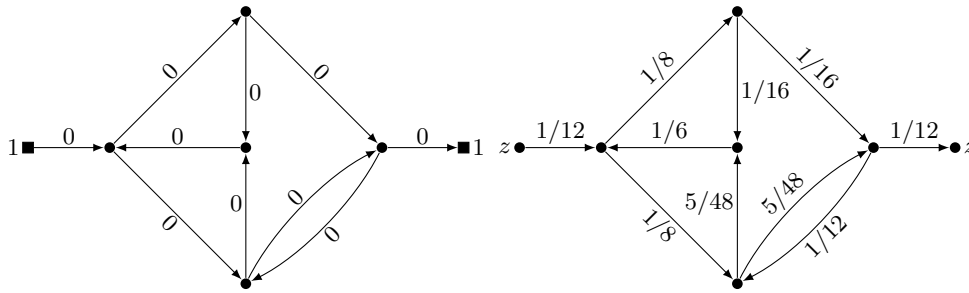
Figure 4 Configurations of splitters and the corresponding vertex in the residual graph. The outgoing arcs from a vertex of the residual graphs are highlighted in red: notice that in a sub-steady-state, the throughputs on these arcs must be equal.

forced to push back flow through its incoming saturated arcs. Thus $t(e_2)$ decreases while $t(e_1)$ increases, by no more than $(t(e_2) - t(e_1))/2$ because of rule 6. Finally in case (d), if we decrease $t(e_3)$, then $t(e_1)$ and $t(e_2)$ must decrease by half as much.

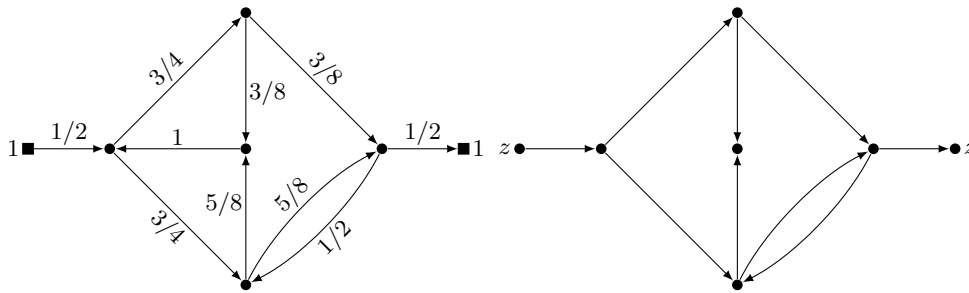
Case (c) presents a challenge due to rule 6, which imposes $t(e_1) \leq t(e_2)$. When $t(e_1) = t(e_2)$, the throughput of e_1 cannot increase, and the throughput of e_2 cannot decrease. We say that e_1 and e_2 are *tight*. In such a case, removing e_1 from F is allowed by rule 6. Fluid arcs e with $t(e) = c(e)$ or saturated arc with $t(e) = 0$ are also *tight*, since we cannot modify their throughput further. Then we define the edge-set of the residual graph to only contain non-tight fluid arcs and reverses of non-tight saturated arcs.

Due to the conservation rule 5, any iterative change to the throughputs of the network must be in accordance with a circulation of the residual graph. Because of rules 6 and 7, some arcs are constrained to have the same throughput. Therefore the chosen circulation itself has similar constraints. This is illustrated in Figure 4, where the arcs that have equal throughput are highlighted in the residual graph. As may be readily checked, those constraints are exactly set on the leaving arcs in the residual graph of each vertex corresponding to a splitter. As for the special vertex z , obtained from the identification of the inputs and the outputs, we may non-deterministically select one of its leaving arc. Then we force all other arcs leaving z to have zero flow, by removing those arcs from the residual graph. From the residual graph, we compute a circulation satisfying each equality constraint. First compute a stationary distribution of a random walk on the residual graph. Then assign to each arc the probability of being the next arc in a random walk from that distribution. This results in a so-called *stationary circulation* (see Figure 5). One must be careful if the residual graph is not strongly connected. Then either we can find a strongly connected subgraph induced by the leaving arcs of some subset of vertices, or the residual graph contains a sink (as in Figure 6). In the former case we can still find a circulation, while in the latter case, we will be able to remove some arc from F .

Once a circulation is found, we increase the throughput as much as possible. This process will result in the creation of at least one sink in the updated residual graph. We show that when the residual graph contains a sink, some arc can be safely removed from F and becomes



■ **Figure 5** Starting from a trivial sub-steady-state, we compute a residual graph and a stationary circulation in this graph (the two vertices marked z should be identified). Then we increase the throughputs accordingly, as much as possible without violating a sub-steady-state rule, by adding $\lambda = 6$ times the circulation at which point some edge reaches its capacity (see Figure 6).



■ **Figure 6** We compute a new residual graph, which does not contain the arc with throughput 1, since this arc cannot increase. Then the existence of a sink prevents us to find a stationary circulation in this residual graph (the two vertices marked z should be identified). We remove from F the incoming arc to the sink with highest throughput, and go to the next iteration.

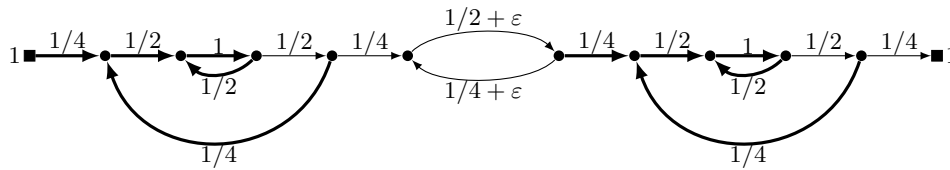
saturated. This bounds the number of steps until the algorithm stops, when z itself becomes a sink. At this point, any arc leaving an input node is either at full capacity or is saturated. Hence rule 3 is satisfied, (t, F) is a steady-state. Summarizing the discussion, we get:

► **Theorem 5.** *There is an algorithm that given a splitter network $G = (I \uplus S \uplus O, E)$ with capacities $c : I \uplus O \rightarrow [0, 1]$, finds a steady-state (t, F) in time $O(|S|^2 + |S| \text{sd}(G_z))$, where G_z is the graph obtained by identifying $I \cup O$ into a single vertex z , and $\text{sd}(G_z)$ denotes the time to compute a stationary distribution on any orientation of a subgraph of G_z .*

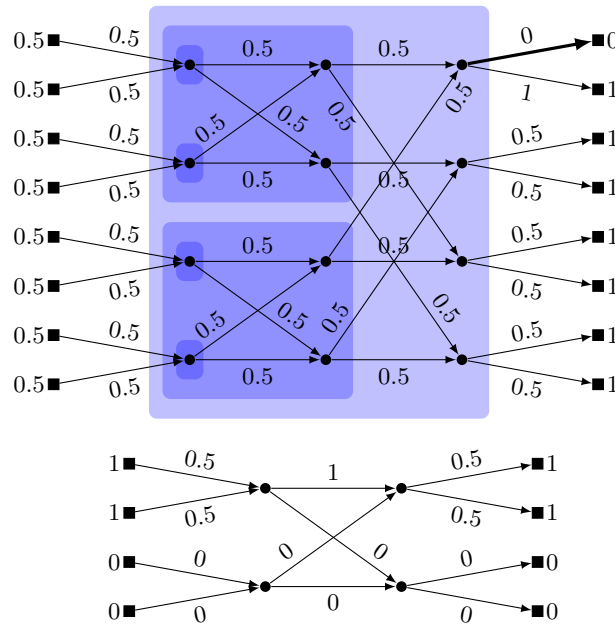
Steady-states are not unique: a directed cycle with no input or output can have any constant throughput on all its arcs. Figure 7 showcases a more interesting network, having one input, one output, and many possible steady-states. However, in this example, all steady-states have the same throughputs on the inputs and outputs. Is there a network with two steady-states having different throughputs on their inputs and outputs? We conjecture that this cannot happen: steady-states are unique up to minor modifications, as in Figure 7. Those modifications would be adding or removing some arcs from F , and adding or subtracting a circulation from the residual graph that leaves the inputs and outputs unchanged.

4 **Balancers**

We now define load-balancing networks and their properties. The goal of a load-balancing network is to divide some input flow evenly between several output belts. In the simplest case, the output belts can receive an arbitrarily large flow (up to the capacity of the belt). In



■ **Figure 7** A network having several steady-states. Any value for ε between 0 and $\frac{1}{2}$ gives a steady-state.

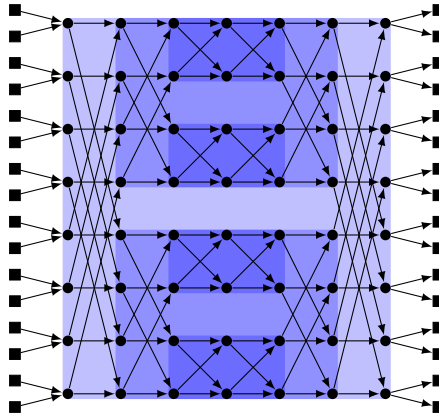


■ **Figure 8** On top, the simple balancer of order 3, with a steady-state that is not balanced when some output capacity is not 1. The capacity of each input (resp. output) is given at their left (resp. right). Below, a simple balancer of order 2, with a steady-state with total throughput less than both the total input capacity and the total output capacity.

more general cases, some outputs may be restricted but we still want the flow to be divided as evenly as possible, without limiting the total throughput available. We distinguish three properties of load-balancing networks. The first of these properties considers networks where the output capacities are not constrained.

► **Definition 6.** A splitter network $G = (I \uplus S \uplus O, E)$ is a balancer if for any $c : I \uplus O \rightarrow [0, 1]$ such that for each output $o \in O$, $c(o) = 1$, there is a steady-state (t, F) for (G, c) with t constant on $\delta^-(O)$. An (n, p) -balancer is defined as a balancer with $|I| = n$ inputs and $|O| = p$ outputs.

When $|I| = |O| = 2^k$, the simple balancer of order k is a balancer network. It can be defined recursively: a simple balancer of order $k + 1$ is made from two simple balancers of order k in parallel. We identify each pair of outputs with equal index from the two balancers, creating a new splitter whose leaving arcs go to new output nodes. The recursive process is highlighted by blue boxes in Figure 8. A drawback of the simple balancer occurs when the output capacities are not uniformly 1. Then the balancing property is lost, as can be seen on the network in the top half of Figure 8.



■ **Figure 9** A Beneš network of order 4 with the recursive structure being made explicit.

Another limitation of simple balancers is that the total throughput at steady-state is not as much as we could expect. A simple upper bound on the total throughput is $\min\{c(I), c(O)\}$. It is reasonable to expect from a load-balancing network to always reach that bound. However, simple balancers do not have this property, as shown by the example on the bottom half of Figure 8. Improving over the definition of simple balancer, the concept of throughput-unlimited balancer imposes a maximized global throughput.

► **Definition 7.** A balancer $G = (I \uplus S \uplus O, E)$ is throughput-unlimited if for any $c : I \uplus O \rightarrow [0, 1]$, there is a steady-state (t, F) for (G, c) such that total throughput $t(I) = t(O)$ is maximized at $\min\{c(I), c(O)\}$.

Notice that it has to be balancing only when the output capacities are uniformly 1. Beneš networks are throughput-unlimited networks with $|O| = |I| = 2^k$. They can be described as gluing two simple balancers, where the second balancer is reversed, see Figure 9. Observe that Beneš networks are their own reverses.

On the negative side, Beneš network are still not balancing when output capacities are not uniformly 1, for instance one could extend the steady-state in the network on the left side of Figure 8 to a steady-state in a Beneš network with the same throughputs. This calls for a stronger property, that a network should be load-balancing and throughput-unlimited for any capacity function. This is the notion of universal balancer.

► **Definition 8.** A splitter network $G = (I \uplus S \uplus O, E)$ is universally balancing if for each capacity $c : I \uplus O \rightarrow [0, 1]$, there is a steady-state (t, F) and $\alpha, \beta \in \mathbb{R}_{\geq 0}$ such that

- (i) for each input i , $t(\delta^+(i)) = \min\{c(i), \alpha\}$,
- (ii) for each output o , $t(\delta^-(o)) = \min\{c(o), \beta\}$.
- (iii) the total throughput $T := t(\delta^+(I))$ equals $\min\{c(I), c(O)\}$.

In the extended version of this work, we will show how to build a universal balancer with $|I| = |O| = 2^k$. From such a universal balancer, by ignoring any set of inputs and outputs (setting their capacities to 0), we can make balancers with arbitrary numbers of inputs and outputs. We will also prove that every balancer presented here contains $\Theta(n \log n)$ splitters where n is the number of inputs and outputs.

► **Proposition 9.** The number of splitters in the simple balancer, Beneš network and universal network of order k are respectively $S(k) = k \cdot 2^{k-1}$, $B(k) = (2k - 1) \cdot 2^{k-1}$, and $U(k) = (k + 1)2^{k+2}$.

5 Lower bounds on the number of splitters

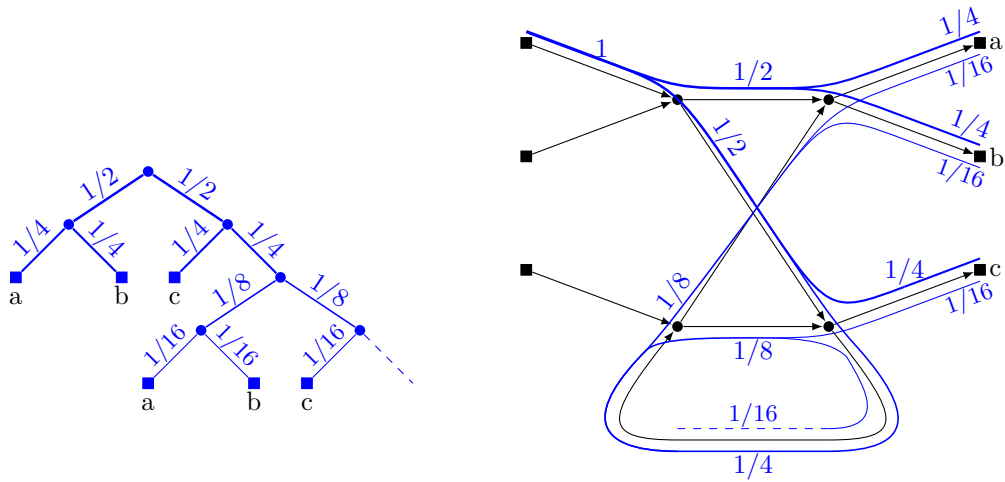
Our next goal is to provide an $\Omega((n+p)\log(n+p))$ lower bound on the number of splitters in a (n, p) -balancer. We begin with what may seem as an unrelated problem: sampling in a discrete probability distribution. Given a fair coin that can be tossed arbitrarily often, how to choose an outcome in $\{1, \dots, d\}$, with probabilities given by a distribution $\pi \in [0, 1]^d$? First, consider the case when $\pi(i)$ is a rational for each $i \in \{1, \dots, d\}$, say $\pi(i) = p_i/q$ where q is a common denominator. Then a sequence of coin tossing can be described as a (possibly infinite) binary decision tree, with each leaf labeled with a sampled value. Here we present a construction of such a tree. Start from a single vertex, which serves as the root. Grow the tree in repeated iterations. At each iteration, add two children to every unlabelled leaf. As soon as the deepest level of the tree contains at least q leaves, label p_i of these leaves with i , for each $i \in \{1, \dots, d\}$. Once labeled, each leaf becomes definitive and will not grow anymore. The process goes on by once again growing the unlabelled leaves, as long (possibly infinitely) as some unlabelled leaf exists. After the tree is completed, the tree can be optimized using a simple trick repeated multiple times. If at any depth d , two leaves share a common label, move them under a common parent, then replace these two leaves with a single leaf at depth $d - 1$ bearing the same label. This process can be generalized to irrational probabilities, and gives a sampling algorithm that minimizes the number of coins tossed:

► **Theorem 10** ([13]). *Let $\pi \in [0, 1]^d$ a discrete probability distribution (so $\mathbb{1}\pi = 1$). Then the minimum expected number of coin tosses necessary to sample an element with probability distribution π is $\sum_{i=1}^d \sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k(\pi_i)$. This minimum is achieved by a binary decision tree where at each depth k and for each $i \in \llbracket 1, d \rrbracket$, the number of leaves with label i is $\text{binary}_k(\pi_i)$ (the value of the bit of weight 2^k in the binary expansion of π_i).*

Consider a splitter network, and think of the flow as discrete, arbitrarily small items. An item enters the network from some input, then meets splitters repeatedly until reaching an output. When an item arrives at a splitter with both outgoing arcs being fluid, it will continue on any of the two outgoing arcs, without preference for one over the other because the splitter is fair. It implies that, from the perspective of this single item, the splitter network behaves like a coin-tossing network, with each splitter corresponding to a coin toss. If the network is a balancer, the sampled distribution is the uniform distribution on O .

Formally, when all the arcs remains fluid, increasing a single input capacity from 0 to 1 results in a non-decreasing throughput on each arc. Because all arcs are still fluid, the sub-steady-state algorithm performs a single iteration. Therefore the increase in throughputs follows a single stationary circulation. As illustrated on Figure 10, it is obtained from the embedding of a binary decision tree T onto the splitter network. The increase in throughput on an arc e is the sum of probabilities of the edges mapped to e . Furthermore, in a balancer network, the increase of throughput is the same on every output. This implies that, as we progressively increase each input capacity from 0 to 1, each binary decision tree must uniformly sample from O .

In each binary decision tree, label each edge e with the probability of its usage during sampling. The sum of these labels represents the expected number of tosses, and can be bounded as shown in Theorem 10. When mapped into the splitter network, for an arc e , the sum of these labels on each edge of the tree mapped to e is the additional throughput on e . By summing over all the binary decision tree, we get that the sum of all labels is at most the number of outgoing arcs of all splitters, that is $2|S|$. Applied on balancers, it yields:



■ **Figure 10** The infinite decision tree (in blue) used to sample uniformly over a three-element set $\{a, b, c\}$ can be embedded from any input into a $(3, 3)$ -balancer. Moreover, the sum of the probabilities of the 3 trees, one from each input, will be at most one on any arc, which shows that this network is indeed a simple balancer.

► **Theorem 11.** *Let $G = (I \uplus S \uplus O, E)$ be an (n, p) -balancer, such that when all input capacities are 1, the steady-state has no saturated arc. Then*

$$|S| \geq \frac{1}{2} |I| |O| \sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k \left(\frac{1}{|O|} \right)$$

For a balancer with $|I| = |O| = 2^k$, since $\sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k \left(\frac{1}{|O|} \right) = \frac{k}{2^k}$, we get a lower bound of $k2^{k-1}$ splitters, matching the value of $S(k)$. Therefore the simple balancer of order k is optimal among all balancer networks without any saturated arcs in their steady-states. By extending this argument to steady-states with saturated arcs, we can remove that restriction, albeit at the cost of halving the lower bound.

Consider the various configurations of fluid and saturated arcs incident to a splitter, illustrated in Figure 3. If a splitter has two fluid outgoing arcs, any additional flow is evenly distributed between the two outputs, akin to the probabilities of a coin toss. If a splitter has two incoming saturated arcs, by rule 8, its outgoing arcs are saturated or at full capacity. In an augmenting circulation, the throughput on those arcs may only decrease by the same quantity by rule 6: the splitter still acts as a coin toss, but on the flow that is pushed back. Otherwise, a positive change in throughput on an incoming arc will be followed by an increase on a single outgoing fluid arc or a decrease on a single incoming saturated arc. Similarly a negative change of throughput on an outgoing saturated arc will impact only one other arc. Any additional unit of flow entering the splitter would be routed deterministically. Therefore, in the embedding of a binary decision tree into the splitter network, a node cannot be mapped to such a vertex, and no coin toss occurs here. Thus any splitter, depending on which of its incident arcs are fluid, acts as either a coin toss or a deterministic router. Thus, even in the presence of saturated arcs, we can embed a binary decision tree, by mapping each edge to a directed path in the residual graph. The inner nodes of any such path are *deterministic* splitters, while its extremities are *tossing* splitters. As a consequence of the sub-steady-stat algorithm, the throughput on each arc increases until it becomes saturated,

then decreases. Therefore its throughput varies by at most 2 during the whole algorithm. This limits the extent to which an arc can be utilized by the embeddings of binary decision trees, leading us to the following conclusion:

► **Theorem 12.** *Let $G = (I \uplus S \uplus O, E)$ be an (n, p) -balancer. Then*

$$|S| \geq \frac{1}{4} |I| |O| \sum_{k \in \mathbb{N}} \frac{k}{2^k} \text{binary}_k \left(\frac{1}{|O|} \right)$$

6 Perspectives

We formalized splitter networks and their steady-states, and presented various load-balancing designs. The ability to design universal balancers enables the simulation of networks with integral capacities: each arc is replicated according to its capacity, and each splitter is replaced by a universal balancer. A universal balancer is fair by the balancing property, and maximizing by the unlimited-throughput property, effectively generalizing splitters. Our definition of splitter network can also be extended to support arc capacities natively, with most of the proofs requiring only minor modifications. In an extended version of this paper [5], we will demonstrate how to simulate any rational capacity. Given an arbitrary rational value between 0 and 1, we will design a splitter network, with a single input and a single output, and achieving this value as maximal throughput. However, simulating irrational capacities is not feasible, as the steady-state throughput is a solution to a linear system of inequations.

Although our continuous model is convenient for modeling the expected throughput of splitter networks, Factorio's belt systems operates discretely. Therefore, the observed throughputs in Factorio's splitter networks are only approximations of those theorized by our model. Further investigation into the disparities between the discrete and continuous splitter networks is necessary to accurately apply our findings to Factorio.

We have left several questions unanswered. The most fundamental remains regarding the uniqueness of the steady-state throughput. While it is possible for a single splitter network to admit multiple steady-states, we have yet to encounter a network with two steady-states that yield different throughputs on their outputs.

Our lower bounds for the number of splitters in balancers have a constant multiplicative gap across all designs, indicating they are not tight. For simple balancers of order k , this gap is closed when we forbid saturated arcs in the steady-state of the balancer. Consequently, leveraging saturation is necessary to further reduce the number of splitters in load-balancing networks. Furthermore, it is worth investigating stronger lower bounds in the context of universal balancers.

Factorio allows splitters to be configured to prioritize either an outgoing arc, or an incoming arc. Utilizing this feature, the universal network described in [20] achieves a significantly smaller size compared to our design. Our technique still establishes a lower bound on the number of fair splitters. In general, what is the minimum size achievable for networks utilizing these more general splitters? It is straightforward to extend the definition of steady-state to accommodate unfair splitters. Additionally, in the extended version of this paper, we will provide complexity results for the problem of global throughput maximization, when we can choose which arcs to prioritize in each splitter or a subset of those splitters.

As a last series of questions, consider a network whose steady-state, when all inputs and outputs have capacity 1, has no saturated arcs. If the augmenting flow from any single input is uniformly distributed across the outputs, then the network is a balancer. This provides a polynomial-time procedure for determining whether a network is a balancer, subject to the absence of saturation. Is it feasible to devise a general procedure to decide whether a splitter network is balancing, throughput unlimited or universal?

References

- 1 Václav E Beneš. On rearrangeable three-stage connecting networks. *The Bell System Technical Journal*, 41(5):1481–1492, 1962.
- 2 Václav E Beneš. Permutation groups, complexes, and rearrangeable connecting networks. *Bell System Technical Journal*, 43(4):1619–1640, 1964.
- 3 BoardGameGeek. Boardgame category: transportation. <https://boardgamegeek.com/boardgamecategory/1011/transportation>.
- 4 Bonnie S. Boardman and Caroline C. Krejci. Simulation of production and inventory control using the computer game factorio. In *ASEE 2021 Gulf-Southwest Annual Conference*, 2021.
- 5 Basile Couëtoux, Bastien Gastaldi, and Guyslain Naves. The steady-states of splitter networks, 2024. URL: <https://arxiv.org/abs/2404.05472>.
- 6 Chase Covello, Hyunjang Jung, and Bryan C. Watson. Using graph theory to investigate the role of expertise on infrastructure evolution: A case study examining the game factorio. In *Conference on Systems Engineering Research*, pages 297–311. Springer, 2023.
- 7 Miguel Coviello Gonzalez and Marek Chrobak. Towards a theory of mixing graphs: A characterization of perfect mixability. *Theoretical Computer Science*, 845:98–121, 2020.
- 8 Yefim A. Dinits. The method of scaling and transportation problems. *Studies in Discrete Mathematics, Moscow*, pages 46–57, 1973.
- 9 Shivam Duhan, Chengming Zhang, Wenyu Jing, and Mingqi Li. Factory optimization using deep reinforcement learning ai. *Purdue Undergraduate Research Conference*, 57, 2019.
- 10 Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM (JACM)*, 35(4):921–940, 1988.
- 11 Ketcheson, David. Mathematics Stackexchange: Belt Balancer problem (Factorio). <https://math.stackexchange.com/questions/1775378/belt-balancer-problem-factorio>.
- 12 Donald E Knuth. *The Art of Computer Programming: Sorting and Searching, Volume 3*. Pearson Education, 1998.
- 13 Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In JF Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428. Addison-Wesley, 1976.
- 14 Legnagi, Alessandro and Montini, Axel. VeriFactory. <https://github.com/alegnagi/verifactory/>.
- 15 Andre Leue. *Verification of Factorio Belt Balancers using Petri Nets*. PhD thesis, Bachelorarbeit, Darmstadt, Technische Universität Darmstadt, 2021.
- 16 MatthaeusHarris. Factorio belts are Turing-complete. https://www.reddit.com/r/factorio/comments/lc25cx/factorio_belts_are_turing_complete/.
- 17 Carol A. Meyers and Andreas S. Schulz. Integer equal flows. *Operations Research Letters*, 37(4):245–249, 2009.
- 18 Amandeep Parmar. *Integer programming approaches for equal-split network flow problems*. PhD thesis, Georgia Institute of Technology, 2007.
- 19 Sean Patterson, Joan Espasa, Mun See Chang, and Ruth Hoffmann. Towards automatic design of factorio blueprints. *arXiv preprint arXiv:2310.01505*, 2023.
- 20 pocarski. Universal 8-8: Perfectly Balanced, as All Things Should Be. <https://web.archive.org/web/20230922022806/https://alt-f4.blog/ALTF4-27/>.
- 21 R-O-C-K-E-T. factorio-SAT: Enhancing the Factorio experience with SAT solvers. <https://github.com/R-O-C-K-E-T/Factorio-SAT>.
- 22 Kenneth N. Reid, Iliya Miralavy, Stephen Kelly, Wolfgang Banzhaf, and Cedric Gondro. The factory must grow: automation in factorio. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 243–244, 2021.
- 23 K. Srinathan, Pranava R. Goundan, MVN Ashwin Kumar, R. Nandakumar, and C. Pandu Rangan. Theory of equal-flows in networks. In *Computing and Combinatorics: 8th Annual International Conference, COCOON 2002 Singapore, August 15–17, 2002 Proceedings 8*, pages 514–524. Springer, 2002.
- 24 Wube Software. Factorio. <https://www.factorio.com/>.

How Did They Design This Game? Swish: Complexity and Unplayable Positions

Antoine Dailly ✉

Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, Clermont-Auvergne-INP, LIMOS, 63000 Clermont-Ferrand, France

Pascal Lafourcade ✉ 

Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, Clermont-Auvergne-INP, LIMOS, 63000 Clermont-Ferrand, France

Gaël Marcadet ✉ 

Université Clermont-Auvergne, CNRS, Mines de Saint-Étienne, Clermont-Auvergne-INP, LIMOS, 63000 Clermont-Ferrand, France

Abstract

SWISH is a competitive pattern recognition card-based game, in which players are trying to find a valid cards superposition from a set of cards, called a “swish”. By the nature of the game, one may expect to easily recover the logic of the SWISH’s designers. However, no justification appears to explain the number of cards, of duplicates, but also under which circumstances no player can find a swish. In this work, we formally investigate SWISH. In the commercial version of the game, we observe that there exist large sets of cards with no swish, and find a construction to generate large sets of cards without swish. More importantly, in the general case with larger cards, we prove that SWISH is NP-complete.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness; Theory of computation → Backtracking; Mathematics of computing → Combinatorial algorithms

Keywords and phrases Game, Complexity, Algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.10

Funding *Antoine Dailly*: This author was supported by the International Research Center “Innovation Transportation and Production Systems” of the I-SITE CAP 20-25 and by the ANR project GRALMECO (ANR-21-CE48-0004).

Gaël Marcadet: This author was supported by the DataLake-For-Nuclear (D4N) project funded by the BPI institute.

Acknowledgements We thank the anonymous referees for their useful suggestions and remarks.

1 Introduction

SWISH is a pattern recognition card game designed in 2011 by Zvi Shalem and Gali Shimoni and published by the company ThinkFun [20]. It works as the famous game SET [4, 6, 13], each player having to find a *swish* among the 16 cards present on the table before their opponents do. SWISH includes 60 transparent cards where each card contains one points and one circle, coming in four colors. Players simultaneously try to create a swish by spotting two or more cards that can be laid on top of one another in some manner so that every point fits in a circle of the same color as we can see in Figure 1 (no two points or circles can meet). Create a swish, and you claim the cards used, with new cards then being laid out. Whoever claims the most cards wins the game.

To play this game, it is important to note that the cards are transparent and can be rotated or flipped through *vertical axial symmetry*, *horizontal axial symmetry* or *central symmetry*, as described in Figure 2 where one card can be rotated or flipped in three other positions.



© Antoine Dailly, Pascal Lafourcade, and Gaël Marcadet; licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

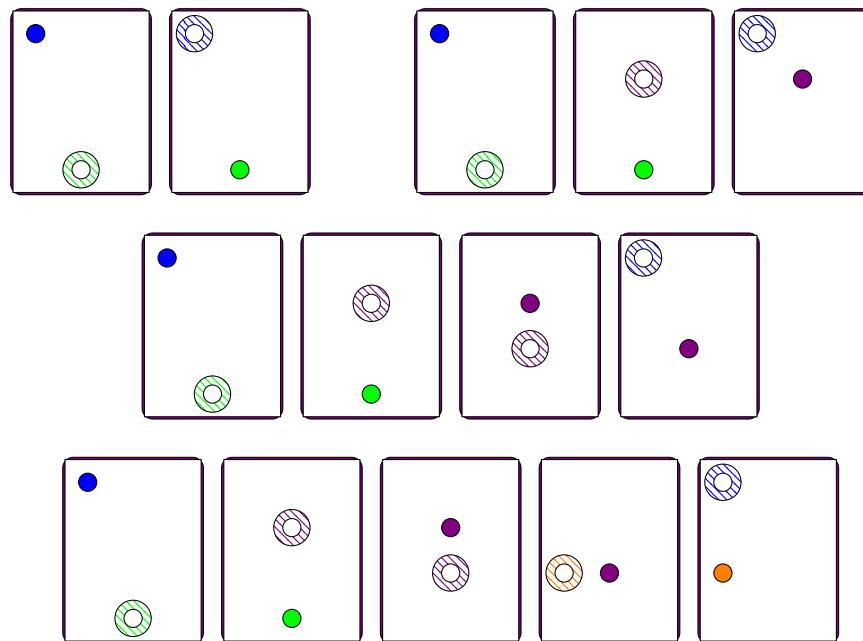
Editors: Andrei Z. Broder and Tami Tamir; Article No. 10; pp. 10:1–10:19

Leibniz International Proceedings in Informatics

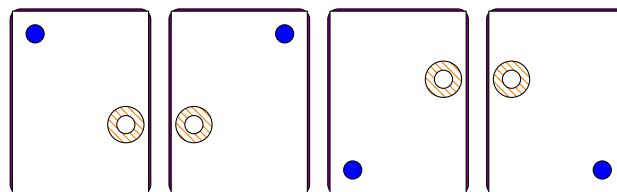


LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 How Did They Design This Game? Swish: Complexity and Unplayable Positions



■ **Figure 1** SWISH examples: on the first line a SWISH with 2 cards on the left, and a 3 cards SWISH. On the second line a 4 cards SWISH and on the last line a 5 cards SWISH.

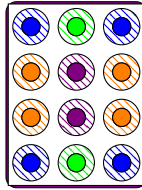


■ **Figure 2** Example of flipping and rotating a card.

1.1 Swish cards

There are 60 transparent cards in the commercial version of SWISH, following a grid structure of height 4 and width 3. The cards are obtained by placing a point in each of the four possible positions (accounting for symmetries), and then a circle in each of the other possible positions. For the points in the left column, the circle can be in 11 positions. For the points in the middle column, due to axial symmetry, the circle can be in 7 positions. Note that this only generates 36 cards, but there are 24 cards which are duplicated, reaching a total of 60 cards.

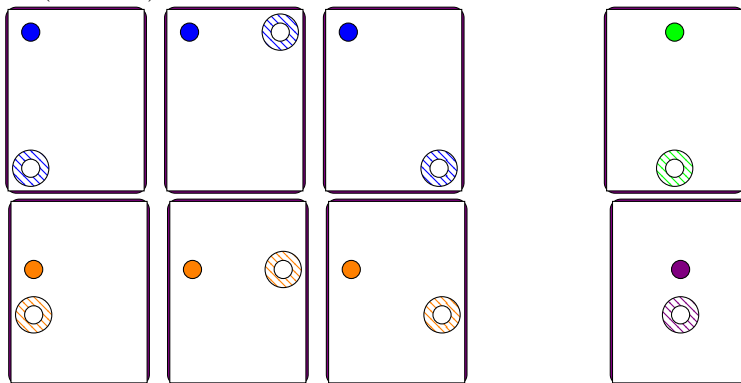
Each position of the grid is associated with a particular color. The colors represent the position of a point or a circle (blue is for a corner, green for the middle column and the top and bottom rows, purple for the middle column and the middle rows, orange for the middle rows and the left and right columns), so they are here to help the player. The game can be played with single-colored cards.



We present all $60=16+12+12+10+10$ transparent cards of the board game SWISH.

Same color cards

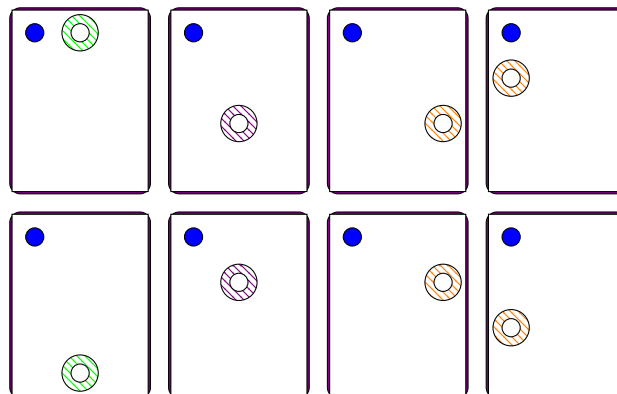
First, we have all cards where the circle and the point are of the same color, which appear twice in the deck (16 cards). These cards exist in double in order to form swish of size 2.



We then have cards where the point and the circle are of different colors.

12 blue point cards

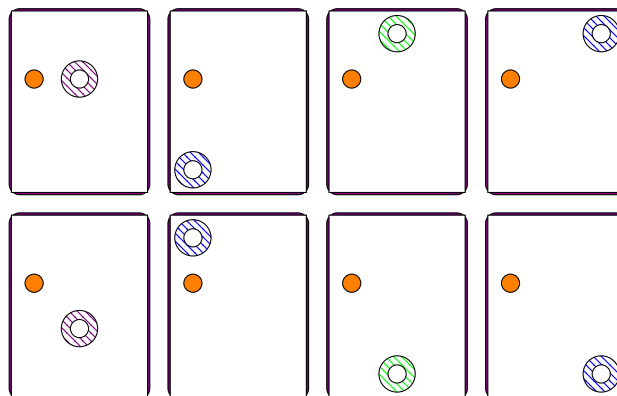
We give all bicolored with a single blue point. The cards on the first row appear twice in the deck (8 cards) and the ones on the second row appear once (4 cards) for a total of 12 cards.



12 orange point cards

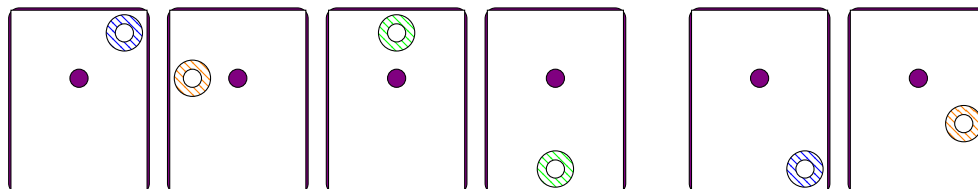
We give all bicolored with a single orange point. The cards on the first row appear twice in the deck (8 cards) and the ones in the second row appear once (4 cards) for a total of 12 cards.

10:4 How Did They Design This Game? Swish: Complexity and Unplayable Positions



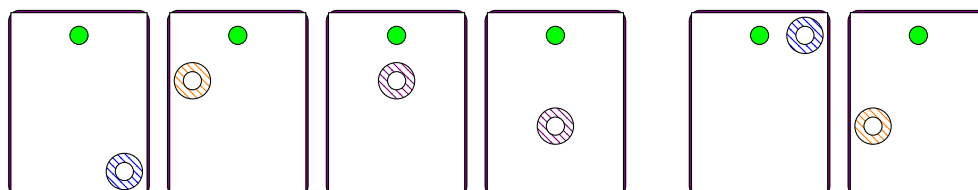
10 purple point cards

We give all bicolored with a single purple point. The first four cards on the left appear twice in the deck (8 cards) and the two last cards on the right appear once (2 cards) for a total of 10 cards.



10 green point cards

We give all bicolored with a single green point. The first four cards on the left appear twice in the deck (8 cards) and the two last ones on the right appear once (2 cards) for a total of 10 cards.



1.2 Generalizing Swish

Since the board game SWISH is played on cards of height 4 and width 3, it is trivial to find a large swish among a given set of cards with a brute-force algorithm (even though it can be difficult for human players). Hence, we propose a generalization of SWISH in order to explore the computational complexity of the game. Creating general version of games is a standard way of studying their complexity outside of the often small and thus solvable standard positions, as this was done for SET itself [6, 13], and other commercial games such as Othello [12], Scrabble [14], Hanabi [2], Kingdomino [16], Backgammon [21], The Crew [18]; but also for already complex games such as Hex [9], Chess [10], Go [15, 19, 22] or Shogi [1]. For more results on the complexity of games, either combinatorial or commercial, and either standard or generalized, we refer the reader to [3, 5, 7, 11].

The generalized version of SWISH is played on cards of height h and width w . Cards can have one or several symbols, which can be points or circles. For a given card C and two integers a and b (with $1 \leq a \leq h$ and $1 \leq b \leq w$), we denote by $C[a][b]$ the spot in row a and column b . Other than that, the generalized version is played the exact same way as the board game version: from a set \mathcal{C} of cards, the players try to create a *swish*, that is, a subset $\mathcal{S} \subseteq \mathcal{C}$ such that every point meets a circle, every circle meets a point, and no two points or two circles meet. The cards can still be flipped or rotated, which can also be seen as applying axial (vertical or horizontal) or central symmetry.

Since the cards are drawn from the deck at random, the players cannot anticipate what is going to come next. Hence, we will assume that they will try to maximize their given score at each round of the game. Thus, the question that we ask is the following: given a set of cards, can we find a swish that is as large as possible? This optimization question leads to the following decision problem:

SWISH

Instance: A set \mathcal{C} of cards, an integer k .

Question: Is there a swish $\mathcal{S} \subseteq \mathcal{C}$ such that $|\mathcal{S}| \geq k$?

1.3 Contributions and outline

Our results are twofold. First, in Section 2, we study the computational complexity of SWISH. We begin with the most basic case of SWISH, that is, if there is only one symbol per card:

► **Theorem 1.** *SWISH can be solved in polynomial time if there is one symbol per card.*

We then prove that SWISH is NP-complete in the general case, even with as few as three symbols per card. The proof uses an intermediary step through a more constrained variant of SWISH.

► **Theorem 2.** *SWISH is NP-complete, even if there are at most three symbols per card.*

This leaves only the case of two symbols per card open. Then, in the same line as [4], we study in Section 3 how many cards there can be in a *no-swish position*, that is, a set that does not contain any swish. Note that, for the base game, the rules are to play with a set of 16 cards at a time, implying that this is enough to guarantee finding a swish, but we found a no-swish position of 28 cards. Furthermore, we construct no-swish positions for the generalized version of SWISH with no duplicate cards, that contain a very high fraction (depending on the parity of the width and length, roughly half in the worst case) of the total possible cards.

2 The computational complexity of Swish

We first prove the following result, which covers the most basic case for SWISH:

► **Theorem 1.** *SWISH can be solved in polynomial time if there is one symbol per card.*

Proof of Theorem 1. The algorithm is as follows. First, associate the cards by duplicates. Two cards are duplicates if, after applying an axial or a central symmetry to one of them, they are identical. For any set of duplicates of size more than 4, remove duplicate cards until there are exactly 4 of them (this is because no more than 4 duplicates can be used in the same swish). Then, construct the *compatibility graph* G : each card C is a vertex, and there is an edge $C_i C_j$ if (wlog) there is a point in $C_i[a][b]$ and a circle in $C_j[a][b]$. Now, we just have

10:6 How Did They Design This Game? Swish: Complexity and Unplayable Positions

to find a maximum-size matching M of G ; if $|M| \geq k$, then we answer YES, otherwise, we answer NO. Note that this only works since each card has exactly one symbol: once a card has been paired with another card, it cannot be paired with another card, except through flipping or rotating it if it has a duplicate.

The algorithm clearly is polynomial-time, since trimming the duplicates can be done in linear time through a hash table, constructing the compatibility graph takes polynomial time, and the maximum matching is polynomial-time solvable [8]. ◀

We now focus on the NP-hardness of the generalized version of SWISH. We are interested in minimizing the number of symbols per card, to get closer to the commercial version of SWISH. In order to prove Theorem 2, we are going to go through three intermediary lemmas. First, we are going to prove that a more constrained variant, SIMPLE-SWISH, is NP-complete, even with at most four symbols per card. Then, we are going to show how to adapt the reduction in order to have the cards have at most three symbols. Finally, we are going to reduce SIMPLE-SWISH to SWISH.

The game SIMPLE-SWISH is a restricted variant of SWISH. The rules are exactly the same, except that we fix a top and a left side for the cards, and that we can neither flip nor rotate them (hence, it is forbidden to apply symmetry to cards). This gives us the following decision problem:

SIMPLE-SWISH

Instance: A set \mathcal{C} of cards, an integer k .

Question: Is there a simple-swish $\mathcal{S} \subseteq \mathcal{C}$ such that $|\mathcal{S}| \geq k$?

► **Lemma 3.** *SIMPLE-SWISH is NP-complete, even if there are at most four symbols per card.*

Proof. We will reduce from MAX-(2,3)-SAT, a restriction of the classical MAX-SAT problem, which was proved NP-complete in [17].

MAX-(2,3)-SAT

Instance: A formula ϕ in CNF such that every clause is of size 2 and every variable appears in at most 3 clauses, an integer k .

Question: Is there an assignment of the variables such that at least k clauses are verified?

Let ϕ be a MAX-(2,3)-SAT formula with n variables x_1, \dots, x_n and m clauses c_1, \dots, c_m , and assume that the variables are ordered within a clause (so each clause has a first variable and a second variable). We will create a set \mathcal{C} of cards the following way. Each card has height $h = \max(m, n)$ and width $w = 6$ (note that we can assume $h \gg 6$).

- For every variable x_i , create the following cards:
 - A card X_i with a point in $X_i[i][1]$ and a circle in $X_i[i][3]$;
 - A card \bar{X}_i with a point in $\bar{X}_i[i][2]$ and a circle in $\bar{X}_i[i][3]$.
 Those two cards are called the *variable cards*, which represent the assignment of the variable x_i .
- For each variable x_i that appears in clauses c_{j_1}, c_{j_2} and c_{j_3} , for each subset $J \subseteq \{j_1, j_2, j_3\}$ (including the empty set), create a card $X_{i,J}$ with a point in $X_{i,J}[i][3]$ and circles in $X_{i,J}[j][3]$ for each $j \in J$.
 Those eight cards are called the *linkage cards*, which represent which clause(s) the variable x_i satisfies.

- For each variable x_i that appears positively in clauses c_j for $j \in J$ (we may have $J = \emptyset$), create a card $X_{i,c}$ with a circle in $X_{i,c}[i][1]$ and points in $X_{i,c}[j][4]$ for each $j \in J$ such that x_i is the first variable of c_j and in $X_{i,c}[i][5]$ for each $j \in J$ such that x_i is the second variable of c_j .

For each variable x_i that appears negatively in clauses c_j for $j \in J$ (we may have $J = \emptyset$), create a card $\overline{X}_{i,c}$ with a circle in $\overline{X}_{i,c}[i][1]$ and points in $\overline{X}_{i,c}[j][4]$ for each $j \in J$ such that \overline{x}_i is the first variable of c_j and in $\overline{X}_{i,c}[i][5]$ for each $j \in J$ such that \overline{x}_i is the second variable of c_j .

Those two cards are called the *satisfying cards*, which represent the fact that the assignment of the variable satisfies some clauses it is in.

- For each clause c_j , create three cards C_j^1 , C_j^2 and $C_j^{1,2}$ with a point in $C_j^1[j][6]$, $C_j^2[j][6]$ and $C_j^{1,2}[j][6]$, and circles in $C_j^1[j][4]$, $C_j^2[j][5]$, $C_j^{1,2}[j][4]$ and $C_j^{1,2}[j][5]$. Those three cards are called the *clause cards*, which represent the fact that the clause c_j is satisfied by its first, second or both variables.

The set C contains every variable, clause, linkage and satisfying card as described above, so $12n + 3m$ cards in total. All those cards have at most four symbols. This reduction is depicted on Figure 3. Let $\ell = 3n + k$. We claim that there is an assignment of the variables satisfying at least k clauses of ϕ if and only if there is a simple-swish on C of size at least ℓ . Note that the reduction is clearly polynomial.

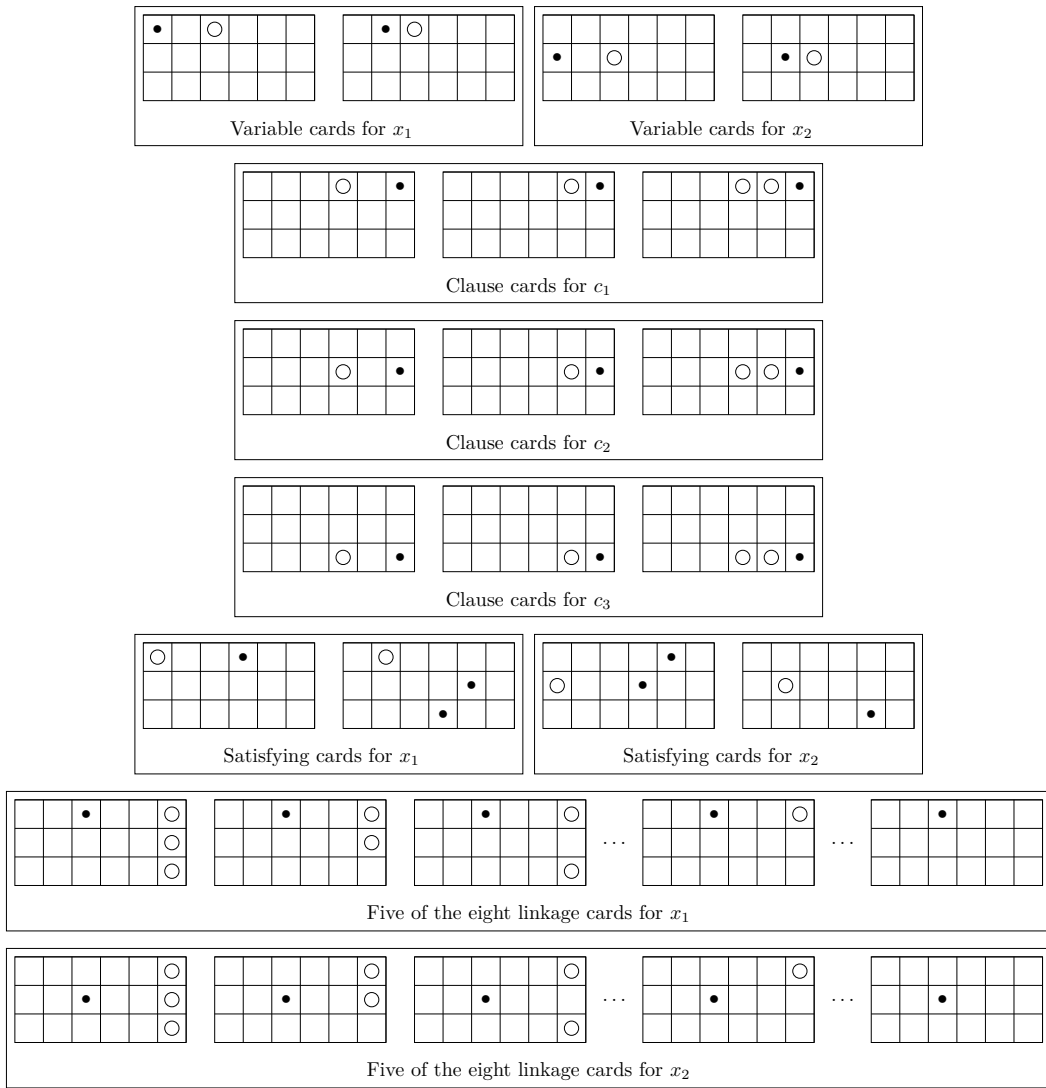
(\Rightarrow) Assume that there is an assignment of the variables satisfying at least k clauses of ϕ .

We construct the following simple-swish S :

- For every variable x_i which is assigned as **True**, add the variable card X_i and the satisfying card $X_{i,c}$ to S ;
- For every variable x_i which is assigned as **False**, add the variable card \overline{X}_i and the satisfying card $\overline{X}_{i,c}$ to S ;
- For every variable x_i , denote by J the set of indices of clauses that are satisfied by the assignment of x_i (we may have $J = \emptyset$), and add the linkage card $X_{i,J}$ to S ;
- For every clause c_j satisfied by the assignment, add the clause card C_j^1 (resp. C_j^2 , $C_j^{1,2}$) to S if c_j is satisfied by its first (resp. second, both) variable.

It is clear that S is a simple-swish. First, two points and circles cannot meet. Then, every point meets a circle and every circle meets a point: the point of each variable card meets the circle of the associated satisfying card, the circle of each variable card meets the point of the associated linkage card, the point of each satisfying card meets the circles of each clause card that are satisfied by the given variable, and the point of each satisfied clause card meets the circle of one of the linkage cards of one of the variables satisfying it. Furthermore, S contains exactly one variable, one linkage and one satisfying card for each variable, as well as one clause card for each satisfied clause, and hence $|S| \geq 3n + k = \ell$.

(\Leftarrow) Assume that there is a simple-swish S of size at least ℓ . Due to the construction of the cards, S can contain at most one variable card, one linkage card and one satisfying card for each variable, as well as at most one clause card for each clause. Hence, there are at least k clause cards in S . For each variable x_i , if $X_i \in S$ assign x_i as **True** and if $\overline{X}_i \in S$ assign x_i as **False** (if none of X_i, \overline{X}_i is in S , then assign x_i as **True** by default). Now, every clause card $C_j \in S$ can only be there if some variable card X_i (resp. \overline{X}_i) such that $x_i \in c_j$ (resp. $\overline{x}_i \in c_j$). This implies that, for every clause card $C_j \in S$, at least one of the two variables in c_j will be assigned in such a way that c_j will be satisfied. Hence, at least k clauses of ϕ will be satisfied. \blacktriangleleft



■ **Figure 3** An example of the reduction of Lemma 3, with $c_1 = (x_1 \vee x_2)$, $c_2 = (x_2 \vee \bar{x}_1)$, $c_3 = (\bar{x}_1 \vee \bar{x}_2)$.

► **Lemma 4.** *SIMPLE-SWISH is NP-complete, even if there are at most three symbols per card.*

Proof of Lemma 4. Assume that there are n SIMPLE-SWISH cards of height h and width w with at most four symbols per card. We will create $4n$ cards with at most three symbols per card. Those cards will be of height $h + n$ and width w (note that we can assume $h \neq w$ and $h + n \neq w$).

For each card C_i with symbols on $C_i[j_1][k_1]$, $C_i[j_2][k_2]$, $C_i[j_3][k_3]$ and $C_i[j_4][k_4]$ (including no symbol), create the four following cards:

- C_i^1 with a point in $C_i^1[h + i][1]$, and $C_i^1[j_1][k_1] = C_i[j_1][k_1]$;
- C_i^2 with a circle in $C_i^2[h + i][1]$, a point in $C_i^2[h + i][2]$, and $C_i^2[j_2][k_2] = C_i[j_2][k_2]$;
- C_i^3 with a circle in $C_i^3[h + i][2]$, a point in $C_i^3[h + i][3]$, and $C_i^3[j_3][k_3] = C_i[j_3][k_3]$;
- C_i^4 with a circle in $C_i^4[h + i][3]$, and $C_i^4[j_4][k_4] = C_i[j_4][k_4]$.

For (C, k) an instance of SIMPLE-SWISH, create a set C' of cards as described above, and let

$(C', 4k)$ be a new instance of SIMPLE-SWISH. Clearly, there is a simple-swish of size at least k in C if and only if there is a simple-swish of size at least $4k$ in C' , and each card in C' has at most three symbols. ◀

► **Observation 5.** *The reduction of Lemma 4 can start from cards with at most n symbols, where n is a constant integer.*

We are now ready to prove our main result:

► **Theorem 2.** *SWISH is NP-complete, even if there are at most three symbols per card.*

Proof. We will reduce from SIMPLE-SWISH. Let (C, k) be a SIMPLE-SWISH position, with C containing cards of height h and width w with at most three symbols per card. We create the set C' as follows. For every card $C_i \in C$, add to C' four cards C_i^1, C_i^2, C_i^3 and C_i^4 of height $2h$ and width $2w$ (the construction assumes that $h \neq w$; if $h = w$, we can adapt it by adding an empty buffer column in the middle of C_i^1, C_i^2, C_i^3 and C_i^4). Set $C_i^1[a][b] = C_i[a][b]$ for $a \leq h$ and $b \leq w$, and no other symbol on C_i^1 . Set $C_i^2[a][w+1-b] = C_i[a][b]$ for $a \leq h$ and $b \leq w$, and no other symbol on C_i^2 . Set $C_i^3[h+1-a][b] = C_i[a][b]$ for $a \leq h$ and $b \leq w$, and no other symbol on C_i^3 . Set $C_i^4[h+1-a][w+1-b] = C_i[a][b]$ for $a \leq h$ and $b \leq w$, and no other symbol on C_i^4 . In other words, each of the four cards is divided in four parts, C_i^1 contains C_i in the top left, C_i^2 contains the vertical axial symmetry of C_i in the top right, C_i^3 contains the horizontal axial symmetry of C_i in the bottom left, and C_i^4 contains the central symmetry of C_i in the bottom right. We now prove that there is a simple-swish of size at least k in C if and only if there is a swish of size at least $4k$ in C' .

(\Rightarrow) Let S be a simple-swish of size at least k in C . We construct S' by taking, for every card $C_i \in S$, the four cards C_i^1, C_i^2, C_i^3 and C_i^4 . By leaving them in their original position, we obtain a swish of size at least $4k$ in C' .

(\Leftarrow) Let S' be a swish of size at least $4k$ in C' . First, we can assume that every card in S' is in its original position. Indeed, using symmetry or a rotation on a card $C_i^j \in C'$ changes it to another card of C' (for instance, using vertical axial symmetry on C_i^2 gives C_i^4). However, when there are two identical cards in a set, only one of them can be used in a swish without using symmetries or rotation. Hence, if a card in S' was used after a symmetry or a rotation, then, we can replace it in S' by the equivalent card with no symmetry or rotation.

Now, there are $4k$ cards in S' , all in their original positions (*i.e.*, no symmetry or rotation was applied to any card). Hence, S' can be subdivided in four subsets S'_1, S'_2, S'_3 and S'_4 , such that $S'_j = \{S_i^j \mid S_i^j \in S'\}$. Each of the S'_j 's is a swish, since the cards in each subset do not interact with each other by construction. By the pigeonhole principle, at least one of the sets S'_j is of size at least k . Let $S = \{S_i \mid S_i^j \in S'_j\}$, S is a swish of size at least k in C . ◀

3 Swish has large unplayable positions

In SWISH, an unplayable position, or *no-swish* position, is a set of cards where no swish exists. Large no-swish positions are particularly interesting for SWISH, since other games tend to not have them (in particular, it is well-known that the commercial version of SET has no unplayable position). In this section, we will be studying no-swish positions for both the commercial and the generalized version of SWISH. We thus focus on cards with exactly two symbols (one circle and one point). Furthermore, for simplification, we assume that no card appears twice (accounting for its possible configurations) in the generalized version.

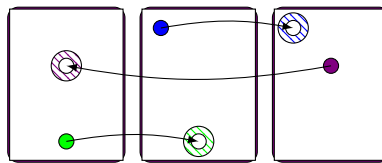
Finding the largest unplayable position for the generalized version is hard, since there are many possible combinations. However, finding the largest no-swish position for the commercial version of the game, containing 60 cards (described in the introduction) is a more

achievable challenge. We will present the largest no-swish position of the commercial version of SWISH, before presenting a construction of a large no-swish position for the generalized version, of which a commercial no-swish position that we found (removing duplicates) is a direct application.

Note that our analysis holds for *rectangular* cards, that is, cards where the height and width differ. Indeed, if the height and width are the same, then there are four more operations that can be applied to change the configuration of the card, which changes the game.

3.1 Commercial no-swish

First of all, we need to give an algorithmic-friendly representation of SWISH, including cards, rotations but also handling the definition of *compatibility* between two cards, at the heart of a swish. By the nature of the game, two cards are said to be *compatible* if the point of the first card meets the circle of the second card. Observe that the compatibility between two cards, generalized to all the cards in SWISH, is very close to a *directed graph* structure, the nodes of the graph being the cards and the arcs being the compatibility between the cards. Following this idea of graph structure to represent compatibility between cards, a swish essentially corresponds to a cycle in the graph, as depicted in Figure 4.

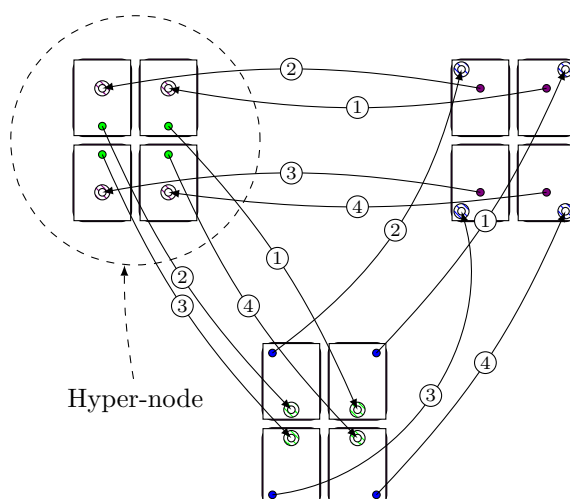


■ **Figure 4** Example of a swish of 3 cards, with explicit compatibility using ordinary directed graph.

At this point, the definition of a swish becomes clearer: A swish is a cycle of length 2 or higher in a graph (that will be constructed from the compatibility relation), each node of the cycle representing the card involved in the swish, and each arc of the cycle corresponding to the compatibility between two consecutive cards. Such a cycle C can be written formally as the set of traversed nodes or cards c_1, \dots, c_n , where for each couple of cards c_i and c_{i+1} , there is a directed arc between c_i and c_{i+1} (with $c_{n+1} = c_1$).

However, this seemingly intuitive graph structure is not sufficient. Recall that in SWISH, a card contains four possible configurations as depicted in Figure 2, and since all four configurations of the card are modeled as a single node, then a cycle may represent a *false* swish: let c_1, c_2, c_3 be three cards where c_1 and c_2 are compatible with respect to some configuration r_1 and r_2 , whereas c_2 and c_3 are compatible with respect to some configuration r'_2 and r_3 with r_2 different from r'_2 . Clearly, the cards c_1, c_2, c_3 do not constitute a swish since a swish is composed of a set of cards and a single configuration for each card composing the swish. Hence, c_1 either matches c_2 using configuration r_2 or c_2 matches c_3 using configuration r'_2 , but both statement cannot be achieved using the same configuration for c_2 .

To fix this issue, we rely on *directed hypergraph*, rather than ordinary directed graph, as depicted in Figure 5. This has the following two major modifications: first, four nodes are used to represent each card, one node for each configuration of the card. For the sake of clarity, such a node representing the configuration of a card is called a *configuration node*. Second, a hyper-node of the hypergraph represents a card including all its configuration nodes. Said differently, an hyper-node corresponds to a set of exactly four configuration nodes. We are now ready to focus on the formal hypergraph-based representation of SWISH.



■ **Figure 5** Example of a SWISH-focused hypergraph containing 3 cards. Four possible swishes (identified by numbers on the arcs) are represented.

Formalization of Swish

In SWISH, a card c_i is defined by the position of the point and the circle, and a card has four possible configurations, denoted by $\{r_{i,1}, r_{i,2}, r_{i,3}, r_{i,4}\}$. Remark that among these configuration nodes, one of them is isomorphic to c_i . In order to be agnostic of the card representation, we denote by \mathcal{D} the domain in which a configuration node $r_{i,j}$ is represented. To obtain information on the compatibility between configuration nodes, we define a **Match** : $\mathcal{D} \times \mathcal{D} \mapsto \{\text{true}, \text{false}\}$ algorithm allowing us to identify if two configuration nodes $r_{i,j}$ and $r_{i',j'}$ match, meaning that the point in $r_{i,j}$ meets the circle in $r_{i',j'}$. Obviously, the exact definition of the **Match** algorithm highly depends on the representation of the configuration node space \mathcal{D} . We also define two configuration node manipulation algorithms **FlipLeft** : $\mathcal{D} \mapsto \mathcal{D}$ and **FlipUp** : $\mathcal{D} \mapsto \mathcal{D}$, allowing respectively to apply axial symmetries to a configuration node on the left-side and on the up-side, respectively. Observe that the set of four configuration nodes $\{r_{i,1}, r_{i,2}, r_{i,3}, r_{i,4}\}$ derived from the same card c_i , can be rewritten as $\{r_{i,1}, \text{FlipLeft}(r_{i,1}), \text{FlipUp}(r_{i,1}), \text{FlipLeft}(\text{FlipUp}(r_{i,1}))\}$ with $r_{i,1} = c_i$.

We define a *SWISH-focused hypergraph* $\mathcal{G} = (\mathcal{V}, \mathcal{E}, m)$ as follow:

- The set $\mathcal{V} \subseteq \mathcal{D}$ corresponds to the set of configuration nodes $r_{i,j}$ associated to the j -th configuration of the card c_i .
- The set $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ corresponds to the set of compatibility between configuration nodes, where each $e \in \mathcal{E}$, described as the couple $(r_{i,j}, r_{i',j'}) \in \mathcal{V} \times \mathcal{V}$, must be read as the point of the j -th rotation of the card c_i meets the circle of the j' -th rotation of the card $c_{i'}$.
- An additional mapping function $m : \mathcal{V} \mapsto \mathbb{N}$ which maps a configuration node $r_{i,t}$ to an identifier in \mathbb{N} . We implement m such that for a configuration node $r_{i,j}$, the mapping function m outputs i . We rely on this mapping function to identify if two configuration nodes $r_{i,j}$ and $r_{i',j'}$ are representing the same card, by testing whether $m(r_{i,j}) = m(r_{i',j'})$.

Transposing a set of cards $\mathcal{C} = \{c_1, \dots, c_n\}$ into a SWISH-focused hypergraph can be achieved as follow: for each card $c_i \in \mathcal{C}$, denote all four possible configurations of c_i by $r_{i,1}, r_{i,2}, r_{i,3}$ and $r_{i,4}$. The set of arcs \mathcal{E} of the hypergraph can easily be computed by adding, for two configuration nodes $r_{i,j}$ and $r_{i',j'}$, the arc $(r_{i,j}, r_{i',j'})$ in \mathcal{E} if **Match** $(r_{i,j}, r_{i',j'})$ returns **true**. The mapping function is used in our representation to limit the use of each card at

most once, by restricting the evaluation of $m(r_{i,j})$ for every configuration node $r_{i,j} \in \mathcal{V}$ to return i . Observe that at most four configuration nodes can produce the same identifier $i \in \mathbb{N}$, since a card as at most four possible configurations. These configuration nodes associated with the same identifier compose what we call a hypernode. In the following, we denote by **ConstructHGraph** the algorithm which, from a given set of cards \mathcal{C} , outputs the associated SWISH-focused hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, m)$, working as explained above. By construction, the **ConstructHGraph** algorithm has an asymptotic complexity of $\mathcal{O}(|\mathcal{C}|^2)$, since we have to execute the **Match** algorithm for every distinct configuration nodes $r_{i,j}$ and $r_{i',j'}$.

Finding a swish in such hypergraph remains very similar to searching a cycle in an ordinary graph: in a cycle with nodes $r_{1,j_1}, \dots, r_{n,j_n}$, the point of each configuration node r_{i,j_i} has to meet the circle in $r_{i+1,j_{i+1}}$, which can be checked by testing **Match**($r_{i,j_i}, r_{i+1,j_{i+1}}$). The only one additional constraint is that the set of configuration nodes $r_{1,j_1}, \dots, r_{n,j_n}$ contained in the cycle has to respect the condition that for all $i, i' \in \{1, \dots, n\}$ with $i \neq i'$, we have $m(r_{i,j_i}) \neq m(r_{i',j_{i'}})$, ensuring the cycle to traverse each hyper-node at most once and hence preventing the use of the same card several times.

Computation of large no-swish positions

Thanks to the **ConstructHGraph** algorithm, we are able to define the **NoSwishSet** algorithm which, given a set of cards $\mathcal{C} = \{c_1, \dots, c_n\}$, outputs a subset $\mathcal{C}' \subseteq \mathcal{C}$ where \mathcal{C}' contains no swish of any length. Following the hypergraph modelization, deciding if a given set of cards does not contain any swish can be trivially formalized as $\text{HasNoSwish}(\mathcal{C}) = \neg \text{HasSwish}(\mathcal{C})$, which must be read as “check if the given set of cards contains a swish and return the negation of the result”. To verify if a set of cards \mathcal{C} contains a swish, the set of cards will be encoded as a SWISH-focused hypergraph, since the behavior of **HasSwish** is to decide if there exists some cycle in the hypergraph visiting at most once (and possibly not) each hypernode. In the following, we denote by **FindCycle** the algorithm which, given a SWISH-focused hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, m)$ and a starting configuration node $r_{i,j} \in \mathcal{V}$ used to start the cycle search, outputs a cycle C respecting the above conditions, or \perp if no cycle can be found.

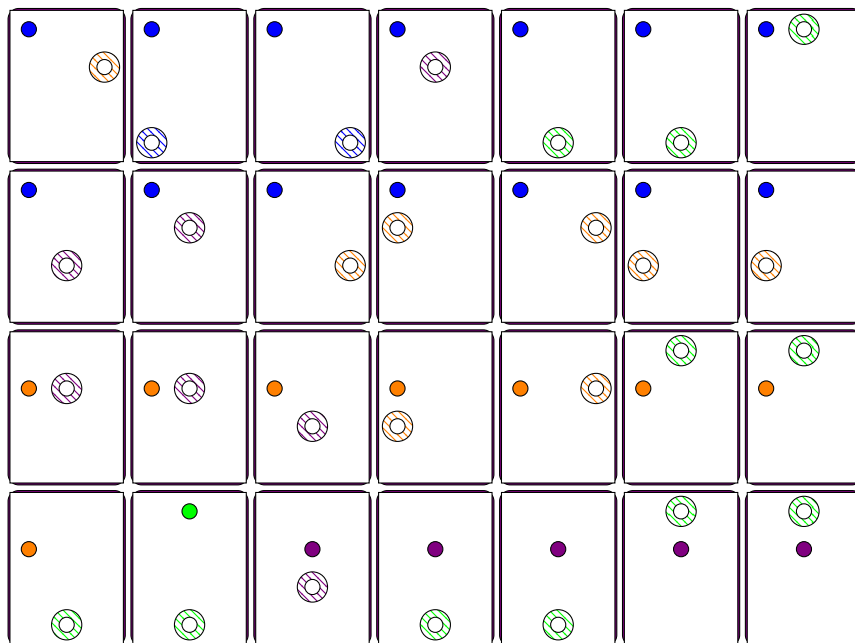
Let us explain the internal behavior of **HasSwish**, taking as an input a set of cards \mathcal{C} . First, the algorithm constructs the SWISH-focused hypergraph $\mathcal{G} \leftarrow \text{ConstructHGraph}(\mathcal{C})$ where $\mathcal{G} = (\mathcal{V}, \mathcal{E}, m)$. Then, since we do not know in advance a configuration node being in a cycle (if one exists), we have to test every configuration node of the \mathcal{G} as the starting point for a cycle, leading to repeat the **FindCycle** algorithm $|\mathcal{V}|$ times. If, for every configuration node, no cycle can be found, then it is clear that no swish exists and hence **HasSwish** returns \perp . Otherwise, one cycle has been found and we end the algorithm by returning \top . The asymptotic complexity of **HasSwish** is $\mathcal{O}(|\mathcal{V}|^2 + |\mathcal{V}| \cdot (|\mathcal{V}| + |\mathcal{E}|)) = \mathcal{O}(|\mathcal{V}|^2 + |\mathcal{V}| \cdot |\mathcal{E}|)$.

Since **HasNoSwish** simply negates the output of **HasSwish**, then the **HasNoSwish** algorithm has a quadratic asymptotic complexity. However, our problem is not limited to find a swish, but rather to find a subset \mathcal{C}' of the set \mathcal{C} such that \mathcal{C}' does not contain any swish. Since we are working on the commercial version of SWISH, which has 60 cards in total, we can use the naive approach consisting of checking for each possible subset \mathcal{C}' of \mathcal{C} if it contains a swish, and exclude this subset if it is the case. This exhaustive search is implemented in practice by using divide-and-conquer: a recursive algorithm taking as parameters a current set of cards \mathcal{C} and the set of remaining cards \mathcal{R} , first extracts from \mathcal{R} a card c and calls itself a first time with the parameters $\mathcal{C} \cup \{c\}$ and $\mathcal{R} \setminus \{c\}$, and a second time with the parameters \mathcal{C} and $\mathcal{R} \setminus \{c\}$. When the set \mathcal{R} is empty, then the algorithm runs **HasNoSwish**(\mathcal{C}) and returns the set $\{\mathcal{C}\}$ if it does not contain any swish, and returns \perp otherwise.

Furthermore, we are able to optimize the no-swish set search using the following heuristic: suppose that \mathcal{C} is a set of cards such that $\text{HasNoSwish}(\mathcal{C})$ fails, meaning that \mathcal{C} contains a swish. Then, for *any* set of cards \mathcal{C}' , the execution of $\text{HasNoSwish}(\mathcal{C} \cup \mathcal{C}')$ also fails. This remark holds since adding a card in the set of cards \mathcal{C} is the same as inserting new configuration nodes and arcs in the hypergraph. As a result, possibly one or more swish are created, but certainly do not delete any existing swish (*i.e.*, cycle) from the hypergraph. We take advantage of this remark to prune the recursive call tree, by checking during the recursion if \mathcal{C} contains a swish, and halt the recursion if a swish is detected.

Results

With our algorithm, we have obtained a largest no-swish position containing 28 cards, which is close to half the number of cards in the commercial version of SWISH. This no-swish position is depicted in Figure 6. Note that it contains duplicates.



■ **Figure 6** No-swish set of cards.

3.2 Generalized no-swish

We begin by focusing on rectangular cards. The basis of our method consists in dividing the cards in four quarters. For one quarter, we fix a point in some position. We then lock its four symmetric positions in the other quarters. This defines a “cross” in the middle of the card, cornered by the four locks. We then generate one card by position in this cross, with a circle in each. Finally, we add one circle on two of the three locks, generating two more cards. For the odd width and height cases, we also have to manage the middle row and column independently. We will, for each possible parity of height and width, give the total number of cards; then explain our strategy to create a large no-swish set, and compute the ratio between those two numbers.

Subdivision of the cards into quarters

We assume that the cards are rectangular. Each card is divided in four quarters, each of size hw . If h or w is odd, then, there is an additional row or column in-between the quarters. The top left quarter is denoted by Q_1 , the top right by Q_2 , the bottom left by Q_3 and the bottom right by Q_4 . Note that there is a bijection between the coordinates (a, b) in Q_1 and the set $\{1, \dots, hw\}$, with $i = (a - 1)w + b$.

Even-even cards

Assume first that the cards have width $2w$ and height $2h$. The set \mathcal{T} containing all possible cards has size:

$$|\mathcal{T}| = \sum_{i=1}^{hw} (4hw - 1) = 4(hw)^2 - hw.$$

We construct the following set \mathcal{S} of cards. For each $i \in \{1, \dots, hw\}$ with $i = (a - 1)w + b$, we create the $4(hw - i) + 2$ following cards, all with a point in $C[a][b]$:

- For each $j \in \{i + 1, \dots, hw\}$ with $j = (c - 1)w + d$, we create four cards, one with a circle in $C[c][d]$ (so in Q_1), one with a circle in $C[c][2w + 1 - d]$ (so in Q_2), one with a circle in $C[2h + 1 - c][d]$ (so in Q_3), and one with a circle in $C[2h + 1 - c][2w + 1 - d]$ (so in Q_4);
- We create two additional cards, one with a circle in $C[c][2w + 1 - d]$ and one with a circle in $C[2h + 1 - a][d]$.

It is easy to see that \mathcal{S} is a no-swish set. Indeed, to create a swish using a card created at step $i = (a - 1)w + b$, one cannot use any card created at step $i' > i$, since none of them has a circle in (a, b) , even using the symmetries. Furthermore, there is no swish using only cards created at step i , since there are only three of them meeting in (a, b) after some symmetries, but they do not form a swish, and thus leave an uncovered point in (a, b) . Hence, a swish using such a card would need to use cards from some step $i' < i$, but doing so will again leave an uncovered point, which will need to be covered using a card from some step $i'' < i'$, and so on until we reach step 1, for which no card can cover the point in the corner.

The construction of \mathcal{S} is depicted on Figure 7. Let us now evaluate its size:

$$|\mathcal{S}| = \sum_{i=1}^{hw} (4(hw - i) + 2) = 2(hw)^2.$$

Hence, the ratio $\frac{|\mathcal{S}|}{|\mathcal{T}|}$ tends to $\frac{1}{2}$ when h and w tend to infinity, so we constructed a no-swish set containing roughly half of the possible cards.

Even-odd cards

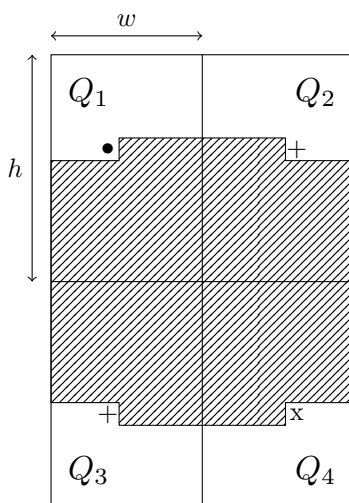
Assume now that the cards have width $2w + 1$ and height $2h$. The set \mathcal{T} containing all possible cards has size:

$$|\mathcal{T}| = \sum_{i=1}^{hw} (2h(2w + 1) - 1) + \sum_{i=1}^h (2hw + 2h - 1) = 2h^2(2w^2 + 2w + 1) - h(w + 1).$$

Note that this coincides with the described cards of the commercial version of SWISH.

We construct the following set \mathcal{S} of cards. For each $i \in \{1, \dots, hw\}$ with $i = (a - 1)w + b$, we create the $4(hw - i) + 2(h + 1 - i) + 2$ following cards, all with a point in $C[a][b]$:

- For each $j \in \{i + 1, \dots, hw\}$ with $j = (c - 1)w + d$, we create four cards, one with a circle in $C[c][d]$ (so in Q_1), one with a circle in $C[c][2w + 2 - d]$ (so in Q_2), one with a circle in $C[2h + 1 - c][d]$ (so in Q_3), and one with a circle in $C[2h + 1 - c][2w + 2 - d]$ (so in Q_4);



■ **Figure 7** Construction of a no-swish set for even-even cards. We place a point in the dotted position, and one card for each possible circle in the area filled with lines, as well as two cards with circles in the two positions with a +. We repeat this for every position in Q_1 .

- For each $j \in \{a, \dots, h\}$, we create two cards, one with a circle in $C[j][w+1]$ and one with a circle in $C[2h+1-j][w+1]$ (so both circles are in the middle column);
- We create two additional cards, one with a circle in $C[c][2w+2-d]$ and one with a circle in $C[2h+1-a][d]$.

Furthermore, for each $i \in \{1, \dots, h\}$, we create the $2(hw - wi) + 2(h - i) + 1$ following cards, all with a point in $C[i][w+1]$:

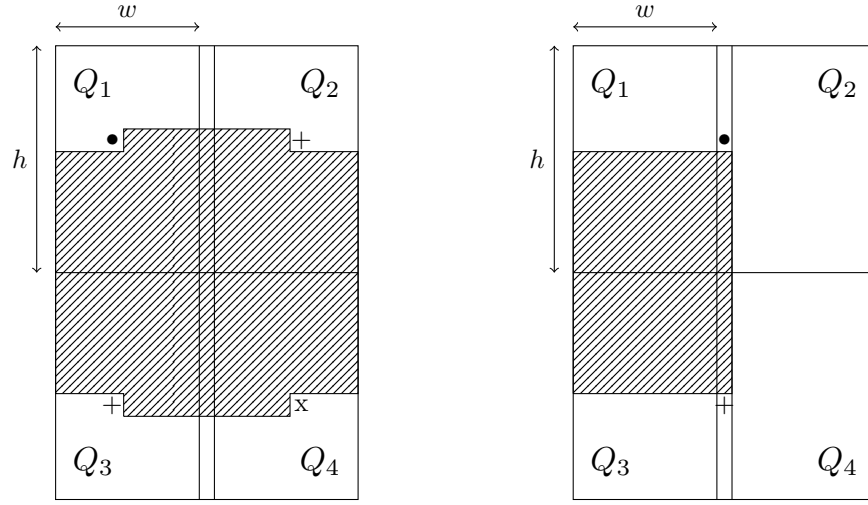
- For each $j \in \{i+1, \dots, h\}$ and $k \in \{1, \dots, w\}$, we create two cards, one with a circle in $C[j][k]$ (so in Q_1) and one with a circle in $C[2h+1-j][k]$ (so in Q_3);
- For each $j \in \{i+1, \dots, h\}$, we create two cards, one with a circle in $C[j][w+1]$ and one with a circle in $C[2h+1-j][w+1]$ (so both circles are in the middle column);
- We create one additional card with a circle in $C[2h+1-i][w+1]$.

Again, it is easy to see that \mathcal{S} is a no-swish set (the proof follows the same arguments as above).

The construction of \mathcal{S} is depicted on Figure 8. Let us now evaluate its size:

$$|\mathcal{S}| = \sum_{i=1}^{hw} (4(hw - i) + 2(h + 1 - i) + 2) + \sum_{i=1}^h (2(hw - wi) + 2(h - i) + 1) = h^2(w^2 + 3w + 1).$$

Note that, by using $h = 2$ and $w = 1$, we obtain $|\mathcal{S}| = 20$, which corresponds to the optimal no-swish position found by the `NoSwishSet` algorithm on the commercial version of `SWISH` (excluding duplicates). Hence, the ratio $\frac{|\mathcal{S}|}{|\mathcal{T}|}$ tends to $\frac{1}{4}$ when h and w tend to infinity, so we constructed a no-swish set containing roughly a quarter of the possible cards.



(a) We place a point in the dotted position, and one card for each possible circle in the area filled with lines, as well as two cards with circles in the two positions with a +. We repeat this for every position in Q_1 .

(b) We place a point in the dotted position, and one card for each possible circle in the area filled with lines, as well as one card with a circle in the position with a +. We repeat this for every position in the first half of the middle column.

■ **Figure 8** Construction of a no-swish set for even-odd cards. There are two sub-constructions.

Odd-odd cards

Assume finally that the cards have width $2w + 1$ and height $2h + 1$. The set \mathcal{T} containing all possible cards has size:

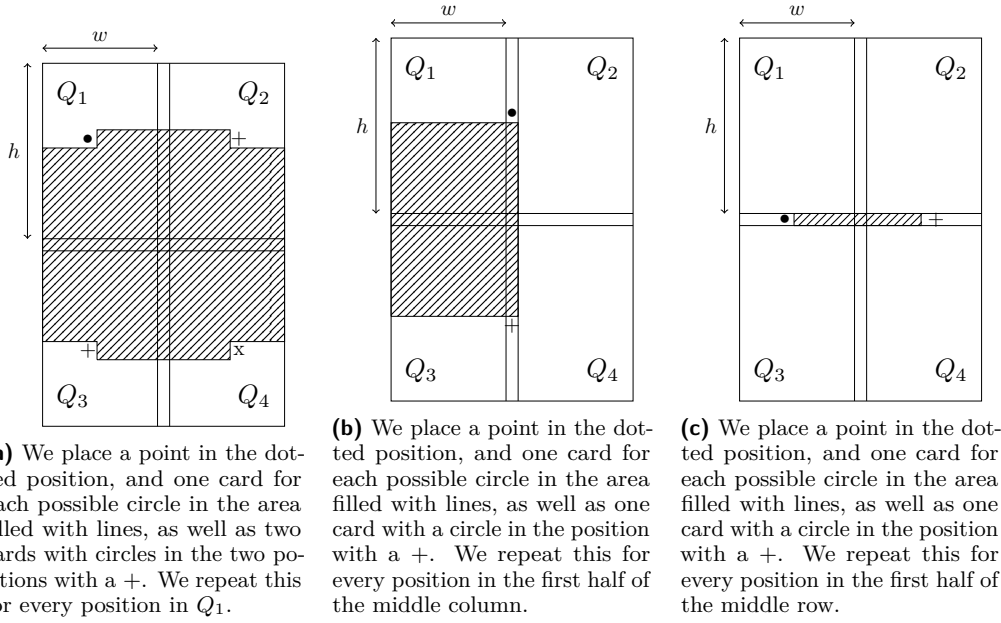
$$\begin{aligned}
 |\mathcal{T}| &= \sum_{i=1}^{hw} ((2h + 1)(2w + 1) - 1) + \sum_{i=1}^h ((2h + 1)w + 2h) \\
 &\quad + \sum_{i=1}^w (h(2w + 1) + 2w) + wh + w + h \\
 &= 4(hw)^2 + hw(4w + 4h + 3) + h(2h + 1) + w(2w + 1).
 \end{aligned}$$

We construct the following set \mathcal{S} of cards. For each $i \in \{1, \dots, hw\}$ with $i = (a - 1)w + b$, we create the $4(hw - i) + 2(h + 1 - i) + 2w + 3$ following cards, all with a point in $C[a][b]$:

- For each $j \in \{i + 1, \dots, hw\}$ with $j = (c - 1)w + d$, we create four cards, one with a circle in $C[c][d]$ (so in Q_1), one with a circle in $C[c][2w + 2 - d]$ (so in Q_2), one with a circle in $C[2h + 2 - c][d]$ (so in Q_3), and one with a circle in $C[2h + 2 - c][2w + 2 - d]$ (so in Q_4);
- For each $j \in \{a, \dots, h\}$, we create two cards, one with a circle in $C[j][w + 1]$ and one with a circle in $C[2h + 2 - j][w + 1]$ (so both circles are in the middle column);
- For each $j \in \{1, \dots, 2w + 1\}$, we create one card with a circle in $C[h + 1][j]$ (so the circle is in the middle row);
- We create two additional cards, one with a circle in $C[c][2w + 2 - d]$ and one with a circle in $C[2h + 2 - a][d]$.

Furthermore, for each $i \in \{1, \dots, h\}$, we create the $2(hw - wi) + 2(h - i) + w + 2$ following cards, all with a point in $C[i][w + 1]$:

- For each $j \in \{i + 1, \dots, h\}$ and $k \in \{1, \dots, w\}$, we create two cards, one with a circle in $C[j][k]$ (so in Q_1) and one with a circle in $C[2h + 2 - j][k]$ (so in Q_3);



■ **Figure 9** Construction of a no-swish set for odd-odd cards. There are three sub-constructions.

- For each $j \in \{i + 1, \dots, h\}$, we create two cards, one with a circle in $C[j][w + 1]$ and one with a circle in $C[2h + 2 - j][w + 1]$ (so both circles are in the middle column);
- For each $j \in \{1, \dots, w + 1\}$, we create one card with a circle in $C[h + 1][j]$ (so the circle is in the middle row);
- We create one additional card with a circle in $C[2h + 2 - i][w + 1]$.

Finally, for each $i \in \{1, \dots, w\}$, we create the $2(w - i) + 2$ following cards, all with a point in $C[h + 1][i]$:

- For each $j \in \{i + 1, w\}$, we create two cards, one with a circle in $C[h + 1][j]$ and one with a circle in $C[h + 1][2h + 2 - j]$ (so both circles are in the middle row);
- We create two additional cards, one with a circle in $C[h + 1][2w + 2 - i]$ and one with a circle in $C[h + 1][w + 1]$.

Again, using the same argument as above, \mathcal{S} is a no-swish set.

The construction of \mathcal{S} is depicted on Figure 9. Let us now evaluate its size:

$$\begin{aligned}
 |\mathcal{S}| &= \sum_{i=1}^{hw} (4(hw - i) + 2(h + 1 - i) + 2w + 3) + \sum_{i=1}^h (2(hw - wi) + 2(h - i) + w + 2) \\
 &\quad + \sum_{i=1}^w (2(w - i) + 2) \\
 &= hw(hw + 3h + 2w + 2) + h(h + 1) + w(w + 1).
 \end{aligned}$$

Hence, the ratio $\frac{|\mathcal{S}|}{|\mathcal{T}|}$ tends to $\frac{1}{4}$ when h and w tend to infinity, so we constructed a no-swish set containing roughly a quarter of the possible cards.

3.3 Large no-swish positions

In the above subsection, we presented how to construct large no-swish positions for the general version of SWISH with rectangular cards, up to half the total number of cards for the even-even case. Note that the even-odd construction does give a set of the maximum

possible size for the commercial version, as found with the NoSwishSet algorithm. However, we only know that those positions are maximal (*i.e.*, adding any card creates a swish), not whether they are of maximum size. Since they do contain a very high ratio of all possible cards, we conjecture that our method is optimal, in that no no-swish set of a size highest than the ones we construct can exist.

Furthermore, note that the case where $h = w$ is still open, but that in this case no-swish positions should be of a smaller size, since more rotations and symmetries can be applied to the cards, and thus it is easier to create a swish. We leave this case for future consideration.

4 Conclusion & Open Problems

In this work, we initiated a study of SWISH and showed interesting properties. First, by studying SWISH with cards of arbitrary size with three or more symbols, we proved that the complexity of finding a swish is NP-complete. Then, we proposed two distinct algorithms to find large no-swish positions: an exponential algorithm to find the largest set of commercial cards (*i.e.*, cards of original game SWISH), finding a large set of 28 cards, but also a polynomial-time algorithm to construct a set of arbitrarily sized, rectangular cards having two symbols, returning almost half of the possible set of cards.

Some questions remains unanswered, that we leave as open problems. The complexity of solving SWISH, being shown to be polynomial for cards of one symbol and NP-complete for cards with three symbols, remains unclear for cards having 2 symbols. In addition, the optimality of the returned no-swish positions using our algorithm for the generalized SWISH has not been proven and it remains open whether or not it is possible to find a larger no-swish set. As an independent topic of interest, we still hardly understand how the game has been constructed, in particular the motivation to duplicate some cards and not others.

References

- 1 Hiroyuki Adachi, Hiroyuki Kamekawa, and Shigeki Iwata. Shogi on $n \times n$ board is complete in exponential time. *Trans. IEICE*, 70:1843–1852, 1987.
- 2 Jean-François Baffier, Man-Kwun Chiu, Yago Diez, Matias Korman, Valia Mitsou, André van Renssen, Marcel Roeloffzen, and Yushi Uno. Hanabi is np-hard, even for cheaters who look at their cards. *Theoretical Computer Science*, 675:43–55, 2017.
- 3 Elwyn R Berlekamp, John H Conway, and Richard K Guy. *Winning ways for your mathematical plays*. AK Peters/CRC Press, 2004.
- 4 Fábio Botler, Andrés Cristi, Ruben Hoeksma, Kevin Schewior, and Andreas Tönnis. SUPER-SET: A (Super)Natural Variant of the Card Game SET. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FUN.2018.12.
- 5 Kyle Burke. Combinatorial game rulesets, 2024. URL: <http://kyleburke.info/rulesetTable.php>.
- 6 Kamalika Chaudhuri, Brighten Godfrey, David Ratajczak, and Hoeteck Wee. On the complexity of the game of set, 2003.
- 7 Erik D Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *International Symposium on Mathematical Foundations of Computer Science*, pages 18–33. Springer, 2001.
- 8 Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.
- 9 Shimon Even and Robert Endre Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the ACM (JACM)*, 23(4):710–719, 1976.

- 10 Aviezri S Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n . In *International Colloquium on Automata, Languages, and Programming*, pages 278–293. Springer, 1981.
- 11 Robert A Hearn and Erik D Demaine. *Games, puzzles, and computation*. CRC Press, 2009.
- 12 Shigeki Iwata and Takumi Kasai. The othello game on an $n \times n$ board is pspace-complete. *Theoretical Computer Science*, 123(2):329–340, 1994.
- 13 Michael Lampis and Valia Mitsou. The computational complexity of the game of set and its theoretical applications. In *LATIN 2014: Theoretical Informatics: 11th Latin American Symposium, Montevideo, Uruguay, March 31–April 4, 2014. Proceedings 11*, pages 24–34. Springer, 2014.
- 14 Michael Lampis, Valia Mitsou, and Karolina Soltys. Scrabble is pspace-complete. In Evangelos Kranakis, Danny Krizanc, and Flaminia Luccio, editors, *Fun with Algorithms*, pages 258–269, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- 15 David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM (JACM)*, 27(2):393–401, 1980.
- 16 Viet-Ha Nguyen, Kévin Perrot, and Mathieu Vallet. Np-completeness of the game kingdom-otm. *Theoretical Computer Science*, 822:23–35, 2020.
- 17 Venkatesh Raman, B. Ravikumar, and S. Srinivasa Rao. A simplified np-complete maxsat problem. *Information Processing Letters*, 65(1):1–6, January 1998. doi:10.1016/S0020-0190(97)00223-8.
- 18 Frederick Reiber. The crew: The quest for planet nine is np-complete. *arXiv preprint arXiv:2110.11758*, 2021.
- 19 John Michael Robson. The complexity of go. *Proc. IFIP, 1983*, 1983.
- 20 G. Shimoni and Z. Shalem. Swish, 2011. URL: <https://www.thinkfun.com/products/swish/>.
- 21 R Teal Witter. Backgammon is hard. In *International Conference on Combinatorial Optimization and Applications*, pages 484–496. Springer, 2021.
- 22 David Wolfe. Go endgames are pspace-hard. *intelligence*, 9(7):6, 2000.

Hamiltonian Paths and Cycles in NP-Complete Puzzles

Marnix Deurloo 

Utrecht University, The Netherlands

Mitchell Donkers

Utrecht University, The Netherlands

Mieke Maarse

Utrecht University, The Netherlands

Benjamin G. Rin¹  

Utrecht University, The Netherlands

Karen Schutte 

Utrecht University, The Netherlands

Abstract

We show that several pen-and-paper puzzles are NP-complete by giving polynomial-time reductions from the Hamiltonian path and Hamiltonian cycle problems on grid graphs with maximum degree 3. The puzzles include *Dotchi Loop*, *Chains*, *Linesweeper*, *Arukone₃* (also called *Numberlink₃*), and *Araf*. In addition, we show that this type of proof can still be used to prove the NP-completeness of *Dotchi Loop* even when the available puzzle instances are heavily restricted. Together, these results suggest that this approach holds promise in general for finding NP-completeness proofs of many pen-and-paper puzzles.

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

Keywords and phrases Hamiltonicity, NP-completeness, complexity theory, pen-and-paper puzzles

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.11

Acknowledgements We would like to thank Hein Duijf, Jonathan Grube, Rosalie Iemhoff, Dominik Klein, Michael Moortgat, and the anonymous referees for their helpful comments and suggestions.

1 Introduction

Pen-and-paper puzzles are often NP-complete. Famous among such results are the NP-completeness of Sudoku [26] and Minesweeper [17], not least because of the worldwide popularity of these particular puzzles. But innumerable other such results exist (see, e.g., [1], [2], [4], [7], [18], and [19], among countless others). Typically, to say a puzzle is NP-complete means that the problem of deciding whether a given instance of the puzzle is solvable is an NP-complete problem. Other problems regarding puzzles may also be considered (see, e.g., [24]), but the present paper addresses only solvability problems.

Often, the proofs of NP-completeness for pen-and-paper puzzles work by reduction from generally useful NP-complete problems such as 3SAT. However, some puzzles may resist this approach, or at least seem more naturally suited to be proved NP-complete by other reduction strategies. In particular, some puzzles take place on a grid and require the solver to construct a kind of path or loop. For puzzles in this category, it seems plausibly viable to prove their NP-completeness by reducing a variant of the Hamiltonian path or cycle problem

¹ Corresponding author. Authors listed alphabetically.





■ **Figure 1** A grid graph (left) and a grid graph with specified terminal nodes s and t (right).

to them – particularly, a variant involving grid graphs. The present paper² considers five³ puzzles, four of which (*Dotchi Loop*, *Chains*, *Linesweeper*, *Arukone₃*) directly require finding paths or cycles on square grids. The other, *Araf*, requires the solver to construct *regions*, but even there, our reduction shows how to force one such region to be shaped as a (Hamiltonian) path. In all cases, we find the Hamiltonian path or cycle problem on grid graphs with maximum degree 3 to be a successful basis for an NP-completeness proof. We surmise that this approach is broadly applicable to a wide array of other pen-and-paper puzzles.⁴

1.1 Preliminaries

► **Definition 1.** A grid graph is a finite graph $G = \langle V, E \rangle$ whose nodes have integer coordinates, with edges between all and only pairs of nodes with Euclidean distance 1. That is, $V \subseteq \mathbb{Z} \times \mathbb{Z}$ and $E = \{(\langle x, y \rangle, \langle x', y' \rangle) \in (\mathbb{Z} \times \mathbb{Z}) \times (\mathbb{Z} \times \mathbb{Z}) \mid |x - x'| + |y - y'| = 1\}$.

When discussing the (non-)existence of a path from a node s (the start node) to t (the end node), we call s and t terminal nodes. Such a path is a cycle if $s = t$ and is Hamiltonian if it visits each node in G exactly once.

► **Example 2.** In Figure 1 we see two grid graphs. The first has Hamiltonian paths, but no cycle. The second has a Hamiltonian path from specified node s to specified node t .

► **Definition 3.** The problem HC3G (the Hamiltonian cycle problem on grid graphs with max. degree 3) is the problem of determining whether a given grid graph with max. degree 3 has a Hamiltonian cycle.

The problem HP3G (the Hamiltonian path problem on grid graphs with max. degree 3) is the problem of determining whether a given grid graph with max. degree 3 and specified nodes s and t has a Hamiltonian path from s to t .

It is well known since [22, Thm. 2] that both of these problems are NP-complete:

► **Theorem 1** (Papadimitriou & Vazirani, see [22, Thm. 2]). *HP3G is NP-complete.*

² This work has its origins in four bachelor theses completed between 2019 and 2023 by authors of this paper under supervision of the fourth author (see [5], [6], [20], [23]).

³ In Section 7, we also consider a restricted class of *Dotchi Loop* instances, for a total of six NP-completeness proofs.

⁴ This proof strategy is not without precedent. For example, such an approach is taken in [3] to prove the complexity of Amazons puzzles, and [26] proves the ASP-completeness of Slitherlink. The present paper’s earliest results, Theorems 3 and 4, were found by the third author in [20], albeit without restricting the graphs to maximum degree 3, making the proofs more complex than needed. Later, between the completion of the present paper and its appearance in print, we came to learn of some recent findings posted on ArXiv by Hadyn Tang for many loop and path puzzles [25]. These findings show that under certain conditions, puzzles can be proved NP-complete by constructing only gadgets for grid graph nodes with degree exactly 3, removing the need for gadgets of degree 1 or 2. Indeed, these conditions seem to obtain for at least some of the puzzles examined below, such as *Dotchi Loop* and *Linesweeper*. Accordingly, it is possible to simplify proofs wherever such gadgets are not needed. However, such gadgets are typically straightforward to construct anyway, and in any case we often find their inclusion instructive. It remains unclear how many puzzles, if any, are amenable to our approach but not Tang’s.



■ **Figure 2** Four node shapes. (Rotations are also possible.)

► **Corollary 1.** *HC3G is NP-complete.*

While the corollary is not explicitly stated in [22], it can be proven by the same proof method as used for Theorem 1 but with one less step.⁵ An explicit proof can also be found in [3].

Although the authors of [22] may not have predicted the future explosive popularity of pen-and-paper puzzles, nor the academic interest in their computational complexity, their result seems almost ready-made for NP-completeness proofs of many such puzzles – or, more precisely, NP-hardness proofs. Typically, proving that these puzzles are *in* NP is trivial, as verifying the correctness of a given solution attempt is easy to do in polynomial time. The present paper’s results are no exception. Moreover, with respect to NP-hardness, the reader can verify that each reduction presented here is easily computable in polynomial time. Accordingly, we focus hereafter only on presenting the reductions. The early ones have relatively simple gadgets, but in subsequent proofs the intricacy increases.

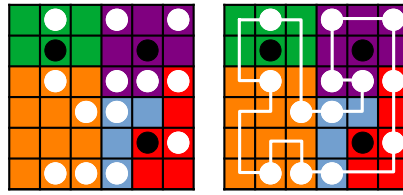
1.2 General scheme

We call gadgets that represent nodes *rooms*. Rooms typically contain portions corresponding to edges, called *hallways* (or *corridors*). Rooms are usually square, or nearly so, and are surrounded on all sides by *walls* – arrangements of cells that prevent the simulated path/cycle from passing through them. Walls are typically a key – perhaps *the* key – subgadget for the success of the sorts of reductions presented in this paper. Their function is to force the simulated path/cycle to go only in the directions we want (i.e., to adjacent rooms) rather than wandering chaotically. On a side of a room with a hallway, the room’s wall will naturally have a gap serving as an *exit* (or *entrance*, since the graph is undirected).

In the case of reductions from HP3G, rooms come in one of four distinct shapes (up to rotation), depending on their degree: a given graph can have one type of degree-1 node, two types of degree-2 node, and one type of degree-3 node (see figure 2). Additionally, a room’s design may differ depending on whether the node it represents is the graph’s starting node s , ending node t , or neither. However, since degree-1 nodes must always be terminal in any Hamiltonian path, we need not design gadgets for nonterminal degree-1 nodes.⁶

⁵ The proof of Theorem 1 works by reduction from a known NP-complete variant of the Hamiltonian cycle problem on directed graphs, wherein one of the steps involves transforming nodes of a given directed graph into nodes of a grid graph with max. degree 3. Here, pains are taken to make the resulting grid graph suitable for checking the presence of a certain path (from specified nodes s and t) rather than a cycle (see [22, Fig. 2]). By dispensing with this effort and simply skipping this step (i.e., ignoring [22, Fig. 2b] and instead transforming all nodes of the directed graph in accordance with [22, Fig. 2a]), we obtain a proof of the NP-completeness of HC3G.

⁶ Trivially, the reduction can easily map a graph containing such a node to any unsolvable puzzle instance. So we can assume the given graph contains no such nodes. On another note, for similar reasons, we can also assume the graph is connected.



■ **Figure 3** A *Dotchi Loop* puzzle instance (left) and its solution (right) [12]. Colors indicate regions.

In the case of reductions from HC3G, we can completely ignore *all* degree-1 nodes, since no graph with a degree-1 node can have a Hamiltonian cycle.⁷ This leaves only three types of room to construct for such reductions, corresponding to (b)-(d) in Figure 2. That said, degree-1 rooms are usually easy to construct, so we can just as well include such gadgets in our proof and have our reduction transform graphs with degree-1 nodes into corresponding puzzle instances just like any other graph, with the understanding that such instances will necessarily be unsolvable.

2 *Dotchi Loop*

We begin with a proof for *Dotchi Loop*. The reduction is simple, but it straightforwardly demonstrates the use of HC3G for such proofs. In Section 7, we present a more complex proof showing *Dotchi Loop* to be NP-complete even when the set of possible puzzle instances is heavily restricted in a certain way.

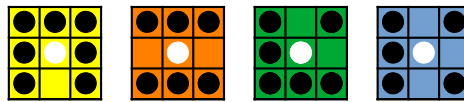
Dotchi Loop [12] is a Japanese pen-and-paper puzzle consisting of a grid of cells divided into contiguous and non-overlapping *regions*. Some cells are empty and some contain a white or black circle. The solver must join cells orthogonally to create a single closed loop that passes through all white circles and avoids all black circles. (See Figure 3.) It is not necessary to visit every empty cell. The loop is forbidden to cross or overlap itself. Within any given region, the loop must either turn 90° at all white circles or pass straight through all white circles of that region. For brevity, we call any White circle through which the loop passes straight a *straight circle*. We call the others *turn circles*. For example, the white circle in the northwest region is straight and the ones in the southwest region are turn circles. A region is *straight* (respectively, *turn*) if its white circles are straight (respectively, turn).

► **Theorem 2.** *Dotchi Loop is NP-complete.*

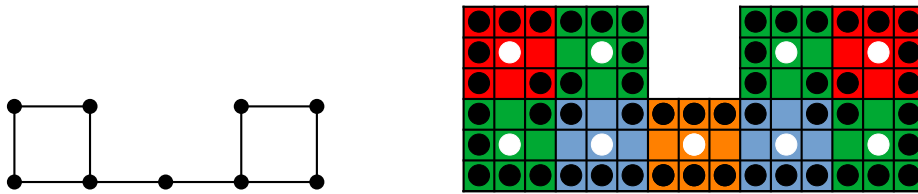
Proof. We give a reduction from HC3G. Wall subgadgets are easy to construct by using rows of black circles. With these, we can straightforwardly build rooms with appropriate hallways. Each fits in a 3×3 area and contains a white circle at the center. We fill the remaining space in each room with black circles, except for pathways toward the central cell from every available entrance. See Figure 4. Each room is its own region. As with all these reductions, rooms are placed in similar relative positions to those of the corresponding nodes and edges in the input graph. An example can be seen in Figure 5.

To confirm the correctness of the reduction, observe that white circles in each room guarantee, per the rules, that a solution's loop will necessarily visit that room. So the similarly shaped cycle in the given grid graph is Hamiltonian. Conversely, if the grid graph is Hamiltonian, a similarly shaped loop can clearly cover all white cells. This completes the proof. ◀

⁷ Again, such a graph can trivially be mapped to any unsolvable instance of the puzzle at hand. And, again, we can do the same with any graph that isn't connected.



■ **Figure 4** All *Dotchi Loop* rooms (up to rotation). The first is unnecessary (see Section 1.2).

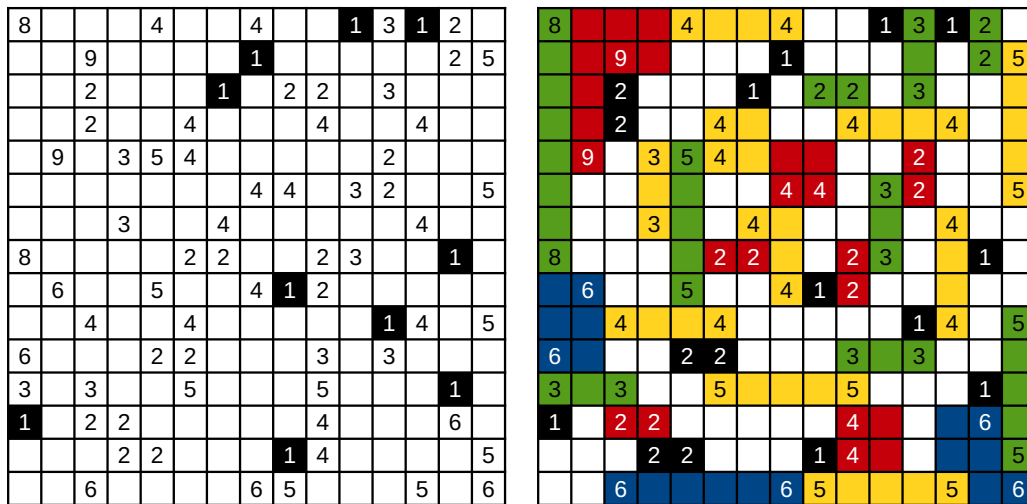


■ **Figure 5** A graph (left) and corresponding *Dotchi Loop* puzzle instance (right). Colors distinguish neighboring regions (though distinct regions with no shared border may share colors in the figure).

► **Example 4.** The graph in Figure 5 has nine nodes. The corresponding puzzle instance has nine rooms. Each room here is a separate *Dotchi Loop* region.

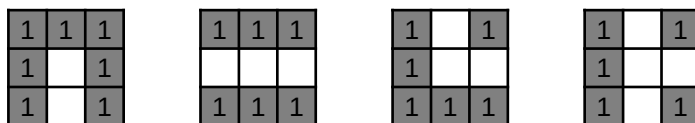
3 Chains

A Chains [13] puzzle instance consists of a grid of cells, some of which contain natural numbers. The solver must join cells orthogonally such that each cell numbered $i > 1$ is connected by a “line” (path) of length i to some other cell also containing i . Cells with number $i = 1$ must be in lines of length 1 (i.e., to themselves). Lines cannot overlap themselves or other paths. See Figure 6.

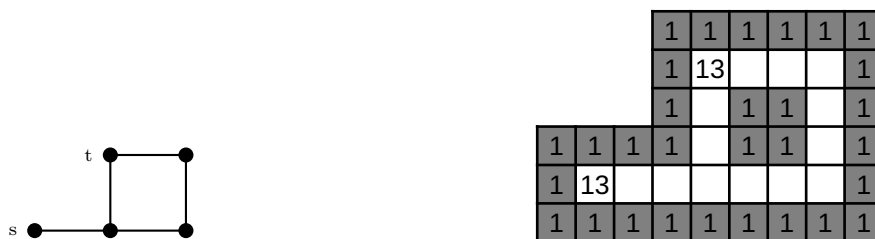


■ **Figure 6** A Chains puzzle instance (left) and its solution (right) [14].

► **Theorem 3.** *Chains is NP-complete.*



■ **Figure 7** Chains nonterminal room gadgets. (Rotations are also possible.)



■ **Figure 8** A graph (left) and equivalent Chains puzzle instance (right).

Proof. We give a reduction from HP3G. Since lines may not cross, walls are simply constructed out of cells containing 1s. Room gadgets consist of 3×3 blocks of cells. The gadgets for each nonterminal node type are presented in Figure 7. As usual, rooms are placed in relative positions corresponding to those of the given graph's nodes.

The rooms representing terminal nodes are constructed identically to the nonterminal nodes, except that we add the number $3m - 2$ to their central cells, where m is the number of nodes in the graph. See Figure 8 for an example.

To verify the reduction's correctness, first observe that if a Hamiltonian path exists between s and t , then the two corresponding central cells can indeed connect via an identically shaped line of length $3(m - 1) + 1 = 3m - 2$, since (i) the number of cells needed to move from one room's central cell to the next is always three, (ii) the path from s to t uses $m - 1$ edges, so the similarly shaped line in the puzzle connects $m - 1$ pairs of central cells from adjacent rooms, and (iii) cell s itself must be counted in the total length.

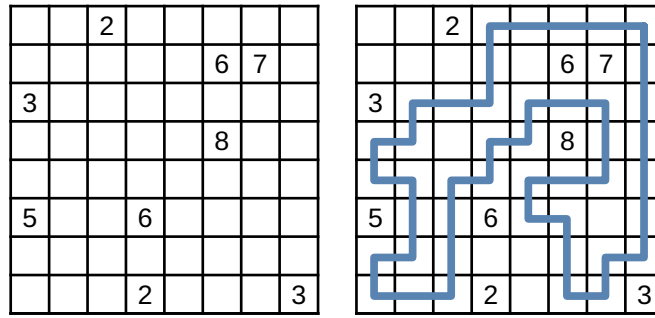
Conversely, the puzzle instance is constructed in such a way that there is virtually no flexibility for the solver. From the center of any room, any movement of three steps leads to the center of another room. Therefore, a line connecting the two cells numbered $3m - 2$ (which must have length $3m - 2$, by the rules) visits exactly m rooms, making the similarly shaped path from s to t in the graph Hamiltonian. ◀

► **Example 5.** The graph in Figure 8 has five nodes, so $3m - 2 = 3(5) - 2 = 13$.

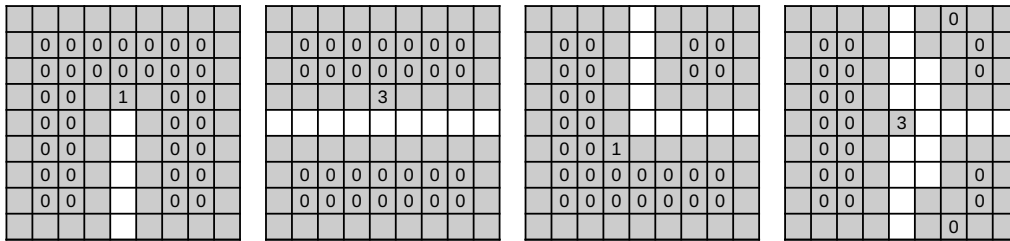
4 Linesweeper

Linesweeper [15], like Chains, consists of a grid of cells, some of which contain nonnegative⁸ integers. The solver must find a loop through the grid such that every numbered cell is orthogonally or diagonally adjacent to precisely that number of cells visited by the loop. An example is shown in Figure 9.

⁸ Note that typical instances of Linesweeper have no cells with 0s, but the puzzle's definition does explicitly allow them – see [21].



■ **Figure 9** A Linesweeper puzzle instance (left) and its solution (right) [16].



■ **Figure 10** All Linesweeper rooms (up to rotation). Cells clearly inaccessible to the solution loop are shaded as a visual aid. Note that we could also use only the last room and add 0s to block exits.

► **Theorem 4.** *Linesweeper is NP-complete.*

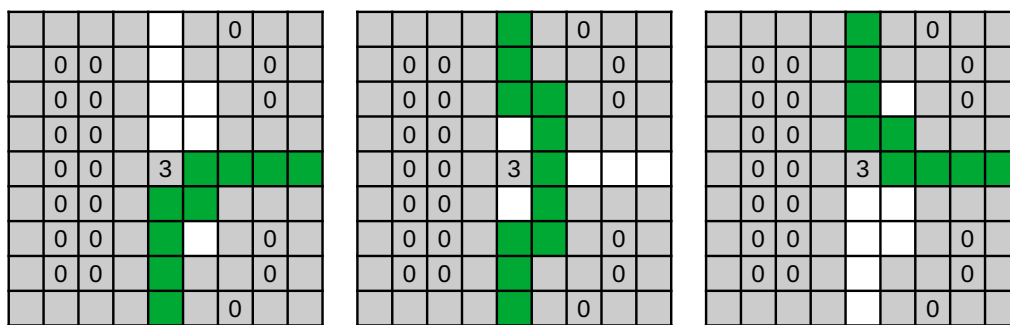
Proof. We give a reduction from HC3G. Walls are created by use of 0s. Under the rules, the loop cannot touch any cells adjacent to a cell containing a 0, so each 0-cell creates a 3×3 impenetrable area. Room gadgets are shaped rather similarly to those of Chains and *Dotchi Loop*, though larger to accommodate the peculiar needs of the degree-3 room. In or adjacent to the center of each room is a cell with the number 1 or 3, depending on the room type. See Figure 10.

It is straightforward to see that the gadgets other than the degree-3 room are solvable⁹ in exactly one way. The degree-3 room is solvable in three ways, displayed in Figure 11. A full example of the reduction is shown in Figure 12.

To verify the reduction’s correctness, we first note that if a given graph has a Hamiltonian cycle, the corresponding Linesweeper instance is clearly solvable by a loop of similar shape. For the converse, suppose the puzzle instance is solvable. Observe that the 1 or 3 in each room guarantees that the solution loop must visit that room. As noted above, every room without degree 3 can be traversed by the loop in only one way, and so it is easy to see that our needed cycle in the graph takes precisely the same shape traversing the corresponding node. For the degree-3 rooms, the reader can verify that the three solutions presented

⁹ We acknowledge that only a puzzle instance as a whole, not a subset (such as a single room) can be formally said to be solved or have a solution. However, as a mild abuse of terminology, we may say throughout this paper that an indication (usually pictorial, but possibly written) of things for a puzzle solver to do with respect to a given room is a *solution* to the room if the indication would describe part of a correct solution attempt for an entire puzzle instance containing that room. We will say the room is *solved* if a solution (in this sense) has been given for it.

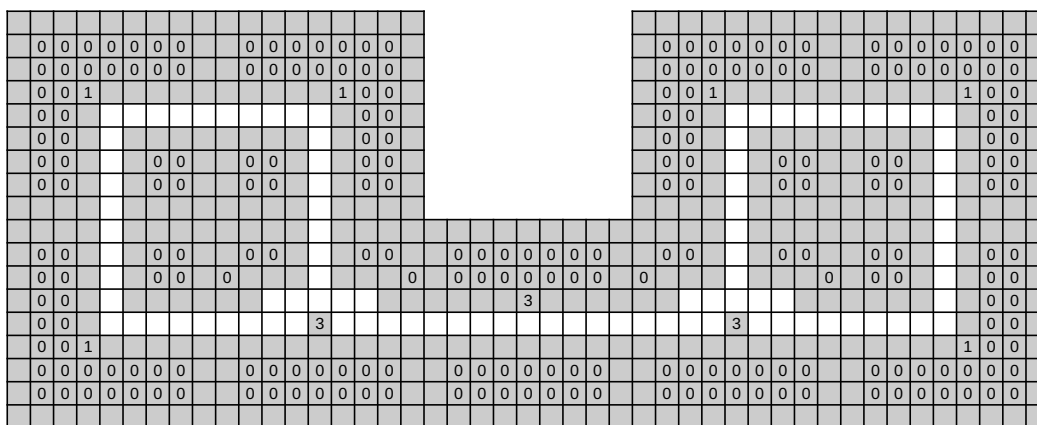
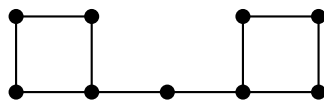
11:8 Hamiltonian Paths and Cycles in NP-Complete Puzzles



■ **Figure 11** Three solutions for a degree-3 Linesweeper room.

in Figure 11 are the only possible ways to solve such rooms. Consequently, we see that whichever direction the solution loop takes through a degree-3 room, a Hamiltonian cycle through the input graph can take the same direction. Thus a given grid graph with max. degree 3 is Hamiltonian if and only if the corresponding Linesweeper instance is solvable, as required. ◀

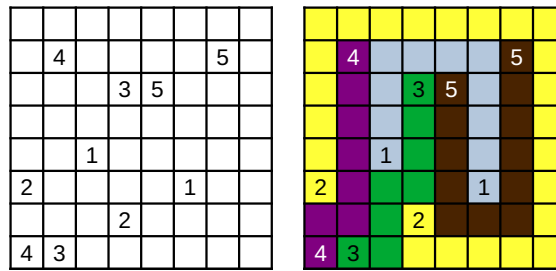
► **Example 6.** In Figure 12 we see a full example of the reduction.



■ **Figure 12** A graph and its corresponding Linesweeper puzzle instance.

5 *Arukone*₃

*Arukone*₃ (written *Arukone*³ in [11]) is another puzzle consisting of a grid of blank and natural-numbered cells, with the additional stipulation that each number that is present appears exactly twice. This puzzle is a variant of *Arukone* (also called Number Link, Nanbarinku, or



■ **Figure 13** An $Arukone_3$ puzzle instance (left) and its solution (right) [11].

Flow – see [9]) and $Arukone_2$ (see [10]), which have already been considered.¹⁰ The $Arukone_3$ variant contains just one added rule beyond $Arukone_2$ (the 2×2 rule – see below), but this rule markedly affects the nature of the puzzle.

The goal is to connect each pair of equal numbers with a path (called a *line*) of orthogonally adjacent cells that never crosses itself or another line and never covers a 2×2 area. The puzzle is solved when all number pairs are connected and all empty cells are filled. Figure 13 shows an example.

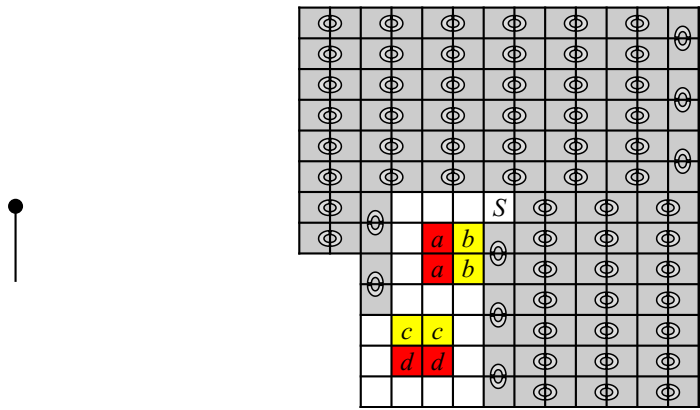
► **Theorem 5.** $Arukone_3$ is NP-complete.

Proof. Our reduction for this puzzle is from HP3G. Accordingly, the grid graph should have two distinguished terminal nodes, s and t . Recall that this makes reductions from HP3G require the construction of more room types than in reductions from HC3G. In the case of $Arukone_3$, the two rooms corresponding to nodes s and t will share a number S in their central cells. We call the line connecting the two S -cells (assuming the puzzle instance is solvable) the *Hamilton line*. In this proof, there is no relevant distinction to make between a starting room and ending room, so we have only one gadget design for, say, the degree-3 terminal room. (In the proof of Theorem 6 for $Araf$, such a distinction is necessary and more room types will therefore be needed.)

In our reduction to $Arukone_3$, the “walls” will consist of many number pairs arranged as dominoes (see, e.g., Figure 14). Since the numbers in a given domino can only be connected to each other, and since these dominoes are so tightly packed together, the only connection possible for them is a line of length 2. In general, we say that two equally numbered cells are connected *directly* if the line connecting them has length 2, and *indirectly* otherwise. In our diagrams, wall cells are shaded for readability. Additionally, rather than display a unique number pair for each domino within walls, we simply draw a ring (see, e.g., the bottom-right horizontal dominoes in Figure 14) to indicate that this pair of adjacent cells must be directly connected in any solution.¹¹ However, not all dominoes will be displayed this way; some (non-wall) dominoes that may be possible to connect indirectly will be shown with letters standing for natural numbers – in the present figure, a, b, c , and d . (As discussed above, the central S is also a natural number.) It is easy to see that the walls constructed here serve their intended purpose sufficiently: since lines cannot intersect, the mass of directly connected dominoes prevents another line (in particular, the Hamilton line) from getting through.

¹⁰ See [1]. Also notable is [19], which shows the NP-completeness of a further $Arukone$ variant.

¹¹ For most of these dominoes, the need for direct connection is obvious because they have no empty space around them. For some – e.g., the vertical dominoes on the left of Figure 14 – direct connection is forced because any indirect connection would violate the 2×2 rule.



■ **Figure 14** A degree-1 node and $Arukone_3$ degree-1 room. Colors are only a visual aid.

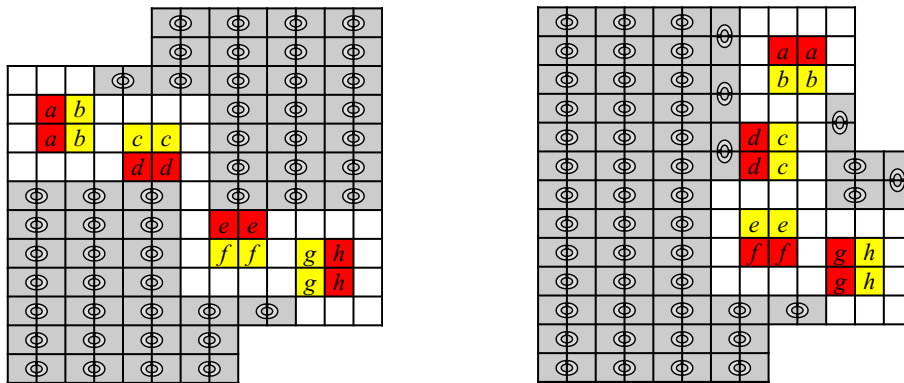
Rooms in this reduction are 13×13 square areas, minus ten cells for each corridor out. These cells are occupied by the adjacent room. The room in Figure 14 has one corridor, which consists of all the empty cells in the southwest quadrant forming a rough δ -shape. The exit is west of the cc domino. Later it will become clearer how rooms in this reduction fit together.

One oddity in this reduction, in contrast to the rest in this paper, is that rooms in this reduction are tilted 45° clockwise from the nodes they represent. For example, Figure 14 depicts a degree-1 room in which the outgoing corridor heads southwest, representing a degree-1 node with an outgoing edge to the *south*.

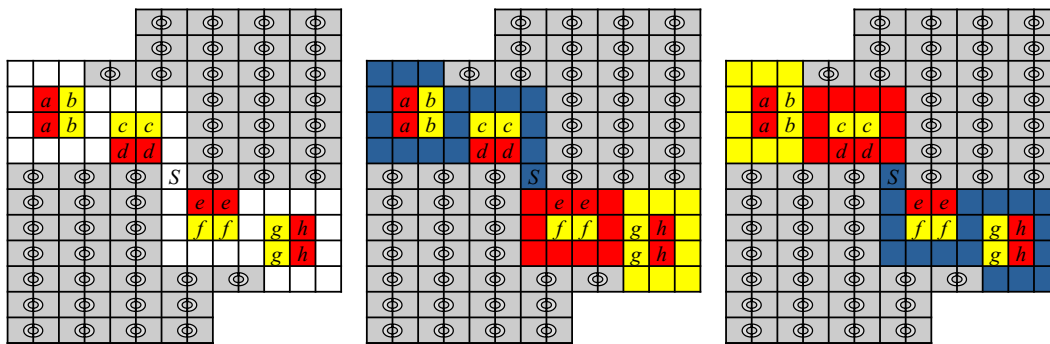
The aforementioned non-wall dominoes in Figure 14 with numbers $a, b, c,$ and d form what we call *pillars*. The two dominoes aa and bb together comprise one pillar and the dominoes cc and dd comprise another. In general, a pillar diagonally touching the central cell (here, the a - b pillar) is called an *inner pillar* and a pillar more distant (here, c - d) is called an *outer pillar*. In the present gadget, all four of these dominoes must be connected directly (the degree-1 room bears only one possible solution, wherein pillar dominoes must connect directly to avoid blocking the line from S). However, in other gadgets this will not always hold. As we will see, the purpose of pillars, in rooms with more than one possible solution, is to be able to connect indirectly and fill the cells of corridors not in use by the Hamilton line. Since the rules of $Arukone_3$ require that all cells be filled, pillars are a key part of the construction that make the reduction work. We will see, furthermore, that when a pillar domino needs to connect indirectly to fill unused corridor cells, it can do so without breaking the 2×2 rule.

Figure 15(a) depicts a nonterminal straight degree-2 room with northwest and southeast corridors, corresponding to a nonterminal straight degree-2 node with west and east edges. Figure 15(b) depicts a nonterminal corner degree-2 room with northeast and southeast corridors, corresponding to a nonterminal corner degree-2 node with north and east edges.

The straight room, we claim, is solvable in exactly one way, with no possibility for the pillars to have indirect connections – similarly to the degree-1 room. In this unique solution, the empty cells must be all filled by a single line passing completely through the room. (It will turn out that this line is the Hamilton line.) To verify this, observe by inspection that the central cell cannot be reached by any possible line that would indirectly connect a domino within this room. So the central cell must be in some line extending outside the room. Since the central cell, like any cell without a number, can never be the endpoint its line, its line



■ **Figure 15** Nonterminal degree-2 straight and corner *Arukone*₃ rooms.



■ **Figure 16** An *Arukone*₃ terminal degree-2 straight room and its two solutions.

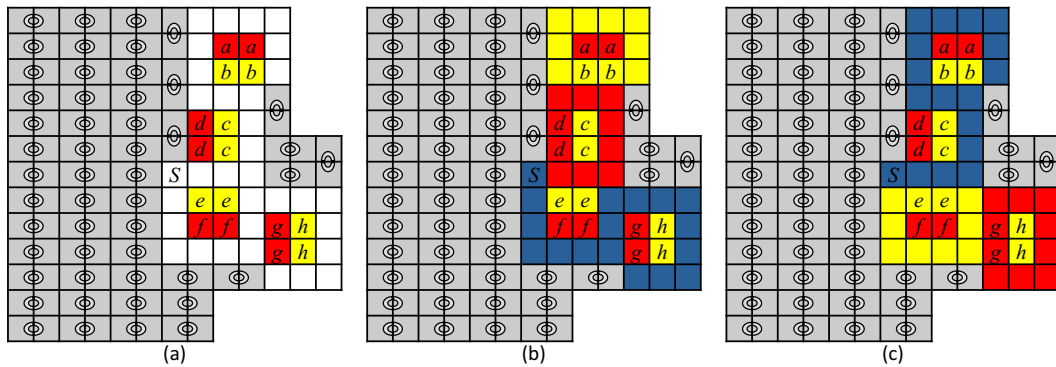
must extend out through both exits (in the process, filling both of the two corridors entirely). In sum, this room is solved only by running a line from one exit to the other, passing through the center cell along the way. This behavior matches that of nonterminal straight degree-2 nodes in solved HP3G instances.

The corner room is similar. Its solution requires a single line containing the center, both exits, and (consequently) all other white cells. The only difference in the argument is in the reason why the center is inaccessible to pillar connection lines. For the straight room, it was because of physical inaccessibility, but in the corner room, the 2×2 rule plays a role in preventing an inner pillar domino's line from filling the center. In fact, we can see by inspection that all pillar domino lines in the corner room must connect directly. As with the straight room, the unique possible solution for the present gadget matches that of the corresponding node type in solved HP3G instances.

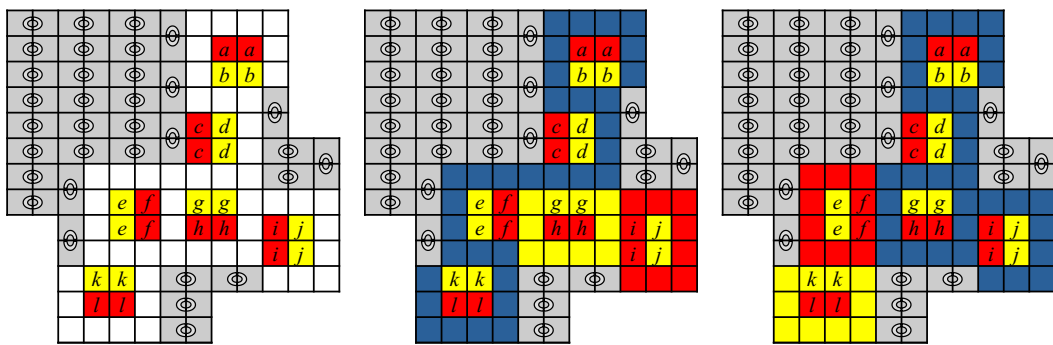
In contrast to these two nonterminal degree-2 rooms, the terminal degree-2 rooms can (unsurprisingly) be solved two ways, neither of which can use both corridors (because the cell with number *S* cannot be a middle point of its line). The two possible solutions, shown in Figure 16, correspond precisely to the two possible behaviors of terminal straight degree-2 nodes in solved HP3G instances. Here the use of pillars to fill otherwise empty corridor space exemplifies our earlier description.

Figure 17 shows the terminal corner degree-2 room. We claim this room is solvable in two ways, much like the terminal room already discussed. Note that in Figure 17 there are two possible cells for the Hamilton line to visit in its first step from cell *S*. If it visits the cell

11:12 Hamiltonian Paths and Cycles in NP-Complete Puzzles



■ **Figure 17** An $Arukone_3$ terminal degree-2 corner room and its two solutions.

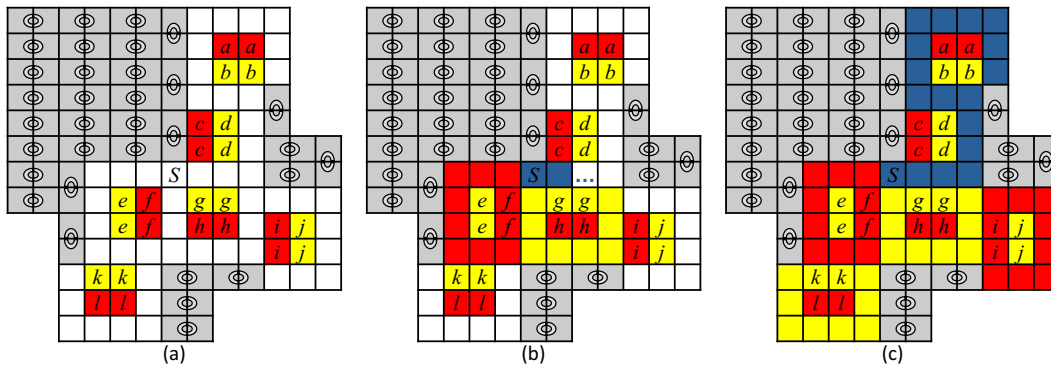


■ **Figure 18** An $Arukone_3$ nonterminal degree-3 room and two of three solutions.

south of S , then it cannot also visit any cells north of the center row, as that would leave behind white cells that pillar dominoes could never fill without breaking the 2×2 rule. By a similar argument, if the line originating from S heads east, then it cannot visit any cells *south* of the center row. In either case, the cells not filled by the Hamilton line can and must be filled by the pillars, as seen in the figure.

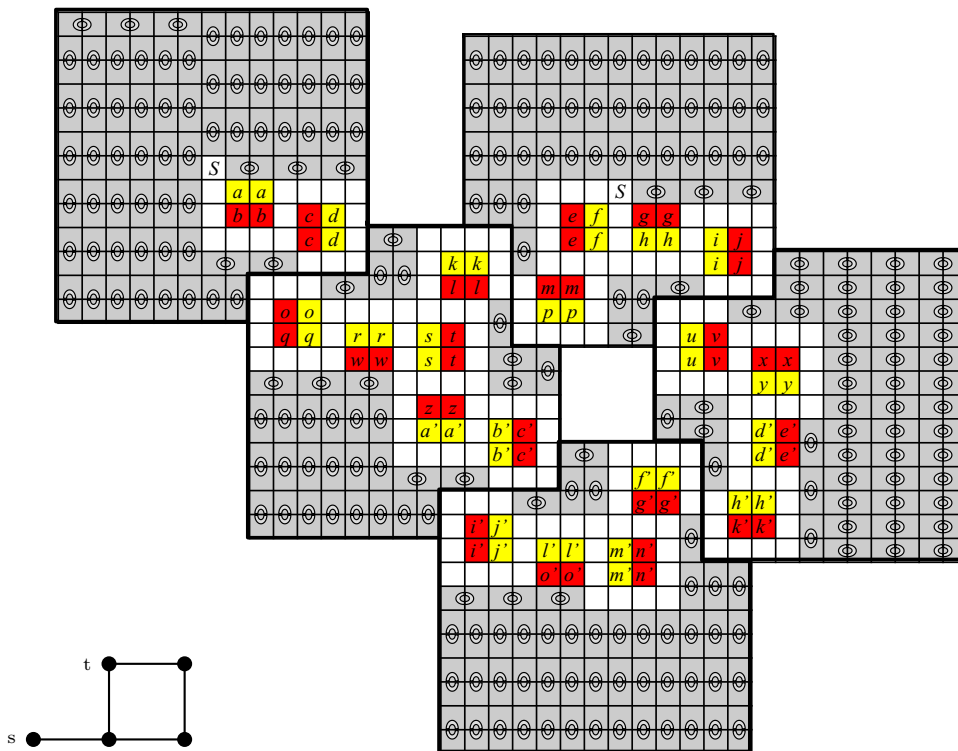
Figure 18 displays a nonterminal degree-3 room. We claim there are precisely three possible solutions (two, up to symmetry), in concordance with nonterminal degree-3 nodes in solved HP3G instances. First, note that the central cell must, as before, be part of a line that traverses precisely two of the exits. If they are the northeast and southwest exits, then the solution must be as in Figure 18(b) (other attempts – for instance, using the three white cells in the central column – fail for reasons similar to those previously discussed). Likewise, if they are the northeast and southeast, the solution must be as in the third figure. The last case (southwest and southeast) is symmetrical to this.

Finally, Figure 19 displays a terminal degree-3 room. As noted for the other terminal rooms, the cell numbered S must be an endpoint of its line, so its line must visit precisely one of the three cells adjacent to it. (The other two must be filled by inner pillars.) If it moves east, then the other two cells neighboring S must be filled by inner pillars (see Figure 19(b)), after which there is no option but to take the northeast exit (Figure 19(c)). The other two cases – moving one cell south or west (not shown) – are similar (forcing east and south exits, respectively). Again, in this gadget we see all, and only, the possible behaviors corresponding to those of terminal degree-3 nodes in HP3G instances.

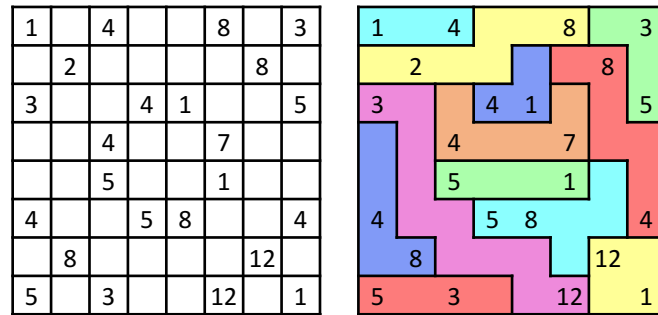


■ **Figure 19** $Arukone_3$ terminal degree-3 room (a), a partial solution (b), and a full solution (c).

This completes the discussion of individual gadgets. An example puzzle instance showing how they fit together is given in Figure 20. Extra walls can be added to make the instance rectangular. This completes the proof. ◀



■ **Figure 20** A graph and its corresponding $Arukone_3$ instance. (Further walls in white space not shown.) The instance has a 45° clockwise tilt compared to the graph.



■ **Figure 21** An *Araf* puzzle instance (left) and its solution (right). [8].

6 *Araf*

Araf, like *Linesweeper* and *Chains*, consists of a grid of cells that each may contain a natural number. (Unlike *Arukone*₃, *Araf* need not have precisely two of each present number.) The goal is to divide the grid into contiguous regions such that (i) each cell is part of exactly one region, (ii) each region contains exactly two numbers, and (iii) the size of each region’s area is strictly between those two numbers. Figure 21 presents an example.

► **Theorem 6.** *Araf is NP-complete.*

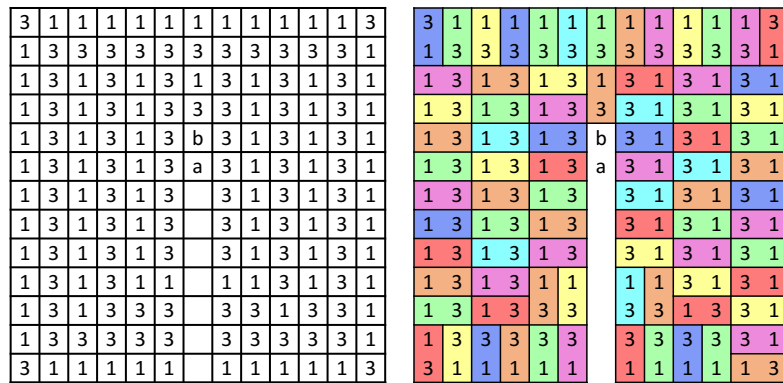
Proof. We give a reduction from HP3G. In this reduction, it will be necessary to construct eleven rooms, not seven as in *Arukone*₃. This is because we have separate designs for rooms representing start nodes and end nodes, rather than a single design for both types of terminal node for any given node degree/shape. We will begin with the rooms for start nodes, then present the rooms for nonterminal nodes, and then finally present the rooms for end nodes.

Let us say two cells are *connected* in a solution if they occupy the same region. We may at times identify a numbered cell with its number, if no confusion arises. Thus we may say, e.g., that the numbers *a* and *b* in a given room are connected.

In all rooms we construct, the walls are built out of dominoes that each consist of a 1 and a 3 (see Figure 22). By rule (iii), the two domino cells must be together in a region of area exactly 2, as long as no nearby numbered cells provide other options. We construct the rooms in such a way that no problem with this arises. Technically, while it is conceivable that a large area filled with wall dominoes could be solved in more than one way (e.g., if a horizontal 3-1 domino is above a horizontal 1-3 domino, these four cells can be paired off vertically or horizontally), these differences have no meaningful effect on the puzzle solution or our proof. In particular, the region corresponding to the grid graph’s Hamiltonian path (hereafter, the *Hamilton region*) is unaffected. Accordingly, we will not distinguish between solutions with identical Hamilton regions that differ only by small variations such as with the 3-1 pairings.

Importantly, all the rooms in this proof are constructed in such a way that they can’t affect each other. The sides of each room are filled with 1s and there is a 3 in each corner. Thus the cells on the side of a room will be unable to share a region with cells on the side of an adjacent room. Therefore, it will be impossible for a solution to have any region include cells from two or more gadgets, with the exception of the Hamilton region.

If the grid graph has a Hamiltonian path, the size of the Hamilton region in the puzzle is straightforward to calculate. As we will see, the start room will have nine cells available for the Hamilton region, the end room will have seven, and the nonterminal rooms will each have thirteen. Hence the region has $9 + 7 + 13(n - 2) = 13n - 10$ cells, where *n* is the number of nodes in the grid graph.



■ **Figure 22** An *Araf* degree-1 start room (left) and solution.

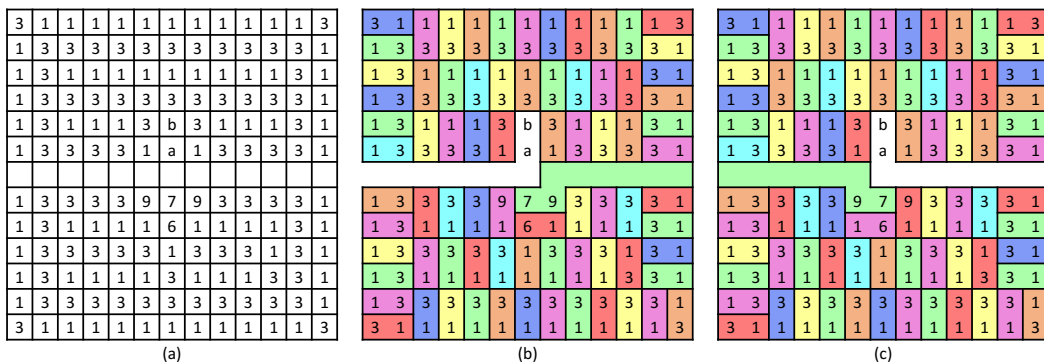
We now define $a = 13n - 11$ and $b = a + 2 = 13n - 9$. The room representing the start node will contain a domino with a and b , arranged so that these two numbers must connect in any solution. Their region, with forced size $13n - 10$, will be the Hamilton region.

Note that the rest of the numbers in the puzzle will all be 9 or less, so the numbers a and b will be by far the greatest.

In Figure 22 the degree-1 start room is shown. There are eighty 3-1 regions and two (large) numbers a and b , which we now argue must share a region (the Hamilton region). Observe that b cannot connect to one of the 3s next to it – for, if it did, its region would need to have size at least 4, which is not possible in the given room. So b must connect to a . A similar observation will hold for the other start rooms.

Since a and b are connected, the 3s and 1s in this gadget must connect to each other. Since each such connection yields a region of size 2 under *Araf*'s rules, there must be $80 \cdot 2 = 160$ cells filled. There are $13 \times 13 = 169$ total cells in this gadget, so the nine remaining cells, including a and b , must be occupied by the Hamilton region.

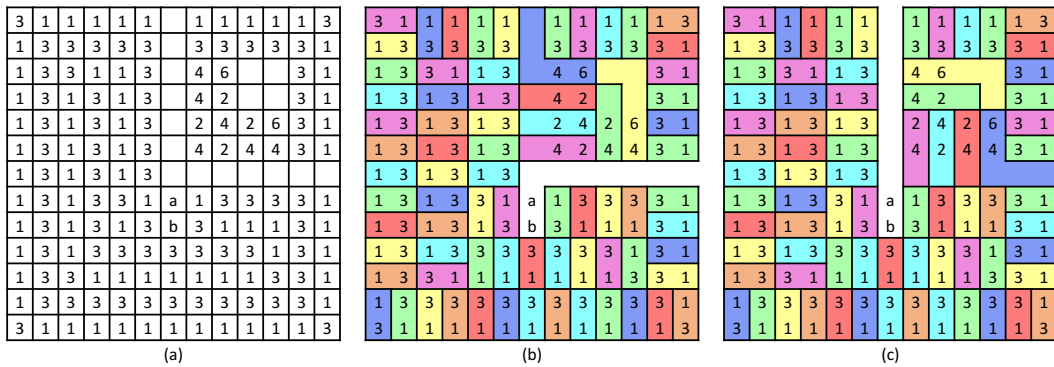
Figure 23 shows the straight degree-2 start room. We argue that a solution for it must contain seventy-four 3-1 regions, one 9-7 region, one 9-1 region and one 6-1 region. This results in at least $74 \cdot 2 + 8 + 2 + 2 = 160$ filled cells.



■ **Figure 23** An *Araf* degree-2 straight start room (left) and solutions.

For all 1s except those adjacent to a or 6, it is obvious that they must connect to a neighboring 3. For each of the 1s next to a , if it would connect elsewhere then the only options would be a 7 or 9. However, the 7 is impossible because this would block the a and b ,

11:16 Hamiltonian Paths and Cycles in NP-Complete Puzzles



■ **Figure 24** An *Araf* degree-2 corner start room (left) and solutions.

whose region requires far more than just two cells. Likewise for the far 9. For the close 9 (two spaces below the 1), connecting would leave a 3 from the same quadrant as the 1 unpaired, forcing the 3 to connect to the 7 or remaining 9. But this, even if somehow possible, would again block off a and b . So each 1 next to a must connect to a neighboring 3.

The 7 must now connect to a 9, as it cannot connect to a 6 by the rules of *Araf*. The resulting 9-7 region requires exactly eight cells. To avoid blocking the a and b , this region must be as in Figure 23(b) or (c). All other cells in the central row must remain open to be filled by the b - a (Hamilton) region.

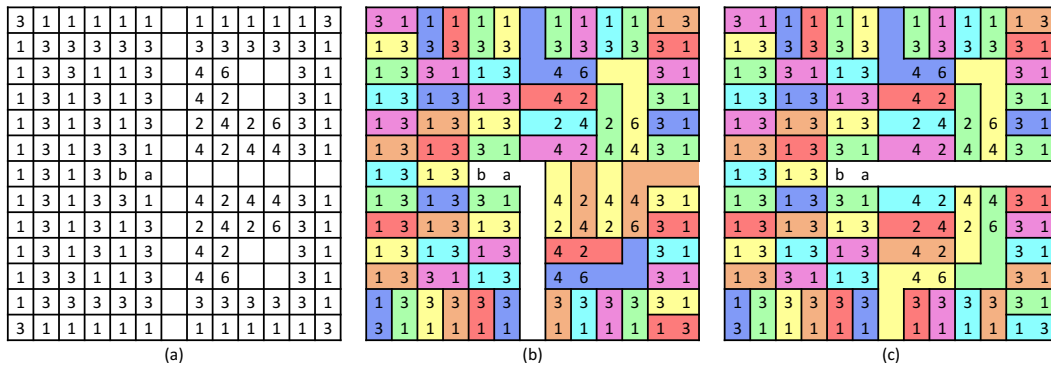
We thus see that the remaining 9 cannot connect to the adjacent 3, because the resulting region would need at least four cells, including two central row cells. Hence the 9 must connect to the adjacent 1, forming a region that can and must use just two cells.

Finally, the 6-1 region can also only occupy two cells, because of the limited space around it. Adding this up with the other regions, 160 cells are filled, leaving nine for the Hamilton region. This can be done exactly two ways, depicted in Figure 23.

Figure 24 shows the corner degree-2 start room. We argue that a solution must have sixty-nine 3-1 regions, four 4-2 regions, and two 6-4 regions. This results in $69 \cdot 2 + 4 \cdot 3 + 2 \cdot 5 = 160$ filled cells. So the remaining nine cells must be filled by the Hamilton region.

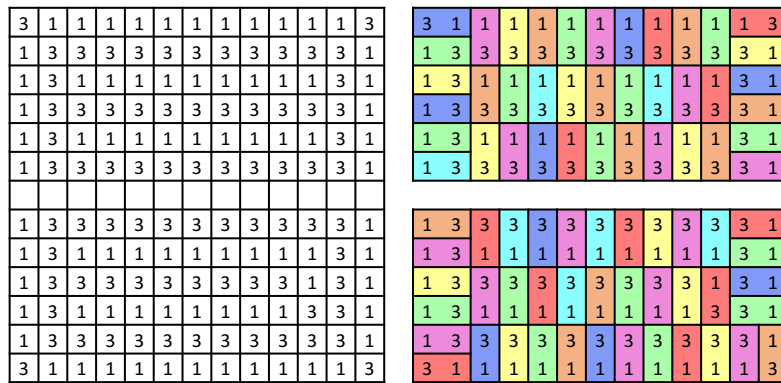
Here, the 6s must connect to the adjacent 4s because those 4s have no alternative. Further, the b cannot connect to a 3 as argued previously, so it must connect to a and form the Hamilton region. The 1 to the right of a cannot make a 1-4 region, as that blocks the east hallway while also forcing a 2-4 region above to block the north hallway. This leaves two, somewhat symmetrical, ways to solve the room, as seen in Figure 24.

Figure 25 shows the degree-3 start room, constructed similarly to the corner degree-2 start room but with a southeast quadrant mirroring the northeast. Forced in its solution are fifty-eight 3-1 regions, eight 4-2 regions, and four 6-4 regions. This results in $58 \cdot 2 + 8 \cdot 3 + 4 \cdot 5 = 160$ filled cells, leaving nine for the Hamilton region. In each of the two eastern quadrants, the numbers' regions can together fill either a 7×6 or 6×7 rectangular area, resulting in three ways to solve the room. Two are shown in the figure. The third, with Hamilton region heading north, mirrors the first.

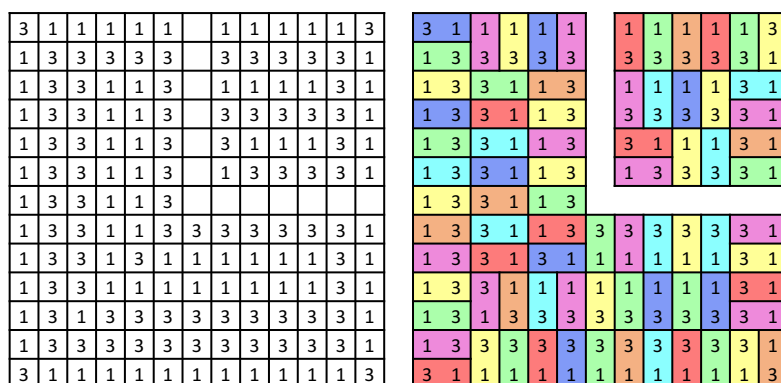


■ **Figure 25** An *Araf* degree-3 start room (left) and two of three solutions.

We now present the nonterminal rooms. As degree-1 nonterminal rooms are unnecessary, we begin in Figures 26 and 27 with the degree-2 nonterminal rooms, which require no comment.



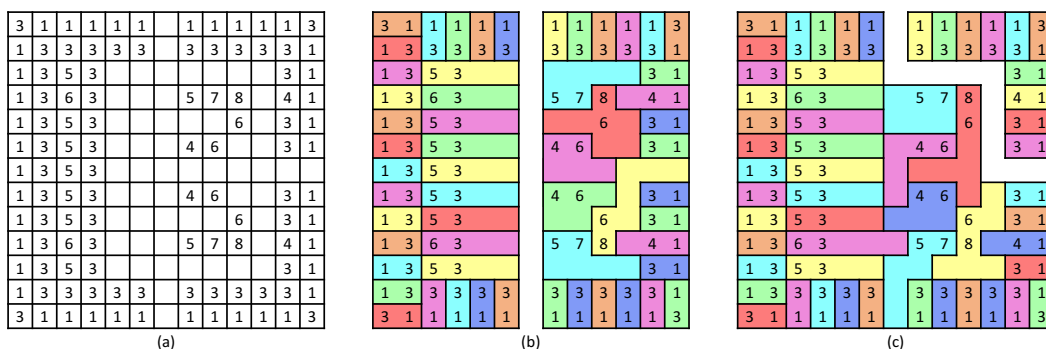
■ **Figure 26** An *Araf* degree-2 straight nonterminal room (left) and its solution.



■ **Figure 27** An *Araf* degree-2 corner nonterminal room (left) and its solution.

The degree 3 nonterminal room, shown in Figure 28, is more complex. We claim it has thirty-nine 3-1 regions, two 4-1 regions, seven 5-3 regions, two 6-3 regions, two 7-5 regions, two 8-6 regions and two 6-4 regions. Two solutions are shown, and a third symmetrical to the second is also possible.

11:18 Hamiltonian Paths and Cycles in NP-Complete Puzzles

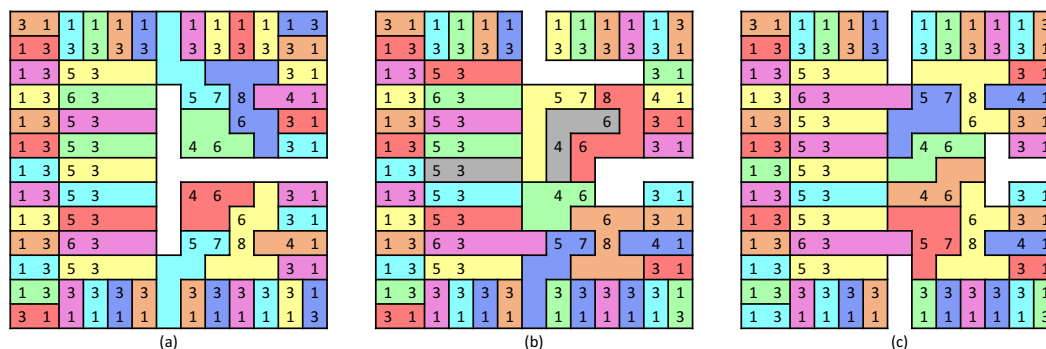


■ **Figure 28** An *Araf* degree-3 nonterminal room (left) and two of the solutions.

Note that all cells in the west side will always be filled. In the east is some flexibility. However, observe that all 1s must connect to the adjacent 3s and 4s. This leaves two areas that each contain numbers 4, 5, 6, 6, 7, and 8. In each, since a 6 can never connect to 5 or 7, the 6s must connect to the 8 and 4. So the 5 must connect to 7. This yields regions of exactly five, exactly six, and exactly seven cells.

The two 6-3 regions can fill either four or five cells and the two 4-1 regions can fill two or three cells. This means that the total area covered by regions of numbers in the room is between 154 and 158 cells, leaving between eleven and fifteen for the Hamilton region.

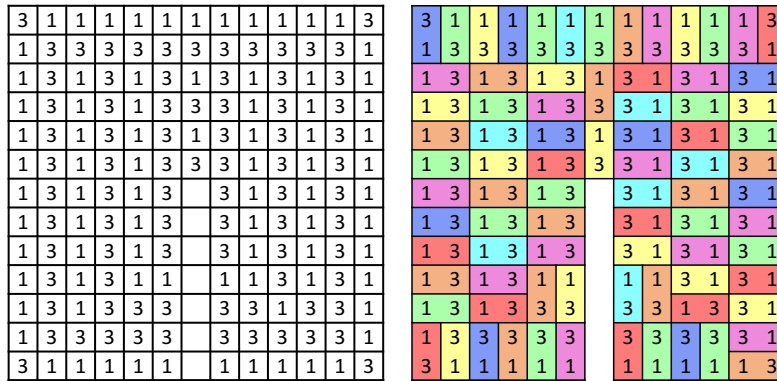
The images in Figure 28 show the intended ways to solve this room, but unlike the previous rooms, this one offers freedom for the solver to place regions in other ways that are at least locally acceptable within the rules. See Figure 29. We argue, however, that these lead to global failure for solving the puzzle instance as a whole, or at any rate do not render solvable any puzzle instances whose corresponding graphs have no Hamiltonian paths.



■ **Figure 29** Alternative solution attempts for *Araf* degree-3 nonterminal room.

First we note that since eleven or more cells are unfilled by regions of numbers in the room itself, the Hamilton region must visit there. The only question is whether/how it exits. But failing to exit, as in image (a), leaves the Hamilton region with at least three endpoints: the start room, the end room, and this room. The only way for this to occur is if there is some room (with degree 3) elsewhere whose three hallways are all used. But this is impossible, as it would require nineteen cells for the Hamilton region (the normal thirteen plus six more), which exceeds the maximum (fifteen) established above. Solution attempts such as (b) and (c) can be similarly disregarded.

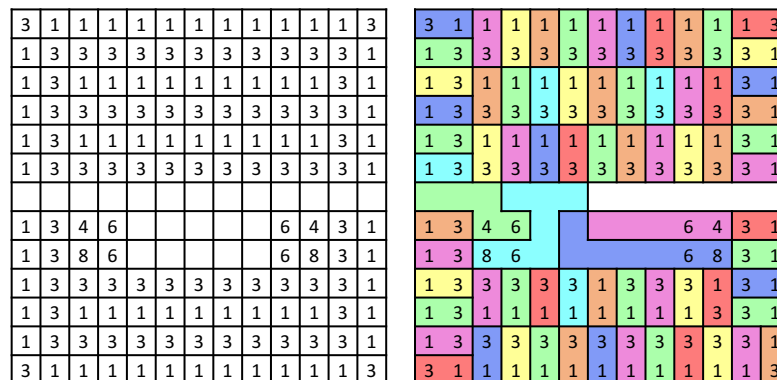
Having now handled all start and nonterminal rooms, we finally turn to end rooms. All of these except for the straight degree-2 room are constructed identically to their corresponding start rooms, but with a 3-1 or 1-3 domino in place of a and b . Their solutions are the same as the start rooms', except with the two cells formerly occupied by a and b now enclosed within a two-cell region, leaving seven (not nine) cells for occupation by the Hamilton region. We show just one example (Figure 30).



■ **Figure 30** An *Araf* degree-1 end room (left) and its solution.

For the case of the straight degree-2 end room, Figure 31 shows its layout and one of its two solutions. The other is symmetrical. We claim that the room contains sixty-nine 3-1 regions, two 8-6 regions and two 6-4 regions. This results in $69 \cdot 2 + 2 \cdot 7 + 2 \cdot 5 = 162$ filled cells, with the remaining seven cells being filled by the Hamilton region.

Observe that each 6 adjacent to an 8 must connect to the 8, because the only alternative is to connect to a 3, which would leave a 1 optionless. The other 6s must then connect to their neighboring 4s, since the 4s have no other option.

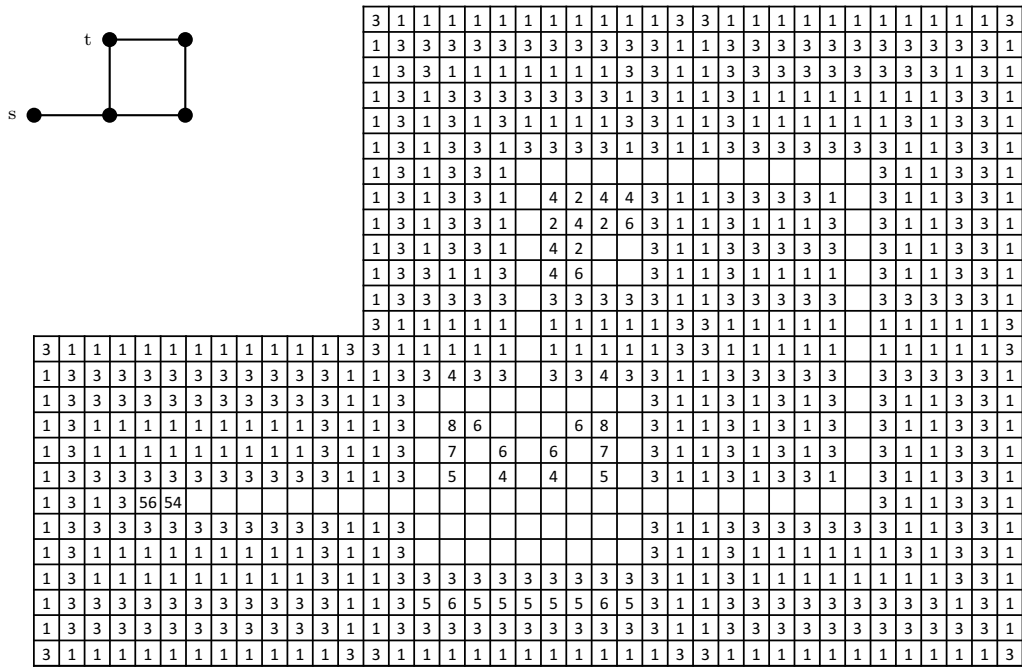


■ **Figure 31** An *Araf* degree-2 end room (left) and one of two solutions.

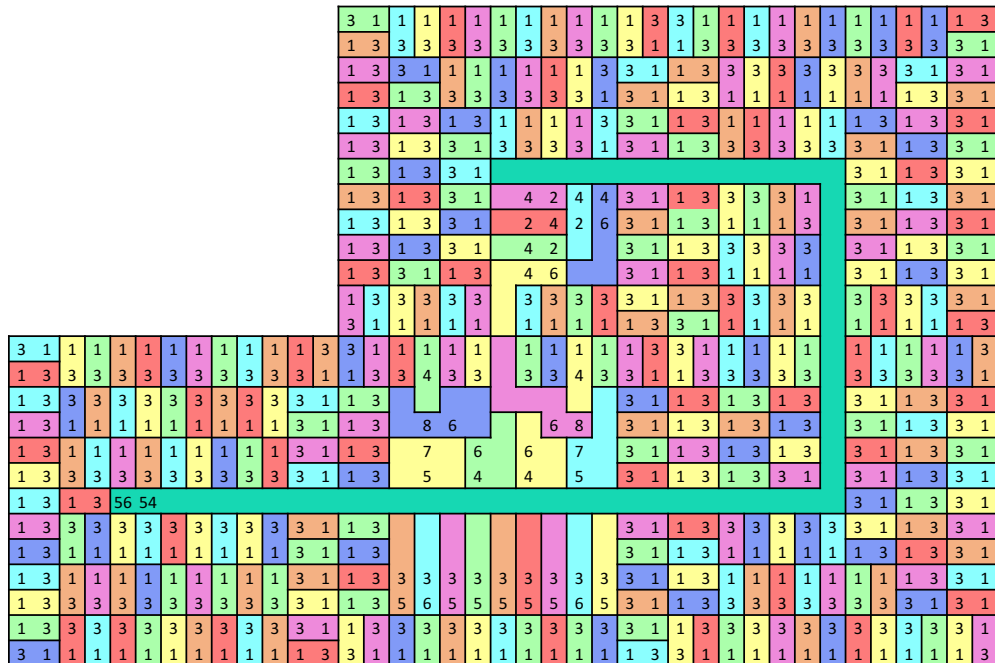
Having covered all the gadgets, we conclude the proof. In Example 7 we show a full example of an *Araf* puzzle instance that simulates a grid graph. ◀

▶ **Example 7.** The graph in Figure 32 has five nodes, so we have $a = 5(13) - 11 = 54$ and $b = a + 2 = 56$. Figure 33 gives the puzzle instance's solution.

11:20 Hamiltonian Paths and Cycles in NP-Complete Puzzles



■ Figure 32 Example graph and its corresponding *Araf* instance.



■ Figure 33 *Araf* example solution with 56-54 region representing a Hamiltonian path.

7 *Dotchi Loop, revisited*

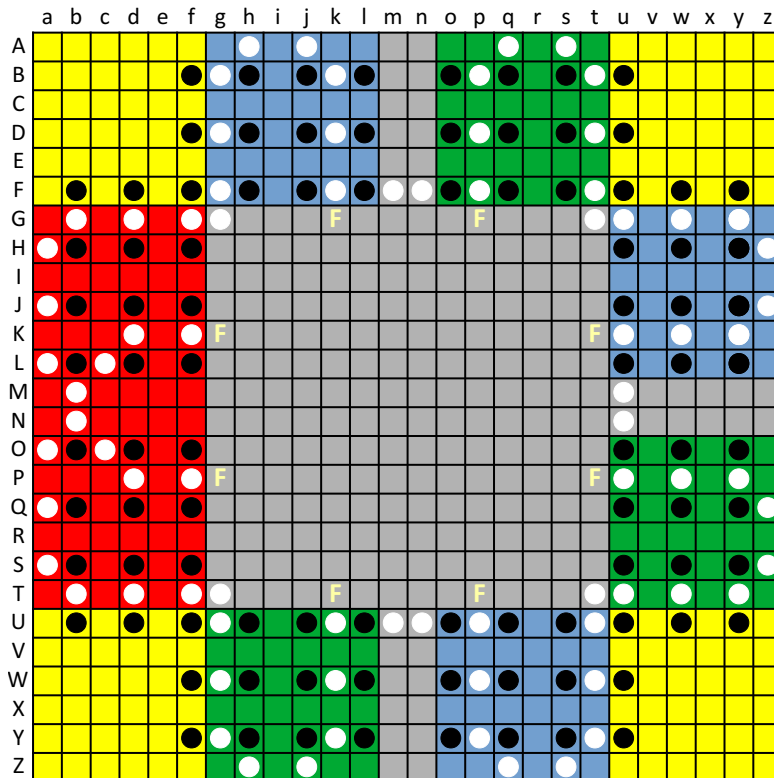
We now come full circle to our first puzzle, *Dotchi Loop*. In this section we show that *Dotchi Loop* remains NP-complete even when the set of available puzzle instances is restricted so that black circles are forbidden from being orthogonally or diagonally adjacent. Let us call the solvability problem for *Dotchi Loop* under these conditions *Restricted Dotchi Loop*. It is easy to see that the formation of walls in *Restricted Dotchi Loop* is much more difficult than in unrestricted *Dotchi Loop* (Section 2).

► **Theorem 7.** *Restricted Dotchi Loop is NP-complete.*

Proof. We again give a reduction from HC3G. Although the formation of walls in the manner of our earlier proof is difficult or impossible, the key idea in our construction is now the use of *the loop itself* to form a sort of wall. Here we make use of the fact that the loop is forbidden to cross itself. By forcing the loop to surround each room, we give it no opportunity to wander astray.

The degree-3 room is shown in Figure 34. Here, the letter F is not part of the gadget, but just a label for our reference (see below). Degree-2 rooms (not shown) are constructed similarly, but with changes we now describe.

All rooms have 6×6 regions in the corners with black circles arranged as in the figure. The **Gg-Gt-Tg-Tt** square is also the same as in the figure. On a side with no exit, between the two 6×6 regions in the corners is a rectangle like the **Ga-Gf-Ta-Tf** rectangle shown, rotated appropriately. In a side with an exit, there is a rectangle resembling the **Gu-Gz-Tu-Tz** rectangle shown, rotated appropriately. For example, the degree-2 corner room with hallways north and east is identical to the degree-3 room depicted, except with its **Ug-Ut-Zg-Zt** rectangle replaced by a left-rotated copy of the **Ga-Gf-Ta-Tf** rectangle.



■ **Figure 34** *Restricted Dotchi Loop* degree-3 room. There are twelve distinct regions. As in Figure 5, colors distinguish neighboring regions, but distant regions may be shown with similar color.

We now see how the loop is forced to form its own self-impenetrable wall around the room, save for an opening at each of exactly two of the 2×6 areas mentioned earlier. These areas are the “corridors” of this reduction and each one joins with one from a neighboring room (as they are aligned at the center). Accordingly, we now see that the rooms work as they need to. ◀

8 Discussion and Future Research

8.1 Application of HP3G and HC3G

We have seen how HP3G and HC3G are useful graph-theoretic computational problems for giving reductions to some pen-and-paper puzzles, particularly those occurring on square grids with a kind of path- or loop-finding aspect. But even *Araf* was successfully proven NP-complete using one of these, despite the lack of any overt requirement to construct a path or loop in the puzzle’s rules. Therefore, HP3G and HC3G seem to be at least moderately broadly applicable problems that should be taken seriously as candidates for future such reductions – certainly, at least, when proofs using common choices such as 3SAT are not forthcoming.

8.2 Restricted puzzle versions

In general, we find it interesting to consider whether a given puzzle remains NP-complete when the set of available puzzle instances is restricted in some way. We explored this in Section 7 for *Dotchi Loop*. In deliberately making the task difficult by eliminating the means to create easy walls, we became forced to find an alternative approach using the loop’s path itself. Similar such results may be available for the other puzzles. For example, we may consider Linesweeper without the use of 0-cells (despite footnote 8). We suspect that HP3G and HC3G can still serve as bases for reductions to these restricted puzzles, but the task gets more challenging the more tools we deprive ourselves of using to (for example) build walls.

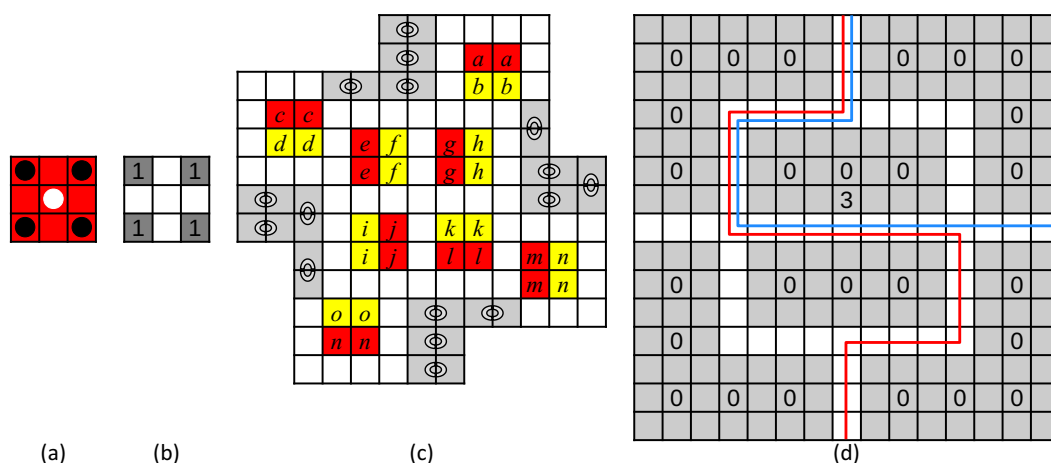
Restricted versions of puzzles compare to the original versions in a way analogous to how 3SAT compares to SAT, or HP3G and HCG3 respectively compare to the Hamiltonian path and cycle problems on arbitrary graphs. As potential bases for reductions in future NP-completeness proofs, they require the construction of fewer gadgets than the originals. But even in the likely scenario that no future NP-completeness proofs use these particular puzzles as bases for reductions, we still find the possibility of nontrivially restricted versions being NP-complete inherently interesting.

8.3 HP4G and HC4G

As a side remark, we are grateful for the existence of Theorem 1 and Corollary 1, which have allowed us to give reductions without needing to construct degree-4 rooms. Had we been forced to construct such rooms in addition to the others, our task would have been rather harder. While a degree-4 room is in some cases very simple to construct after seeing the lower-degree rooms, in other cases their design is not so trivial. *Dotchi Loop*, *Chains*, and *Arukone₃* are easy cases (see Figure 36(a)-(c)). Linesweeper is more challenging, but could be done as in Figure 36(d), if the lower-degree rooms are appropriately padded with 0s to match the dimensions. But *Araf* and Restricted *Dotchi Loop* bear no constructions we have been able to find. For *Araf*, the lower-degree rooms do not fit a clear pattern that can be straightforwardly extrapolated to degree 4, and overall we simply have not found any design that works. For Restricted *Dotchi Loop*, such a pattern very clearly does exist, but a naive

11:24 Hamiltonian Paths and Cycles in NP-Complete Puzzles

attempt to design degree-4 rooms this way allows a loop to enter and exit the same degree-4 room twice, resulting in some non-Hamiltonian graphs getting mapped to solvable puzzle instances. (By contrast, the rooms in Figure 36 all prevent this.) Although these last two puzzles are provably NP-complete anyway, and degree-4 rooms are of course possible to build “the long way” by reducing $\text{HP4G} \rightarrow \text{HP3G} \rightarrow \text{Araf}$ and $\text{HC4G} \rightarrow \text{HC3G} \rightarrow \text{Restricted DL}$, we wonder as a matter of pure curiosity whether these puzzles admit “direct” degree-4 room designs – i.e., ones as small and natural as those of the lower-degree rooms.




■ **Figure 36** Degree-4 rooms of *Dotchi Loop*, *Chains*, *Arukone₃*, and *Linesweeper*. Rooms (b) and (c) can be made terminal by inserting an appropriate number in the center. Room (d) is given with two of six solutions shown.

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. *Journal of Information Processing*, 23(3):239–245, 2015. doi:10.2197/IPSJJIP.23.239.
- 2 Aviv Adler, Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu, and Jayson Lynch. Tatamibari is NP-complete. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms (FUN 2021)*, volume 157 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FUN.2021.1.
- 3 Michael Buro. Simple Amazons endgames and their connection to Hamilton circuits in cubic subgrid graphs. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games, Second International Conference, CG 2000, Hamamatsu, Japan, October 26-28, 2000, Revised Papers*, volume 2063 of *Lecture Notes in Computer Science*, pages 250–261. Springer, 2001. doi:10.1007/3-540-45579-5_17.
- 4 Erik D. Demaine, Jayson Lynch, Mikhail Rudoy, and Yushi Uno. Yin-Yang Puzzles are NP-complete. In Meng He and Don Sheehy, editors, *Proceedings of the 33rd Canadian Conference on Computational Geometry, CCCG 2021, August 10-12, 2021, Dalhousie University, Halifax, Nova Scotia, Canada*, pages 97–106, 2021.
- 5 Marnix Deurloo. On the NP-completeness of Dotchi Loop and restricted sets thereof. Bachelor’s thesis, Utrecht University, 2023.
- 6 Mitchell Donkers. The NP-completeness of pen and paper puzzles. Bachelor’s thesis, Utrecht University, 2021. URL: <https://studenttheses.uu.nl/handle/20.500.12932/1383>.

- 7 Markus Holzer and Oliver Ruepp. The troubles of interior design—A complexity analysis of the game Heyawake. 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 198–212. Springer, 2007. doi:10.1007/978-3-540-72914-3_18.
- 8 Otto Janko and Angela Janko. Araf. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Araf/index.htm>.
- 9 Otto Janko and Angela Janko. Arukone. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Arukone/index.htm>.
- 10 Otto Janko and Angela Janko. Arukone². Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Arukone-2/index.htm>.
- 11 Otto Janko and Angela Janko. Arukone³. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Arukone-3/index.htm>.
- 12 Otto Janko and Angela Janko. Dotchi loop. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Dotchi-Loop/index.htm>.
- 13 Otto Janko and Angela Janko. Ketten. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Ketten/index.htm>.
- 14 Otto Janko and Angela Janko. Ketten, nr. 7. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Ketten/007.a.htm>.
- 15 Otto Janko and Angela Janko. Linesweeper. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Linesweeper/index.htm>.
- 16 Otto Janko and Angela Janko. Linesweeper, nr. 082. Accessed on 2024-02-20. URL: <https://www.janko.at/Raetsel/Linesweeper/082.a.htm>.
- 17 Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22:9–15, 2000. doi:10.1007/BF03025367.
- 18 Jonas Kölker. Kurodoko is NP-complete. *Journal of Information Processing*, 20(3):694–706, 2012. doi:10.2197/ipsjjip.20.694.
- 19 Kotsuma Kouichi and Takenaga Yasuhiko. NP-completeness and enumeration of Number Link puzzle. *IEICE Technical Report. Theoretical Foundations of Computing*, 109(465):1–7, 2010.
- 20 Mieke Maarse. The NP-completeness of some lesser known logic puzzles. Bachelor’s thesis, Utrecht University, 2019. URL: <https://studenttheses.uu.nl/handle/20.500.12932/33867>.
- 21 Jak Marshall. Play my puzzle game: Linesweeper. September 30, 2010. URL: <https://103percent.blogspot.com/2010/09/play-my-new-puzzle-game.html>.
- 22 Christos H. Papadimitriou and Umesh V. Vazirani. On two geometric problems related to the travelling salesman problem. *Journal of Algorithms*, 5(2):231–246, June 1984. doi:10.1016/0196-6774(84)90029-4.
- 23 Karen Schutte. The NP-completeness of three logic puzzles. Bachelor’s thesis, Utrecht University, 2021.
- 24 Allan Scott, Ulrike Stege, and Iris Rooij. Minesweeper may not be NP-complete but is hard nonetheless. *The Mathematical Intelligencer*, 33.4:5–17, 2011. doi:10.1007/s00283-011-9256-x.
- 25 Hadyn Tang. A framework for loop and path puzzle satisfiability NP-hardness results, 2022. arXiv:2202.02046.
- 26 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-A(5):1052–1060, 2003. URL: http://search.ieice.org/bin/summary.php?id=e86-a_5_1052.

Card-Based Cryptography Meets Differential Privacy



Reo Eriguchi  

National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Kazumasa Shinagawa  

Ibaraki University, Japan

National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Takao Murakami  

The Institute of Statistical Mathematics, Tachikawa, Japan

National Institute of Advanced Industrial Science and Technology, Tokyo, Japan

Abstract

Card-based cryptography studies the problem of implementing cryptographic algorithms in a visual way using physical cards to demonstrate their security properties for those who are unfamiliar with cryptography. In this paper, we initiate the study of card-based implementations of differentially private mechanisms, which are a standard privacy-enhancing technique to publish statistics of databases while protecting the privacy of any particular individual. We start with giving the definition of differential privacy of card-based protocols. As a feasibility result, we present three kinds of protocols using standard binary cards for computing the sum of parties' binary inputs, $f(x_1, \dots, x_n) = \sum_{i=1}^n x_i$ for $x_i \in \{0, 1\}$, under differential privacy. Our first protocol follows the framework of output perturbation, which provides differential privacy by adding noise to exact aggregation results. The protocol needs only two shuffles, and the overheads in the number of cards and the error bound are independent of the number n of parties. Our second and third protocols are based on Randomized Response, which adds noise to each input before aggregation. Compared to the first protocol, they improve the overheads in the number of cards and the error bound in terms of differential privacy parameters at the cost of incurring a multiplicative factor of n . To address a technical challenge of generating non-uniform noise using a finite number of cards, we propose a novel differentially private mechanism based on the hypergeometric distribution, which we believe may be of independent interest beyond applications to card-based cryptography.

2012 ACM Subject Classification Security and privacy → Information-theoretic techniques

Keywords and phrases Card-based cryptography, Differential privacy, Secure computation

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.12

Funding *Reo Eriguchi*: This work was supported in part by JST CREST Grant Number MJCR22M1 and JST AIP Acceleration Research JPMJCR22U5.

Kazumasa Shinagawa: This work was supported in part by JSPS KAKENHI 21K17702 and 23H00479, and JST CREST Grant Number MJCR22M1.

Takao Murakami: This work was supported in part by JSPS KAKENHI JP22H00521.

1 Introduction

With the rapid development of cryptography, various kinds of cryptographic primitives have been proposed and allowed secure data processing on sensitive data. However, most of these primitives are supposed to be implemented by computers and, as such, often lead to complicated algorithm design. As a result, there remains a gap in non-experts' understanding of the security properties, which may prevent active social implementations.



© Reo Eriguchi, Kazumasa Shinagawa, and Takao Murakami; licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 12; pp. 12:1–12:20

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

To address this problem, *card-based cryptography* [7, 6] studies the problem of implementing cryptographic algorithms in a visual way using physical cards and demonstrates their security properties for those who are unfamiliar with cryptography. So far, many card-based cryptographic protocols have been proposed to implement secure multiparty computation (e.g., [6, 16, 19, 22]) and zero-knowledge proofs [18, 11].

Recently, the concept of *differential privacy* [10] has been attracting a lot of attention as the gold standard for rigorous privacy guarantees. Differential privacy is a mathematical concept introduced in [8, 9] to quantify the privacy loss associated with any publication of statistics of databases. For example, consider the simplest task of computing the sum of n parties' private inputs. If the exact aggregation result is published, an adversary colluding with $n - 1$ parties can deduce the input of the remaining party from the result, which in principle cannot be prevented only by secure computation techniques. Differentially private mechanisms make results untraceable back to individuals by perturbing them with the addition of noise. Due to its strong privacy and robustness guarantees, many differentially private mechanisms have been proposed and deployed in privacy-preserving data analysis of, e.g., users' location information [2, 26] and social network data [21].

We note that differentially private mechanisms were previously supposed to be implemented using computers in the literature. This seems to be in part because the mechanisms usually need complicated processes to generate noise drawn from non-uniform probability distributions (e.g., Laplace or Bernoulli distribution [10]). Towards the further deployment of privacy-enhancing techniques, it is important to demonstrate differentially private mechanisms in an easier-to-understand way. However, the problem of implementing differentially private mechanisms using cards has never been considered prior to this work.

1.1 Our Results

In this paper, we initiate the study of card-based implementations of differentially private mechanisms. We start with giving the definition of differential privacy of card-based protocols. Our definition is inspired by the framework of [3] defining differential privacy of (non-card-based) distributed protocols. As a feasibility result, we present three kinds of protocols using standard binary cards for computing the sum of parties' binary inputs, $f(x_1, \dots, x_n) = \sum_{i=1}^n x_i$ for $x_i \in \{0, 1\}$, under differential privacy. Computing the binary sum function is one of the most fundamental problems within the context of differential privacy [3, 4, 10, 24].

Our first protocol is based on output perturbation, which provides differential privacy by adding noise to exact aggregation results. The protocol needs only two shuffles, and the overheads in the number of cards and a bound on the mean squared error (MSE) are independent of the number n of inputs. A technical challenge is how to generate non-uniform noise using a finite number of cards. Along the way, we propose a novel differentially private mechanism based on the hypergeometric distribution, which we believe may be of independent interest beyond applications to card-based cryptography.

Our second and third protocols are based on input perturbation, which adds noise to each input before aggregation. Compared to the above protocol, they improve the overheads in the number of cards and the error bound in terms of differential privacy parameters at the cost of incurring a multiplicative factor of n . Our third protocol even reduces the number of shuffles to one. While not apparent in asymptotic notations, we empirically show that the second protocol ensures a smaller number of cards and a smaller MSE in concrete parameter settings. The detailed comparison is shown in Table 1 and Section 6.

We note that our first protocol can be described in the traditional model of card-based protocols introduced in [16]. On the other hand, our second and third protocols assume that parties apply private reveals to cards, which is not allowed in the model of [16]. While

■ **Table 1** Comparison of our card-based protocols.

Protocol	MSE	# cards	# shuffles
$\Pi_{k,\ell}^{\text{HG}}$ (Section 4)	$O(\epsilon^{-4} \ln \delta^{-1})$	$n + O(\epsilon^{-5} \ln \delta^{-1})$	2
$\Pi_{k,\ell}^{\text{RR}}$ (Section 5.1)	$O(\epsilon^{-2}n)$	$O(\epsilon^{-1}n)$	n
$\Pi_{k,\ell}^{\text{RR}'}$ (Section 5.2)	$O(\epsilon^{-2}n)$	$O(\epsilon^{-1}n)$	1

ϵ, δ denote differential privacy parameters and n denotes the number of parties (see Section 2 for the definition). We show the asymptotic performance when ϵ tends to 0 and use the approximation $e^\epsilon \approx 1 + \epsilon + \epsilon^2/2$.

such private operations could be easily implemented in practice and are assumed by a prior work [13], constructing protocols without any private operations has been considered as theoretically important in the literature. For that, we show that private operations in our second and third protocols can be removed at the cost of doubling the number of cards and requiring n more shuffles.

1.2 Overview of Our Techniques

We here provide an overview of our protocols. More detailed descriptions and security proofs are given in the following sections.

Our protocols assume standard binary cards with \heartsuit and \clubsuit . A framework to guarantee differential privacy is roughly categorized into output perturbation and input perturbation: The former first computes an exact result privately and then perturbs it with the addition of noise; The latter first perturbs private inputs and then computes a target function on the noisy values.

1.2.1 Our Protocol Based on the Hypergeometric Distribution

Our first protocol is based on output perturbation. The private computation of the sum $f(\mathbf{x}) = \sum_{i \in [n]} x_i$ of binary inputs is straightforward: If parties submit face-down cards following the encoding rule $\heartsuit = 1, \clubsuit = 0$, then the number of \heartsuit s in a resulting sequence of cards is equal to $f(\mathbf{x})$. A main technical challenge is thus how to implement the addition of noise providing differential privacy using cards. One of the most common choices for the noise distribution is the binomial distribution $\text{Bin}(k, 1/2)$ with the number of trials k and the success probability $1/2$ [1]. A naïve card-based implementation of binomial distributions would be that for each $i = 1, 2, \dots, k$, parties uniformly permutes a pair of two cards with \heartsuit and \clubsuit , and adds one of them to the above sequence in a face-down manner. Although it indeed generates binomial samples, this naïve implementation needs a large number of shuffle operations proportional to k . A state-of-the-art analysis [1] shows that k should be chosen as $k = \Omega(\epsilon^{-2} \ln \delta^{-1})$ to guarantee (ϵ, δ) -differential privacy. Concretely, k should be larger than 2000 for $\epsilon = 0.5$ and $\delta = 10^{-6}$, and even larger than 20000 for $\epsilon = 0.1$ and $\delta = 10^{-6}$.

To reduce the number of shuffles, we prepare a supplementary sequence of randomly shuffled cards containing equal numbers of \heartsuit s and \clubsuit s, and choose k cards from it without replacement. Intuitively, if the number of \heartsuit s in the sequence is sufficiently larger than k , then the number of \heartsuit s in the k draws approximately follows the binomial distribution $\text{Bin}(k, 1/2)$. Since we sample cards without replacement, our method requires only a single shuffle to prepare the supplementary sequence for generating noise. It is important to note that the number of \heartsuit s in the sequence is actually a finite value. A technical challenge is thus that the noise does not exactly follow the binomial distribution but follows the *hypergeometric distribution*, which precisely describes the distribution of the number of \heartsuit s in

k draws without replacement from a sequence of cards containing equal numbers of \heartsuit s and \clubsuit s¹. We present for the first time the differential privacy guarantee of a mechanism adding noise drawn from the hypergeometric distribution, and also present a utility guarantee in terms of the mean squared error (MSE). We believe that differentially private mechanisms based on the hypergeometric distribution may be of independent interest beyond applications to card-based cryptography.

1.2.2 Our Protocols Based on Randomized Response

Our second and third protocols are based on input perturbation. Specifically, we focus on a traditional mechanism called *Randomized Response* [10, 24], which guarantees differential privacy by having parties flip their input bits with a probability $p = 1/(e^\epsilon + 1)$. A technical challenge here is how to implement biased coins using cards.

Our first realization is a direct implementation of the above procedure: We prepare n supplementary sequences of ℓ cards each consisting of randomly permuted k \heartsuit s and $\ell - k$ \clubsuit s such that $p \approx k/\ell$ and let the i -th party privately open a card in the i -th sequence and flip his input if and only if he draws \heartsuit . In Section 5.1, we carefully analyze the impact of the finite approximation of the probability p on differential privacy, which is not a straightforward problem as there are known attacks on naïve implementations of algorithms assuming real arithmetic [14].

A possible drawback of the above implementation is that the number of shuffle operations grows linearly in the number n of parties since n supplementary sequences should be independently prepared. To reduce the number of shuffles, we propose an alternative implementation: We prepare *one* supplementary sequence consisting of randomly permuted k \heartsuit s and $\ell - k$ \clubsuit s such that $p \approx k/\ell$ and let the i -th party privately open the i -th card in the sequence and flip his input if and only if he draws \heartsuit . This method allows us to prepare the supplementary sequence with only a single shuffle. On the other hand, a more careful analysis of privacy and utility is necessary since the states of cards drawn by parties are no more independent. Note that this kind of challenge has not been encountered in the prior computer-based implementations of Randomized Response or its variants [24, 10, 23] where parties can locally generate independent randomness.

Finally, both of the above implementations require parties to apply private reveals to cards. While such private operations could be easily implemented in practice, it has also been considered as theoretically important to construct protocols without any private operations (i.e., those following the traditional model of card-based protocols [16]). We also show that private operations can be removed by emulating the local computations done by parties with card-based secure computation protocols. In the above implementations, parties need to privately compute the XOR of their inputs and the states of cards drawn from supplementary sequences. We emulate these computations with an efficient card-based XOR protocol without any private operations [17]. As a result, we obtain variant protocols removing private operations at the cost of doubling the number of cards and requiring n more shuffles. Note that our first protocol based on the hypergeometric distribution can be described following the model of [16] as it assumes no private operation.

¹ The hypergeometric distribution can be defined in a more general setting where a sequence contains different numbers of \heartsuit s and \clubsuit s.

2 Preliminaries

Notations. For $n \in \mathbb{N}$, define $[n] = \{i \in \mathbb{Z} : 1 \leq i \leq n\}$. If a random variable z follows a probability distribution \mathcal{D} , we write $z \leftarrow \mathcal{D}$. Let $\ln x$ denote the base-e logarithm of x , where e is the Napiers constant.

2.1 Card-based Protocols

Card. In this paper, we use *binary cards* whose front sides are either \clubsuit or \heartsuit and back sides are both $?$. We assume that two cards with the same symbol are indistinguishable. We use the encoding $\clubsuit = 0$ and $\heartsuit = 1$ throughout the paper except in Section 7.

Shuffle. A *shuffle* is an operation that applies a random permutation to a sequence of face-down cards, where the permutation is chosen by some probability distribution. It is assumed that no party guesses which permutation is chosen from the shuffle.

A *complete shuffle* is a shuffle that applies a uniformly random permutation to a sequence of face-down cards, which is denoted by $[\cdot]$. For example, a complete shuffle for a sequence of three cards results in one of the six sequences each with probability $1/6$ as follows:

$$\left[\begin{array}{ccc} 1 & 2 & 3 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \rightarrow \left[\begin{array}{ccc} 1 & 2 & 3 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \text{ or } \left[\begin{array}{ccc} 1 & 3 & 2 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \text{ or } \left[\begin{array}{ccc} 2 & 3 & 1 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \text{ or } \left[\begin{array}{ccc} 2 & 1 & 3 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \text{ or } \left[\begin{array}{ccc} 3 & 1 & 2 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \text{ or } \left[\begin{array}{ccc} 3 & 2 & 1 \\ \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right].$$

A *pile-scramble shuffle* [12] is a shuffle that divides a sequence of cards into multiple *piles* of the same number of cards and applies a random permutation to a sequence of piles, which is denoted by $[\cdot | \cdot \cdots \cdot]$. For example, a pile-scramble shuffle for a sequence of three piles each having two cards results in one of the six sequences each with probability $1/6$ as follows:

$$\left[\begin{array}{cc|cc|cc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \right] \rightarrow \left\{ \begin{array}{l} \begin{array}{ccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \\ \begin{array}{ccc} 1 & 2 & 5 & 6 & 3 & 4 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \\ \begin{array}{ccc} 3 & 4 & 5 & 6 & 1 & 2 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \\ \begin{array}{ccc} 3 & 4 & 1 & 2 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \\ \begin{array}{ccc} 5 & 6 & 1 & 2 & 3 & 4 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \\ \begin{array}{ccc} 5 & 6 & 3 & 4 & 1 & 2 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \end{array} \right.$$

Protocol. Suppose that there are n parties each having an input $x_i \in D$, where the input domain D is a finite set. A card-based protocol consists of three phases: the setup phase, the computation phase, and the output phase. In the setup phase, *supplementary cards* are prepared. Using shuffles, they are drawn from a probability distribution independent of parties' inputs. Here, the front sides of them are hidden from all parties. In the computation phase, parties repeat one of the following operations:

- **Input:** Each party submits a face-down card according to his/her input. They are called *main cards*. If necessary, it is allowed to perform *private reveals* for a subset of the supplementary cards (e.g., [13, 20, 25]), where a designated party privately reads the symbol of a face-down card.
- **Shuffle:** A random permutation is applied to the current sequence of cards consisting of the main cards and the supplementary cards. It is assumed that no party guesses which permutation is chosen from the shuffle.
- **Insertion:** Some of the supplementary cards are inserted to the main cards.

In the output phase, parties open all cards in the current sequence and determine the output value. Note that if parties do not perform private reveals, then a protocol can be described in the traditional model of card-based protocols [16].

We evaluate the space complexity of a card-based protocol Π by the total number of cards used to execute Π , which we denote by $\#\text{Card}(\Pi)$. We also evaluate the computational complexity of Π by the total number of shuffle operations since shuffling is the most costly operation in practice [15]. We denote it by $\#\text{Shuffle}(\Pi)$.

2.2 Differential Privacy

Following the terminology in [3], we say that two n -dimensional vectors $\mathbf{x} = (x_i)_{i \in [n]}$, $\mathbf{x}' = (x'_i)_{i \in [n]}$ are T -neighboring for a subset $T \subseteq [n]$ if $x_i = x'_i$ for any $i \in T$ and $x_i \neq x'_i$ for at most one $i \notin T$. If \mathbf{x} and \mathbf{x}' are \emptyset -neighboring, we simply say that they are neighboring. For a finite set D , we define the *sensitivity* of a function $f : D^n \rightarrow \mathbb{Z}$ as

$$\Delta = \max_{\substack{\mathbf{x}, \mathbf{x}' \in D^n: \\ \text{neighboring}}} |f(\mathbf{x}) - f(\mathbf{x}')|.$$

We say that two probability distributions $\mathcal{D}_1, \mathcal{D}_2$ over a set U are (ϵ, δ) -DP close if for any subset $S \subseteq U$, it holds that $\Pr[y \leftarrow \mathcal{D}_1 : y \in S] \leq e^\epsilon \cdot \Pr[y \leftarrow \mathcal{D}_2 : y \in S] + \delta$.

3 Differentially Private Card-based Protocols

We start with giving the definition of differential privacy of card-based protocols. Our definition is inspired by the framework of [3] defining differential privacy of (non-card-based) distributed protocols. In this paper, we consider an adversary corrupting a set T of at most $n - 1$ parties. We assume that the adversary is *semi-honest*, that is, she tries to learn information from her view during the protocol but does not deviate from the protocol specifications. Let $\text{View}_{\Pi, T}(\mathbf{x})$ denote the view of the adversary during the execution of a card-based protocol Π on input $\mathbf{x} = (x_1, \dots, x_n)$, which consists of the inputs of the corrupted parties and the information (e.g., the states of cards) that they can learn during the execution of Π .

► **Definition 1.** *Let ϵ, δ be non-negative numbers. We say that a card-based protocol Π is (ϵ, δ) -differentially private if for any set T of at most $n - 1$ parties and any pair $(\mathbf{x}, \mathbf{x}')$ of T -neighboring vectors, $\text{View}_{\Pi, T}(\mathbf{x})$ and $\text{View}_{\Pi, T}(\mathbf{x}')$ are (ϵ, δ) -DP close.*

We evaluate the utility of a protocol Π with respect to a function $f : D^n \rightarrow \mathbb{Z}$ by its mean squared error (MSE) defined as

$$\text{MSE}_f(\Pi) = \max_{\mathbf{x} \in D^n} \mathbb{E} \left[|\Pi(\mathbf{x}) - f(\mathbf{x})|^2 \right],$$

where $\Pi(\mathbf{x})$ is a random variable corresponding to the output of Π on input \mathbf{x} .

In this paper, we focus on the setting in which every party has a bit $x_i \in \{0, 1\}$ and they compute the binary sum $f(x_1, \dots, x_n) = x_1 + \dots + x_n$. Note that the sensitivity of f is $\Delta = 1$.

4 Our Protocol Based on the Hypergeometric Distribution

The hypergeometric distribution is a probability distribution of the number Z of \heartsuit s in k cards chosen uniformly at random from a sequence consisting of $m - \ell$ \clubsuit s and ℓ \heartsuit s. Formally, we define the distribution as follows:

► **Definition 2.** Let k, ℓ, m be positive integers such that $k \leq \ell$ and $k + \ell \leq m$. A random variable Z follows the hypergeometric distribution $\text{HG}(m, \ell, k)$ if its probability mass function is given by

$$\Pr[Z = z] = p_{\text{HG}}(z) = \frac{\binom{\ell}{z} \binom{m-\ell}{k-z}}{\binom{m}{k}}, \quad z = 0, 1, \dots, k.$$

First, we show that hypergeometric distributions are able to provide differential privacy.

► **Proposition 3.** Let k, ℓ be positive integers such that $k < \ell$, and α, β be real numbers such that

$$\alpha \geq \frac{\ell}{\ell - k} \text{ and } \beta > 1.$$

Let $f : D^n \rightarrow \mathbb{Z}$ be a function with sensitivity Δ . Let ϵ and δ be real numbers such that

$$\epsilon \geq \Delta \ln(\alpha\beta) \text{ and } \delta \geq \exp\left(-\frac{k}{2} \left(\frac{\beta-1}{\beta+1} - \frac{2\Delta}{k}\right)^2\right). \quad (1)$$

Define a randomized algorithm \mathcal{M} as

$$\mathcal{M}(\mathbf{x}) = f(\mathbf{x}) + z, \quad z \leftarrow \text{HG}(2\ell, \ell, k).$$

Then, for any pair $(\mathbf{x}, \mathbf{x}')$ of neighboring vectors, $\mathcal{M}(\mathbf{x})$ and $\mathcal{M}(\mathbf{x}')$ are (ϵ, δ) -DP close.

Proof. It is sufficient to show that

$$\Pr[\mathcal{M}(\mathbf{x}) \in S] \leq e^\epsilon \cdot \Pr[\mathcal{M}(\mathbf{x}') \in S] + \delta$$

for any subset $S \subseteq \mathbb{Z}$. Let $y = f(\mathbf{x})$, $y' = f(\mathbf{x}')$. We assume that $y \leq y'$. The case of $y \geq y'$ can be dealt with similarly. Since $z \leftarrow \text{HG}(2\ell, \ell, k)$ takes values between 0 and k , we may assume that $S \subseteq \{s \in \mathbb{Z} : y \leq s \leq y' + k\}$. Letting $z_0 = k/(\beta + 1)$, we decompose S into three subsets: $S_1 = \{s \in S : s \leq y' + z_0\}$, $S_2 = \{s \in S : y' + z_0 < s \leq y + k\}$, and $S_3 = \{s \in S : y + k < s \leq y' + k\}$. We will show that

$$\Pr[\mathcal{M}(\mathbf{x}) \in S_1] \leq \delta \text{ and } \Pr[\mathcal{M}(\mathbf{x}) = s] \leq e^\epsilon \cdot \Pr[\mathcal{M}(\mathbf{x}') = s] \quad (\forall s \in S_2).$$

If this is shown, since $\Pr[\mathcal{M}(\mathbf{x}) \in S_3] = \Pr[z \leftarrow \text{HG}(2\ell, \ell, k) : z > k] = 0$, we obtain that

$$\begin{aligned} \Pr[\mathcal{M}(\mathbf{x}) \in S] &\leq \Pr[\mathcal{M}(\mathbf{x}) \in S_1] + \sum_{s \in S_2} \Pr[\mathcal{M}(\mathbf{x}) = s] \\ &\leq \delta + \sum_{s \in S_2} e^\epsilon \cdot \Pr[\mathcal{M}(\mathbf{x}') = s] \\ &\leq e^\epsilon \cdot \Pr[\mathcal{M}(\mathbf{x}') \in S] + \delta. \end{aligned}$$

First, since $y' \leq y + \Delta$ and the mean of $\text{HG}(2\ell, \ell, k)$ is $k\ell/(2\ell) = k/2$, the Chernoff inequality [5] implies that

$$\Pr[\mathcal{M}(\mathbf{x}) \in S_1] \leq \sum_{0 \leq z \leq \Delta + z_0} p_{\text{HG}}(z) \leq \exp(-2t^2k),$$

where

$$t = \frac{1}{2} - \frac{\Delta + z_0}{k} = \frac{1}{2} \left(\frac{\beta-1}{\beta+1} - \frac{2\Delta}{k} \right)$$

12:8 Card-Based Cryptography Meets Differential Privacy

We thus obtain that

$$\Pr[\mathcal{M}(\mathbf{x}) \in S_1] \leq \exp\left(-\frac{k}{2}\left(\frac{\beta-1}{\beta+1} - \frac{2\Delta}{k}\right)^2\right) \leq \delta.$$

Next, let $s \in S_2$ and set $z = s - y, z' = s - y'$. We then obtain that $\max\{z - \Delta, z_0\} \leq z' \leq z \leq k$, and

$$\begin{aligned} \frac{\Pr[\mathcal{M}(\mathbf{x}) = s]}{\Pr[\mathcal{M}(\mathbf{x}') = s]} &= \frac{p_{\text{HG}}(z)}{p_{\text{HG}}(z')} \\ &= \frac{\binom{\ell}{z} \binom{\ell}{k-z}}{\binom{\ell}{z'} \binom{\ell}{k-z'}} \\ &= \prod_{z' < i \leq z} \frac{k+1-i}{i} \prod_{\ell-k+z' < i \leq \ell-k+z} \frac{2\ell-k+1-i}{i} \\ &\leq \left(\frac{k+1}{z'+1} - 1\right)^{z-z'} \left(\frac{2\ell-k+1}{\ell-k+z'+1} - 1\right)^{z-z'} \\ &\leq \left(\frac{k}{z_0} - 1\right)^{z-z'} \alpha^{z-z'} \quad (\because z' \geq z_0) \\ &= (\alpha\beta)^{z-z'}. \end{aligned}$$

Here, we use the fact that

$$\ell \geq \frac{\alpha}{\alpha-1}k \geq \frac{\alpha}{\alpha-1}(k-1) - \frac{\alpha+1}{\alpha-1}z_0$$

and hence $(\ell - z_0)/(\ell - k + z_0 + 1) \leq \alpha$. Since $\alpha\beta > 1$, we obtain that

$$\frac{\Pr[\mathcal{M}(\mathbf{x}) = s]}{\Pr[\mathcal{M}(\mathbf{x}') = s]} \leq (\alpha\beta)^\Delta \leq e^\epsilon \quad \blacktriangleleft$$

We show a protocol $\Pi_{k,\ell}^{\text{HG}}$ based on the hypergeometric distribution in Figure 1. The following theorem shows the differential privacy, MSE and complexities of $\Pi_{k,\ell}^{\text{HG}}$.

► **Theorem 4.** *Let ϵ, δ be positive real numbers such that $\delta < 1/\sqrt{e}$. Let k, ℓ be integers such that*

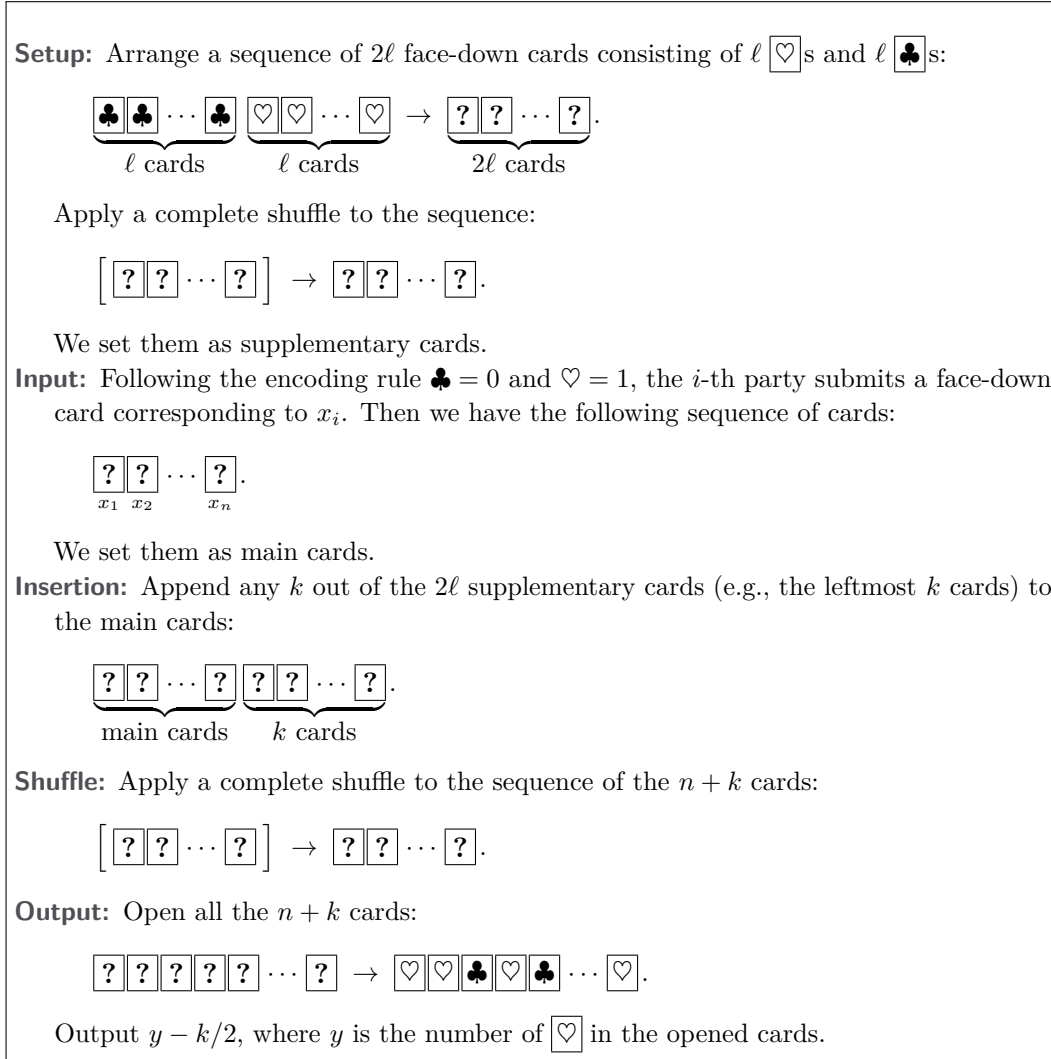
$$k \geq 4 \left(\frac{e^\epsilon + 1 + \epsilon}{e^\epsilon - 1 - \epsilon}\right)^2 \ln \frac{1}{\delta} + \frac{2(e^\epsilon + 1 + \epsilon)}{e^\epsilon - 1 - \epsilon} \quad \text{and} \quad \ell \geq \left(1 + \frac{1}{\epsilon}\right)k. \quad (2)$$

Then, the protocol $\Pi_{k,\ell}^{\text{HG}}$ satisfies (ϵ, δ) -differential privacy. The MSE of $\Pi_{k,\ell}^{\text{HG}}$ with respect to $f : \{0, 1\}^n \ni (x_i)_{i \in [n]} \mapsto \sum_{i \in [n]} x_i \in \mathbb{Z}$ is

$$\text{MSE}_f(\Pi_{k,\ell}^{\text{HG}}) = \frac{k(2\ell - k)}{4(2\ell - 1)}. \quad (3)$$

The complexities of $\Pi_{k,\ell}^{\text{HG}}$ are

$$\#\text{Card}(\Pi_{k,\ell}^{\text{HG}}) = n + 2\ell = n + O\left(\frac{e^{2\epsilon}}{\epsilon(e^\epsilon - 1 - \epsilon)^2} \ln \frac{1}{\delta}\right) \quad \text{and} \quad \#\text{Shuffle}(\Pi_{k,\ell}^{\text{HG}}) = 2.$$



■ **Figure 1** A protocol $\Pi_{k,\ell}^{\text{HG}}$.

Proof. First, we show the differential privacy of the protocol $\Pi_{k,\ell}^{\text{HG}}$. Let T be a set of corrupted parties such that $|T| \leq n - 1$, and let $\mathbf{x} = (x_i)_{i \in [n]}$, $\mathbf{x}' = (x'_i)_{i \in [n]} \in \{0, 1\}^n$ be T -neighboring inputs. Define Y (resp. Y') be random variables corresponding to the number y computed during the execution of $\Pi_{k,\ell}^{\text{HG}}$ on input \mathbf{x} (resp. \mathbf{x}'). Note that $x_i = x'_i$ for all $i \in T$ and the cards opened during the protocol are a uniformly random permutation of y \heartsuit s and $n + k - y$ \clubsuit s. Thus, the distributions of $\text{View}_{\Pi_{k,\ell}^{\text{HG}}, T}(\mathbf{x})$ and $\text{View}_{\Pi_{k,\ell}^{\text{HG}}, T}(\mathbf{x}')$ can be simulated from Y and Y' , respectively. From the post-processing property of differential privacy, it is sufficient to show that Y and Y' are (ϵ, δ) -DP close.

If parties' inputs are \mathbf{x} , then the number of \heartsuit s included in the main cards is $f(\mathbf{x})$ just after all parties submit their cards. Furthermore, the number of \heartsuit s included in k cards drawn from supplementary cards follows the distribution $\text{HG}(2\ell, \ell, k)$. Thus, the number Y of \heartsuit s included in the main cards at the end of the protocol follows the same distribution as $\mathcal{M}(\mathbf{x}) = f(\mathbf{x}) + z$, $z \leftarrow \text{HG}(2\ell, \ell, k)$. Similarly, Y' follows the same distribution as $\mathcal{M}(\mathbf{x}')$.

12:10 Card-Based Cryptography Meets Differential Privacy

Define α and β as $\alpha = \ell/(\ell - k)$ and $\beta = e^\epsilon/(1 + \epsilon)$, respectively. Since $\epsilon > 0$, we have that $\beta > 1$. Furthermore, since $\ell \geq (1 + \epsilon^{-1})k$, it holds that $\alpha \leq 1 + \epsilon$. We then obtain that

$$\alpha\beta \leq e^\epsilon. \quad (4)$$

We will show that

$$\frac{k}{2} \left(\frac{e^\epsilon - 1 - \epsilon}{e^\epsilon + 1 + \epsilon} - \frac{2}{k} \right)^2 \geq \ln \delta^{-1}. \quad (5)$$

If this is shown, the condition (4) and the assumption that the sensitivity of f is $\Delta = 1$ imply the condition (1), and hence it follows from Proposition 3 that Y and Y' are (ϵ, δ) -DP close. Let

$$a = \frac{e^\epsilon - 1 - \epsilon}{e^\epsilon + 1 + \epsilon}, \quad b = \sqrt{2 \ln \delta^{-1}}, \quad \text{and } t = \sqrt{k}.$$

Then, the condition (5) is equivalent to $(at - 2/t)^2 \geq b^2$, i.e., $at^2 - bt - 2 \geq 0$. Furthermore, it is equivalent to

$$k = t^2 \geq \frac{b^2}{2a^2} \left(1 + \sqrt{1 + \frac{8a}{b^2}} \right) + \frac{2}{a}.$$

Since $\delta \leq 1/\sqrt{e}$, we have that $a \leq 1$ and $b \geq 1$. Thus, it holds that

$$\frac{b^2}{2a^2} \left(1 + \sqrt{1 + \frac{8a}{b^2}} \right) + \frac{2}{a} \leq \frac{2b^2}{a^2} + \frac{2}{a} = 4 \left(\frac{e^\epsilon + 1 + \epsilon}{e^\epsilon - 1 - \epsilon} \right)^2 \ln \frac{1}{\delta} + \frac{2(e^\epsilon + 1 + \epsilon)}{e^\epsilon - 1 - \epsilon}.$$

The condition (5) then follows from the condition (2).

Finally, we analyze the utility of $\Pi_{k,\ell}^{\text{HG}}$. If parties' inputs are \mathbf{x} , the output of the protocol is given as $y - k/2 = f(\mathbf{x}) + z - k/2$, $z \leftarrow \text{HG}(2\ell, \ell, k)$. Since the mean of the hypergeometric distribution $\text{HG}(2\ell, \ell, k)$ is $k/2$, $\text{MSE}_f(\Pi_{k,\ell}^{\text{HG}})$ is equal to the variance of $\text{HG}(2\ell, \ell, k)$. We therefore obtain (3). \blacktriangleleft

5 Our Protocols Based on Randomized Response

Randomized Response [10, 24] guarantees differential privacy by having parties flip their input bits with a certain probability p . Specifically, for a privacy parameter $\epsilon > 0$, let p be such that

$$\frac{1}{e^\epsilon + 1} \leq p < \frac{1}{2}. \quad (6)$$

We define an algorithm \mathcal{R}_p as follows: On input $x \in \{0, 1\}$, \mathcal{R}_p chooses $r \in \{0, 1\}$ according to the Bernoulli distribution with parameter p , i.e.,

$$\Pr[r = 1] = p \quad \text{and} \quad \Pr[r = 0] = 1 - p,$$

and then outputs $y = x \oplus r$. The condition (6) implies that $\Pr[\mathcal{R}_p(x) = b] \leq e^\epsilon \cdot \Pr[\mathcal{R}_p(1 - x) = b]$ for any $x, b \in \{0, 1\}$. Hence $\mathcal{R}_p(0)$ and $\mathcal{R}_p(1)$ are $(\epsilon, 0)$ -DP close.

Setup: Arrange n sequences of ℓ face-down cards each consisting of k \heartsuit s and $\ell - k$ \clubsuit s:

$$\underbrace{\heartsuit \heartsuit \cdots \heartsuit}_{k \text{ cards}} \underbrace{\clubsuit \clubsuit \cdots \clubsuit}_{\ell - k \text{ cards}} \rightarrow \underbrace{? ? \cdots ?}_{\ell \text{ cards}}.$$

Apply a complete shuffle to each of the n sequences:

$$[? ? \cdots ?] \rightarrow [? ? \cdots ?].$$

We set the whole sequence as supplementary cards, and call the i -th sub-sequence as the i -th sequence of the supplementary cards.

Input: The i -th party performs a private reveal for any card (e.g., the leftmost one) in the i -th sequence of the supplementary cards. Let $r_i \in \{\clubsuit, \heartsuit\}$ be the opened symbol. Following the encoding rule $\clubsuit = 0$ and $\heartsuit = 1$, the i -th party submits a face-down card corresponding to $x_i \oplus r_i$. Then we have the following sequence of cards:

$$\underbrace{?}_{x_1 \oplus r_1} \underbrace{?}_{x_2 \oplus r_2} \cdots \underbrace{?}_{x_n \oplus r_n}.$$

We set them as main cards.

Output: Open all the n cards:

$$[? ? ? ? ? \cdots ?] \rightarrow [\heartsuit \clubsuit \clubsuit \heartsuit \heartsuit \cdots \clubsuit].$$

Output $z = \frac{y - nk/\ell}{1 - 2k/\ell}$, where y is the number of \heartsuit in the opened cards.

■ **Figure 2** A protocol $\Pi_{k,\ell}^{\text{RR}}$.

5.1 A Direct Implementation

Our first realization, denoted by $\Pi_{k,\ell}^{\text{RR}}$, is a direct implementation of the above procedure: We prepare n sequences each consisting of randomly permuted k \heartsuit s and $\ell - k$ \clubsuit s such that $p \approx k/\ell$ and let the i -th party privately open a card in the i -th sequence and flip his input if and only if he draws \heartsuit . The formal description of $\Pi_{k,\ell}^{\text{RR}}$ is given in Figure 2.

The following theorem shows the differential privacy, MSE and complexities of $\Pi_{k,\ell}^{\text{RR}}$.

► **Theorem 5.** *Let ϵ be a positive real number. Let k, ℓ be integers such that*

$$\ell \geq \frac{3(e^\epsilon + 1)}{e^\epsilon - 1} \text{ and } \frac{1}{e^\epsilon + 1} \leq p := \frac{k}{\ell} \leq \frac{e^\epsilon + 2}{3(e^\epsilon + 1)}. \quad (7)$$

Then, the protocol $\Pi_{k,\ell}^{\text{RR}}$ satisfies $(\epsilon, 0)$ -differential privacy. The MSE of $\Pi_{k,\ell}^{\text{RR}}$ with respect to $f : \{0, 1\}^n \ni (x_i)_{i \in [n]} \mapsto \sum_{i \in [n]} x_i \in \mathbb{Z}$ satisfies

$$\text{MSE}_f(\Pi_{k,\ell}^{\text{RR}'}) = \frac{np(1-p)}{(1-2p)^2} \leq \frac{n(e^\epsilon + 2)(2e^\epsilon + 1)}{(e^\epsilon - 1)^2}.$$

The complexities of $\Pi_{k,\ell}^{\text{RR}}$ are

$$\#\text{Card}(\Pi_{k,\ell}^{\text{RR}}) = n(\ell + 1) = O\left(\frac{ne^\epsilon}{e^\epsilon - 1}\right) \text{ and } \#\text{Shuffle}(\Pi_{k,\ell}^{\text{RR}}) = n.$$

12:12 Card-Based Cryptography Meets Differential Privacy

Proof. To begin with, it holds that

$$\frac{e^\epsilon + 2}{3(e^\epsilon + 1)} - \frac{1}{e^\epsilon + 1} = \frac{e^\epsilon - 1}{3(e^\epsilon + 1)} \geq \frac{1}{\ell}.$$

Hence, there indeed exists integers k, ℓ satisfying the condition (7).

To see the differential privacy of the protocol $\Pi_{k,\ell}^{\text{RR}}$, let $Y_i(\mathbf{x})$ denote a random variable corresponding to the state of the card that the i -th party submits to main cards when parties' inputs are \mathbf{x} . Let T be a set of corrupted parties such that $|T| \leq n - 1$, and let $\mathbf{x} = (x_i)_{i \in [n]}, \mathbf{x}' = (x'_i)_{i \in [n]} \in \{0, 1\}^n$ be T -neighboring inputs. For any $i \in [n]$, the distribution of the state r_i of the card that the i -th party draws from supplementary cards is given as $\Pr[r_i = \clubsuit] = 1 - p$ and $\Pr[r_i = \heartsuit] = p$. Furthermore, since the 1-to- n -th sub-sequences are prepared independently, r_1, \dots, r_n are independent. Thus, if we encode $\heartsuit = 1, \clubsuit = 0$, then $Y_i(\mathbf{x}) = \mathcal{R}_p(x_i)$. Similarly, we have that $Y_i(\mathbf{x}') = \mathcal{R}_p(x'_i)$. We also have that $p < 1/2$ since

$$\frac{1}{2} - \frac{e^\epsilon + 2}{3(e^\epsilon + 1)} = \frac{e^\epsilon - 1}{6(e^\epsilon + 1)} > 0.$$

The condition (6) is then satisfied and the differential privacy of the algorithm \mathcal{R}_p implies that $(Y_i(\mathbf{x}))_{i \notin T}$ and $(Y_i(\mathbf{x}'))_{i \notin T}$ are $(\epsilon, 0)$ -DP close. The adversary's view during the execution of the protocol on input \mathbf{x} (resp. \mathbf{x}') can be simulated from $((x_i)_{i \in T}, (Y_i(\mathbf{x}))_{i \notin T})$ (resp. $((x_i)_{i \in T}, (Y_i(\mathbf{x}'))_{i \notin T})$). Since $x_i = x'_i$ ($\forall i \in T$), the post-processing property implies that $\Pi_{k,\ell}^{\text{RR}}$ is $(\epsilon, 0)$ -differentially private.

To analyze the utility of $\Pi_{k,\ell}^{\text{RR}}$, let $\mathbf{x} \in \{0, 1\}^n$. For ease of notations, we write $Y_i = Y_i(\mathbf{x})$, $s = \sum_{i \in [n]} x_i$. Note that $Y_i = 1$ if and only if the i -th party submits \heartsuit to main cards, and $Y_i = 0$ if and only if he submits \clubsuit . Furthermore, $\sum_{i \in [n]} Y_i$ is equal to the total number y of \heartsuit s included in main cards.

Since $x^2 = x$ if $x \in \{0, 1\}$, the expectation and variance of Y_i are given by

$$\mathbb{E}[Y_i] = 1 \cdot \Pr[Y_i = 1] = (1 - 2p)x_i + p \text{ and}$$

$$\text{Var}[Y_i] = \mathbb{E}[Y_i^2] - (\mathbb{E}[Y_i])^2 = p(1 - p) + (1 - 2p)^2 x_i - (1 - 2p)^2 x_i^2 = p(1 - p).$$

Since $\mathbb{E}\left[\left(\sum_{i \in [n]} Y_i - np\right)/(1 - 2p)\right] = s$, the expectation of an output z of $\Pi_{k,\ell}^{\text{RR}}$ is $s = f(\mathbf{x})$. Hence, $\text{MSE}_{\Pi_{k,\ell}^{\text{RR}}}(f)$ is given by the variance of z . Since the 1-to- n -th sub-sequences of supplementary cards are prepared independently, Y_1, \dots, Y_n are independent and

$$\text{Var}[z] = \frac{1}{(1 - 2p)^2} \text{Var}\left[\sum_{i \in [n]} Y_i\right] = \frac{1}{(1 - 2p)^2} \sum_{i \in [n]} \text{Var}[Y_i] = \frac{np(1 - p)}{(1 - 2p)^2}.$$

On the other hand, $g(t) := t(1 - t)/(1 - 2t)^2$ is monotonically increasing with respect to t . We therefore conclude that $\text{Var}[z] \leq n(e^\epsilon + 2)(2e^\epsilon + 1)/(e^\epsilon - 1)^2$. \blacktriangleleft

5.2 Reducing the Number of Shuffles

A possible drawback of our first realization is that the number of shuffles grows linearly in the number of parties. In this section, we propose an alternative implementation denoted by $\Pi_{k,\ell}^{\text{RR}'}$: We prepare *one* supplementary sequence consisting of randomly permuted k \heartsuit s and $\ell - k$ \clubsuit s such that $p \approx k/\ell$ and let the i -th party privately open the i -th card in the sequence and flip his input if and only if he draws \heartsuit . The formal description of $\Pi_{k,\ell}^{\text{RR}'}$ is given in Figure 3.

The following theorem shows the differential privacy, MSE and complexities of $\Pi_{k,\ell}^{\text{RR}'}$.

Setup: Arrange a sequence of ℓ face-down cards consisting of k \heartsuit s and $\ell - k$ \clubsuit s:

$$\underbrace{\heartsuit \heartsuit \cdots \heartsuit}_k \underbrace{\clubsuit \clubsuit \cdots \clubsuit}_{\ell - k} \rightarrow \underbrace{? ? \cdots ?}_{\ell}.$$

Apply a complete shuffle to the sequence:

$$[? ? \cdots ?] \rightarrow ? ? \cdots ?.$$

We set them as supplementary cards.

Input: The i -th party performs a private reveal for the i -th card in the supplementary cards. Let $r_i \in \{\clubsuit, \heartsuit\}$ be the opened symbol. Following the encoding rule $\clubsuit = 0$ and $\heartsuit = 1$, the i -th party submits a face-down card corresponding to $x_i \oplus r_i$. Then we have the following sequence of cards:

$$\underbrace{?}_{x_1 \oplus r_1} \underbrace{?}_{x_2 \oplus r_2} \cdots \underbrace{?}_{x_n \oplus r_n}.$$

We set them as main cards.

Output: Open all the n cards:

$$? ? ? ? ? \cdots ? \rightarrow \heartsuit \clubsuit \clubsuit \heartsuit \heartsuit \cdots \clubsuit.$$

Output $z = \frac{y - nk/\ell}{1 - 2k/\ell}$, where y is the number of \heartsuit in the opened cards.

■ **Figure 3** A protocol $\Pi_{k,\ell}^{\text{RR}'}$.

► **Theorem 6.** Let ϵ be a positive real number and assume that $n \geq 2$. Let k, ℓ be integers such that

$$\alpha := \frac{n}{\ell} \leq \frac{e^\epsilon - 1}{5e^\epsilon} \quad \text{and} \quad \frac{1 + \alpha e^\epsilon}{e^\epsilon + 1} \leq p := \frac{k}{\ell} \leq \frac{1 + 2\alpha e^\epsilon}{e^\epsilon + 1}. \quad (8)$$

Then, the protocol $\Pi_{k,\ell}^{\text{RR}'}$ satisfies $(\epsilon, 0)$ -differential privacy. The MSE of $\Pi_{k,\ell}^{\text{RR}'}$ with respect to $f : \{0, 1\}^n \ni (x_i)_{i \in [n]} \mapsto \sum_{i \in [n]} x_i \in \mathbb{Z}$ satisfies

$$\text{MSE}_f(\Pi_{k,\ell}^{\text{RR}'}) \leq \frac{25n(1 + 2\alpha e^\epsilon)(1 - 2\alpha)(1 + 4\alpha)e^\epsilon}{(e^\epsilon - 1)^2}.$$

The complexities of $\Pi_{k,\ell}^{\text{RR}'}$ are

$$\#\text{Card}(\Pi_{k,\ell}^{\text{RR}'}) = n + \ell = O\left(\frac{ne^\epsilon}{e^\epsilon - 1}\right) \quad \text{and} \quad \#\text{Shuffle}(\Pi_{k,\ell}^{\text{RR}'}) = 1.$$

Proof. To begin with, it holds that

$$\frac{1 + 2\alpha e^\epsilon}{e^\epsilon + 1} - \frac{1 + \alpha e^\epsilon}{e^\epsilon + 1} \geq \frac{1}{\ell}. \quad (9)$$

Indeed, since $\alpha = n/\ell$, the inequality (9) is equivalent to $n \geq (e^\epsilon + 1)/e^\epsilon$. Since $n \geq 2$ and $2e^\epsilon > e^\epsilon + 1$, (9) actually holds. Thus, there exists an integer k satisfying the condition (8).

To see the differential privacy of the protocol $\Pi_{k,\ell}^{\text{RR}'}$, let T be a set of corrupted parties such that $|T| \leq n - 1$, and let $\mathbf{x} = (x_i)_{i \in [n]}, \mathbf{x}' = (x'_i)_{i \in [n]} \in \{0, 1\}^n$ be T -neighboring inputs.

12:14 Card-Based Cryptography Meets Differential Privacy

Let $H = [n] \setminus T$. Then $x_i \neq x'_i$ for some $i \in H$. We assume that $x_i = 0, x'_i = 1$. The case of $x_i = 1, x'_i = 0$ can be dealt with similarly. Let $H_i = H \setminus \{i\}$. For $j \in [n]$, define R_j as a random variable corresponding to the state $r_j \in \{\clubsuit, \heartsuit\}$ of the card that the j -th party draws from supplementary cards. For $j \in [n]$, define $Y_j(\mathbf{x})$ as a random variable corresponding to the state of the card that the j -th party submits when parties' inputs are \mathbf{x} . Similarly, we define $Y_j(\mathbf{x}')$ as a corresponding random variable when parties' inputs are \mathbf{x}' . For a subset $S \subseteq [n]$, we denote $R_S = (R_j)_{j \in S}, Y_S(\mathbf{x}) = (Y_j(\mathbf{x}))_{j \in S}, Y_S(\mathbf{x}') = (Y_j(\mathbf{x}'))_{j \in S}$.

Then, the joint view of corrupted parties in T is given as $\text{View}_{\Pi_{k,\ell}^{\text{RR}}, T}(\mathbf{x}) = (Y_H(\mathbf{x}), R_T)$ and $\text{View}_{\Pi_{k,\ell}^{\text{RR}}, T}(\mathbf{x}') = (Y_H(\mathbf{x}'), R_T)$. For any outcome (y_H, r_T) of $(Y_H(\mathbf{x}), R_T)$, it holds that

$$\begin{aligned} & \Pr \left[\text{View}_{\Pi_{k,\ell}^{\text{RR}}, T}(\mathbf{x}) = (y_H, r_T) \right] \\ &= \Pr[R_T = r_T] \Pr[Y_H(\mathbf{x}) = y_H \mid R_T = r_T] \\ &= \Pr[R_T = r_T] \cdot \sum_{r_{H_i}} \Pr[R_{H_i} = r_{H_i}] \Pr[Y_H(\mathbf{x}) = y_H \mid R_T = r_T, R_{H_i} = r_{H_i}], \end{aligned}$$

where r_{H_i} ranges over the set of all outcomes of R_{H_i} . Since $Y_{H_i}(\mathbf{x})$ is uniquely determined by R_{H_i} , we have that

$$\begin{aligned} & \Pr \left[\text{View}_{\Pi_{k,\ell}^{\text{RR}}, T}(\mathbf{x}) = (y_H, r_T) \right] \\ &= \Pr[R_T = r_T] \cdot \sum_{r_{H_i}} \Pr[R_{H_i} = r_{H_i}] \Pr[Y_i(\mathbf{x}) = y_i \mid R_{[n] \setminus \{i\}} = r_{[n] \setminus \{i\}}]. \end{aligned}$$

Let

$$\begin{aligned} P(y_i) &= \Pr[Y_i(\mathbf{x}) = y_i \mid R_{[n] \setminus \{i\}} = r_{[n] \setminus \{i\}}] \text{ and} \\ P'(y_i) &= \Pr[Y_i(\mathbf{x}') = y_i \mid R_{[n] \setminus \{i\}} = r_{[n] \setminus \{i\}}]. \end{aligned}$$

Suppose that $r_{[n] \setminus \{i\}}$ is composed of $n-1-j$ \clubsuit s and j \heartsuit s. Since we assume that $x_i = 0 = \clubsuit$ and $x'_i = 1 = \heartsuit$, $P(\clubsuit)$ and $P'(\heartsuit)$ are equal to the probability of the event that the i -th party draws $r_i = \clubsuit$ from supplementary cards, and hence we obtain that

$$P(\clubsuit) = P'(\heartsuit) = \frac{\ell - n + 1 - k + j}{\ell - n + 1}.$$

In addition, $P(\heartsuit)$ and $P'(\clubsuit)$ are equal to the probability of the event that the i -th party draws $r_i = \heartsuit$ from supplementary cards, and hence we have that

$$P(\heartsuit) = P'(\clubsuit) = \frac{k - j}{\ell - n + 1}.$$

Therefore, it holds that

$$\frac{P(y_i)}{P'(y_i)} \leq \max \left\{ \frac{k - j}{\ell - n + 1 - k + j}, \frac{\ell - n + 1 - k + j}{k - j} \right\}.$$

Since $0 \leq j \leq n-1$, we obtain that

$$\begin{aligned} \frac{P(y_i)}{P'(y_i)} &\leq \max \left\{ \frac{k}{\ell - n + 1 - k}, \frac{\ell - k}{k - n + 1} \right\} \\ &= \max \left\{ \frac{p}{(1-p) - (n-1)/\ell}, \frac{1-p}{p - (n-1)/\ell} \right\} \\ &\leq \max \left\{ \frac{p}{(1-p) - \alpha}, \frac{1-p}{p - \alpha} \right\}. \end{aligned}$$

On the other hand, we have that

$$\max \left\{ \frac{p}{(1-p) - \alpha}, \frac{1-p}{p - \alpha} \right\} \leq e^\epsilon \iff \frac{\alpha e^\epsilon + 1}{e^\epsilon + 1} \leq p \leq \frac{(1-\alpha)e^\epsilon}{e^\epsilon + 1}$$

and

$$\frac{(1-\alpha)e^\epsilon}{e^\epsilon + 1} \geq \frac{1+2\alpha e^\epsilon}{e^\epsilon + 1} \iff \frac{n}{\ell} \leq \frac{e^\epsilon - 1}{3e^\epsilon}.$$

Thus, it follows from the condition (8) that $P(y_i) \leq e^\epsilon \cdot P'(y_i)$. We therefore conclude that

$$\begin{aligned} & \Pr \left[\text{View}_{\Pi_{k,\ell}^{\text{RR}}, T}(\mathbf{x}) = (y_H, r_T) \right] \\ &= \Pr[R_T = r_T] \sum_{r_{H_i}} \Pr[R_{H_i} = r_{H_i}] P(y_i) \\ &\leq \Pr[R_T = r_T] \sum_{r_{H_i}} \Pr[R_{H_i} = r_{H_i}] e^\epsilon P'(y_i) \\ &= e^\epsilon \Pr[R_T = r_T] \cdot \sum_{r_{H_i}} \Pr[R_{H_i} = r_{H_i}] \Pr[Y_i(\mathbf{x}') = y_i \mid R_{[n] \setminus \{i\}} = r_{[n] \setminus \{i\}}] \\ &= e^\epsilon \Pr \left[\text{View}_{\Pi_{k,\ell}^{\text{RR}}, T}(\mathbf{x}') = (y_H, r_T) \right]. \end{aligned}$$

To analyze the utility of the protocol, let $\mathbf{x} \in \{0, 1\}^n$. For ease of notations, we write $Y_i = Y_i(\mathbf{x})$, $s = \sum_{i \in [n]} x_i$. Note that $Y_i = 1$ if and only if the i -th party submits \heartsuit to main cards, and $Y_i = 0$ if and only if the i -th party submits \clubsuit . Furthermore, $\sum_{i \in [n]} Y_i$ is equal to the total number y of \heartsuit s included in main cards.

The expectations of Y_i and Y_i^2 are

$$\mathbb{E}[Y_i] = 1 \cdot \Pr[Y_i = 1] = (1 - 2p)x_i + p \text{ and } \mathbb{E}[Y_i^2] = 1^2 \cdot \Pr[Y_i = 1] = (1 - 2p)x_i + p.$$

In particular, the expectation of an output z of $\Pi_{k,\ell}^{\text{RR}'}$ is $s = f(\mathbf{x})$ and hence $\text{MSE}_{\Pi_{k,\ell}^{\text{RR}}}(f)$ is equal to the variance of z .

Since $x^2 = x$ if $x \in \{0, 1\}$, the variance of Y_i is

$$\text{Var}[Y_i] = \mathbb{E}[Y_i^2] - (\mathbb{E}[Y_i])^2 = p(1-p) + (1-2p)^2 x_i - (1-2p)^2 x_i^2 = p(1-p).$$

For any $i \neq j$, if $x_i = x_j = 0$, then

$$\Pr[Y_i = 1, Y_j = 1] = \frac{\binom{\ell-2}{k-2}}{\binom{\ell}{k}} = \frac{k(k-1)}{\ell(\ell-1)} =: a_1.$$

If $x_i = 1, x_j = 0$ or $x_i = 0, x_j = 1$, then

$$\Pr[Y_i = 1, Y_j = 1] = \frac{\binom{\ell-2}{k-1}}{\binom{\ell}{k}} = \frac{k(\ell-k)}{\ell(\ell-1)} =: a_2.$$

If $x_i = x_j = 1$, then

$$\Pr[Y_i = 1, Y_j = 1] = \frac{\binom{\ell-2}{k}}{\binom{\ell}{k}} = \frac{(\ell-k)(\ell-k-1)}{\ell(\ell-1)} =: a_3.$$

We have that

$$\mathbb{E}[Y_i Y_j] = \Pr[Y_i = 1, Y_j = 1] = (1-x_i)(1-x_j)a_1 + ((1-x_i)x_j + x_i(1-x_j))a_2 + x_i x_j a_3.$$

12:16 Card-Based Cryptography Meets Differential Privacy

Thus, the covariance of Y_i and Y_j is

$$\begin{aligned} \text{Cov}[Y_i, Y_j] &= \mathbb{E}[Y_i Y_j] - \mathbb{E}[Y_i] \mathbb{E}[Y_j] \\ &= (a_1 - p^2) + (-a_1 + a_2 - (1 - 2p)p)(x_i + x_j) + (a_1 - 2a_2 + a_3 - (1 - 2p)^2)x_i x_j \\ &= -\frac{k(\ell - k)}{\ell^2(\ell - 1)} + \frac{2k(\ell - k)}{\ell^2(\ell - 1)}(x_i + x_j) - \frac{4k(\ell - k)}{\ell^2(\ell - 1)}x_i x_j \end{aligned}$$

We thus obtain that

$$\begin{aligned} \text{Var} \left[\sum_{i \in [n]} Y_i \right] &= \sum_{i \in [n]} \text{Var}[Y_i] + \sum_{i \neq j} \text{Cov}[Y_i, Y_j] \\ &= p(1 - p)n - \frac{k(\ell - k)}{\ell^2(\ell - 1)}n(n - 1) + \frac{4k(\ell - k)}{\ell^2(\ell - 1)}(n - 1)s - \frac{4k(\ell - k)}{\ell^2(\ell - 1)} \sum_{i \neq j} x_i x_j \\ &\leq p(1 - p)n + \frac{4p(1 - p)}{\ell}n^2 \\ &\leq \frac{(1 + 2\alpha e^\epsilon)(e^\epsilon - 2\alpha e^\epsilon)}{(e^\epsilon + 1)^2}(1 + 4\alpha)n. \end{aligned}$$

On the other hand, the condition (8) implies that

$$1 - 2p \geq 1 - \frac{2(1 + 2\alpha e^\epsilon)}{e^\epsilon + 1} = \frac{e^\epsilon - 1 - 4\alpha e^\epsilon}{e^\epsilon + 1} \geq \frac{e^\epsilon - 1}{5(e^\epsilon + 1)} > 0.$$

Thus, the variance of z is upper bounded by

$$\text{Var} \left[\frac{\sum_{i \in [n]} Y_i - np}{1 - 2p} \right] = \frac{1}{(1 - 2p)^2} \text{Var} \left[\sum_{i \in [n]} Y_i \right] \leq \frac{25n(1 + 2\alpha e^\epsilon)(1 - 2\alpha)(1 + 4\alpha)e^\epsilon}{(e^\epsilon - 1)^2}. \blacktriangleleft$$

6 Performance Evaluation

We evaluate our proposed protocols based on the following performance metrics:

- Number of cards: The total number of main cards and supplementary cards.
- Error: The mean squared error with respect to the binary sum $f(x_1, \dots, x_n) = \sum_{i \in [n]} x_i$.
- Number of shuffles: The total number of shuffles in the protocol, including the preparation of supplementary cards.

Table 1 in Section 1.1 shows the performance of our protocols in the asymptotic setting where $\epsilon \rightarrow 0$. Here, we use the approximation $e^\epsilon \approx 1 + \epsilon + \epsilon^2/2$. We set k and ℓ to the minimum integers that satisfy the conditions in Theorems 4, 5, and 6.

Figure 4 shows the performance of our protocols for concrete values of n , ϵ , and δ . We set $n \in \{100, 1000\}$, $\epsilon \in \{0.1, 0.2, \dots, 1, 1.2, \dots, 5.0\}$, and $\delta = 10^{-6}$, and plot the number of cards and the MSE.

Below, we highlight the advantage of each of the protocols $\Pi_{k,\ell}^{\text{HG}}$, $\Pi_{k,\ell}^{\text{RR}}$, and $\Pi_{k,\ell}^{\text{RR}'}$.

$\Pi_{k,\ell}^{\text{HG}}$: The error and the number of shuffles do not depend on n . The additive overhead in the number of cards, i.e., $O(\epsilon^{-5} \ln \delta^{-1})$, is independent of n . In contrast, $\Pi_{k,\ell}^{\text{RR}}$ and $\Pi_{k,\ell}^{\text{RR}'}$ suffer from a larger number of cards and a larger error when n becomes larger, as shown in Figure 4.

$\Pi_{k,\ell}^{\text{RR}}$: When ϵ is close to 0, $\Pi_{k,\ell}^{\text{RR}}$ achieves a smaller number of cards and a smaller error than $\Pi_{k,\ell}^{\text{HG}}$ and $\Pi_{k,\ell}^{\text{RR}'}$, as shown in Figure 4. In addition, $\Pi_{k,\ell}^{\text{RR}}$ achieves pure differential privacy (i.e., $\delta = 0$).

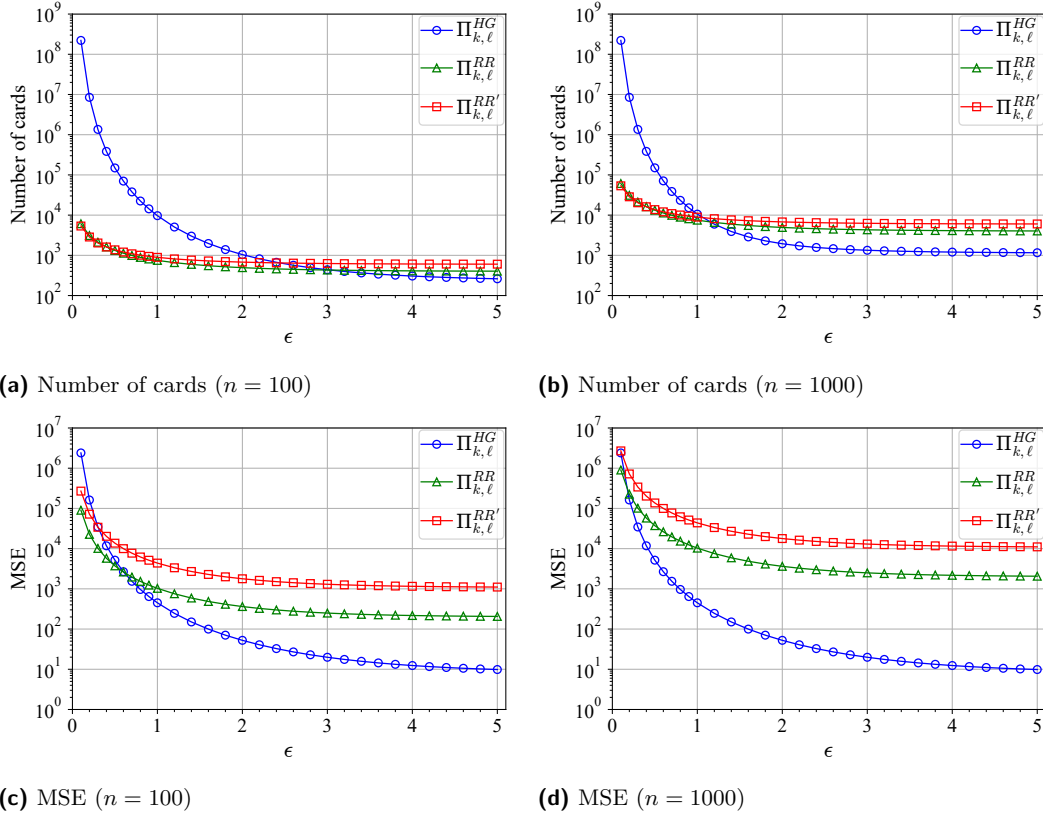


Figure 4 The number of cards and MSE of our protocols.

$\Pi_{k,\ell}^{RR'}$: The number of shuffles is only one. $\Pi_{k,\ell}^{RR'}$ achieves asymptotically the same upper bound on the number of cards and the error as $\Pi_{k,\ell}^{RR}$. It also achieves pure differential privacy (i.e., $\delta = 0$).

7 Removing Private Operations

In the protocols $\Pi_{k,\ell}^{RR}$ and $\Pi_{k,\ell}^{RR'}$, parties need to perform private reveals and privately decide which cards to submit based on the results. While such private operations could be easily realized in practice, it has also been considered as theoretically important to construct protocols without any private operations in the literature (i.e., those following the traditional model of card-based protocols [16]). In this section, we show a variant $\tilde{\Pi}_{k,\ell}^{RR}$ (resp. $\tilde{\Pi}_{k,\ell}^{RR'}$) of $\Pi_{k,\ell}^{RR}$ (resp. $\Pi_{k,\ell}^{RR'}$) where parties do not perform private reveals at the cost of doubling the number of cards and requiring n more shuffles. Note that the protocol $\Pi_{k,\ell}^{HG}$ can be described following the model of [16] as it assumes no private operation.

Our solution is to emulate private XOR operations done by each party with an existing XOR protocol without private reveals [17]. To this end, we first modify a way of encoding bits: We encode 0 into a pair of cards $\clubsuit\heartsuit$ and 1 into $\heartsuit\clubsuit$, instead of encoding 0 into \clubsuit and 1 into \heartsuit . To preserve the structure of encoding, we consider a pair of cards encoding a bit as a minimum unit. In particular, we replace every complete shuffle with a pile-scramble shuffle. That is, whenever we shuffle m cards in $\Pi_{k,\ell}^{RR}$ and $\Pi_{k,\ell}^{RR'}$, we shuffle m pairs of cards in such a way that the pairs are uniformly permuted but the order of cards in each pair is preserved.

Next, the XOR protocol in [17] allows parties to perform the following conversion of cards:

$$\underbrace{\boxed{?} \boxed{?}}_a \underbrace{\boxed{?} \boxed{?}}_b \rightarrow \underbrace{\boxed{?} \boxed{?}}_{a \oplus b}.$$

In the above protocol, parties do not perform any private operation, and the trace of states of cards is independent of inputs a, b or an output $a \oplus b$.² We modify $\Pi_{k,\ell}^{\text{RR}}$ and $\Pi_{k,\ell}^{\text{RR}'}$ as follows: Whenever a party randomizes his input bit, he first submits a pair of face-down cards encoding x_i , picks a pair of face-down cards encoding a random bit r_i , and then computes their XOR with the protocol in [17]. Since the XOR protocol in [17] requires no additional card and only one pile-scramble shuffle, the cost for executing n instances of the XOR protocol is n pile-scramble shuffles.

Finally, observe that the final states of odd-numbered cards in main cards in $\tilde{\Pi}_{k,\ell}^{\text{RR}}$ and $\tilde{\Pi}_{k,\ell}^{\text{RR}'}$ is equal to the final states of main cards in the original protocols $\Pi_{k,\ell}^{\text{RR}}$ and $\Pi_{k,\ell}^{\text{RR}'}$, respectively. We thus calculate the number y of \heartsuit s in the odd-numbered cards and output $z = \frac{y - nk/\ell}{1 - 2k/\ell}$.

The security of the XOR protocol ensures that the trace of states visible to parties is simulated from that of $\Pi_{k,\ell}^{\text{RR}}$ or $\Pi_{k,\ell}^{\text{RR}'}$. Hence, the resultant protocols $\tilde{\Pi}_{k,\ell}^{\text{RR}}$ and $\tilde{\Pi}_{k,\ell}^{\text{RR}'}$ achieve the same level of differential privacy and MSE as the original protocols. On the other hand, due to the structure of encoding and the additional shuffles to execute the XOR protocol, the complexities of $\tilde{\Pi}_{k,\ell}^{\text{RR}}$ and $\tilde{\Pi}_{k,\ell}^{\text{RR}'}$ are given as follows:

$$\begin{aligned} \#\text{Card}(\tilde{\Pi}_{k,\ell}^{\text{RR}}) &= 2n(\ell + 1) = O\left(\frac{ne^\epsilon}{e^\epsilon - 1}\right), \quad \#\text{Shuffle}(\tilde{\Pi}_{k,\ell}^{\text{RR}}) = 2n, \\ \#\text{Card}(\tilde{\Pi}_{k,\ell}^{\text{RR}'}) &= 2(n + \ell) = O\left(\frac{ne^\epsilon}{e^\epsilon - 1}\right), \quad \text{and } \#\text{Shuffle}(\tilde{\Pi}_{k,\ell}^{\text{RR}'}) = n + 1. \end{aligned}$$

References

- 1 Naman Agarwal, Ananda Theertha Suresh, Felix Xinnan X Yu, Sanjiv Kumar, and Brendan McMahan. cpSGD: Communication-efficient and differentially-private distributed SGD. In *Advances in Neural Information Processing Systems*, pages 7564–7575, 2018.
- 2 Miguel E. Andrés, Nicolás E. Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS'13)*, pages 901–914, 2013.
- 3 Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In *Advances in Cryptology – CRYPTO 2008*, pages 451–468, 2008.
- 4 Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Advances in Cryptology – EUROCRYPT 2019*, pages 375–403, 2019.
- 5 Václav Chvátal. The tail of the hypergeometric distribution. *Discrete Mathematics*, 25(3):285–287, 1979.
- 6 Claude Crépeau and Joe Kilian. Discreet solitary games. In *Advances in Cryptology – CRYPTO'93*, pages 319–330, 1994.

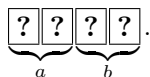
² We show a self-contained exposition of the XOR protocol in [17] in Appendix A.

- 7 Bert Den Boer. More efficient match-making and satisfiability the five card trick. In *Advances in Cryptology – EUROCRYPT’ 89*, pages 208–217, 1990.
- 8 Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: Privacy via distributed noise generation. In *Advances in Cryptology – EUROCRYPT 2006*, pages 486–503, 2006.
- 9 Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography*, pages 265–284, 2006.
- 10 Cynthia Dwork and Aaron Roth. *The Algorithmic Foundations of Differential Privacy*. Now Publishers, 2014.
- 11 Ronen Gradwohl, Moni Naor, Benny Pinkas, and Guy N. Rothblum. Cryptographic and physical zero-knowledge proof systems for solutions of Sudoku puzzles. *Theory of Computing Systems*, 44(2):245–268, 2009.
- 12 Rie Ishikawa, Eikoh Chida, and Takaaki Mizuki. Efficient card-based protocols for generating a hidden random permutation without fixed points. In *Unconventional Computation and Natural Computation*, pages 215–226, 2015.
- 13 Yoshifumi Manabe and Hibiki Ono. Secure card-based cryptographic protocols using private operations against malicious players. In *Innovative Security Solutions for Information Technology and Communications*, pages 55–70, 2021.
- 14 Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 650–661, 2012.
- 15 Daiki Miyahara, Itaru Ueda, Yu-ichi Hayashi, Takaaki Mizuki, and Hideaki Sone. Analyzing execution time of card-based protocols. In *Unconventional Computation and Natural Computation*, pages 145–158, 2018.
- 16 Takaaki Mizuki and Hiroki Shizuya. A formalization of card-based cryptographic protocols via abstract machine. *International Journal of Information Security*, 13(1):15–23, 2014.
- 17 Takaaki Mizuki and Hideaki Sone. Six-card secure AND and four-card secure XOR. In *Frontiers in Algorithmics*, volume 5598, pages 358–369, 2009.
- 18 Valtteri Niemi and Ari Renvall. Solitaire zero-knowledge. *Fundamenta Informaticae*, 38(1,2):181–188, 1999.
- 19 Takuya Nishida, Yu-ichi Hayashi, Takaaki Mizuki, and Hideaki Sone. Card-based protocols for any Boolean function. In *Theory and Applications of Models of Computation*, pages 110–121, 2015.
- 20 Hibiki Ono and Yoshifumi Manabe. Efficient card-based cryptographic protocols for the Millionaires’ problem using private input operations. In *Asia Joint Conference on Information Security (AsiaJCIS)*, pages 23–28, 2018.
- 21 Sofya Raskhodnikova and Adam Smith. *Differentially Private Analysis of Graphs*, pages 543–547. Springer, 2016.
- 22 Kazumasa Shinagawa and Koji Nuida. A single shuffle is enough for secure card-based computation of any Boolean circuit. *Discrete Applied Mathematics*, 289:248–261, 2021.
- 23 Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *Proceedings of the 26th USENIX Security Symposium (USENIX’17)*, pages 729–745, 2017.
- 24 Stanley L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.
- 25 Yohei Watanabe, Yoshihisa Kuroki, Shinnosuke Suzuki, Yuta Koga, Mitsugu Iwamoto, and Kazuo Ohta. Card-based majority voting protocols with three inputs using three cards. In *2018 International Symposium on Information Theory and Its Applications (ISITA)*, pages 218–222, 2018.
- 26 Tianqing Zhu, Gang Li, Wanlei Zhou, and Philip S. Yu. *Differential Privacy and Applications*. Springer, 2017.

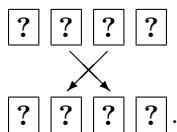
A The XOR Protocol in [17]

Mizuki and Sone [17] proposed the following four-card XOR protocol, which takes commitments to a, b with the two-card encoding $\clubsuit\heartsuit = 0$, $\heartsuit\clubsuit = 1$ and outputs a commitment to $a \oplus b$ without additional cards:

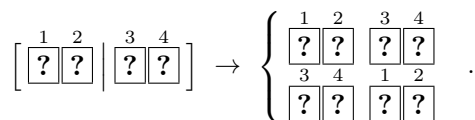
1. Arrange the input commitments as follows:



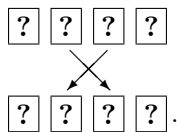
2. Rearrange the order of the sequence as follows:



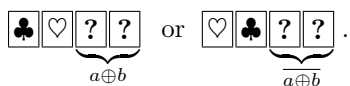
3. Apply a pile-scramble shuffle with two piles (also known as a *random bisection cut*):



4. Rearrange the order of the sequence as follows:



5. Reveal the leftmost two cards and determine the output commitment as follows:



The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples

Simone Faro¹ ✉ 🏠 

Department of Mathematics and Computer Science, University of Catania, Italy

Francesco Pio Marino ✉

Department of Mathematics and Computer Science, University of Catania, Italy
Univ Rouen Normandie, INSA Rouen Normandie, Université Le Havre Normandie, Normandie
Univ, LITIS UR 4108, CNRS NormaSTIC FR 3638, IRIB, Rouen, F-76000, France

Andrea Moschetto

Department of Mathematics and Computer Science, University of Catania, Italy

Arianna Pavone ✉ 

Department of Mathematics and Computer Science, University of Palermo, Italy

Antonio Scardace

Department of Mathematics and Computer Science, University of Catania, Italy

Abstract

Sampled String Matching is presented as an efficient solution to the string matching problem, aiming to tackle the space constraints of indexed string matching while purportedly reducing search times for online solutions. Despite the problem's inception dating back to 1991, practical solutions have only recently emerged. These purportedly accelerate online searches by up to 35 times compared to conventional methods, achieved through a partial index occupying a mere 5% of the text size.

This paper delves into the intricacies of one of the latest and most effective text sampling techniques, character distance sampling, which revolves around sampling distances between characters of a selected alphabet within the text. Specifically, we introduce fake samples while remaining honest! In other words, the study reveals that, interestingly, strategically introducing fake samples within the sampled sequence slashes the required index space by almost half, just avoid compromising the algorithm's correctness. Additionally, since efficiency is everything, this approach, in turn, purportedly enhances the algorithm's efficiency under specific conditions.

2012 ACM Subject Classification Theory of computation → Pattern matching; Information systems → Information retrieval

Keywords and phrases string matching, sampling

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.13

Funding *Simone Faro*: Supported by the National Centre for HPC, Big Data and Quantum Computing, Project CN00000013, affiliated to Spoke 10, co-founded by the European Union - NextGenerationEU.

Arianna Pavone: Supported by PNRR project ITSERR - Italian Strengthening of the ESFRI RI RESILIENCE.

1 Introduction

In the intricate world of computer science, the pursuit of pinpointing patterns within a text stands as a formidable challenge, spanning across diverse fields such as natural language processing, information retrieval, and computational biology. This endeavor, dubbed *string*

¹ Corresponding author



matching in academic circles, involves the task of identifying every occurrence of a given pattern x , spanning a length of m , within a text y , stretching to a length of n . Both sequences are crafted from characters drawn from an alphabet Σ of size σ .

While the methods for storing data may vary, textual data remains a stalwart pillar of information storage. This reliance is particularly evident in literature and linguistics, where data take the form of vast corpora and extensive dictionaries. Yet, this reliance extends into the realm of computer science, where vast volumes of data are stored in linear files. Even in the domain of molecular biology, where biological entities are often simplified to sequences of nucleotides or amino acids, the need for swift text-searching solutions persists, propelling researchers to seek ever-faster methodologies.

Applications necessitate two distinct approaches: *online* and *offline* string matching. The former contends with unprocessed text, demanding real-time scrutiny during the search operation. Its worst-case time complexity clocks in at $\Theta(n)$, a milestone initially conquered by the venerable Knuth-Morris-Pratt (KMP) algorithm [16]. However, the holy grail of average time complexity, $\Theta(\frac{n \log_{\sigma} m}{m})$ [20], finds its manifestation in the Backward-Dawg-Matching (BDM) algorithm [4], a gem in the algorithmic crown.

While legions of string matching algorithms strive for sub-linear performance in practice [5], the Boyer-Moore-Horspool algorithm [2, 13] deserves a standing ovation for its resounding success and the avalanche of subsequent research it has inspired.

On the flip side, solutions embracing the second approach aim to expedite searches through judicious preprocessing, erecting data structures that render search operations a breeze, at least proportional to the pattern's length. Dubbed *indexed searching*, this methodology enjoys a cornucopia of efficient solutions. Notable mentions include those leveraging suffix trees [1], boasting a $O(m + occ)$ worst-case time, suffix arrays [17], offering a respectable $O(m + \log n + occ)$ [17], and the formidable FM-index [12] (Full-text index in Minute space), a compressed titan born of the Burrows-Wheeler transform, deftly balancing input compression with swift substring queries.

But ah, the catch! Despite their dazzling time performance², the voracious appetite for space exhibited by full-index data structures, such as suffix-trees and suffix-arrays, dwarfs that of the text itself, ranging from 4 to 20 times its girth. And while the FM-Index appears leaner, often weighing in at less than the text's own bulk, its construction demands almost as much space as a full-index, leaving many practical applications gasping for breath.

Fear not, for the *sampled string matching* emerges as the solution to our space-hungry conundrum, offering a ray of hope amidst the darkness of bloated data structures!

1.1 Sampled String Matching

A more apt remedy for the quandary lies in the realm of *sampled string matching*, first delineated by Vishkin in 1991 [19]. Here, the strategy involves fashioning a succinct sampled version of the text and then applying any online string matching algorithm directly onto this trimmed version. While this technique may unearth potential pattern occurrences more swiftly, the caveat is that each discovery within the sampled text necessitates subsequent validation within the original corpus. Nonetheless, a sampled-text approach flaunts several virtues: firstly, it often boasts ease of implementation, standing head and shoulders above its more labyrinthine counterparts. Secondly, it exhibits a penchant for parsimony, requiring

² A fast offline solution's search speed is as swift as a cheetah's sprint, typically under 1 millisecond per query.

only a trifling amount of additional space. Moreover, it's no slouch in the speed department, often zipping through searches. Furthermore, it's not averse to updates, allowing for swift alterations to the underlying data structure.

Besides Vishkin's theoretical breakthrough, a more pragmatic incarnation of sampled string matching has emerged, courtesy of Claude *et al.* [3], leveraging an alphabet reduction technique (OTS). Their brainchild touts an extra space overhead of a mere 14% of the text's size, while clocking in at speeds up to 5 times faster than traditional online string matching algorithms on English texts. It thus earns its stripes as one of the premier solutions for such search approach.

Their ingenuity doesn't stop there. They've also dabbled in indexing the sampled text, concocting a sampled suffix array by indexing the sampled positions of the text. Although reminiscent of the sparse suffix array [14], this variant dances to its own beat, with divergent sampling properties birthing distinct search algorithms and performance metrics.

More recently, Faro *et al.* have injected a dose of innovation into the sampling sphere with their *Character Distance Sampling* (CDS) approach [7–11]. In practical terms, through sampling absolute positions of some specific characters in the text, called *pivot characters*, their method has yielded speedups of up to a factor of 9 on English texts, while demanding a mere pittance of additional space, ranging from 11% to 2.8% of the text's size. This translates to a whopping 50% reduction in search times compared to the previous approach (OTS).

1.2 Our Contribution

In this paper, we propose a variation of the CDS method that enhances it in terms of both space efficiency and search performance. The ingenious tweak involves introducing a set of additional false samples of the pivot characters, amusingly dubbed *fake samples*, which marginally increase the number of elements in the partial index. Paradoxically, this leads to a whopping three-quarters reduction in the overall space required to represent the data structure, all while ensuring algorithmic correctness. Quite a nifty advantage, wouldn't you say, especially considering the significant boost in search performance it offers?

The crux of the new approach lies in storing distances between pivot characters rather than their absolute positions within the text. This allows us to reduce the space used but introduces the problem of direct addressing of positions within the original text. But fear not, for we shall delve into the details in due course giving a solution to this problem.

The structure of the paper unfolds as follows: In Section 2 we briefly summarize the CDS method as it is used to efficiently solve the string matching problem. In Section 3, we lay out the conceptual foundation of the new sampling approach, delineating its costs and benefits. Section 5 presents an empirical analysis comparing the new approach to the standard CDS methodology in terms of space utilization and search time efficiency. Finally, our findings and insights are encapsulated in Section 6.

2 Characters Distance Sampling in Brief

In this section, we embark upon a concise yet comprehensive description of the methodology employed in crafting the partial-index built in the *Character Distance Sampling* (CDS).

For this purpose, let y be the input text, of length n , and let x be the input pattern, of length m , both over an alphabet Σ of size σ . We assume that all strings can be treated as vectors starting at position 1. Thus we refer to $x[i]$ as the i -th character of the string x , for $1 \leq i \leq m$, where m is the size of x .

13:4 The Great Textual Hoax: Boosting Sampled String Matching with Fake Samples

We select a sub-alphabet $C \subseteq \Sigma$ to serve as the *set of pivot characters*. Using this designated pivots, we sample the text y by calculating the distances between consecutive occurrences of any pivot character $c \in C$ within y . Formally, our sampling methodology is based on the following definition of *position sampling* within a text.

► **Definition 1 (Position Sampling).** *Let y be a text of length n , let $C \subseteq \Sigma$ be the set of pivot characters and let n_c be the number of occurrences of any $c \in C$ in the input text y .*

First we define the position function, $\delta : \{1, \dots, n_c\} \rightarrow \{1, \dots, n\}$, where $\delta(i)$ is the position of the i -th occurrence of any occurrence of the pivot character c in y . Formally we have

$$\begin{aligned} (i) \quad & 1 \leq \delta(i) < \delta(i+1) \leq n && \text{for each } 1 \leq i \leq n_c - 1 \\ (ii) \quad & y[\delta(i)] \in C && \text{for each } 1 \leq i \leq n_c \\ (iii) \quad & y[\delta(i) + 1.. \delta(i+1) - 1] \text{ contains no pivot characters} && \text{for each } 0 \leq i \leq n_c \end{aligned}$$

where in (iii) we assume that $\delta(0) = 0$ and $\delta(n_c + 1) = n + 1$.

Then the position sampled version of y , indicated by \dot{y} , is a numeric sequence, of length n_c , defined as

$$\dot{y} = \langle \delta(1), \delta(2), \dots, \delta(n_c) \rangle. \quad (1)$$

► **Example 2.** Suppose $y = \text{“agaacgcagtata”}$ is a sequence of length 13, over the alphabet $\Sigma = \{a, c, g, t\}$. Let $C = \{a\}$ be the set of pivot characters. Thus the *position sampled version* of y is $\dot{y} = \langle 1, 3, 4, 8, 11, 13 \rangle$. Specifically the first occurrence of character “a” is at position 1 ($y[1] = \text{“a”}$), its second occurrence is at position 3 ($y[3] = \text{“a”}$), and so on.

► **Definition 3 (Characters Distance Sampling).** *Let $C \subseteq \Sigma$ be the set of pivot characters, let $n_c \leq n$ be the number of occurrences of any pivot character in the text y and let δ be the position function of y . The characters distance function is defined by $\Delta(i) = \delta(i+1) - \delta(i)$, for $1 \leq i \leq n_c - 1$, as the distance between two consecutive occurrences of any pivot character in y .*

Then the characters-distance sampled version of the text y is a numeric sequence, indicated by \bar{y} , of length $n_c - 1$ defined as

$$\bar{y} = \langle \Delta(1), \Delta(2), \dots, \Delta(n_c - 1) \rangle = \langle \delta(2) - \delta(1), \delta(3) - \delta(2), \dots, \delta(n_c) - \delta(n_c - 1) \rangle \quad (2)$$

► **Example 4.** Let $y = \text{“agaacgcagtata”}$ be a text of length 13, over the alphabet $\Sigma = \{a, c, g, t\}$. Let $C = \{a\}$ be the set of pivot characters. Thus the character distance sampling version of y is $\bar{y} = \langle 2, 1, 4, 3, 2 \rangle$. Specifically $\bar{y}[1] = \Delta(1) = \delta(2) - \delta(1) = 3 - 1 = 2$, while $\bar{y}[3] = \Delta(3) = \delta(4) - \delta(3) = 8 - 4 = 4$, and so on.

In practical scenarios, particularly when dealing with large alphabets, the set of pivot characters may comprise only one character. Consequently, for the sake of simplicity, we will frequently refer to the pivot character in the singular form, rather than mentioning the entire set of pivot characters.

The approach of sampled string matching utilizing CDS maintains a partial index, which is represented by the position-sampled version of the text y . The size of this index is $32n_c$ bits, assuming that this index resides in memory and is readily available for any search operation on the text. When there arises a need to search for a pattern x of length m within y , a preprocessing step is executed on the pattern to compute its sampled version \bar{x} . It can be straightforwardly proved that an occurrence of x in y corresponds to an occurrence of \bar{x} in \bar{y} , hence it suffices to utilize any string matching algorithm to locate the occurrences of \bar{x} in \bar{y} to solve the problem. However, the reverse scenario is not necessarily true, implying that occurrences of \bar{x} in \bar{y} may not align with occurrences of x in y . Consequently, for each occurrence of \bar{x} in \bar{y} , referred to as a candidate occurrence, a validation check in y is required.

Given that the validation process demands $O(m)$ computational time, the entire search operation will consume $O(mn)$ time. Nonetheless, envisioning modifications to the fundamental procedure to ensure that the overall search operation, despite the checks, remains linear in time is not challenging (for further details, refer to [8]).

An essential aspect to highlight in our discourse is that the CDS-based approach does not explicitly maintain the character-distance sampled version \bar{y} of the text. Instead, it maintains the position-sampled version \dot{y} of the text. Indeed, \bar{y} solely retains the distances between the pivot characters and lacks direct ties to the original positions of these pivot characters within the text. Consequently, directly verifying every candidate occurrence becomes impracticable. This issue is addressed by retaining the text \dot{y} , which holds the positions, and computing \bar{y} on-the-fly during the search. The i -th element of \bar{y} can indeed be computed in constant time using the relationship $\bar{y}(i) = \dot{y}(i + 1) - \dot{y}(i)$.

The CDS-based sampled string matching approach has demonstrated remarkable effectiveness in practical applications, boasting a significant reduction in search times by up to 40 times compared to standard online exact string matching techniques. Remarkably, this enhancement is achieved while incurring a relatively minimal cost, as it entails the construction of a partial index merely equivalent to 2% of the text size. Moreover, sampled string matching has exhibited exceptional flexibility, rendering it adept at addressing text searching challenges, even in the approximate realm. Notably, Faro *et al.* [11] recently introduced the run-length text sampling, tailored for approximate searches. This technique proves well-suited, for instance, for tasks such as *Order Preserving pattern matching* [15].

In addition to its commendable space and time efficiency, sampled string matching offers a plethora of other advantageous features. For instance, ease of programming stands out as a notable advantage, with the construction of the partial index typically being a swift and straightforward process. Moreover, the inherent flexibility of the data structure allows it to seamlessly adapt to text variations. This means that minor alterations in the text, such as character deletions or insertions, can be effortlessly reflected in the corresponding index.

However the one described above is not without its share of pitfalls or weaknesses. One such challenge is the variability in performance based on the choice of pivot character. Consequently, strategic consideration must be given to selecting the pivot character, striking a balance between partial index size and execution times. Research indicates that in the case of the English language the pivot character ranked 8th tends to offer best performances [8].

Another factor to consider is that if the pattern is exceptionally short and lacks occurrences of the pivot character, resorting to a standard string search within the text becomes necessary. Additionally, this method may not yield significant advantages when applied to texts with small alphabets, as the benefits in terms of space efficiency may not be realized. However, studies by Faro *et al.* [9, 10] have proved the efficacy of a technique leveraging condensed alphabets to expand the underlying alphabet size and achieve markedly improved performance.

3 The Cunning Hoax of Fake Samples

Let's point out the paradox of text sampling based on position distance!

While it does indeed seek to reduce the burden of representation by sampling elements, it naively inflates the space required for each element by a whopping factor of 4. Imagine, if you will, the humble character of a text, typically content with a mere 8 bits for its expression, suddenly finding itself encased in the luxurious padding of a 32-bit integer!

In the realm of extreme scenarios, where the sampled positions sprawl to encompass more than a quarter of the input sequence, we encounter a most peculiar predicament. The very act of sampling, intended to lighten the load, paradoxically threatens to devour more

memory than the original text itself. Consider, for instance, the vast genomic landscape, where the four noble bases – A, C, G, and T – hold court. Here, the uniform distribution of these alphabetic constituents renders sampling based on position distance a pointless endeavor, yielding a partial index that, although it manages to significantly improve search performance, rivals the text in sheer magnitude of memory consumption. A nice scam!

In this section we present a useful strategy to avoid this unpleasant and somewhat embarrassing situation at a very low cost in almost all practical cases. Although our discussion focuses on the CDS sampling method, the proposed technique is suitable for any distance-based text sampling method between sampled locations where a certain limit is imposed on the representation space of each distance.

To do this, we assume that p_1 and p_2 are two consecutive sampled positions, and we also assume that $d = p_2 - p_1$ is the distance between these two positions. We use the symbol γ to denote the *distance representation bound* (DRB), a value related to the memory space used for the representation of each individual value of the sequence. In this context each value of the sequence of distances is represented using exactly γ bits, so that each distance contained in the sequence should have a value between 0 and $2^\gamma - 1$.

As we have already discussed, this constraint on the representation of distances introduces the problem of not being able to represent all those distances whose value is greater than or equal to the limit 2^γ , making it inapplicable in many practical cases.

Our solution to this problem involves the introduction of a certain additional, arbitrarily large number of samples, which can allow us to decompose a distance $d \geq 2^\gamma$ into a sequence of distances whose value is contained within the limit imposed on us. Since these additional samples are not foreseen in the actual sampling, we will call them *fake samples*, to underline the fact that they are not sampled positions but rather fictitious and therefore incorrect values, introduced exclusively for the purpose of ensuring that all distances can fall within the 2^γ limit that we have imposed for the representation of the sampled text.

To avoid ambiguity in the sequence representation, the decomposition should be non-ambiguous. For this reason, among the many possible alternatives, this work proposes a decomposition that involves the introduction of a false sample, if necessary, every $2^\gamma - 1$ characters of the original text. In other words, if two samples are at a distance $d = p_2 - p_1 \geq 2^\gamma$, we introduce a false sample at position $p'_1 = p_1 + 2^\gamma - 1$ and we operate recursively on the residual interval $p_2 - p'_1$, provided that it is greater than or equal to 2^γ . An interval of d positions between two real samples will then be decomposed, as needed, into a sequence of $\lceil d/2^\gamma \rceil$ intervals of size $2^\gamma - 1$, made exception for the last interval whose size will be equal to $d \bmod (2^\gamma)$. The above is achieved through the insertion of fake samples. An example of the Sampling procedure is shown on **Algorithm 1**.

More formally we introduce the following definition of *Fake Distance Decomposition*.

► **Definition 5** (Fake Distance Decomposition). *Let d be an integer value representing the distance between two sampled text positions. The fake distance decomposition $[d]_\gamma$ for a given DRB γ is a numeric sequence*

$$[d]_\gamma = \langle d_0, d_1, \dots, d_{k-1} \rangle$$

such that:

- (a) $k = \lceil d/2^\gamma \rceil$;
- (b) $d_i = 2^\gamma - 1$, for $0 \leq i < k - 1$;
- (c) $d_{k-1} = d \bmod (2^\gamma)$.

By Definition 5 it turns out that the sum of the distances obtained from a fake distance decomposition $[d]_\gamma$ results in the value of the original distance. Formally we have

$$\begin{aligned}
 \sum_{u \in [d]_8} u &= \sum_{i=0}^{k-1} d_i \\
 &= \sum_{i=0}^{k-2} (2^\gamma - 1) + (d \bmod (2^\gamma)) \\
 &= (2^\gamma - 1) \times \left\lfloor \frac{d}{2^\gamma} \right\rfloor + (d \bmod (2^\gamma)) \\
 &= d
 \end{aligned} \tag{3}$$

Observe also that the fake distance decomposition of a distance value $d < 2^\gamma$ leaves the original sequence unchanged, since $k = \lceil d/2^\gamma \rceil = 1$. This can be expressed by the relationship $[d]_\gamma = \langle d \rangle$, if $d < 2^\gamma$.

► **Example 6.** Let $\gamma = 8$ be the distance representation bound which assumes a binary representation of any distance value by using 8 bits, and therefore a maximum representable value equal to $2^\gamma - 1 = 255$. Then, if we take into account the distance values 841, 134, 256 and 255, we have the following fake distance decompositions:

$$\begin{aligned}
 [841]_8 &= \langle 255, 255, 255, 76 \rangle \\
 [134]_8 &= \langle 134 \rangle \\
 [265]_8 &= \langle 255, 10 \rangle \\
 [255]_8 &= \langle 255 \rangle
 \end{aligned}$$

The definition of Fake Distance Decomposition given above can be easily extended in order to decompose a distance sequence and make it representable by using γ bits for each individual sample value. We then introduce the following definition of *Faked Distance Sampling*.

► **Definition 7 (Fake Distance Sampling).** Let $y = \langle d_0, d_1, \dots, d_{n-1} \rangle$ be a sequence of distances, of length n , obtained from any kind of distance sampling performed on an input text. The fake distance sampled version of y , with DRB γ , is a numeric sequence, $[y]_\gamma$, obtained by the concatenation of the fake distance decompositions of its values. Formally

$$[y]_\gamma = [d_0]_\gamma + [d_1]_\gamma + \dots + [d_{n-1}]_\gamma$$

► **Example 8.** Let $y = \langle 3, 124, 15, 255, 7, 9, 15, 262, 9, 841, 3 \rangle$ be a sequence of distances obtained from any kind of sampling performed on the text y . According to Definition 7 the fake distance sampling of y , using a DRB value equal to 8, is given by

$$[y]_8 = \langle 3, 124, 15, 255, 7, 9, 15, \underline{255, 7, 9}, \underline{255, 255, 255, 76}, 3 \rangle$$

where we underlined the sub-sequences obtained by a fake distance decomposition.

► **Example 9.** Let $y = \langle 3, 124, 15, 255, 7, 9, 32, 15, 262, 9, 841, 3 \rangle$ be a sequence of distances obtained from any kind of sampling performed on the text y and let $x = \langle 15, 262, 9 \rangle$ be a sequence of distances obtained from the same kind of sampling performed on the pattern x .

■ **Algorithm 1** Fake-Position-Distance-Sampling($y, n, pivot$).

Data: a string y , its length m and the selected pivot character
Result: The faked character distance representation of y

```

1  $\bar{y} \leftarrow \langle \rangle$ 
2  $j \leftarrow 0$ 
3  $prev \leftarrow 0$ 
4 for  $i \leftarrow 0$  to  $n - 1$  do
5   if ( $y[i] = pivot$ ) then
6      $a \leftarrow i - prev$ 
7     while ( $a > 255$ ) do
8        $\bar{y}[j] \leftarrow 255$ 
9        $j \leftarrow j + 1$ 
10       $a \leftarrow a - 255$ 
11     end
12      $\bar{y}[j] \leftarrow a$ 
13      $j \leftarrow j + 1$ 
14   end
15 return  $\bar{y}$ 
16 end

```

According to Definition 5 the fake sample sequences are as follows:

$$[y]_8 = \langle 3, 124, 15, 255, 7, 9, 32, 15, 255, 7, 9, 255, 255, 255, 76, 3 \rangle$$

$$[x]_8 = \langle 15, 255, 7, 9 \rangle$$

Therefore with this kind of representation, if we intend to search $[x]_8$ into $[y]_8$, we will have two different *verify*.

The first verify will be at position 7, will give negative response, due for the different original distances in fact, the 255, 7 of the pattern was originally a 262 differently from the one in the text. The second one at the position 12 which can also give positive response depends on the original text and pattern. It's noteworthy to highlight that employing solely the *CDS* representation without employing the *Fake Decomposition*, the number of verification steps would be reduced to just one at position 12. This consideration arises from the fact that the occurrence at position 7 would not match the sampled pattern if it were not decomposed.

3.1 How Much Space Does This Hoax Cost?

In the realm of text sampling, the spatial demand for string matching emerges as a pivotal consideration. It delineates the additional space, relative to the text's size, that a given solution consumes to tackle the task.

Let's consider the partial index derived from our innovative approach: a clever concoction that, despite utilizing more characters than the original technique (refer to Figure 1), owes its efficiency to the inclusion of a certain number of fake samples. Despite the incorporation of these fake samples, our experimental analysis conducted on an English text³ reveals only a marginal deviation in the sizes of these two indices, maintaining a comparable character count.

³ In our experiments, we utilized the 100MB dataset of English texts sourced from *Pizza and Chili* [18].

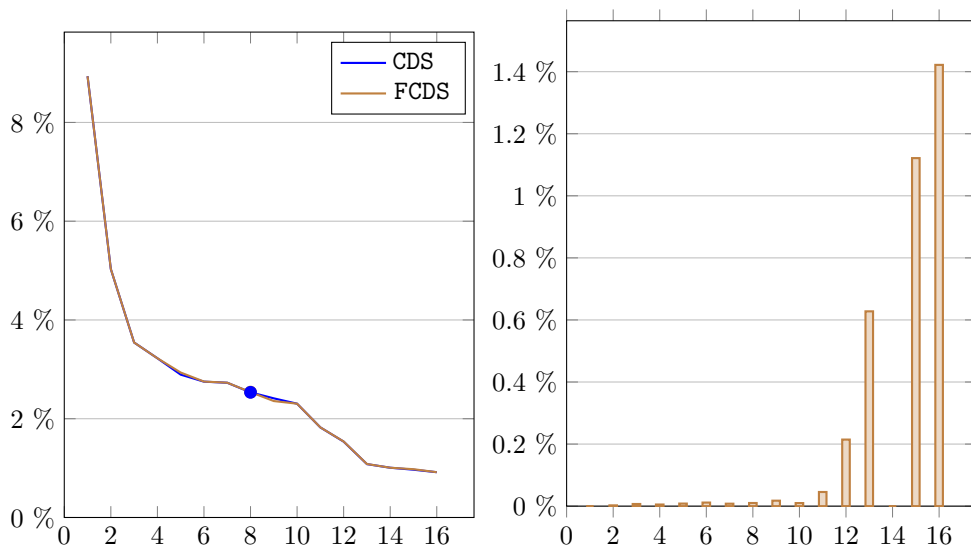
Specifically Figure 1 compares CDS and Faked-CDS approaches. On the left we show the number of elements contained in the two indexes depending on the rank of the pivot character, computed as a percentage of the number of characters of the text on which the indexes are built. We can see how the values are almost identical with very slight variations: this suggests that only a negligible number of samples are added to the original index.

This can be attributed to the fact that only a small fraction of the distances between occurrences of the pivot character exceed the imposed 2^7 threshold. This observation is confirmed in Figure 1, on the right, which shows the percentage of fake samples added in the construction of the partial index in the Faked-CDS approach. As a percentage, the addition of false samples we are compelled to include rises with the rank of the pivot character used in index construction, albeit remaining negligible.

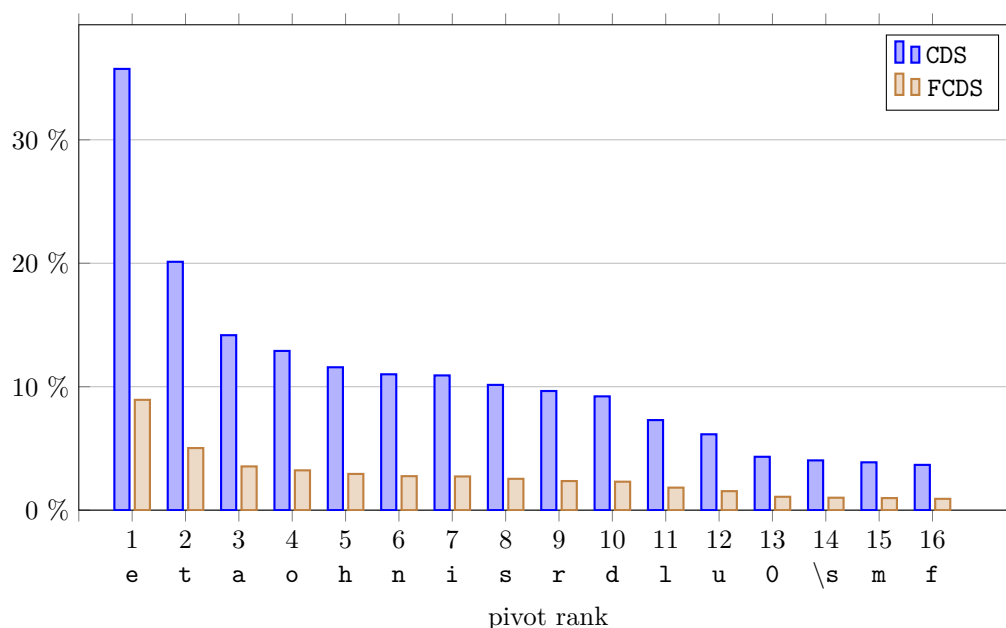
It is very interesting to observe how the pivot of rank 14, which is identified by the character `\s` (the blank space) in our dataset, reduces the number of fake samples to 0. This is due to the fact that this character has a more uniform distribution within the text and this means that two occurrences of the same character are never more than 256 characters apart.

However, the devil lies in the details! Upon closer examination of the space required by these data structures, a significant contrast becomes apparent. While the previous index allocates 32 bits for each element, our method showcases newfound parsimony, utilizing a mere 8 bits for distance representation. Behold the marvel: a nearly fourfold reduction in effective space compared to its predecessor.

To highlight this aspect, Figure 2 shows the additional amount of space required for storing the two data structures, computed as a percentage of the space required for storing the text on which the indexes are built. As expected, the space required by the Faked-CDS approach is almost four times less than that required by the standard CDS approach.



■ **Figure 1** CDS and Faked-CDS approaches compared. On the left, the number of elements contained in the two indexes, calculated as a percentage of the number of characters of the text on which the indexes are built. On the right, the percentage of fake samples added in the construction of the partial index in the Faked-CDS approach, calculated relative to the number of characters in the text. In both cases, the x-axis identifies the rank of the pivot character used to construct the two indices.



■ **Figure 2** CDS and Faked-CDS approaches compared in terms of additional amount of space required for storing the two data structures. The percentage is computed as a relative to the space required for storing the text on which the indexes are built. The x-axis identifies the rank of the character used as a pivot in the construction of the index. The corresponding character for each rank is showed below each rank (\s is the space charcater).

Ultimately, the new approach manages to reduce the space required to store the partial index by a factor ranging from 69% to 73% with respect to the standard CDS approach, despite utilizing more samples in its representation. Quite a feat, considering it also significantly enhances search performance (see Section 5 for details). Specifically, employing the pivot of rank 8, which yields the best search time performance [8], the new approach yields an index occupying only 2.5% of the text size, compared to the standard CDS method's 10%.

In Section 5, we will delve into an analysis of performance, examining both time and space metrics.

4 The Details of the Search Algorithm

The search phase in a sampling-based searching approach entails the straightforward application of any exact string matching algorithm. Its task? To ferret out all instances of the sampled pattern lurking within the sampled text. For each of these candidate occurrences, a thorough verification is then undertaken within the text. The goal? To sift through the text and discern whether the candidate occurrence is a real occurrence or merely a cunning impostor.

In the conventional rendition of the CDS approach, an additional amount of space was allocated to ensure a direct mapping of each pivot character's position within the index to its corresponding position within the original text. In simpler terms, every index element $y[i]$ denotes the position of the i -th pivot character within the text, facilitating swift navigation from the sampled text. Consequently, upon identifying a candidate occurrence, one could swiftly access the corresponding text position for verification, requiring only $O(m)$ time.

Algorithm 2 Search-FCDS($x, m, y, n, \bar{x}, \bar{m}, \bar{y}, \bar{n}$).

Data: a pattern x of length m , a text y of length n and their faked decomposition
Result: the number of occurrences of x in y

```

1  $hbc \leftarrow []$ 
2 for  $i \leftarrow 0$  to  $\Sigma$  do
3    $hbc[i] \leftarrow m$ 
4 end
5 for  $i \leftarrow 0$  to  $\bar{m}$  do
6    $hbc[\bar{x}[i]] \leftarrow \bar{m} - i - 1$ 
7 end
8  $s \leftarrow 0$ 
9  $count \leftarrow 0$ 
10  $last\_position \leftarrow 0$ 
11  $last\_position\_index \leftarrow 0$ 
12 while  $while(s \leq \bar{n} - \bar{m})$  do
13    $i \leftarrow 1$ 
14   while  $i < \bar{m}$  and  $\bar{x}[i - 1] == \bar{y}[s + i - 1]$  do
15      $i = i + 1$ 
16   end
17   if  $i == \bar{m}$  then
18      $position \leftarrow s$ 
19     while  $position > last\_position\_index$  do
20        $last\_position = last\_position + \bar{y}[position]$ 
21        $position = position - 1$ 
22     end
23      $count = count + verify(x, m, y, last\_position)$ 
24   end
25    $s = s + hbc[\bar{y}[s + \bar{m} - 1]]$ 
26   return  $count$ 
27 end

```

However, the advent of the novel sampling representation embraced by the Fake Decomposition approach alters this dynamic. This approach opts to store the distances $\bar{y}[i]$ between consecutive occurrences of pivot characters rather than their positions, thereby relinquishing the explicit mapping of pivot character positions. Consequently, the dilemma arises: how does one access the corresponding text position to verify each candidate occurrence?

By working with the distances between consecutive positions of pivot characters, we can derive the mapping $\dot{y}[i]$ of the i -th character of the index using the following formula:

$$\dot{y}[i] = \sum_{j=0}^i \bar{y}[j]$$

While this approach theoretically lends itself to direct application through a linear index scan algorithm, such as the renowned KMP algorithm [16], let's not be too hasty in celebrating its efficacy. Assuming we've computed the mapping for all positions less than i , accessing the i -th position of the index theoretically enables us to compute the direct mapping of the i -th pivot character using the formula $\dot{y}[i] = \dot{y}[i - 1] + \bar{y}[i]$.

However, despite its apparent simplicity and linear execution time, let's not overlook the practical performance implications. The most efficient string matching approaches are those employing forward jumps [5], allowing for the avoidance of scanning extensive text portions. The Boyer-Moore [2] and Horspool [13] algorithms are among the most representative elements of this family. Yet, such approaches are unable to update the mapping on the text in constant time. Oh, the joys of theoretical elegance versus practical pragmatism!

In this study, we scrutinize two potential remedies for this dilemma:

- The first approach, shown in Figure 2, entails retracing the mapping steps, starting from the last checked position. Put simply, we retain the position v of the last index position from which a text check was conducted. Assuming we've already established a direct mapping for that position, we proceed to scan all text positions from the v -th to the i -th to determine the direct mapping for the new verification position i . This method boasts swift indexing during the search phase, as mapping concerns are deferred until a candidate occurrence surfaces. However, its Achilles' heel lies in the verification phase, which could prove arduous if the last verification occurred significantly earlier in the text. We will refer to this solution as Faked-CDS (FCDS).

- In our second approach, shown in Algorithm 3, we aimed to integrate a technique reminiscent of the one used in the *OTS* approach [3]. Much like their method, we chose to periodically link elements in the sampled text to their positions in the original text. This deliberate choice was made to simplify the backtracking process, sparing us the effort of computing positions by summing distances. Furthermore, it enables quick identification of a character's position, assuming it has already been mapped. Consequently, computing positions for unmapped characters is noticeably accelerated in real-world scenarios.

In a more formal sense, we define a parameter $k \geq 1$ and allocate an array ρ of $\lceil n/k \rceil$ positions, serving as a link between the sampled elements of the partial index and their actual positions in the text. Specifically, within the partial index, one element is sampled every k , totaling $\lceil n/k \rceil$ elements. Consequently, the additional space required amounts to $4 \times \lceil n/k \rceil$ bytes. When verifying position i with v as the index of the last verified position, it becomes necessary to backtrack the mapping, commencing from the larger value between the mapping of v and $\rho[\lceil i/k \rceil]$.

The advantage of this second approach lies in its ability to circumvent the need for extensive backtracking through text, thereby reducing verification times. Of course, the downside is the inevitable consumption of extra memory to accommodate the mapping array. We will refer to this solution as Faked-CDS + Mapping (FCDS+).

5 Experimental Evaluation

In this section, we present experimental results in order to evaluate the performances of the sampling approaches presented in this paper.

In particular, we tested the original CDS approach (CDS), the Faked-CDS approach (FCDS) and the Faked-CDS enhanced with mapping (FCDS+) approach. The latter has been implemented using values of k in the set $\{8, 16, 32, 64\}$. However, the experimental results for these 4 variants are essentially identical, so we will simply unify the values of the 4 variants. Moreover we choose to set $\gamma = 8$ to conduct our experiments, where we remember that γ is the number of bits we use to represent each single element of the sampled text.

In all three instances, we opted for the pivot character ranked 8 within the text, identified as the letter **s**, as depicted in Figure 2. This particular choice was made based on its observed superior performance in terms of execution times, as detailed in [8].

Algorithm 3 Search-FCDS+($x, m, y, n, \bar{x}, \bar{m}, \bar{y}, \bar{n}, \rho, k$).

Data: a pattern x of length m , a text y of length n , their sampled versions and the mapping array

Result: the number of matches

```

1  $hbc \leftarrow []$ 
2 for  $i \leftarrow 0$  to  $\Sigma$  do
3    $hbc[i] \leftarrow m$ 
4 end
5 for  $i \leftarrow 0$  to  $\bar{m}$  do
6    $hbc[\bar{x}[i]] \leftarrow \bar{m} - i - 1$ 
7 end
8  $s \leftarrow 0$ 
9  $count \leftarrow 0$ 
10 while  $s \leq \bar{n} - \bar{m}$  do
11    $i \leftarrow 1$ 
12   while  $i < \bar{m}$  and  $\bar{x}[i - 1] == \bar{y}[s + i - 1]$  do
13      $i = i + 1$ 
14   end
15   if  $i == \bar{m}$  then
16      $position \leftarrow \rho[s/k]$ 
17      $idx \leftarrow s - (s \% k)$ 
18     while  $idx < s$  do
19        $position = position + \bar{y}[idx]$ 
20        $idx = idx + 1$ 
21     end
22      $count = count + verify(x, m, y, position)$ 
23   end
24    $s = s + hbc[\bar{y}[s + \bar{m} - 1]]$ 
25 return  $count$ 
26 end

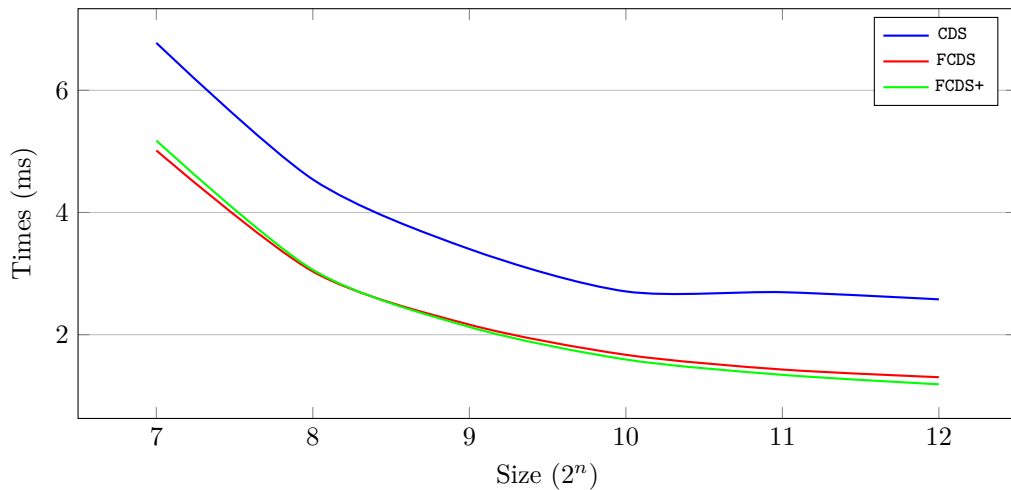
```

The algorithms have been implemented using the C programming language, and have been tested using the SMART tool [6] and executed locally on a MacBook Pro with 4 Cores, a 2.7 GHz Intel Core i7 processor, 16 GB RAM 2133 MHz LPDDR3, 256 KB of L2 Cache and 8 MB of Cache L3.⁴ During the compilation we use the -O3 optimization option.

Comparisons were conducted in terms of search times. For our experiments, we utilized a 100MB dataset of English texts sourced from *Pizza and Chili* [18]. We employed various pattern sizes ranging from 2^7 to 2^{12} . The space used to maintain the partial indexes and any mapping arrays is shown in the following table.

Name	Index Size	Mapping Size	Samples	Fake Samples
CDS	10.14 MB	-	2536790	-
FCDS	2.53 MB	-	2537048	258
FCDS+	2.53 MB	1.26 MB	2537048	258

⁴ The SMART tool is available online for download at <http://www.dmi.unict.it/~faro/smart/> or at <https://github.com/smart-tool/smart>.



■ **Figure 3** Running times of three sampled string matching algorithms for searching a 100 MB English text and averaging over 1000 different runs.

For each sequence in the dataset, we randomly selected 1000 patterns extracted from the text and computed the average running time over these 1000 runs.

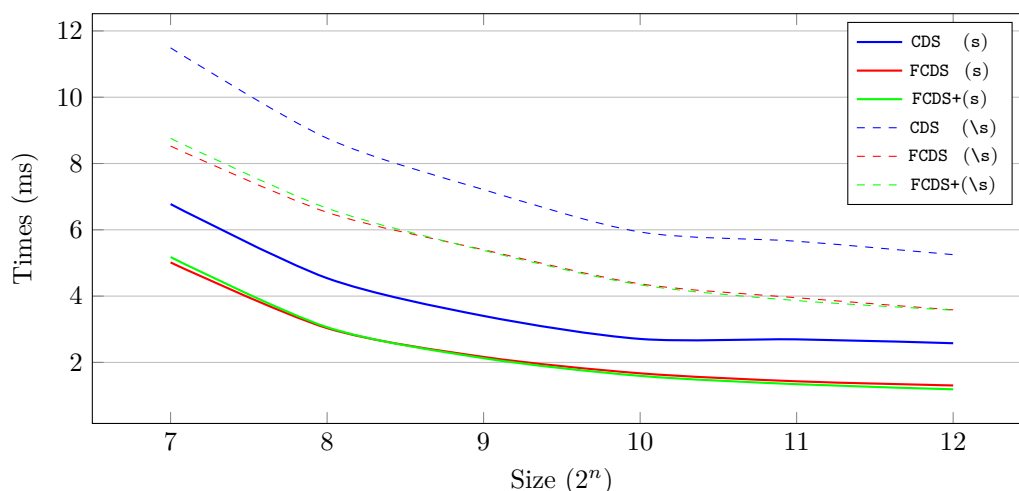
As illustrated in Figure 3, despite utilizing less space, as demonstrated in previous sections, the *Fake Decomposition* method achieves superior performance in practice, particularly in terms of searching time.

According to experimental results, it turns out that the newly proposed methods (FCDS and FCDS+) allows a speed-up ranging between 33.7% and 55.1%. Despite the increase in the size of the partial-index with the new method, several factors contribute to the observed improvement in performance. Firstly, the enlarged size of the sampled pattern, a known accelerator in classical string matching, likely plays a role, as exact string matching algorithms generally perform better with larger patterns. Secondly, the Character Distance Sampling algorithms, while susceptible to performance degradation in scenarios with fewer than 2 distances represented in the sampled pattern, benefit from the increased number of characters required for the representation, thereby averting their worst-case scenario. Another notable speed-up is observed in the searching phase, which no longer requires the computation of distances by subtracting consecutive positions; rather, each item in our sampled text already represents a distance itself.

While FCDS consistently demonstrates a space reduction of approximately 75%, FCDS+ utilizes an additional data structure to accommodate the position mapping, resulting in a space reduction ranging from 62.5% (if the position mapping is stored every 8 different characters) to 72% (with a mapping value of 32).

It is noteworthy that while both the FCDS and FCDS+ approaches demonstrate comparable performances, nuanced disparities are discernible. Notably, FCDS exhibits marginally superior efficacy for short patterns, whereas FCDS+ surpasses as pattern length extends. The rationale underlying this phenomenon lies in the examination of candidate occurrences identified by the search within the partial index.

For short patterns, the proliferation of candidate occurrences escalates, rendering the verification times in FCDS easily justifiable. The proximity of consecutive checks minimizes the impact of backtracking on search efficiency. Consequently, the inclusion of an additional mapping table mildly penalizes the FCDS+ methodology.



■ **Figure 4** Performances of the algorithms with a 100 MB English text averaging over 1000 different executions. Solid lines indicate running times obtained with a partial index constructed using the 8-th ranked pivot character, specifically representing the character 's'. Dashed lines indicate running times obtained with a partial index constructed using the the 14-th pivot character, representing the blank space character '\s', which generates no fake samples.

Conversely, with longer patterns, the candidate occurrences diminish significantly, exacerbating the performance degradation induced by backtracking in the FCDS approach. Herein lies the challenge: reconstructing the occurrence's position from distant points within the text. In contrast, FCDS+ capitalizes on the mapping table, curtailing backtracking distances substantially.

Lastly, we venture into a comparative analysis of the two methodologies delineated in this study, particularly examining scenarios where distinct pivot characters are employed. A shared observation emerges regarding the potential rationale for adopting the “blank space” character (ranked 14 within the text) as a pivot. Notably, its utilization within the FCDS approach obviates the generation of fake samples. One might conjecture that leveraging this character as a pivot could be pragmatically advantageous when performance metrics are comparable, owing to its minimal impact on partial index construction.

Regrettably, the time performances are far from commensurate, as elucidated in Figure 4. It becomes apparent that sampling predicated on the rank 8 character yields a notable 40% to 50% reduction in processing times compared to its “blank space” counterpart.

6 Conclusions

We introduced a novel method for storing sampled versions of text and patterns based on character distance sampling. Our approach involved an initial analysis of a new decomposition method, which significantly reduces the space requirements for each occurrence of any pivot character. Subsequently, we developed two distinct searching processes aimed at outperforming the original *Character Distance Sampling* (CDS) algorithm. Our algorithms demonstrate improved efficiency compared to the previous method by a factor ranging between 33.7% and 55.1%, while achieving a space reduction ranging from 63% to 75%.

Given the promising results achieved with this new technique, it would be interesting to apply it to other derived versions of the *CDS*, such as the offline and condensed variants. It would also be intriguing to investigate how the *CDS* algorithm performs when combined with other online string matching algorithms, to explore the possibility of achieving new and interesting performance improvements. We intend to explore these directions in our future work.

References

- 1 Alberto Apostolico. The myriad virtues of subword trees. In Alberto Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words*, pages 85–96, Berlin, Heidelberg, 1985. Springer Berlin Heidelberg.
- 2 Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, October 1977. doi:10.1145/359842.359859.
- 3 Francisco Claude Faust, Gonzalo Navarro, Hannu Peltola, Leena Salmela, and Jorma Tarhio. String matching with alphabet sampling. *Journal of Discrete Algorithms*, 11, December 2010. doi:10.1016/j.jda.2010.09.004.
- 4 M. Crochemore. Speeding up two string-matching algorithms. *Algorithmica*, 12(4):247–267, 1994. doi:10.1007/BF01185427.
- 5 Simone Faro and Thierry Lecroq. The exact online string matching problem: A review of the most recent results. *ACM Comput. Surv.*, 45(2), March 2013. doi:10.1145/2431211.2431212.
- 6 Simone Faro, Thierry Lecroq, Stefano Borzi, Simone Di Mauro, and Alessandro Maggio. The string matching algorithms research tool. In *Proceedings of the Prague Stringology Conference 2016*, pages 99–111. Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016. URL: <http://www.stringology.org/event/2016/p09.html>.
- 7 Simone Faro and Francesco Pio Marino. Reducing time and space in indexed string matching by characters distance text sampling. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2020, Prague, Czech Republic, August 31 - September 2, 2020*, pages 148–159. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2020. URL: <http://www.stringology.org/event/2020/p13.html>.
- 8 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Efficient online string matching based on characters distance text sampling. *Algorithmica*, 82(11):3390–3412, 2020. doi:10.1007/S00453-020-00732-4.
- 9 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Enhancing characters distance text sampling by condensed alphabets. In Claudio Sacerdoti Coen and Ivano Salvo, editors, *Proceedings of the 22nd Italian Conference on Theoretical Computer Science, Bologna, Italy, September 13-15, 2021*, volume 3072 of *CEUR Workshop Proceedings*, pages 1–15. CEUR-WS.org, 2021. URL: <https://ceur-ws.org/Vol-3072/paper1.pdf>.
- 10 Simone Faro, Francesco Pio Marino, and Arianna Pavone. Improved characters distance sampling for online and offline text searching. *Theor. Comput. Sci.*, 946:113684, 2023. doi:10.1016/J.TCS.2022.12.034.
- 11 Simone Faro, Francesco Pio Marino, Arianna Pavone, and Antonio Scardace. Towards an efficient text sampling approach for exact and approximate matching. In Jan Holub and Jan Zdárek, editors, *Prague Stringology Conference 2021, Prague, Czech Republic, August 30-31, 2021*, pages 75–89. Czech Technical University in Prague, Faculty of Information Technology, Department of Theoretical Computer Science, 2021. URL: <http://www.stringology.org/event/2021/p07.html>.
- 12 Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, July 2005. doi:10.1145/1082036.1082039.
- 13 R. Nigel Horspool. Practical fast searching in strings. *Software: Practice and Experience*, 10(6):501–506, 1980. doi:10.1002/spe.4380100608.

- 14 Juha Kärkkäinen and Esko Ukkonen. Sparse suffix trees. In Jin-Yi Cai and Chak Kuen Wong, editors, *Computing and Combinatorics*, pages 219–230, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.
- 15 Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S. Iliopoulos, Kunsoo Park, Simon J. Puglisi, and Takeshi Tokuyama. Order-preserving matching. *Theoretical Computer Science*, 525:68–79, 2014. Advances in Stringology. doi:10.1016/j.tcs.2013.10.006.
- 16 Donald E. Knuth, James H. Morris, Jr., and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977. doi:10.1137/0206024.
- 17 Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993. doi:10.1137/0222058.
- 18 Gonzalo Navarro Paolo Ferrigna. *Pizza&Chili*. Available online: pizzachili.dcc.uchile.cl/, 2005.
- 19 Uzi Vishkin. Deterministic sampling—a new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, 1991. doi:10.1137/0220002.
- 20 Andrew Chi-Chih Yao. The complexity of pattern matching for a random string. *SIAM Journal on Computing*, 8(3):368–387, 1979. doi:10.1137/0208029.

PackIt!: Gamified Rectangle Packing

Thomas Garrison ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Marijn J. H. Heule ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Bernardo Subercaseaux ✉ 

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract

We present and analyze PACKIT!, a turn-based game consisting of packing rectangles on an $n \times n$ grid. PACKIT! can be easily played on paper, either as a competitive two-player game or in *solitaire* fashion. On the t -th turn, a rectangle of area t or $t + 1$ must be placed in the grid. In the two-player format of PACKIT! whichever player places a rectangle last wins, whereas the goal in the solitaire variant is to perfectly pack the $n \times n$ grid. We analyze necessary conditions for the existence of a perfect packing over $n \times n$, then present an automated reasoning approach that allows finding perfect games of PACKIT! up to $n = 50$ which includes a novel SAT-encoding technique of independent interest, and conclude by proving an NP-hardness result.

2012 ACM Subject Classification Mathematics of computing → Combinatorics; Theory of computation → Discrete optimization

Keywords and phrases PackIt!, rectangle packing, SAT, NP-hardness

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.14

Related Version *Full Version*: <https://arxiv.org/abs/2403.12195>

Supplementary Material

Software (Code associated to the paper): <https://github.com/bsubercaseaux/packit>
archived at `swh:1:dir:4b7a9bb37e64301305bf01082703a27c23cae84f`

Funding The authors are partially supported by the National Science Foundation (NSF) grant CCF-2108521.

Acknowledgements We thank FUN2024 reviewers for their feedback and suggestions. We also thank Richard Green, for his comments and his blog post about our paper. The last author thanks Abigail Kamenetsky for her help with a web implementation of the game.

1 Introduction

Pen-and-paper games have not only stimulated bored high school students for centuries, but also attracted the attention of mathematicians and computer scientists alike. From Tic-Tac-Toe to Conway’s *Sprouts* [10], passing through *Dots and Boxes* [6], Sudoku, Hangman [1], and Nim [4], simple pen-and-paper games have had a long lasting impact in combinatorial game theory (e.g., the Sprague-Grundy theorem) and have offered landmark computational challenges (e.g., Sudokus require 17 clues to have a unique solution [13]). In this paper we introduce a new pen-and-paper game, PACKIT!, and explore both mathematical and computational challenges concerning it.

1.1 Definition of PackIt!

The game proceeds by turns, and takes place over an $n \times n$ grid that we shall denote G . The main principle of PACKIT! is very simple: on turn t (starting from 1), a rectangle r_t of area t or $t + 1$ must be placed into G without intersecting any of the already placed rectangles.



© Thomas Garrison, Marijn J. H. Heule, and Bernardo Subercaseaux;
licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

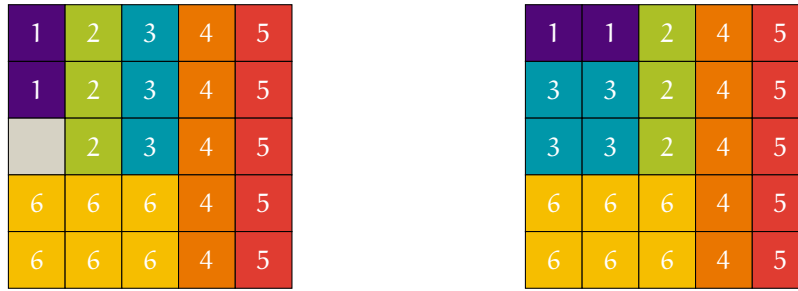
Editors: Andrei Z. Broder and Tami Tamir; Article No. 14; pp. 14:1–14:19



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

14:2 PackIt!: Gamified Rectangle Packing



(a) An *imperfect* game of PACKIT!.

(b) A *perfect* game of PACKIT!.

■ **Figure 1** Illustration of a couple of games of PACKIT!. Each rectangle a_t is labeled with t and depicted in a different color.

Formally, at the beginning of the game one defines the set of *used cells of the grid* as $U_0 := \emptyset$. On turn t , the corresponding player chooses $r_t = (h_t, v_t, x_t, y_t)$, with $h_t \cdot v_t \in \{t, t + 1\}$, and $0 \leq x_t, y_t < n$. Define the cells used by this rectangle as the set

$$A_t := \{x_t, x_{t+1}, \dots, x_{t+h_t-1}\} \times \{y_t, y_{t+1}, \dots, y_{t+v_t-1}\},$$

so that the requirement for a valid turn is that $A_t \cap U_{t-1} = \emptyset$. After a valid turn, one sets $U_t := U_{t-1} \cup A_t$. Figure 1 illustrates some examples.

PackIt! as a game. PACKIT! can be played as a *solitaire* game, where the goal of the game is to complete a *perfect packing*, that is, to play so that after a valid sequence of turns it holds that $U_t = \{0, \dots, n-1\} \times \{0, \dots, n-1\}$. As depicted in Figure 1, we say the final board of such a game corresponds to a *perfect game of PACKIT!*. For two players, it suffices to alternate turns and when a player cannot play a valid turn, he or she is declared the *loser*. At this point, we suggest the reader to directly experiment with PACKIT!. A version of the game is available for *solitaire* mode at <https://packit.surge.sh>.

Organization. The main question about PACKIT! is:

for which values of n the $n \times n$ grid admits a perfect game of PACKIT!?

Section 2 presents arithmetic results that represent the initial steps toward answering this question. Then, Section 3 discusses the complexity of PACKIT!, showing that a particular version of the solitaire game is NP-hard. Finally, Section 4 is devoted to analyzing this question from a computational perspective. We present an initial backtracking implementation, which is then improved by a more complicated approach leveraging a novel SAT encoding.

2 Arithmetic Results

A perfect game of PACKIT! can be conceptually divided into two aspects:

- **(Rectangle selection)** As we denote by $|A_t|$ the area of the rectangle used in turn t , it must hold that in a perfect game of PACKIT! we have

$$\sum_t |A_t| = n^2.$$

Moreover, in order to fit every rectangle r_t of dimensions $h_t \times v_t$, it must hold that $\max(h_t, v_t) \leq n$. We will say that such a sequence of choices is a *valid rectangle sequence*.

- **(Packing Aspect)** Even if a sequence of area choices is valid, it can be the case that it is not possible to use such area choices in a perfect game of PACKIT!.

This section focuses on studying perfect games through the lens of the first aspect, as it is sometimes enough to determine the *tileability/untileability* of grids. Despite PACKIT! being originally defined for a square grid, from now on we consider $m \times n$ grids as most of our ideas generalize nicely in that setting. Without loss of generality we will assume $n \geq m$ throughout the paper.

In order to state our results, we will need a couple of definitions. We denote by T_k the k -th *triangle number*, defined as $T_k = \sum_{i=1}^k i = \frac{k(k+1)}{2}$. Then, for any positive integer r , we denote by $\tau(r) = \arg \max_k \{T_k \mid T_k \leq r\}$.

An initial observation to understand whether an $m \times n$ grid admits a perfect packing is that the number of rectangles used in perfect PACKIT! games depends entirely on the grid area $m \cdot n$, and not on its precise width or height

► **Lemma 1.** *For an $m \times n$ grid there is a unique number $K(m, n)$ such that if the $m \times n$ grid admits a perfect PACKIT! game, then such a packing must use exactly $K(m, n)$ rectangles. In particular, $K(m, n) = \tau(m \cdot n)$.*

Proof. Assume, expecting a contradiction that for some $m \times n$ grid there are two sequences $A := (|A_1|, \dots, |A_{K_1}|)$ and $A' := (|A'_1|, \dots, |A'_{K_2}|)$, with $K_1 \neq K_2$, that can be used for perfect packings. Now, note that we must have

$$\sum_{t=1}^{K_1} |A_t| = m \cdot n = \sum_{t=1}^{K_2} |A'_t|. \tag{1}$$

By the game rules, we have that

$$\sum_{t=1}^{K_1} |A_t| \geq \sum_{t=1}^{K_1} t = T_{K_1}, \quad \text{and} \quad \sum_{t=1}^{K_1} |A_t| \leq \sum_{t=1}^{K_1} (t+1) = T_{K_1+1} - 1.$$

Using the same analysis for A' , and Equation (1), we get

$$\max(T_{K_1}, T_{K_2}) \leq m \cdot n \leq \min(T_{K_1+1}, T_{K_2+1}) - 1.$$

As $K_1 \neq K_2$, let us assume without loss of generality that $K_1 > K_2$. Using that T is an increasing sequence, we have

$$T_{K_1} \leq m \cdot n \leq T_{K_2+1} - 1. \tag{2}$$

Now, as K_1 is an integer, $K_1 > K_2$ implies $K_1 \geq K_2 + 1$, from where Equation (2) becomes $T_{K_1} \leq m \cdot n \leq T_{K_1} - 1$, a clear contradiction. To obtain the second part of the lemma, note that when $K(m, n) := K_1 = K_2$ we get

$$T_{K(m,n)} \leq m \cdot n \leq T_{K(m,n)+1} - 1,$$

from where it follows by the definition of τ that $K(m, n) = \tau(m \cdot n)$. ◀

We can now define the notion of *gap*, which intuitively represents the number of turns t in which a rectangle of area $t + 1$ must be chosen. Let us say that any turn t at which a rectangle of area $t + 1$ is chosen is an *expansion turn*.

► **Definition 2.** *For any $m \times n$ grid, we define its gap, $\gamma(m, n)$, as*

$$\gamma(m, n) = m \cdot n - T_{\tau(m \cdot n)}.$$

14:4 PackIt!: Gamified Rectangle Packing

► **Lemma 3.** *For any sequence of turns that results in a perfect packing of an $m \times n$ grid, the number of expansion turns is exactly $\gamma(m, n)$.*

Proof. By Lemma 1, there must be exactly $K(m, n) = \tau(m \cdot n)$ turns in such a sequence. If for every turn $t \in \{1, \dots, \tau(m \cdot n)\}$, a rectangle of area t were to be chosen, then the total area used would be exactly

$$\sum_{t=1}^{\tau(m \cdot n)} t = T_{\tau(m \cdot n)}.$$

Given that the total area used must be $m \cdot n$, we conclude there must be exactly $m \cdot n - T_{\tau(m \cdot n)}$ expansion turns. ◀

The next ingredient to analyze whether an $m \times n$ grid admits a perfect packing has to do with prime numbers, as if the area of a rectangle is a prime number p , then the only possible rectangles are $p \times 1$ or $1 \times p$, which can limit our ability to pack it. We define the set $P(m, n)$ as

$$P(m, n) = \{p \mid n < p \leq K(m, n) \text{ and } p \text{ is prime}\}.$$

As the next results show, the comparison between the *gap* of a grid and the size of its corresponding P set plays a crucial role in understanding whether or not it allows a perfect packing. In particular, Theorem 4 shows how small gaps can forbid perfect packings, whereas Theorem 5 shows how large gaps can also be problematic.

► **Theorem 4 (Small gap).** *For any $m \times n$ grid, if $\gamma(m, n) < |P(m, n)|$, then the grid does not allow a perfect game of PACKIT!.*

1	2	2	3	3	3
4	4	7	7		5
4	4	7	7		5
6	6	7	7		5
6	6	7	7		5
6	6				5

■ **Figure 2** Illustration of the impossibility result for $n = 6$ resulting from Theorem 4. Even though turns 1 through 6 use the minimal possible area, the choice of area 8 on turn 7 is enough to make turn 9 possible, as only 8 empty cells remain (which is invariant under the concrete choice of packing).

Before a formal proof, let us present some intuition. Theorem 4 considers a gap that is “too small”, as the following example shows. Consider $m = n = 6$. One can easily check that, $\tau(6 \cdot 6) = 8^1$, and therefore the gap results in

$$\gamma(m, n) = m \cdot n - T_{\tau(m \cdot n)} = 6 \cdot 6 - \frac{8 \cdot 9}{2} = 0.$$

¹ A general formula for $\tau(r)$ is not too hard to derive. In particular, $\tau(r) = \left\lfloor \frac{\sqrt{8r+1}}{2} - \frac{1}{2} \right\rfloor$.

Then, $K(m, n) = \tau(m \cdot n) = 8$, and thus $P(m, n) = \{7\}$. As $K(m, n) = 8$, any perfect packing of the 6×6 grid will consist of 8 rectangles. We claim that in turn 7, the area chosen must be 7, or in other words, that choosing a rectangle of area 8 in turn 7 would forbid a perfect packing. To see this, consider expecting a contradiction that a rectangle of area 8 is chosen on turn 7, and notice that then on the first 8 turns the smallest sum of areas we can achieve would be

$$1 + 2 + 3 + 4 + 5 + 6 + 8 + 8 = 37 > 36,$$

a contradiction. On the other hand, given 7 is a prime number, the only rectangles of area 7 are a 1×7 or a 7×1 rectangle, neither of which can be packed into a 6×6 grid. As either area choice for turn 7 leads to a contradiction, we conclude it is not possible to have a perfect game of PACKIT! over the 6×6 grid. This example is illustrated in Figure 2, and is generalized in the next proof.

Proof of Theorem 4. Let $p \in P(m, n)$. At turn p , one must choose between area p or area $p + 1$. If area p is chosen, then the rectangle must be either $1 \times p$ or $p \times 1$, due to the primality of p . However, by the definition of the set $P(m, n)$ we have $p > n \geq m$, and thus neither the $1 \times p$ nor the $p \times 1$ rectangle can be packed into the $m \times n$ grid. Assume, expecting a contradiction, that $\gamma(m, n) < |P(m, n)|$ and there exists a sequence of turns leading to a perfect packing for the $m \times n$ grid. As a result of the previous argument, every turn $p \in P(m, n)$ must be an expansion turn. As the number of expansion turns is equal to $\gamma(m, n)$ by Lemma 3, we have $\gamma(m, n) \geq |P(m, n)|$, which directly contradicts the assumption. ◀

► **Theorem 5 (Large gap).** For any $m \times n$ grid, let $\mathbf{1}_{K_p}$ be the indicator variable corresponding to whether $K(m, n) + 1$ is a prime number or not. Then, the condition

$$\gamma(m, n) > K(m, n) - |P(m, n)| - \mathbf{1}_{K_p}$$

implies the $m \times n$ grid does not allow a perfect game of PACKIT!.

Before the proof, let us present some intuition for Theorem 5. Consider $m = n = 18$ (this example is illustrated in Figure 3). As a result, $\tau(18 \cdot 18) = 24$, and therefore the gap is

$$\gamma(m, n) = m \cdot n - T_{\tau(m \cdot n)} = 18 \cdot 18 - \frac{24 \cdot 25}{2} = 24.$$

We also have $K(m, n) = \tau(m \cdot n) = 24$, implying that any perfect packing of the 18×18 grid will consist of $K(m, n) = 24$ rectangles. We claim that on turn 18, both choices of area, 18 and 19, lead to contradictions. Let us see what happens if area 18 is chosen on turn 18. In this case, even if area $t + 1$ is chosen on every turn $t \neq 18$, the maximum sum of the areas we can achieve is

$$2 + 3 + \dots + 17 + 18 + \mathbf{18} + 20 + \dots + 25 = 323 < 324,$$

implying the 324 cells of the 18×18 grid cannot be covered. On the other hand, if area 19 is chosen on turn 18, we run into a different issue: as 19 is a prime number it only allows for the rectangles 1×19 or 19×1 , neither of which can be packed into the 18×18 grid. As both cases lead to an impossibility, we conclude it is not possible to have a perfect game of PACKIT! over the 18×18 grid. The proof for Theorem 5 generalizes this example.

14:6 PackIt!: Gamified Rectangle Packing

5	5	5	5	5	5	23	23	23	23	23	23	23	23	23	23	23	23
14	4	4	4	4	4	23	23	23	23	23	23	23	23	23	23	23	23
14	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
14	12		11	11	11	11	11	11	11	11	11	11	11	2	2	2	
14	12		7	7	7	7	7	7	7	7	24	24	24	24	24	19	19
14	12	21	21	10	22	22	20	20	20	8	24	24	24	24	24	19	19
14	12	21	21	10	22	22	20	20	20	8	24	24	24	24	24	19	19
14	12	21	21	10	22	22	20	20	20	8	24	24	24	24	24	19	19
14	12	21	21	10	22	22	20	20	20	8	24	24	24	24	24	19	19
14	12	21	21	10	22	22	20	20	20	8	6	17	17	9	9	19	19
14	12	21	21	10	22	22	20	20	20	8	6	17	17	9	9	19	19
14	12	21	21	10	22	22	20	20	20	8	6	17	17	9	9	19	19
14	12	21	21	10	22	22	18	18	18	8	6	17	17	9	9	19	19
14	12	21	21	10	22	22	18	18	18	1	6	17	17	15	15	15	15
14	12	21	21	10	22	22	18	18	18	1	6	17	17	15	15	15	15
13	13	13	13	13	13	13	18	18	18	3	3	17	17	15	15	15	15
13	13	13	13	13	13	13	18	18	18	3	3	17	17	15	15	15	15

■ **Figure 3** Illustration of the impossibility result for $n = 18$ (Theorem 5). Even though almost each rectangle t has area $t + 1$, except for $t \in \{18, 22\}$ (where $t + 1 > n$ is prime), the total area covered by turn 24 is only $322 = 18^2 - 2$, and naturally it is not possible to fill in the two remaining cells in turn 25.

Proof of Theorem 5. Let $p \in P(m, n)$. As $p \leq K(m, n)$ by definition of $P(m, n)$, turn $p - 1$ is necessarily part of any perfect packing. At turn $p - 1$, one must choose between area $p - 1$ or area p . If area p is chosen, then the rectangle must be either $1 \times p$ or $p \times 1$, due to the primality of p . However, by the definition of the set $P(m, n)$ we have $p > n \geq m$, and thus neither the $1 \times p$ nor the $p \times 1$ rectangle can be packed into the $m \times n$ grid. We conclude that for each $p \in P(m, n)$, the turn $p - 1$ is not an expansion turn.

If $K(m, n) + 1$ is prime, then the rectangle turn $K(m, n)$ cannot be an expansion turn. By definition, $K(m, n) + 1 \notin P(m, n)$, so the number of turns that are not expansion turns is at least $|P(m, n)| + \mathbf{1}_{K_p}$. By Lemma 1, the number of expansion turns is exactly $\gamma(m, n)$, which together with the previous fact implies that the total number of turns is at least

$$|P(m, n)| + \mathbf{1}_{K_p} + \gamma(m, n). \tag{3}$$

Suppose, expecting a contradiction that

$$\gamma(m, n) > K(m, n) - |P(m, n)| - \mathbf{1}_{K_p}, \tag{4}$$

and yet there exists a sequence of turns leading to a perfect packing for the $m \times n$ grid. By

combining Equation (3) and Equation (4), the total number of turns is at least

$$\begin{aligned} |P(m, n)| + \mathbf{1}_{K_p} + \gamma(m, n) &> |P(m, n)| + \mathbf{1}_{K_p} + (K(m, n) - |P(m, n)| - \mathbf{1}_{K_p}) \\ &= K(m, n), \end{aligned}$$

which is a contradiction, given the total number of turns must be exactly $K(m, n)$ according to Lemma 1. \blacktriangleleft

Combining Theorem 4 and Theorem 5, we obtain a range of values for the gap of an $m \times n$ grid in which perfect packings are *a priori* possible. So far, we have not found any examples of $m \times n$ grids whose gap belongs in this range and yet no perfect packings exist. Therefore, we pose the following conjecture

► **Conjecture 6.** *Let $m \leq n$ be positive integers. Then, if*

$$|P(m, n)| \leq \gamma(m, n) \leq K(m, n) - |P(m, n)| - \mathbf{1}_{K_p},$$

it is possible to complete a perfect game of PACKIT! for the $m \times n$ grid.

Interestingly, Theorem 4 is enough to construct infinite families of $n \times n$ grids that do not admit perfect packings.

► **Theorem 7.** *There are infinitely many positive integers n such that the $n \times n$ grid does not admit a perfect game of PACKIT!.*

Proof. By Theorem 4, it suffices to show that there are infinitely many values of n such that $\gamma(n, n) = 1$ and $|P(n, n)| > 1$. First, consider the following claim.

▷ **Claim 8.** For every $n \geq 100$, we have $K(n, n) \geq 1.4n$.

Proof of Claim 8. Let $\ell = \lfloor 1.4n \rfloor$. It suffices to argue that $T_\ell \leq n^2$. As $\ell > n \geq 100$, we have $\ell < \frac{1}{100}\ell^2$, which we can use as follows.

$$T_\ell = \frac{\ell^2 + \ell}{2} \leq \frac{\frac{101\ell^2}{100} + \ell}{2} = 101\ell^2/200,$$

and conclude by noting that

$$101\ell^2/200 \leq \frac{101}{200} \cdot \left(\frac{140}{100}n\right)^2 = \frac{1\,979\,600}{2\,000\,000}n^2 \leq n^2. \quad \triangleleft$$

Now, Schoenfeld proved in [15] that for every $n > 3 \cdot 10^6$, there is always a prime number between n and $(1 + \frac{1}{16957})n$, which applied twice gives us that there are always (at least) two prime numbers between n and $(1 + \frac{1}{16957})^2 n \leq 1.4n$. Therefore, for $n > 3 \cdot 10^6$ we always have $|P(n, n)| > 1$. It remains to prove that $\gamma(n, n) = 1$ infinitely often. We do this by using the theory of generalized Pell's equation. Indeed, the condition $\gamma(n, n) = 1$ can be written, by using notation $K := K(n, n)$, as

$$n^2 - \frac{K(K+1)}{2} = 1, \tag{5}$$

which after multiplying both sides by 8 and rearranging is equivalent to

$$8n^2 - (2K+1)^2 = 7.$$

14:8 PackIt!: Gamified Rectangle Packing

Introducing the variable $t := (2K + 1)$ we consider the following equations.

$$t^2 - 8n^2 = -7, \quad (6)$$

$$\left(t^{(h)}\right)^2 - 8\left(n^{(h)}\right)^2 = 1. \quad (7)$$

While Equation (7) presents an “homogeneous” Pell equation, for which it is well known that infinitely many solutions exist over the positive integers (cf. the problem of square triangular numbers [2]), Equation (6) corresponds to a “non-homogeneous” equation, less frequently studied. Similarly to the theory of ordinary differential equations, we can obtain a set of solutions to the non-homogeneous equation by combining one *initial solution* for it with a set of solutions to its homogeneous counterpart. Indeed, assume the existence of a solution (n_0, t_0) to Equation (6) over the positive integers, and $\left(n_i^{(h)}, t_i^{(h)}\right)$ a sequence of solutions to Equation (7) over the positive integers, whose existence is standard (see e.g., [2]).

▷ **Claim 9.** The sequence (n_i, t_i) , defined as

$$(n_i, t_i) := \left(t_0 t_i^{(h)} + 8n_0 n_i^{(h)}, t_0 n_i^{(h)} + n_0 t_i^{(h)}\right), \quad (8)$$

is an infinite family of solutions of Equation (6) over the positive integers.

Proof of Claim 9. By assumption, (n_0, t_0) is a solution of Equation (6), and $\left(n_i^{(h)}, t_i^{(h)}\right)$ is a solution of Equation (7). Thus, we have

$$\begin{aligned} -7 &= (t_0^2 - 8n_0^2) \left(\left(t_i^{(h)}\right)^2 - 8\left(n_i^{(h)}\right)^2 \right) \\ &= (t_0 + \sqrt{8}n_0)(t_0 - \sqrt{8}n_0) \left(t_i^{(h)} + \sqrt{8}n_i^{(h)} \right) \left(t_i^{(h)} - \sqrt{8}n_i^{(h)} \right) \\ &= \left[(t_0 + \sqrt{8}n_0) \left(t_i^{(h)} + \sqrt{8}n_i^{(h)} \right) \right] \cdot \left[(t_0 - \sqrt{8}n_0) \left(t_i^{(h)} - \sqrt{8}n_i^{(h)} \right) \right] \\ &= \left[\left(t_0 t_i^{(h)} + 8n_0 n_i^{(h)} \right) + \sqrt{8} \left(t_0 n_i^{(h)} + n_0 t_i^{(h)} \right) \right] \\ &\quad \cdot \left[\left(t_0 t_i^{(h)} + 8n_0 n_i^{(h)} \right) - \sqrt{8} \left(t_0 n_i^{(h)} + n_0 t_i^{(h)} \right) \right] \\ &= \left(t_0 t_i^{(h)} + 8n_0 n_i^{(h)} \right)^2 - 8 \left(t_0 n_i^{(h)} + n_0 t_i^{(h)} \right)^2 \\ &= n_i^2 - 8t_i^2. \quad \blacktriangleleft \end{aligned}$$

As we can provide an initial solution $(n_0, t_0) := (11, 31)$ to Equation (6), we conclude by Claim 9 that it has infinitely many solutions over the positive integers. We now finish the proof by the following claim.

▷ **Claim 10.** Every solution (n_i, t_i) to Equation (6) over the positive integers with $n_i > 3 \cdot 10^6$ corresponds to a value of n such that the $n \times n$ grid does not admit a perfect game of PACKIT!.

Proof of Claim 10. Let (n_i, t_i) be a solution to Equation (6) and let us argue that the $n_i \times n_i$ does not admit a perfect game of PACKIT!. First, consider that t_i must be odd, as $t_i^2 = 1 + 8n_i^2$, by Equation (6). Therefore $(t_i - 1)/2$ is a positive integer. We now argue that $(t_i - 1)/2$ indeed matches the definition of $K(n_i, n_i)$. Let us denote $(t_i - 1)/2$ by K' , and we will argue that indeed $K' = K(n_i, n_i)$. To see, this, consider that as Equation (6) has the same set of solutions as Equation (5), it must be the case that

$$n_i^2 - \frac{K'(K' + 1)}{2} = 1,$$

implying that $T_{K'} = n_i^2 - 1 \leq n_i^2$. Moreover, we have that

$$T_{K'+1} = T_{K'} + (K' + 1) = n_i^2 + K' > n_i^2,$$

thereby confirming that $K' = \tau(n_i^2) = K(n_i, n_i)$. Taking $n := n_i$, we have by construction that $\gamma(n, n) = 1$, and as $n > 3 \cdot 10^6$ we have $|P(n, n)| > 1$. Therefore the condition of Theorem 4 applies to n , implying the $n \times n$ grid does not admit a perfect packing. This concludes the proof of the entire theorem. \triangleleft

Let us define notation $\gamma^{-1}(c)$ to denote the set $\{n \in \mathbb{N}^{>0} \mid \gamma(n, n) = c\}$. The previous proof showed that there are infinitely many values of $n \in \gamma^{-1}(1)$ that do not admit perfect packings. We now show a much stronger statement.

► **Theorem 11.** *For every value $c \geq 0$, only a finite number of values $n \in \gamma^{-1}(c)$ allow for a perfect packing of the $n \times n$ grid.*

Proof. By Theorem 4, it suffices to show that for every value $c \geq 0$, there are only finitely many values of n such that

$$|P(n, n)| = \{n < p \leq K(n, n) \mid p \text{ is prime}\} \leq c$$

We will do so by using the following improvement on Bertrand's postulate due to Dusart.

► **Proposition 12** ([8]). *For every value of $n > 3275$, there exists a prime number p such that*

$$n < p \leq n \left(1 + \frac{1}{2 \ln^2 n}\right).$$

In particular, if we apply Proposition 12 exactly $c + 1$ times, we obtain that

$$\left| \left\{ n < p \leq n \left(1 + \frac{1}{2 \ln^2 n}\right)^{c+1} \mid p \text{ is prime} \right\} \right| \geq c + 1, \quad \text{for every } n > 3275.$$

Now, let us see that for every sufficiently large n it holds that

$$n \left(1 + \frac{1}{2 \ln^2 n}\right)^{c+1} \leq K(n, n),$$

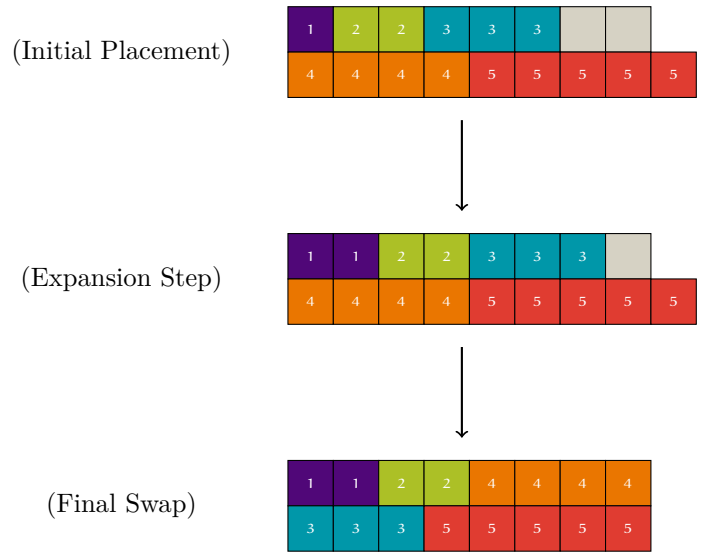
which will be enough to conclude. Indeed, recall that by Claim 8 we have that $K(n, n) \geq 1.4n$ for $n \geq 100$, and hence it only remains for us to show that for sufficiently large n we have

$$\left(1 + \frac{1}{2 \ln^2 n}\right)^{c+1} \leq 1.4,$$

which must be true since the LHS is monotonically decreasing in n and its limit when n goes to infinity is 1. \triangleleft

► **Theorem 13.** *For every even $n \geq 2$, the $2 \times \frac{n^2}{2}$ grid always admits a perfect game of PACKIT!.*

14:10 PackIt!: Gamified Rectangle Packing



■ **Figure 4** Illustration of Case 1 for the proof of Theorem 13, for $n = 4$. In this case $\gamma_n = 1$.

Proof. The proof is constructive. Let $K := K\left(2, \frac{n^2}{2}\right)$. As a first step, we place the first $n - 1$ rectangles (i.e., $1 \times t$ for $t \in \{1, \dots, n - 1\}$) in the first row, one after another, thus covering the first $\frac{n(n-1)}{2} < \frac{n^2}{2}$ cells of the first row. Some of these rectangles will be *expanded* later in order to fill up the first row, meaning that the rectangle $1 \times t$, used in turn t will be replaced by a rectangle $1 \times (t + 1)$. The remaining $K - n - 1$ rectangles, for $t \geq n$, will be placed on the second row. We might have to move some rectangles from the first row to the second row or vice-versa. The proof proceeds by cases over $\gamma\left(2, \frac{n^2}{2}\right)$, which we will abbreviate by γ_n to alleviate notation.

(Case 1: $\gamma_n \leq \frac{n}{2}$). As introduced earlier, the first step is to place the first $n - 1$ rectangles in the row one and the rest in row two. For the moment we do not care if row two is too long or row one too short; we will deal with that in a moment. Next, expand the first γ_n rectangles of row one. Originally, row one was $\frac{n^2}{2} - \frac{n(n-1)}{2} = \frac{n}{2}$ cells too short, and after the expansion of the first γ_n rectangles it is $\frac{n}{2} - \gamma_n$ cells too short. By Lemma 3, the γ_n expansions in row one, guarantee that the total area of rectangles in row one and two adds up to exactly n^2 . As a result, row two must be exactly $\frac{n}{2} - \gamma_n$ cells too large. If γ_n were to be exactly $\frac{n}{2}$, we would be done immediately. Otherwise, we will swap a rectangle from row one with a rectangle from row two. Indeed, note that $r_{\frac{n}{2} + \gamma_n}$, the $1 \times \frac{n}{2} + \gamma_n$ rectangle, is still on row one, and it was not expanded. Therefore, we can swap $r_{\frac{n}{2} + \gamma_n}$ (from row one) with r_n (from row two). As a result, row one has grown by $n - \left(\frac{n}{2} + \gamma_n\right) = \frac{n}{2} - \gamma_n$ cells, and row two has shrunk by the same amount. Therefore both rows have reached their desired length. This case is illustrated in Figure 4.

(Case 2: $\frac{n}{2} < \gamma_n < n - 1$). As before, placing the first $n - 1$ rectangles in row one makes the first row $\frac{n}{2}$ cells too short. Then, if we place rectangles r_n, \dots, r_K in row two, given that in total γ_n expansions are required to achieve the total desired area (Lemma 3), it must be the case that row two is $\gamma_n - \frac{n}{2}$ cells too short. Naively, we would simply expand $\frac{n}{2}$ rectangles in the first row, and $\gamma_n - \frac{n}{2}$ in the second row. However, the second row might contain fewer than $\gamma_n - \frac{n}{2}$ rectangles. To address this, we will transfer a rectangle from row one to row

two, and perform more expansions on row one, which concentrates most of the rectangles. Let us identify which rectangle will be moved from row one to row two. Let us define

$$i = \gamma_n - \frac{n}{2}.$$

Transfer r_i from row one to row two, and expand the first $\gamma_n < n - 1$ of the rectangles in row one. Since γ_n expansions have been made, the total area is exactly $\frac{n^2}{2}$, and thus it only remains to argue that the top row has exactly $\frac{n^2}{2}$ cells covered. This is indeed the case as

$$\frac{n(n-1)}{2} + \gamma_n - i = \frac{n(n-1)}{2} + \gamma_n - \left(\gamma_n - \frac{n}{2}\right) = \frac{n^2}{2}.$$

(Case 3: $\gamma_n \geq n - 1$). Place the first $n - 1$ rectangles in row one and the rest in row two. Expanding all $n - 1$ rectangles in the first row, and then expand $\gamma_n - (n - 1)$ rectangles in the second row. Let $i = \frac{n}{2} - 2$ (if $n \in \{2, 4\}$, the result can be checked manually, and therefore we assume $i \geq 1$ is a valid index for a rectangle). Move rectangle r_i from row one to row two. As in the previous case, it only remains to argue that the number of cells in the top row is exactly $\frac{n^2}{2}$. This is indeed the case as the number is determined by

$$\frac{n(n-1)}{2} + (n-1) - (i+1) = \frac{n(n-1)}{2} + (n-1) - \left(\frac{n}{2} - 1\right) = \frac{n^2}{2}.$$

Having covered all cases, we conclude the entire proof. ◀

3 Complexity Results

In turn t of a game of PACKIT!, the turn in which each of the already placed rectangles was packed into the grid is irrelevant, and therefore a *partially filled grid* G of dimensions $n \times n$ can be represented as an $n \times n$ matrix over $\{0, 1\}$. We will assume this representation uses $O(n^2)$ bits. Consider now the following problem:

PROBLEM: : SolitairePACKIT!
 INPUT : A partially filled grid G , and a turn number t given in binary.
 OUTPUT : Whether it is possible to complete a perfect packing for G starting from turn t .

We will analyze the complexity of SolitairePACKIT! next, but before that, let us remark that the definition of the problem does not require the partial filling of G to be achievable in $t - 1$ turns. We leave the complexity of SolitairePACKIT! with the additional restriction that G must be achievable in $t - 1$ turns as an open problem. That being said, we can present our main complexity result.

► **Theorem 14.** *SolitairePACKIT! is NP-complete.*

Proof. Let $n \times n$ be the the dimensions of G . Membership in NP is easy to see: the certificate is a description of the turns $t, \dots, t + m$, where $m = K(n, n) \leq n^2$, and it suffices to check that at each turn $t + i$, a rectangle of the appropriate area was placed without overlapping with any of the previously placed rectangles. For hardness, we reduce from a variant of the well-known 3 partition problem, proven to be NP-hard by Hulett, Will and Woeginger [12]. The overall reduction is inspired by the analysis of Tetris by Breukelaar et al. [5]. Consider the Restricted-3-Partition problem defined as follows.

14:12 PackIt!: Gamified Rectangle Packing

PROBLEM: :	Restricted-3-Partition
INPUT :	A set of integers, $\mathcal{A} = \{\alpha_1, \dots, \alpha_n\}$, with n a multiple of 3, such that if we define $T := \frac{\sum_{i=1}^n \alpha_i}{(n/3)}$, then $T/4 < \alpha_i < T/2$ for every $i \in [n]$.
OUTPUT :	Whether it is possible to partition \mathcal{A} into $n/3$ sets of 3 elements, all of them having sum exactly T .

Consider now the 4-Restricted-3-Partition, defined exactly as above but with the additional restriction that all numbers α_i are multiples of 4. This additional restriction preserves NP-hardness as every 3-partition P defined as

$$\{4\alpha_1, \dots, 4\alpha_n\} \xrightarrow{P} (\{4\alpha_i, 4\alpha_j, 4\alpha_k\}, \dots, \{4\alpha_x, 4\alpha_y, 4\alpha_z\})$$

is in a one-to-one correspondence with a 3-partition P' defined as

$$\{\alpha_1, \dots, \alpha_n\} \xrightarrow{P'} (\{\alpha_i, \alpha_j, \alpha_k\}, \dots, \{\alpha_x, \alpha_y, \alpha_z\}).$$

We can therefore reduce directly from 4-Restricted-3-Partition. Let \mathcal{A} be an input instance of 4-Restricted-3-Partition. We now show how to construct an associated instance of SolitairePACKIT!. First, we will present the required *gadgets*, which are illustrated in Figure 5.

E-gadgets. An E -gadget consists of a $T \times 3$ grid, in which the first and third column are completely filled, whereas the middle column is completely empty (hence $E(\text{mpty})$ -gadget). An illustration is presented in Figure 5b.

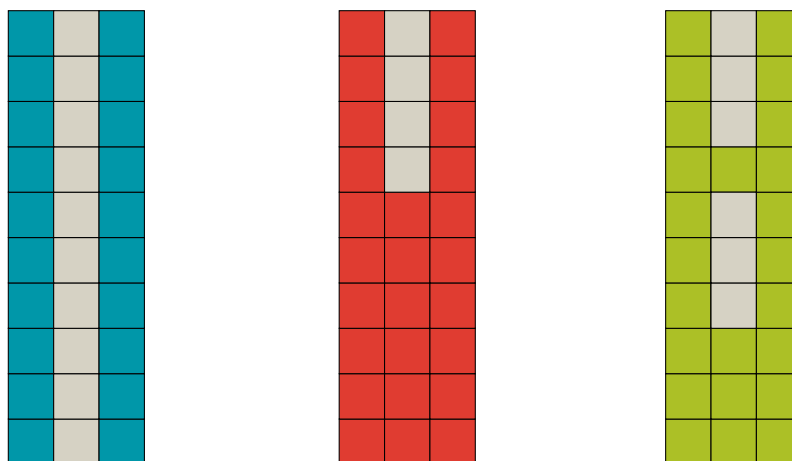
S-gadgets. Given an integer $\alpha \geq 1$, an $S(\alpha)$ -gadget consists of a $T \times 3$ grid, in which the first and third column are completely filled, whereas only the bottom $T - \alpha$ rows of the middle column are filled. In other words, $S(\alpha)$ -gadgets have a *single* “hole” of $\alpha \times 1$, hence their name. An illustration is presented in Figure 5b.

D-gadgets. Given an integer $\alpha \geq 1$, a $D(\alpha)$ -gadget consists of a $T \times 3$ grid, in which the first and third column are completely filled, and the middle column is filled only at row $\alpha + 1$ and rows $\{2\alpha + 2, 2\alpha + 3, \dots, T\}$. In other words, $D(\alpha)$ -gadgets have two “holes” of $\alpha \times 1$, i.e., a *double* hole, hence their name. An illustration is presented in Figure 5c.

With these gadgets, we can now construct a $(T + n) \times (T + n)$ grid as follows. First, horizontally concatenate exactly $n/3$ identical E -gadgets. Next, concatenate an $S(1)$ -gadget to the right of the current construction. Then, for every odd value m such that $3 \leq m < \max(\mathcal{A})$, concatenate a $D(m)$ -gadget to the right of the current construction if $m - 1 \notin \mathcal{A}$, and instead an $S(m + 1)$ -gadget to the right of the current construction otherwise.

Afterwards, if the resulting grid has length $T \times T'$, we complete a $T \times (T + n)$ grid by concatenating a $T \times (T + n - T')$ completely filled grid to the right of the current construction. This is well-defined, meaning that $T' < T + n$, as we show next. First, consider that, as each gadget uses exactly 3 columns, we have

$$\begin{aligned} T' &= 3 \cdot n/3 + 3 \cdot |\{3 \leq m < \max(\mathcal{A}) \mid m \text{ is odd}\}| \\ &\leq n + 3 \left\lceil \frac{\max(\mathcal{A}) - 3}{2} \right\rceil < n + 3 \frac{\max(\mathcal{A})}{2}. \end{aligned}$$



(a) An E -gadget. (b) An $S(4)$ -gadget. (c) A $D(3)$ -gadget.

■ **Figure 5** Illustration of the gadgets for $T = 10$.

Next, consider that

$$T = \left(\sum_{\alpha \in \mathcal{A}} \alpha \right) / (n/3) \leq 3 \max(\mathcal{A}).$$

Then, as $\max(\mathcal{A}) \leq T/2$ by the definition of 4-Restricted-3-Partition, we have

$$T' \leq n + 3 \cdot \frac{\max(\mathcal{A})}{2} \leq n + 3 \cdot \frac{T}{4} < T + n.$$

Finally, to go from the resulting $T \times (T + n)$ grid to a $(T + n) \times (T + n)$ grid it suffices to concatenate a completely filled $n \times (T + n)$ grid at the bottom of the previous grid. This construction is illustrated in Figure 6. We are now ready to prove the correctness of our reduction. Let $G_{\mathcal{A}}$ be the $(T + n) \times (T + n)$ grid constructed by the above process.

► **Lemma 15.** *The instance $(G_{\mathcal{A}}, 1)$ is a Yes-instance for SolitairePACKIT! if and only if \mathcal{A} is a Yes-instance for 4-Restricted-3-Partition.*

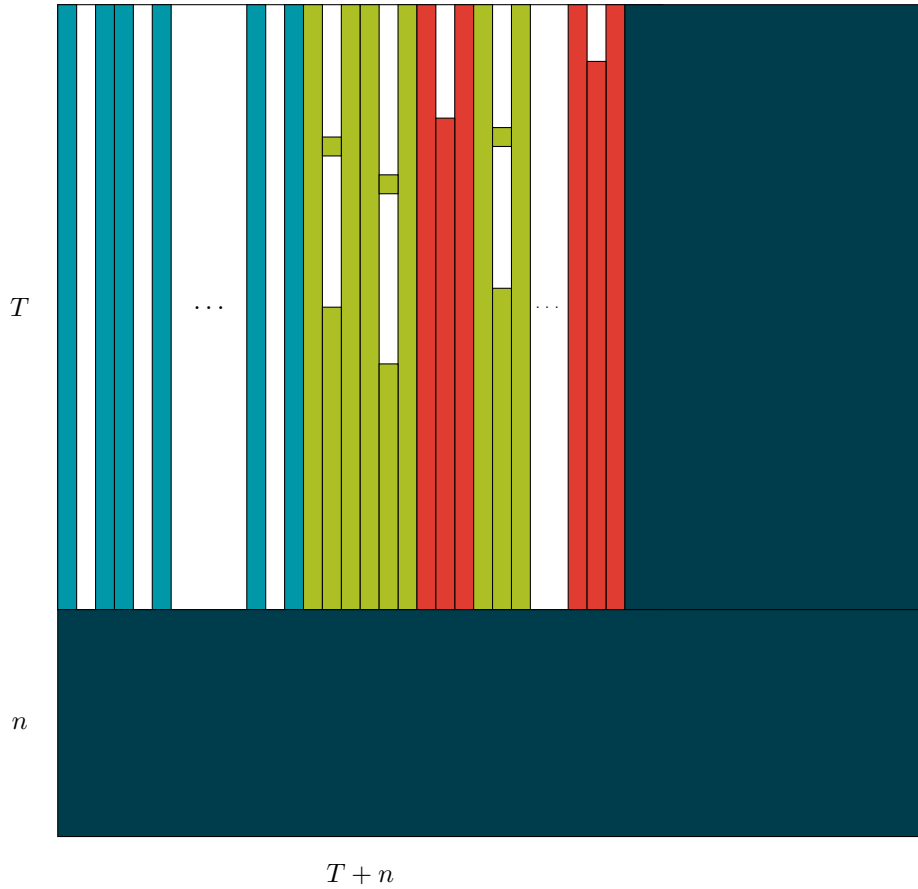
Proof. (\Leftarrow) Let us start with the backward direction since it is simpler. Assume there is a solution to the partition problem with sets $S_1, \dots, S_{n/3}$, where each set has exactly 3 elements and its sum is exactly T . Then, we can complete a perfect packing of G as follows. On each turn $1 \leq t \leq \max(\mathcal{A})$:

Case I) If $t \in \mathcal{A}$, then let i be the index such that $t = \alpha_i$, and j be the index of the set S_j such that $\alpha_i \in S_j$. Then, on this turn we can place a rectangle of dimensions $t \times 1$ into the j -th E -gadget of $G_{\mathcal{A}}$.

Case II) If $t = 4k$ for some positive integer k but $t \notin \mathcal{A}$, then by construction there is a $D(t + 1)$ -gadget, which can be filled by placing a $(t + 1) \times 1$ rectangle on this turn, and a $(t + 1) \times 1$ rectangle on the next turn.

Case III) If $t = 4k + 1$ and $t - 1 \in \mathcal{A}$, then by construction there is an $S(t + 1)$ -gadget, which can be filled by placing a $(t + 1) \times 1$ rectangle on this turn.

Case IV) If $t = 4k + 1$ for some integer k , and $t - 1 \notin \mathcal{A}$, then this turn has been covered in Case II).



■ **Figure 6** Illustration of the construction of $G_{\mathcal{A}}$ for Theorem 14. Note that n could be larger than T , and thus this figure is not necessarily in scale.

Case V) If $t = 4k + 2$ for some integer k , then by construction there is a $D(t + 1)$ -gadget, which can be filled by placing a $(t + 1) \times 1$ rectangle on this turn, and a $(t + 1) \times 1$ rectangle on the next turn.

Case VI) If $t = 4k + 3$ then this turn has been covered in Case V).

As a result of the turns of Case I), every E -gadget will be completely filled since by definition, if $\alpha_i, \alpha_k, \alpha_\ell \in S_j$, then $\alpha_i + \alpha_k + \alpha_\ell = T$. As there are exactly $n/3$ identical E -gadgets in $G_{\mathcal{A}}$, they will all be filled. Note as well that the gadgets used in every case are different. In particular, the only S -gadgets in the construction are for $t + 1 = 4k + 2$ with $t - 1 \in \mathcal{A}$, which are all used by Case III). Similarly, all $D(m)$ -gadgets for $m = 4k + 1$ for some integer k are used by Case II), whereas all $D(m)$ -gadgets for $m = 4k + 3$ are used by Case V). Given all gadgets are perfectly filled up, we have a perfect packing of $G_{\mathcal{A}}$.

(\implies) For the forward direction, assume it is possible to perfectly pack the grid $G_{\mathcal{A}}$ starting from turn 1. Let $G_{\mathcal{A}}^P$ be any perfect packing completing $G_{\mathcal{A}}$. Note immediately that by construction, every rectangle placed in $G_{\mathcal{A}}^P$ from turn 1 onward must have dimension $t \times 1$ for some positive integer t . Intuitively, we will now prove that the choices made in the backward direction of the proof are forced.

► **Definition 16.** For any turn $t \geq 1$, we say the rectangle placed in $G_{\mathcal{A}}^P$ on turn t is proper if either

1. $t = 1$, and the rectangle placed in $G_{\mathcal{A}}^P$ on this turn was a 1×1 rectangle placed in the only $S(1)$ -gadget of $G_{\mathcal{A}}$.
2. $t > 1$ is odd, and $t - 1 \in \mathcal{A}$, and the rectangle placed in $G_{\mathcal{A}}^P$ on this turn was a $(t + 1) \times 1$ placed in the only $S(t + 1)$ -gadget of $G_{\mathcal{A}}$.
3. $t > 1$ is odd and $t - 1 \notin \mathcal{A}$, and the rectangle placed in $G_{\mathcal{A}}^P$ on this turn was a $t \times 1$ placed in one of the two spaces of the only $D(t)$ -gadget of $G_{\mathcal{A}}$.
4. $t \in \mathcal{A}$, and the rectangle placed in $G_{\mathcal{A}}^P$ on this turn was placed in one of the E -gadgets.
5. t is even but $t \notin \mathcal{A}$, and the rectangle placed in $G_{\mathcal{A}}^P$ on this turn was a $(t + 1) \times 1$ placed in one of the two spaces of the only $D(t + 1)$ -gadget of $G_{\mathcal{A}}$.

▷ **Claim 17.** Every turn $t \geq 1$ where a rectangle was placed in $G_{\mathcal{A}}^P$ must have been proper.

Proof of Claim 17. We prove the claim by induction on t . The base case is $t = 1$, for which a single $S(1)$ -gadget exists in the construction, and given that the 1×1 empty space in this gadget must be filled in $G_{\mathcal{A}}^P$, the only turn on which it can be filled is turn 1. Therefore the base case works. For the inductive case, assume the claim holds up to t and let us show it holds for $t + 1$.

- If $t + 1$ is odd and $t \in \mathcal{A}$, then we claim the rectangle placed on turn $t + 1$ must have been a $(t + 2) \times 1$ rectangle in the only $S(t + 2)$ -gadget of $G_{\mathcal{A}}$. Indeed, if this were not the case, said gadget could only have been filled by a $(t + 2) \times 1$ rectangle placed on turn $(t + 2)$, since all previous turns have been proper and thus not placed anything in the $S(t + 2)$ -gadget. However, given there are two empty spaces of size $(t + 3)$ into the only $D(t + 3)$ -gadget of $G_{\mathcal{A}}$ (which must exist since $t \in \mathcal{A} \implies t + 2 \notin \mathcal{A}$ as all elements of \mathcal{A} are multiples of 4), and no previous turns could have placed anything into them as they are proper by inductive hypothesis, then we conclude that on turn $(t + 2)$ a rectangle of size $(t + 3)$ must have been placed into the only $D(t + 3)$ -gadget of $G_{\mathcal{A}}$.
- If $t + 1$ is odd and $t \notin \mathcal{A}$, then given all the previous turns have been proper, it must be that the only $D(t + 1)$ -gadget of $G_{\mathcal{A}}$ has only received a $(t + 1) \times 1$ rectangle placed on turn t , according to (5) in the definition of proper turn. Therefore, a single $(t + 1) \times 1$ empty space remains in the only $D(t + 1)$ -gadget of $G_{\mathcal{A}}$, and it must be that is filled on this turn, as any posterior turns will have rectangles of area at least $t + 2$.
- If $t + 1$ is even but $t + 1 \notin \mathcal{A}$, then given all turns so far have been proper, there are two empty $(t + 2) \times 1$ spaces in the only $D(t + 2)$ -gadget of $G_{\mathcal{A}}$, and given none can be filled after turn $t + 3$, and at most one can be filled in turn $t + 2$, we conclude that turn $t + 1$ must fill one.
- If $t + 1 \in \mathcal{A}$, and this turn were to be improper, then the rectangle placed on this turn must be placed either in an $S(t')$ -gadget or in a $D(t')$ -gadget.

In either case we will reach a contradiction. Note first that $t' > t + 2$: in the construction of $G_{\mathcal{A}}$, as t is odd and $t - 1 \notin \mathcal{A}$, when $m = t$ a $D(t)$ -gadget was created, and the next gadget created is a $S(t + 3)$ -gadget when $m = t + 2$, since $m - 1 \in \mathcal{A}$. Next, note that the remaining empty space on the $S(t')$ -gadget or the $D(t')$ -gadget partially filled on turn $t + 1$ must be at least $t' - (t + 2) > 0$. If $t' - (t + 2) < t + 2$, then that remaining empty space can never be filled in posterior turns, where all rectangles have area at least $t + 2$, a contradiction. Otherwise, $t' - (t + 2) > t + 1$, meaning that $t' > 2t + 3$. Because an $S(t')$ -gadget or a $D(t')$ -gadget exists, we deduce from the construction that $t' \leq \max(\mathcal{A})$. This implies that

$$\max(\mathcal{A}) > t' - 1 > 2t + 2 = 2(t + 1),$$

14:16 PackIt!: Gamified Rectangle Packing

meaning that two elements of \mathcal{A} , namely

$$\alpha_i := \max(\mathcal{A}), \quad \alpha_j := t + 1,$$

hold $\alpha_i > 2\alpha_j$. But by definition of 4-Restricted-3-Partition that would imply the following contradiction:

$$T/4 < \alpha_j < \alpha_i/2 < (T/2)/2 = T/4. \quad \triangleleft$$

By Claim 17, we have that for every $\alpha_t \in \mathcal{A}$, a rectangle of area α_t has been placed inside an E -gadget. Given that $T/2 < \alpha_t < T/4$ for every t , there must be exactly 3 rectangles placed inside every E -gadget. Let $\alpha_i^{(1)}, \alpha_i^{(2)}, \alpha_i^{(3)}$ be the areas of the three rectangles placed inside the i -th E -gadget. As for every i , by hypothesis, the i -th E -gadget is perfectly filled and had t empty cells to be filled, we conclude that $\alpha_i^{(1)} + \alpha_i^{(2)} + \alpha_i^{(3)} = T$, from where it follows that \mathcal{A} is a Yes-instance to the 4-Restricted-3-Partition problem. This concludes the proof of Lemma 15. \blacktriangleleft

Given the reduction presented above can clearly be carried out in polynomial time, we conclude hardness from the correctness proved in Lemma 15, and consequently this finishes the entire proof of Theorem 14. \blacktriangleleft

4 Computing Perfect PackIt! games

Even though Theorem 14 does not directly imply that it is hard to find perfect packings for an $n \times n$ grid (or to decide whether such a packing exist), it arguably gives evidence for this being a hard combinatorial challenge.

In many combinatorial problems SAT-solving can dramatically outperform backtracking approaches. This also happens to be the case for computing perfect PACKIT! games, where even after several optimizations, a backtracking approach only allowed us to find perfect packings up to $n = 20$. In contrast, by using a novel SAT encoding technique we were able to find perfect packings up to $n = 50$ in under 24 hours of computation. As in Section 2, we divide the problem into two stages: (i) finding a set of rectangles (h_t, v_t) such that

- Their total area is n^2 , meaning that $\sum_t h_t \cdot v_t = n^2$.
- The t -th rectangle has area t or $t + 1$, meaning that $h_t \cdot v_t \in \{t, t + 1\}$ for every t .
- All rectangles fit into the $n \times n$ grid, meaning that $\max(h_t, v_t) \leq n$.

and (ii), packing the rectangles obtained in the previous stage without overlaps. Note that due to the area condition, if a valid rectangle selection is packed without overlapping, then they must cover the entire $n \times n$ grid.

For stage (i), we use a pseudo-polynomial dynamic programming approach, similar to the one used for the standard subset sum problem. For stage (ii) we use a sophisticated SAT encoding that uses only $O(n^3)$ many clauses as opposed to the naive $O(n^4)$ encoding. Due to space constraints, both the dynamic programming formulation and the SAT encoding is presented in the extended arXiv version of this paper, at <https://arxiv.org/abs/2403.12195>.

4.1 Computational Results

All experiments have been run on a personal computer with the following specifications:

- MacBook Pro M1, 2020, running Sonoma 14.3
- 16GB of RAM
- 8 cores (but all experiments were run in a single thread).

In terms of software, we experimented with different SAT-solvers, and obtained the best results using the award-winning solver Kissat [3]. We tested every value of n between 5 and 50 and such that neither Theorem 4 nor Theorem 5 applies, and for every value we were able to find a perfect game of PACKIT! in under 24 hours. For each such value, we used the dynamic programming approach to generate a valid selection of rectangles, and simply used the first one obtained. Given the number of valid selections of rectangles is likely exponential in n , it could be that some valid selections are significantly easier to pack than others. The fact that we obtained perfect packings simply using the first valid rectangle selection obtained via dynamic programming confirms the robustness of the SAT approach.

Detailed results are presented in Table 1. As it is common for families of satisfiable formulas, the runtime is not strictly monotone with n , even though the size of the encoding is (both the number of variables and clauses).

■ **Table 1** Computational results for $n \in \{5, \dots, 50\}$. Perfect packings for $n \in \{1, \dots, 4\}$ are trivial.

	n	#vars	#clauses	SAT runtime
	5	141	424	0.0s
(Theorem 4 applies)	6	-	-	-
	7	297	1101	0.0s
	8	375	1482	0.0s
	9	510	2228	0.02s
	10	611	2797	0.02s
	11	780	3921	0.02s
	12	904	4732	0.03s
	13	1037	5673	0.19s
	14	1254	7375	0.16s
	15	1410	8584	0.04s
	16	1661	10838	0.56s
	17	1840	12397	0.20s
(Theorem 5 applies)	18	-	-	-
	19	2327	17184	0.20s
	20	2538	19339	2.47s
	21	2871	23037	2.08s
	22	3105	25582	2.04s
(Theorem 4 applies)	23	-	-	-
	24	3729	33117	4.43s
	25	3995	36396	2.80s
	26	4410	41980	2.69s
	27	4699	45737	23.21s
	28	5148	52283	8.45s
	29	5460	56636	17.24s
(Theorem 5 applies)	30	-	-	-
	31	6278	69109	34.26s
	32	6622	74340	48.17s
	33	7153	83288	36.37s
	34	7520	89207	107.23s
(Theorem 4 applies)	35	-	-	-
	36	8475	105934	747.46s
	37	8874	112997	194.33s
	38	9487	124629	502.20s
	39	9909	132324	442.62s
	40	10556	145392	129.71s
	41	11001	153969	6117.58s
	42	11455	162890	2088.45s
	43	12150	177744	923.03s
	44	12627	187501	579.50s
	45	13356	203857	3185.11s
	46	13856	214540	2188.39s
(Theorem 5 applies)	47	-	-	-
	48	15142	244107	48102.44s
	49	15674	256188	23337.97s
	50	16485	276182	15925.77s

5 Concluding Remarks

We have analyzed several aspects of PACKIT!:

1. Every $2 \times \frac{n^2}{2}$ grid admits a perfect PACKIT! game.
2. For every $n \leq 50$ such that neither Theorem 4 nor Theorem 5 applies, the $n \times n$ grid admits a perfect PACKIT! game. In other words, Conjecture 6 is true for all values of $n \leq 50$.

We hope that both our mathematical and computational techniques can be applicable to similar packing problems. The “Mondrian Art Puzzle” [9, 14] asks for perfect packings of $n \times n$ grids but where all rectangles must use the same area. Recently, the *MIT CompGeom Group* has studied perfect packings for rectangular grids with square pieces [11]. Then, in terms of concrete PACKIT! questions, we pose the following challenges:

1. Prove or refute Conjecture 6.
2. Is there always a perfect packing of the $m \times n$ grid when $\gamma(m, n) = K(m, n)/2$? In this case, exactly half of the turns are expansion turns. In particular, this might be easier to show assuming m and n are even.
3. What is the complexity of PACKIT! as a 2-player game? It is well known that complexity tends to increase in 2-player formulations (see e.g., [7]), so could PACKIT! be complete for the class PSPACE?



In terms of our web implementation of PACKIT!, future work includes the design of an online multiplayer mode, and AIs that could be faced as opponents.

References

- 1 Jér my Barbay and Bernardo Subercaseaux. The Computational Complexity of Evil Hangman. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms (FUN 2021)*, volume 157 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:12, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum f r Informatik. doi:10.4230/LIPIcs.FUN.2021.23.
- 2 Edward J Barbeau. *Pell’s Equation*. Problem Books in Mathematics. Springer, New York, NY, 2003 edition, January 2003.
- 3 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In Tomas Balyo, Nils Froyks, Marijn Heule, Markus Iser, Matti J rvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
- 4 Charles L. Bouton. Nim, A Game with a Complete Mathematical Theory. *Annals of Mathematics*, 3(1/4):35–39, 1901. doi:10.2307/1967631.
- 5 Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kusters, and David Liben-Nowell. Tetris is Hard, Even to Approximate. *International Journal of Computational Geometry & Applications*, 14:41–68, April 2004. doi:10.1142/S0218195904001354.
- 6 Kevin Buchin, Mart Hagedoorn, Irina Kostitsyna, and Max van Mulken. Dots & boxes is PSPACE-complete, 2021. arXiv:2105.02837.
- 7 Erik. D Demaine, William Gasarch, and Mohammad Hajiaghayi. Computational Intractability: A Guide to Algorithmic Lower Bounds. <https://hardness.mit.edu/>.
- 8 Pierre Dusart. *Autour de la fonction qui compte le nombre de nombres premiers*. PhD thesis, Universit  de Limoges, 1998. Th se de doctorat dirig e par Robin, Guy Math matiques appliqu es. Th orie des nombres Limoges 1998. URL: <http://www.theses.fr/1998LIM00007>.

- 9 Natalia García-Colín, Dimitri Leemans, Mia Müßig, and Érika Roldán. There is no perfect mondrian partition for squares of side lengths less than 1001. *arXiv preprint arXiv:2311.02385*, 2023.
- 10 Martin Gardner. Mathematical games. *Scientific American*, 223(4):120–123, October 1970. doi:10.1038/scientificamerican1070-120.
- 11 MIT CompGeom Group, Zachary Abel, Hugo A. Akitaya, Erik D. Demaine, Adam C. Hesterberg, and Jayson Lynch. When can you tile an integer rectangle with integer squares?, 2023. arXiv:2308.15317.
- 12 Heather Hulett, Todd G. Will, and Gerhard J. Woeginger. Multigraph realizations of degree sequences: Maximization is easy, minimization is hard. *Operations Research Letters*, 36(5):594–596, September 2008. doi:10.1016/j.orl.2008.05.004.
- 13 Gary McGuire, Bastian Tugemann, and Gilles Civario. There is no 16-clue sudoku: Solving the sudoku minimum number of clues problem, 2013. arXiv:1201.0749.
- 14 Cooper O’Kuhn. The mondrian puzzle: A connection to number theory, 2018. arXiv:1810.04585.
- 15 Lowell Schoenfeld. Sharper Bounds for the Chebyshev Functions $\theta(x)$ and $\psi(x)$. II. *Mathematics of Computation*, 30(134):337–360, 1976. doi:10.2307/2005976.

Polyamorous Scheduling

Leszek Gąsieniec  

University of Liverpool, UK

Benjamin Smith  

University of Liverpool, UK

Sebastian Wild  

University of Liverpool, UK

Abstract

Finding schedules for pairwise meetings between the members of a complex social group without creating interpersonal conflict is challenging, especially when different relationships have different needs. We formally define and study the underlying optimisation problem: Polyamorous Scheduling.

In Polyamorous Scheduling, we are given an edge-weighted graph and try to find a periodic schedule of matchings in this graph such that the maximal weighted waiting time between consecutive occurrences of the same edge is minimised. We show that the problem is NP-hard and that there is no efficient approximation algorithm with a better ratio than $4/3$ unless $P = NP$. On the positive side, we obtain an $O(\log n)$ -approximation algorithm; indeed, an $O(\log \Delta)$ -approximation for Δ the maximum degree, i.e., the largest number of relationships of any individual. We also define a generalisation of density from the Pinwheel Scheduling Problem, “poly density”, and ask whether there exists a poly-density threshold similar to the $5/6$ -density threshold for Pinwheel Scheduling [Kawamura, STOC 2024]. Polyamorous Scheduling is a natural generalisation of Pinwheel Scheduling with respect to its optimisation variant, Bamboo Garden Trimming.

Our work contributes the first nontrivial hardness-of-approximation reduction for any periodic scheduling problem, and opens up numerous avenues for further study of Polyamorous Scheduling.


2012 ACM Subject Classification Theory of computation \rightarrow Problems, reductions and completeness; Theory of computation \rightarrow Scheduling algorithms

Keywords and phrases Periodic scheduling, Pinwheel scheduling, Edge-coloring, Chromatic index, Approximation algorithms, Hardness of approximation

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.15

Related Version *Full Version*: <https://arxiv.org/abs/2403.00465>

Funding *Sebastian Wild*: Engineering and Physical Sciences Research Council grant EP/X039447/1.

Acknowledgements We are grateful to Viktor Zamaraev for setting us on the right track with the chromatic-index problem, and for several fruitful initial discussions. We also wish to thank Casper Moldrup Rysgaard, Justin Dallant, and Oliver Kim for their helpful contributions; especially Casper, who acted as our rubber duck for a brutally unpolished version of the original hardness-of-approximation reduction. 

1 Introduction

We study a natural periodic scheduling problem faced by groups of regularity-loving polyamorous people: Consider a set of persons and a set of pairwise relationships between them, each with a value representing its neediness, importance, or emotional weight. Find a periodic schedule of pairwise meetings between couples that minimizes the maximal weighted waiting time between such meetings, given that each person can meet with at most one of their partners on any particular day.

Before formally defining the Polyamorous Scheduling Problem (Poly Scheduling for short), we illustrate some features of the problem on an example. Figure 1 shows an instance using the natural graph-based representation: We have vertices for people and weighted



© Leszek Gąsieniec, Benjamin Smith, and Sebastian Wild;
licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 15; pp. 15:1–15:18

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

15:2 Polyamorous Scheduling

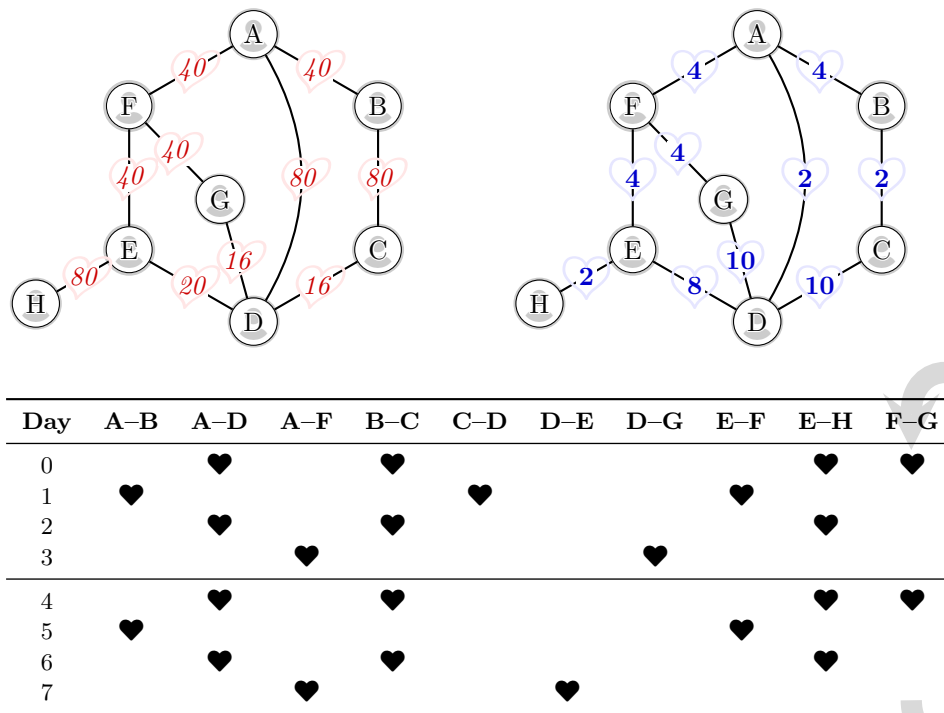


Figure 1 An example Optimisation Polyamorous Scheduling instance with 8 persons: Adam, Brady, Charlie, Daisy, Eli, Frankie, Grace, and Holly. **Top left:** Graph representation with *edge labels* showing the weight (desire growth rates) of each pairwise relationship. **Bottom:** An optimal schedule for the instance. On each day, a set of meetings is scheduled as indicated by ♥s. The schedule has a period of 8 days: after day 7, we start from day 0 again. **Top right:** A decision version of the instance obtained for heat 160. The *edge labels* here are the frequencies with which edges have to be scheduled stay below heat 160.

(undirected) edges for relationships. It is easy to check that the schedule given at the bottom of Figure 1 never schedules more than one daily meeting for any of the 8 persons in the group; in the graph representation, the set of meetings for each day must form a matching. Each day the mutual desire for a meeting experienced by each couple grows by the weight or *desire growth rate* of that relationship¹ – that is, until a meeting occurs and their desire is reset to zero. We will refer to the highest desire ever felt by any pair when following a given schedule as the *heat* of the schedule. The heat of the schedule in Figure 1 is 160: as the reader can verify, no pair ever feels a desire greater than 160 before meeting and resetting their desire to zero. Desire 160 is also attained; e.g., Adam and Daisy are scheduled to see each other every other day, but over the period of 2 days between subsequent meetings, their desire grows to $2 \cdot 80 = 160$.

For the instance in Figure 1, it is easy to show that no schedule with heat < 160 exists. For that, we first convert from desire growth rates to required *frequencies*: Under a heat-160 schedule, a pair with desire growth rate g must meet at least every $\lceil 160/g \rceil$ days. The top-right part of Figure 1 shows the result. It is easy to check that the given schedule indeed achieves these frequencies. However, any further reduction of the desired heat to

¹ “Remember, absence makes the heart grow fonder” [10].
<https://getyarn.io/yarn-clip/ae628721-c1d1-49d1-bd7c-78cbffceabf0>



160 – ε would leave, e.g., Adam hopelessly overcommitted: the relation with Daisy would get frequency $\lfloor (160 - \varepsilon)/80 \rfloor = 1$, forcing them to meet *every* day; but then Brady and Frankie, each with frequency $\lfloor (160 - \varepsilon)/40 \rfloor \leq 3$ cannot be scheduled at all.

While local arguments suffice for our small example, in general, Poly Scheduling is NP-hard (as shown below). We therefore focus this paper on approximation algorithms and inapproximability results.

1.1 Formal Problem Statement

We begin by defining a decision version of Poly Scheduling. In the *Decision Polyamorous Scheduling Problem*, we are given a set of people and pairwise relationships with “attendance frequencies” $f_{i,j}$, and we are trying to find a daily schedule of two-person meetings such that each couple $\{i, j\}$ meets at least every $f_{i,j}$ days. The only constraint on the number of meetings that can occur on any given day is that each person can only participate in at most one of them. A Decision Polyamorous Scheduling instance can naturally be modelled as a graph of people with the edges representing their relationships. Because each person can participate in at most one meeting per day, the edges scheduled on any given day must form a matching in this graph.

► **Definition 1** (Decision Polyamorous Scheduling (DPS)). *A DPS instance $\mathcal{P}_d = (P, R, f)$ (a “decision polycule”) consists of an undirected graph (P, R) where the vertices $P = \{p_1, \dots, p_n\}$ are n persons and the edges R are pairwise relationships, with integer frequencies $f : R \rightarrow \mathbb{N}$ for each relationship.*

The goal is to find an infinite schedule $S : \mathbb{N}_0 \rightarrow 2^R$, such that

- (1) *(no conflicts) for all days $t \in \mathbb{N}_0$, $S(t)$ is a matching in \mathcal{P}_d , and*
- (2) *(frequencies) for all $e \in R$ and $t \in \mathbb{N}_0$, we have $e \in S(t) \cup S(t+1) \cup \dots \cup S(t+f(e)-1)$; or to report that no such schedule exists. In the latter case, \mathcal{P}_d is called infeasible.*

We write $f_{i,j}$ and f_e as shorthands for $f(\{p_i, p_j\})$ resp. $f(e)$. An infinite schedule exists if and only if a *periodic* schedule exists, i.e., a schedule where there is a period $T \in \mathbb{N}$ such that for all t , we have $S(t) = S(t+T)$: any feasible schedule corresponds to an infinite walk in the finite configuration graph of the problem (see Section 3), implying the existence of a finite cycle. A periodic schedule can be finitely described by listing $S(0), S(1), \dots, S(T-1)$.

By relaxing the hard maximum frequencies of meetings between couples to “desire growth rates”, we obtain the Optimisation Polyamorous Scheduling (OPS) Problem. Our objective is to find a schedule that minimizes the “heat”, i.e., the *worst pain of separation* ever felt in the polycule by any couple.

► **Definition 2** (Optimisation Polyamorous Scheduling). *An OPS instance (or “optimisation polycule”) $\mathcal{P}_o = (P, R, g)$ consists of an undirected graph (P, R) along with a desire growth rate $g : R \rightarrow \mathbb{R}_{>0}$ for each relationship in R . An infinite schedule $S : \mathbb{N}_0 \rightarrow 2^R$ is valid if, for all days, $t \in \mathbb{N}_0$, $S(t)$ is a matching in \mathcal{P}_o .*

The goal is to find a valid schedule that minimizes the heat $h = h(S)$ of the schedule where $h(S) = \max_{e \in R} h_e(S)$ and

$$h_e(S) = \sup_{d \in \mathbb{N}} \begin{cases} (d+1) \cdot g(e) & \exists t \in \mathbb{N}_0 : e \notin S(t) \cup S(t+1) \cup \dots \cup S(t+d-1); \\ g(e) & \text{otherwise.} \end{cases}$$

As for DPS, S can be assumed to be periodic without loss of generality, meaning that S is finitely representable.

1.2 Related Work

Polyamorous Scheduling itself has not been studied to our knowledge. Other variants of periodic scheduling have attracted considerable interest recently [24, 18, 1], including FUN [4].

The simplest periodic scheduling problem is arguably *Pinwheel Scheduling*. In Pinwheel Scheduling [19] we are given k positive integer *frequencies* $f_1 \leq f_2 \leq \dots \leq f_k$, and the goal is to find a Pinwheel schedule, i.e., an infinite schedule of tasks $1, \dots, k$ such that any contiguous time window of length f_i contains at least one occurrence of i , for $i = 1, \dots, k$, (or to report the non-existence of such a schedule).

Pinwheel Scheduling is NP-hard [22], but unknown to be in NP [24], (see [14] for more discussion). Poly Scheduling inherits these properties.

The *density* of a Pinwheel Scheduling instance is given by $d = \sum_{i=1}^k 1/f_i$. It is easy to see that $d \leq 1$ is a necessary condition for A to be schedulable, but this is not sufficient, as the infeasible instance $(2, 3, M)$ with $d = \frac{5}{6} + 1/M$, for any $M \in \mathbb{N}$ shows. However, there is a threshold d^* so that $d \leq d^*$ implies schedulability: Whenever $d \leq \frac{1}{2}$, we can replace each frequency f_i by $2^{\lceil \lg(f_i) \rceil}$ without increasing d above 1; then a periodic Pinwheel schedule always exists using the largest frequency as period length. A long sequence of works [19, 6, 20, 26, 7, 13, 11, 14] successively improved bounds on d^* , culminating very recently in Kawamura’s proof [24] that it is indeed a sharp threshold, $d^* = \frac{5}{6}$, confirming the corresponding conjecture of Chan and Chin from 1993 [20]. Generalizations of Pinwheel Scheduling have also been studied, e.g., with jobs of different lengths [17, 12].

Pinwheel Scheduling is a special case of DPS, where the underlying graph (P, R) is a *star*, i.e., a centre connected to k pendant vertices with edges of frequencies f_1, \dots, f_k . Note that it is not generally possible to obtain a polyamorous schedule by combining the local schedule of each person²; see for example a triangle with edge frequencies 2: In the DPS instance $(\{A, B, C\}, \{A-B, B-C, A-C\}, f)$ with $f(e) = 2$ for all edges, the local problem for each person is feasible by alternating between their two partners, but the global DPS instance has no solution. This example also shows that the simple strategy of replacing f_i by $2^{\lceil \lg(f_i) \rceil}$ is not sufficient to guarantee the existence of a schedule for Poly Scheduling. Indeed, it is unclear whether any such constant-factor scaling of frequencies exists which applies to all Poly Scheduling instances.

There are two natural optimisation variants of Pinwheel Scheduling. In *Windows Scheduling* [3] tasks with frequencies are given and the goal is to find a perpetual scheduling that minimizes the *number* of tasks that need to be done *simultaneously* while respecting all frequencies (i.e., the number of channels or servers needed to schedule all tasks). Efficient constant-factor approximation algorithms are known that use the connection to Bin Packing [2] (where we bin tasks by used channels), even when the sets of tasks to schedule changes over time [8].

The *Bamboo Garden Trimming (BGT) Problem* [16, 15] retains the restriction of one task per day, but converts the frequencies into *growth rates* $g_1 \leq \dots \leq g_k$ (of k bamboo plants $1, \dots, k$) and asks to find a perpetual schedule that minimizes the *height* ever reached by any plant. BGT also allows efficient constant-factor approximations whose approximation factor has seen a lively race of successively improvements over last few years: from 2 [16] over $\frac{12}{7} \approx 1.71$ [28], 1.6 [15], and 1.4 [18], down to the current record, $\frac{4}{3} \approx 1.33$, again by Kawamura [24]. As for the Windows Scheduling problem, no hardness of approximation results are known. It remains open whether it is possible to obtain a PTAS for the Bamboo Garden Trimming Problem [15] or the Windows Scheduling Problem. We show that the same is not true for Poly Scheduling (see Theorem 3 below).

² The current state-of-the-art approach in practice, usually via Google Calendar.

As for Pinwheel Scheduling and DPS, Bamboo Garden Trimming is the special case of OPS on star graphs. Although BGT can be approximated well, since it is in general not possible to combine local schedules into a global schedule for a polycule (as noted above), it is not clear whether Poly Scheduling allows an efficient constant-factor approximation.

All mentioned problems above have simple *fractional* counterparts that are much easier to solve and hence provide necessary conditions. Indeed, this is the motivation for density in Pinwheel Scheduling: if we allow a schedule to spend arbitrary fractions of the day on different tasks, we obtain a schedule if and only if the density is at most 1. (Spending a $1/f_i$ fraction on task i each day is best possible). For Windows Scheduling, any valid schedule must partition the tasks into bins (channels/servers), so that each bin admits a Pinwheel schedule. Relaxing the latter constraint to “density at most 1” yields a standard bin packing problem, to which we can apply existing techniques; (packing bins only up to density $5/6$ guarantees a Pinwheel schedule, at the expense of a $6/5$ factor increase in channels). For Bamboo Garden Trimming, the optimal fractional schedule spends a G/g_i fraction of each day with task i , where G is the sum of all growth rates, thus achieving height exactly G . For Poly Scheduling, we can similarly define a fractional problem, but its structure is much richer (see Section 6).

There are further periodic scheduling problems with less direct connections to Poly Scheduling that received attention in the literature. Patrolling problems typically involve periodic schedules: for example, [1] finds schedules for a fleet of k identical robots to patrol (unweighted) points in a metric space, whereas the “Continuous BGT Problem” [15] sends a single robot to points with different frequencies requirements; [25] tasks k robots with patrolling a line or a circle. The underlying geometry in these problems requires different techniques from our work. The *Point Patrolling Problem* studied in [25] can be seen as a “covering version” of Pinwheel Scheduling: each day, we have to assign one of n workers to a single, daily recurring task, where worker i requires a break of a_i days before they can be made to work again. Yet another twist on a patrolling problem is the *Replenishment Problems with Fixed Turnover Times* given in [5], where vertices in a graph have to be visited with given frequencies, but instead of restricting the number of vertices that can be visited per day, the *length of a tour* to visit them (starting at a depot node) shall be minimized.

In the *Fair Hitting Sequence Problem* [9], we are given a collection of sets $\mathcal{S} = \{S_1, \dots, S_m\}$, each consisting of a subset of the set of elements $\mathcal{V} = \{v_1, \dots, v_n\}$. Each set S_j has an urgency factor g_j , which is comparable to the growth rates in BGT instances with one key difference: A set S_j is hit whenever any $v_i \in S_j$ is scheduled. The goal is again similar to BGT; to find a perpetual schedule of elements $v_i \in \mathcal{V}$ that minimizes the time between visits to each set S_j , weighted by g_j . There is also a decision variant, similar to Pinwheel Scheduling in that growth rates are replaced by frequencies. We use a similar layering technique in our approximation algorithm (Section 5) as the $O(\log^2 n)$ -approximation from [9], but we obtain a better approximation ratio for Poly Scheduling. Their $O(\log n)$ -approximation based on randomized rounding does not extend to Poly Scheduling since the used linear program has exponentially many variables for Poly Scheduling (Section 6).

1.3 Our Results

Despite the recent flurry of results on periodic scheduling, Polyamorous Scheduling seems not to have been studied before. Apart from its immediate practical applications, some quirks make Polyamorous Scheduling an interesting combinatorial optimization problem in its own right. The first version of this manuscript used a direct reduction from 3SAT to introduce the following hardness-of-approximation result, which rules out the existence of a PTAS (polynomial-time approximation scheme) for Optimisation Polyamorous Scheduling.

► **Theorem 3** (SAT Hardness of approximation). *Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Optimisation Poly Scheduling problem for any $\delta < \frac{1}{12}$.*

We retain this original proof in the appendix of the extended online version³, both for the record and because we expect future works to expand on the methods it develops. We have, however, since found a substantially simpler and stronger hardness-of-approximation result, Theorem 4, by containing the 3-Regular Chromatic Index Problem as a special case.

► **Theorem 4** (Hardness of approximation). *Unless $P = NP$, there is no polynomial-time $(1 + \delta)$ -approximation algorithm for the Optimisation Poly Scheduling problem for any $\delta < \frac{1}{3}$.*

Though the current form of Theorem 3 follows from Theorem 4, the direct 3SAT reduction is significantly more versatile and we hope to improve the lower bound on the approximation ratio in future work. The core idea of the reduction in Theorem 3 is to force any valid schedule to have a periodic structure with a 3-day period, where edges scheduled on days t with $t \equiv 0 \pmod{3}$ represent the value *True* and edges scheduled on days with $t \equiv 1 \pmod{3}$ represent *False*; the remaining slots, $t \equiv 2 \pmod{3}$, are required to enforce correct propagation along logic gadgets. The appendix, available online, includes a detailed construction of all gadgets, the proof of Theorem 3, and a worked example – the DPS instance corresponding to an example 3-CNF formula.

Theorems 3 and 4 of course imply the NP-hardness of Polyamorous Scheduling; overall, we have 3 independent reductions establishing this. Section 3 surveys these and shows that the best-known upper bound for the complexity of Polyamorous Scheduling is PSPACE. We could thus call Polyamorous Scheduling *very* NP-hard; yet, efficient approximation algorithms are possible. Finding an edge colouring and using a simple round-robin schedule of its colours yields a good approximation if *both* the maximum degree and the ratio between the smallest and the largest desire growth rates are small (Theorem 5).

► **Theorem 5** (Colouring approximation). *For an Optimisation Poly Scheduling instance $\mathcal{P}_o = (P, R, g)$ set $g_{\min} = \min_{e \in R} g(e)$, $g_{\max} = \max_{e \in R} g(e)$, and let Δ be the maximum degree in (P, R) and h^* be the heat of an optimal schedule. There is an algorithm that computes in polynomial time a schedule S of heat h with $\frac{h}{h^*} \leq \min\left\{\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}, \Delta + 1\right\}$.*

A fully general approximation seems only possible with much weaker ratios; we provide an $O(\log \Delta)$ -approximation by applying Theorem 5 to groups with similar weight and interleaving the resulting schedules.

► **Theorem 6** (Layering approximation). *For an Optimisation Poly Scheduling instance $\mathcal{P}_o = (P, R, g)$, let Δ be the maximum degree in (P, R) and h^* be the heat of an optimal schedule. There is an algorithm that computes in polynomial time a schedule S of heat h with $\frac{h}{h^*} \leq 3 \lceil \lg(\Delta + 1) \rceil = O(\log n)$, where $n = |P|$.*

Finally, we generalize the notion of density to Polyamorous Scheduling. As discussed above, density has proven instrumental in understanding the structure of Pinwheel Scheduling and in devising better approximation algorithms, by providing a simple, instance-specific lower bound. For Polyamorous Scheduling, the fractional problem is much richer, and indeed remains nontrivial to solve. We devise a generalization of density⁴ for Poly Scheduling from the dual of the Linear Program (LP) corresponding to a fractional variant of Polyamorous Scheduling, which gives the following instance-specific lower bound.

³ extended online version at <https://arxiv.org/abs/2403.00465>

⁴ Note that poly density describes how tightly the polycule packs meetings together, not the density of its members.

► **Theorem 7** (Fractional lower bound). *Let $\mathcal{P}_o = (P, R, g)$ be an OPS instance with optimal heat h^* . For any set of values $z_e \in [0, 1]$, for $e \in R$, with $\sum_{e \in R} z_e = 1$, we have*

$$h^* \geq \bar{h}(z) = \frac{1}{\max_{M \in \mathcal{M}} \sum_{e \in M} \frac{z_e}{g(e)}}$$

with the maximum ranging over the set of all inclusion-maximal matchings (\mathcal{M}) in (P, R) . The largest value \bar{h}^ of $\bar{h}(z)$ over all feasible z , is the poly density of \mathcal{P}_o .*

The bound implies (and formally establishes) simple ad-hoc bounds such as the following, which corresponds to the lower bound of G on the height in Bamboo Garden Trimming (setting $z_e = g(e)/G$).

► **Corollary 8** (Total growth bound). *Given an OPS instance $\mathcal{P}_o = (P, R, g)$ with optimal heat h^* , let $G = \sum_{e \in R} g(e)$ and m be the size of a maximum matching in (P, R) ; then $h^* \geq G/m$.*

More importantly though, Theorem 7 allows us to define a *poly density* similarly to the Pinwheel Scheduling Problem, and allows us to formulate the most interesting open problem about Poly Scheduling. For a DPS instance $\mathcal{P}_d = (P, R, f)$, define the poly density of \mathcal{P}_d , $\bar{h}^*(\mathcal{P}_d)$, as the poly density of the OPS instance $\mathcal{P}_o = (P, R, 1/f)$ (see also Lemma 10).

► **Open Problem 9** (Poly Density Threshold). *Is there a constant c such that every Decision Poly Scheduling instance $\mathcal{P}_d = (P, R, f)$ with poly density $\bar{h}^*(\mathcal{P}_d) \leq c$ admits a valid schedule?*

2 Preliminaries

In this section, we introduce some general notation and collect a few simple facts about Poly Scheduling used later.

We write $[n..m]$ for $\{n, n+1, \dots, m\}$ and $[n]$ for $[1..n]$. For a set A , we denote its powerset by 2^A . All graphs in this paper are simple and undirected. We denote by $\mathcal{M} = \mathcal{M}(V, E)$ the set of *inclusion-maximal matchings* in graph (V, E) , where matching has the usual meaning of an edge set with no two edges incident to the same vertex. By $\Delta = \Delta(V, E)$, we denote the *maximum degree* in (V, E) . A *pendant vertex* is a vertex with degree 1. The *chromatic index* $\chi_1 = \chi_1(V, E)$ is the smallest number C of “colours” in a proper edge colouring of (V, E) (i.e., the number of disjoint matchings required to cover E); by Vizing’s Theorem [29], we have $\Delta \leq \chi_1 \leq \Delta + 1$ for every graph. Misra and Gries provide a polynomial-time algorithm for edge colouring any graph using at most $\Delta + 1$ colours [27].

Given a schedule $S : \mathbb{N}_0 \rightarrow 2^R$ and an edge $e \in R$, we define the (*maximal*) *recurrence time* $r(e) = r_S(e)$ of e in S as the maximal time between consecutive occurrences of e in S , formally:

$$r_S(e) = \sup_{d \in \mathbb{N}} \begin{cases} d + 1 & \exists t \in \mathbb{N}_0 : e \notin S(t) \cup S(t+1) \cup \dots \cup S(t+d-1); \\ 0 & \text{otherwise.} \end{cases}$$

Using recurrence time, the heat $h = h(S)$ of a schedule S in an OPS instance (P, R, g) is $h(S) = \max_{e \in R} g(e) \cdot r(e)$. Clearly, for any schedule $S : \mathbb{N}_0 \rightarrow 2^R$, we can obtain $S' : \mathbb{N}_0 \rightarrow \mathcal{M}$ by adding edges to $S(t)$ until we have a maximal matching $S'(t) \supseteq S(t)$; then $r_{S'}(e) \leq r_S(e)$ for all $e \in R$ and hence S' is a valid schedule for any DPS instance for which S is valid, and if S schedules an OPS instance with heat $h(S)$ then S' does too, with $h(S') \leq h(S)$.

We use Lemma 10 to reduce OPS to DPS, and Lemma 11 to formalize how DPS solves OPS:

► **Lemma 10** (OPS to DPS). *For every combination of OPS instance $\mathcal{P}_o = (P, R, g)$ and heat value h , there exists a DPS instance $\mathcal{P}_d = (P, R, f)$ such that*

- (1) *any feasible schedule $S : \mathbb{N}_0 \rightarrow 2^R$ for \mathcal{P}_d is a schedule for \mathcal{P}_o with heat $\leq h$, and*
- (2) *any schedule S' for \mathcal{P}_o with heat $h' > h$ is not feasible for \mathcal{P}_d .*

Proof. Consider an OPS polycule $\mathcal{P}_o = (P, R, g)$; we set $\mathcal{P}_d = (P, R, f)$ where $f(e) = \lfloor \frac{h}{g(e)} \rfloor$ for all $e \in R$. Schedules satisfying \mathcal{P}_d when applied to \mathcal{P}_o will allow heat of at most $\max_{e \in R} g(e) \cdot f(e) = \max_{e \in R} g(e) \lfloor \frac{h}{g(e)} \rfloor \leq h$.

Now consider a schedule S' for \mathcal{P}_o with heat $h' > h$. By definition, $h' = \max_{e \in R} r_{S'}(e) \cdot g(e)$, where $r(e) = r_{S'}(e)$ is the recurrence time of e in S' . Assume towards a contradiction that $r(e) \leq f(e)$ for all $e \in R$. This implies that $h' = \max_{e \in R} r(e) \cdot g(e) \leq \max_{e \in R} \lfloor \frac{h}{g(e)} \rfloor \cdot g(e) \leq h$, a contradiction to the assumption. ◀

► **Lemma 11** (DPS to OPS). *Let $\mathcal{P}_d = (P, R, f)$ be a DPS instance. Set $F = \max_{e \in R} f(e)$. There is an OPS instance $\mathcal{P}_o = (P, R, g)$ such that the following holds.*

- (1) *If \mathcal{P}_d is feasible, then \mathcal{P}_o admits a schedule of height $h \leq 1$.*
- (2) *If \mathcal{P}_d is infeasible, then the optimal heat h^* of \mathcal{P}_o satisfies $h^* \geq \frac{F+1}{F}$.*

Proof. Consider a DPS instance $\mathcal{P}_d = (P, R, f)$; we set $\mathcal{P}_o = (P, R, g)$ with $g(e) = 1/f(e)$ for $e \in R$. By definition, any feasible schedule S for \mathcal{P}_d has recurrence time $r_e = r_S(e) \leq f(e)$ for all $e \in R$, so its heat in \mathcal{P}_o is given by $h(S) = \max_{e \in R} r(e) \cdot g(e) = \max_{e \in R} \frac{r(e)}{f(e)} \leq 1$. Conversely, if \mathcal{P}_d is infeasible, then for every $S : \mathbb{N}_0 \rightarrow 2^R$ there exists an edge $e' \in R$ where $r(e') > f(e')$, i.e., $r(e') \geq f(e') + 1$. In \mathcal{P}_o , the heat $h(S)$ must then be $h(S) = \max_{e \in R} r(e) \cdot g(e) \geq r(e') \cdot g(e') \geq \frac{f(e')+1}{f(e')} \geq \frac{F+1}{F}$. ◀

We will often use the *Normal Form* of OPS instances in proofs; this can be assumed without loss of generality but is not generally useful for algorithms unless h^* is known:

► **Lemma 12** (Normal Form OPS). *For every OPS instance $\mathcal{P}_o = (P, R, g)$, there is an equivalent OPS instance $\mathcal{P}'_o = (P, R, g')$ with optimal heat 1 where $g' : R \rightarrow \mathcal{U}$ for $\mathcal{U} = \{1/m : m \in \mathbb{N}_{\geq 1}\}$, i.e., the set of unit fractions. More precisely, for every schedule $S : \mathbb{N}_0 \rightarrow 2^R$ holds: S has optimal heat h^* in \mathcal{P}_o if and only if S has heat 1 in \mathcal{P}'_o . That is, any optimal schedule S^* for either problem is also optimal for the other problem.*

Proof. Let (P, R, g) be an arbitrary OPS instance with optimal heat h^* . Setting $\hat{g}(e) = g(e)/h^*$ yields OPS instance (P, R, \hat{g}) with optimal heat 1. We now start by setting $g'(e) = \hat{g}(e)$ for all $e \in R$. Consider a particular optimal schedule S^* . Suppose that for some edge $e \in R$, we have $g'(e) \notin \mathcal{U}$. In S , there is a maximal separation $r(e) = q \in \mathbb{N}$ between consecutive occurrences of e with $q \cdot g(e) \leq h^*$. But then, increasing $g(e)$ to h^*/q would not affect the heat of S . We can thus set $g'(e) = 1/q$. By induction, we thus obtain $g' : R \rightarrow \mathcal{U}$ without affecting the heat of S . ◀

3 Computational Complexity

One proof of the NP-hardness of the Decision Poly Scheduling (DPS) Problem is that it contains Pinwheel Scheduling as a special case, an NP-hard problem [22]. We show in Section 4 that OPS also contains the Chromatic Index problem as a special case, which gives another proof of the NP-hardness of DPS using the conversion in Lemma 10. Since all good things come in threes, our inapproximability result in the appendix, available online, gives a third independent proof of NP-hardness by reducing 3SAT to DPS.

Upper bounds on the the complexity of DPS are much less clear. Similar to other periodic scheduling problems, characterizing the computational complexity of Poly Scheduling is complicated by the fact that there are feasible instances that require an exponentially large schedule. It is therefore not clear whether Decision Poly Scheduling is in NP since no succinct Yes-certificates are known; this is unknown even for the more restricted Pinwheel Scheduling Problem [24].

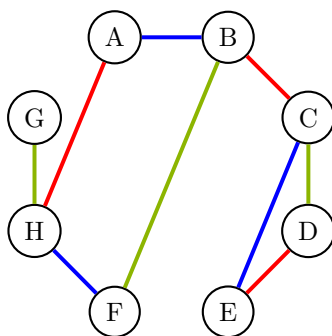
The following simple algorithm shows that DPS is at least in PSPACE (see also [14], [25]): Given the polycule $\mathcal{P}_d = (P, R, f)$ with $|P| = n$ and $|R| = m$, construct the *configuration graph* $\mathcal{G}_c = (V, E)$, where V consists of “countdown vectors” listing for each edge e how many days remain before e has to be scheduled again. $v \in V$ has an outgoing edge for every maximal matching M in $\mathcal{M}(P, R)$, and leads to a successor configuration where all $e \in M$ have their urgency reset to $f(e)$ and all $e \notin M$ have their countdown decremented. Feasible schedules for \mathcal{P}_d correspond to infinite walks in the finite \mathcal{G}_c , and hence must contain a cycle. Conversely, any cycle forms a valid periodic schedule. Our algorithm for DPS thus checks in time $O(|V| + |E|)$ whether \mathcal{G}_c contains a cycle.

The configuration graph \mathcal{G}_c has single exponential size: $V = \{(u_e)_{e \in R} : u_e \in [0..f(e)]\}$ and E has an edge for every matching in (P, R) . So $|E| \leq |V| \cdot 2^m$ (since we have at most $2^{|R|}$ matchings) and $|V| \leq \prod_{e \in R} f(e)$. To further bound this, we use that all $f(e)$ need to be encoded explicitly in binary in the input. $\prod_{e \in R} f(e) \leq \prod_{e \in R} 2^{|f_e|} = 2^{\sum |f_e|} \leq 2^N$ for N the size of the encoding of the input.

To obtain a PSPACE algorithm, we use the polylog-space s - t -connectivity algorithm (using Savich’s Theorem on the NL-algorithm that guesses the next vertex in the path) on \mathcal{G}_c , computing the required part of the graph on-the-fly when queried; this yields overall polynomial space.

4 Unweighted Poly Scheduling & Edge Coloring

Given an OPS instance $\mathcal{P}_o = (P, R, g)$, one can always obtain a feasible schedule from a proper *edge colouring* $c : E \rightarrow [C]$ of the graph (P, R) : any round-robin schedule of the colours is a valid schedule for \mathcal{P}_o , and the number of colours becomes the separation between visits. More formally, we can define a schedule S via $S(t) = \{e \in R : c(e) \equiv t \pmod{C}\}$. An example is shown in Figure 2.



■ **Figure 2** An unweighted polyamorous scheduling instance (that is, an OPS instance where all edges have growth rate 1). Edge colours show one optimal schedule, where every edge is visited exactly every three days: $[3, 3, 3]$, i.e., all red edges are scheduled on days t with $t \equiv 0 \pmod{3}$, all blue edges when $t \equiv 1 \pmod{3}$ and green edges for $t \equiv 2 \pmod{3}$.

15:10 Polyamorous Scheduling

Such a schedule can yield an arbitrarily bad solution to general instances of \mathcal{P}_o , but it gives optimal solutions for a special case: The non-hierarchical polycule \mathcal{P}_u , which is an OPS polycule where all growth rates are $g_{i,j} = 1$ (i.e., an unweighted graph). Recall that any graph with maximal degree Δ can be edge-coloured with at most $\Delta + 1$ colours and clearly needs at least Δ colours.

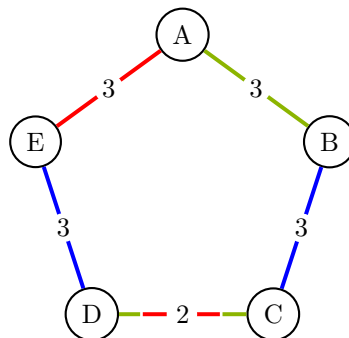
► **Proposition 13** (Unweighted OPS = edge coloring). *An unweighted OPS problem admits a schedule with heat h if and only if the corresponding graph is h -edge-colourable.*

Proof. First note that any k -edge-colouring immediately corresponds to a schedule that visits every edge every k days, since we can schedule all edges e with $c(e) = i$ on days $t \equiv i \pmod C$. Moreover, any schedule with height h must visit every edge at least once within the first h days (otherwise it would grow to desire $> h \cdot 1$). We can therefore assign h colours according to these first h days of the schedule; some edges might receive more than one colour, but we can use any of these and retain a valid colouring using h colours. ◀

Since it is NP-complete to decide whether a graph has chromatic index $\chi_1 = \Delta$ (even when the graph is 3-regular [21]) unweighted Poly Scheduling is NP-hard. This provides a second restricted special case of the problem that is NP-hard, which also gives us the inapproximability result stated in Theorem 4:

Proof of Theorem 4 (page 6). Assume that there is a polynomial-time algorithm A that achieves an approximation ratio of $\frac{4}{3} - \varepsilon$ for some $\varepsilon > 0$. Given an input (V, E) to the 3-Regular Chromatic Index Problem (i.e., given a 3-regular graph, decide whether $\chi_1(G) = \Delta = 3$ or $\chi_1(G) = \Delta + 1 = 4$), we can apply A to (V, E, g) , setting $g(e) = 1$ for all $e \in E$. By Proposition 13, A finds an edge colouring with $c \leq (\frac{4}{3} - \varepsilon) \cdot \chi_1(G)$ colours. If $\chi_1(G) = \Delta = 3$, then $c \leq 4 - 3\varepsilon < 4$, so $c = 3$; if $\chi_1(G) = 4$, then $c \geq 4$. Comparing c to Δ thus determines $\chi_1(G)$ exactly in polynomial time; in particular, for every 3-regular graph, this decides whether $\chi_1(G) = 3$. Since 3-Regular Chromatic Index is NP-complete [21], it follows that $P = NP$. ◀

We close this section with the remark that there are weighted DPS instances where any feasible schedule must “multi-colour” some edges, including the polycule shown in Figure 3. For the general problem, we thus cannot restrict our attention to edge colourings (though they may be a valuable tool for future work).



■ **Figure 3** A discrete polyamorous scheduling instance which is solvable only by assigning multiple colours to the CD edge.

5 Approximation Algorithms

In this section, we present two efficient polynomial-time approximation algorithms for Poly Scheduling, thereby proving Theorems 5 and 6. Throughout this section, we assume a fixed instance $\mathcal{P}_o = (P, R, g)$ of Optimisation Polyamorous Scheduling (OPS) is given.

5.1 Lower Bounds

We first collect a few simple lower bounds used in the analysis later; note that Section 6 has further lower bounds.

► **Lemma 14** (Simple lower bound). *Given an OPS instance $\mathcal{P}_o = (P, R, g)$, set $g_{\min} = \min_{e \in R} g(e)$, $g_{\max} = \max_{e \in R} g(e)$, and $\Delta = \max_{p \in P} \deg(p)$. Any periodic schedule for \mathcal{P}_o has heat $h \geq \max\{\Delta \cdot g_{\min}, g_{\max}\}$.*

Proof. The chromatic number χ_1 of the unweighted graph (P, R) is $\chi_1 \in \{\Delta, \Delta + 1\}$. This means that under any periodic schedule, some edge desires will grow to at least to $\chi_1 \cdot g_{\min} \geq \Delta \cdot g_{\min}$, since we cannot schedule any two edges incident to a degree- Δ node on the same day. Moreover, we cannot prevent the weight- g_{\max} edge from growing to heat g_{\max} . ◀

A second observation is that the lower bound for any subset of the problem is also a lower bound for the problem as a whole:

► **Lemma 15** (Subset bound). *Given two OPS instances $\mathcal{P}_o = (P, R, g)$ and $\mathcal{P}'_o = (P, R', g')$ with $R' \subseteq R$ and $g(e) = g'(e)$ for all $e \in R'$, i. e., \mathcal{P}'_o results from \mathcal{P}_o by dropping some edges. Assume further that any schedule for \mathcal{P}'_o has heat at least h^* . Then, any schedule for \mathcal{P}_o also has heat at least h^* .*

Proof. Suppose there is a schedule S for \mathcal{P}_o of heat $h' < h$. We obtain a schedule S' for \mathcal{P}'_o by dropping all edges $e \notin R'$. (The resulting schedule may have empty days.) By construction, when using S' to schedule \mathcal{P}'_o , all edges in R' will grow to the same heat as in \mathcal{P}_o under S , and hence also to heat $h' < h$. ◀

5.2 Approximation for Almost Equal Growth Rates

We first focus on a special case of OPS instances with “almost equal weights”, which is used as base for our main algorithm. Let the edge weights satisfy $g_{\min} \leq g(e) \leq g_{\max}$ for all $e \in R$. We will show that scheduling a proper edge colouring round-robin gives a $\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}$ approximation algorithm, establishing Theorem 5.

Proof of Theorem 5 (page 6). We compute a proper edge colouring for (P, R) with $\Delta + 1$ colours using the algorithm from [27] and schedule these $\Delta + 1$ matchings in a round-robin schedule. No edge desire will grow higher than $(\Delta + 1) \cdot g_{\max}$ in this schedule. Lemma 14 shows that $\text{OPT} \geq \max\{\Delta \cdot g_{\min}, g_{\max}\}$. The edge-colouring schedule is thus never more than a $\min\{\frac{\Delta+1}{\Delta} \cdot \frac{g_{\max}}{g_{\min}}, \Delta + 1\}$ factor worse than OPT. ◀

5.3 Layering Algorithm

The colouring-based algorithm from Theorem 5 can be arbitrarily bad if desire growth rates are vastly different and Δ is large. For these cases, a more sophisticated algorithm achieves a much better guarantee (Theorem 6). The algorithm consists of 3 steps:

1. breaking the graph into layers (by edge growth rates),

15:12 Polyamorous Scheduling

2. solving each layer using Theorem 5, and
3. interleaving the layer schedules into an overall schedule.

Let L be a parameter to be chosen later. We define layers of $\mathcal{P}_o = (P, R, g)$ as follows. For $i = 0, \dots, L-1$, set $\mathcal{P}_i = (P, R_i, g)$ where

$$R_i = \left\{ e \in R : \frac{g_{\max}}{2^{i+1}} < g(e) \leq \frac{g_{\max}}{2^i} \right\}.$$

Moreover, $\mathcal{P}_L = (P, R_L, g)$ with $R_L = \{e \in R : g(e) \leq \frac{g_{\max}}{2^L}\}$.

Denote by Δ_i , for $i = 0, \dots, L$, the maximal degree in (P, R_i) . Let S_i be the round-robin- $(\Delta_i + 1)$ -colouring schedule from Theorem 5 applied on the OPS instance \mathcal{P}_i . If run in isolation on \mathcal{P}_i , schedule S_i has heat $h_i \leq (\Delta_i + 1)g_{\max}/2^i \leq (\Delta + 1)g_{\max}/2^i$ by the same argument as in Section 5.2. Moreover, for $i < L$, S_i is a $2^{\frac{\Delta_i+1}{\Delta_i}}$ -approximation (on \mathcal{P}_i in isolation); for $i = L$, we can only guarantee a $(\Delta_L + 1)$ -approximation.

To obtain an overall schedule S for \mathcal{P} , we schedule the $L + 1$ layers in round-robin fashion, and within each layer's allocated days, we advance through its schedule as before, i.e., $S(t) = S_{(t \bmod (L+1))}(\lfloor t/(L+1) \rfloor)$. Any advance in layer i is now delayed by a factor $(L + 1)$. Hence S achieves heat at most

$$\bar{h} = \max_{i \in [0..L]} (L + 1) \cdot h_i \leq \max_{i \in [0..L]} (L + 1)(\Delta_i + 1) \cdot \frac{g_{\max}}{2^i}$$

Using Lemma 15 on the layers and Lemma 14, we obtain a lower bound for OPT of

$$\underline{h} = \max \left\{ \max_{i \in [0..L-1]} \Delta_i \cdot \frac{g_{\max}}{2^{i+1}}, g_{\max} \right\}$$

We now distinguish two cases for whether the maximum in \bar{h} is attained for an $i < L$ or for $i = L$. First suppose $\bar{h} = (L + 1)(\Delta_i + 1)g_{\max}/2^i$ for some $i < L$. Since we also have $\underline{h} \geq \Delta_i \cdot g_{\max}/2^{i+1}$, we obtain an approximation ratio of $2(L + 1)\frac{\Delta_i+1}{\Delta_i} \leq 3(L + 1)$ overall in this case. Here, we assume that $\Delta_i \geq 2$; otherwise we have only monogamous couples in this layer and scheduling is trivial, giving $h_i = \Delta_i \cdot g_{\max}/2^i$.

For the other case, namely $\bar{h} = (L + 1)(\Delta_L + 1)g_{\max}/2^L > (L + 1) \cdot (\Delta_i + 1)g_{\max}/2^i$ for all $i < L$, we do not have lower bounds on the edge growth rates. But we still know $\underline{h} \geq g_{\max}$, so we obtain a $(L + 1)(\Delta_L + 1)/2^L$ -approximation overall in this case.

Equating the two approximation ratios suggests to choose L such that $L \approx \lg(\Delta_L + 1) - \lg 3$; with $L = \lceil \lg(\Delta + 1) - \lg 3 \rceil$ and using $\Delta_L \leq \Delta$, we obtain an overall approximation ratio of at most $3(L + 1) \leq 3\lceil \lg(\Delta + 1) \rceil \leq 3\lceil \lg n \rceil$. This concludes the proof of Theorem 6.

6 Fractional Poly Scheduling

In this section, we generalize the notion of density from Pinwheel Scheduling for the Polyamorous Scheduling Problem. For that, we consider the dual of the linear program corresponding to a fractional variant of Poly Scheduling.

6.1 Linear Programs for Poly Scheduling

In the fractional Poly Scheduling problem, instead of committing to a single matching M in (P, R) each day, we are allowed to devote an arbitrary *fraction* $y_M \in [0, 1]$ of our day to M , but then switch to other matchings without cost or delay for the rest of the day (a simple form of scheduling with preemption). The heat of a fractional schedule is again defined

as $\max_{e \in R} r(e)g(e)$, but the recurrence time $r(e)$ now is the maximal time in S before the fraction of days devoted to matchings containing e sum to at least 1. (For a non-preemptive schedule with one matching per day, this coincides with the definition from Section 2.)

Schedules for the fractional problem are substantially easier because there is no need to have different fractions y_M for different days: the schedule obtained by always using the average fraction of time spent on each matching yields the same recurrence times. We can therefore assume without loss of generality that our schedule is given by $S = S(\{y_M\}_{M \in \mathcal{M}})$, with $y_M \in [0, 1]$ and $\sum_{M \in \mathcal{M}} y_M = 1$. S schedules the matchings in some arbitrary fixed order, each day devoting the same y_M fraction of the day to M . Then, recurrence times are simply given by $r_S(e) = 1 / \sum_{M \in \mathcal{M}: e \in M} y_M$.

With these simplifications, we can state the fractional relaxation of Optimisation Poly Scheduling instance $\mathcal{P}_o = (P, R, g)$ as an optimisation problem as follows:

$$\min \quad \bar{h} \tag{1}$$

$$\text{s. t.} \quad \sum_{M \in \mathcal{M}} y_M \leq 1 \tag{2}$$

$$\frac{1}{\sum_{M \in \mathcal{M}: e \in M} y_M} \cdot g_e \leq \bar{h} \quad \forall e \in R \tag{3}$$

$$y_M \in [0, 1] \quad \forall M \in \mathcal{M} \tag{4}$$

Substituting $\bar{h} = 1/\ell$, this is equivalent to the following linear program (LP):

$$\max \quad \ell \tag{5}$$

$$\text{s. t.} \quad \sum_{M \in \mathcal{M}} y_M \leq 1 \tag{6}$$

$$\frac{1}{g_e} \sum_{M \in \mathcal{M}: e \in M} y_M \geq \ell \quad \forall e \in R \tag{7}$$

$$y_M \geq 0 \quad \forall M \in \mathcal{M} \tag{8}$$

The optimal objective value ℓ^* of this LP gives $\bar{h}^* = 1/\ell^*$, the optimal fractional heat.

► **Lemma 16** (Fractional lower bound). *Consider an OPS instance $\mathcal{P}_o = (P, R, g)$ with optimal heat h^* and let $\bar{h}^* = 1/\ell^*$ where ℓ^* is the optimal objective value of the fractional-problem LP from Equation (5). Then $\bar{h}^* \leq h^*$.*

Proof. We use the same approach as in [9, §3]: For any schedule S , $h(S)$ is at least the heat $h_T(S)$ obtained during the first T days only, which in turn is at least $\max_e g(e) \cdot \bar{r}(e)$ for $\bar{r}(e)$ the *average* recurrence time of edge e during the first T days. A basic calculation shows that for the fractions y_M of time spent on matching M during the first T days there exists a value $1/\ell = h(S)(1 - o(T))$, so that we obtain a feasible solution of the LP (5). Hence $1/\ell^* \leq 1/\ell = h(S)(1 - o(T))$. Since these inequalities hold simultaneously for all T , taking the limit as $T \rightarrow \infty$, we obtain $1/\ell^* = \bar{h}^* \leq h(S)$. ◀

The immediate usefulness of Lemma 16 is limited since the number of matchings can be exponential in n .

► **Remark 17** (Randomized-rounding approximation?). One could try to use this LP as the basis of a randomized-rounding approximation algorithm, but since it is not clear how to obtain an efficient algorithm from that, we do not pursue this route here. The simple route taken in [9] cannot achieve an approximation ratio better than $O(\log n)$, so Theorem 6 already provides an equally good deterministic algorithm.

We therefore proceed to the dual LP of Equation (5):

$$\min x \tag{9}$$

$$\text{s. t. } \sum_{e \in R} z_e \geq 1 \tag{10}$$

$$\sum_{e \in M} \frac{z_e}{g_e} \leq x \quad \forall M \in \mathcal{M} \tag{11}$$

$$z_e \geq 0 \quad \forall e \in R \tag{12}$$

While still exponentially large and thus not easy to solve exactly, the dual LP yields the versatile result from Theorem 7.

Proof of Theorem 7 (page 7). Using the given z_e and $x = \max_{M \in \mathcal{M}} \sum_{e \in M} \frac{z_e}{g(e)}$, we fulfil all constraints of Equation (9). The optimal objective value x^* is hence $x^* \leq x$. By the duality of LPs, we have $x^* \geq \ell^*$ for ℓ^* the optimal objective value of Equation (5). Together with Lemma 16, this means $h^* \geq \bar{h}^* = 1/\ell^* \geq 1/x^* \geq 1/x$. ◀

6.2 Poly Density

Theorem 7 gives a more explicit way to compute the *poly density* \bar{h}^* than the primal LP, but it is unclear whether it can be computed exactly in polynomial time. Given the more intricate global structure of Poly Scheduling, \bar{h}^* is necessarily more complicated than the density of Pinwheel Scheduling. A particularly interesting open problem for Poly Scheduling is whether a sufficiently low poly density implies the existence of a valid (integral) schedule.

Specific choices for z_e in Theorem 7 yield several known bounds:

- Setting $z_e = g_e/G$ for $G = \sum_{e \in R} g_e$ yields Corollary 8.
- Fix any subset $R' \subseteq R$. Now set $z_e = g_e/C$ if $e \in R'$ and 0 otherwise, where $C = \sum_{e \in R'} g_e$. The maximum from Theorem 7 then simplifies to $\frac{1}{C} \max_{M \in \mathcal{M}} |M \cap R'|$, so

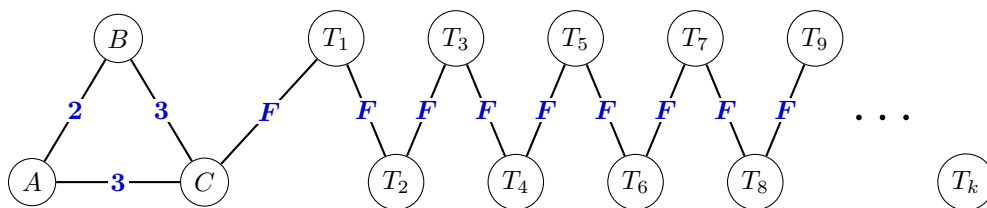
$$h^* \geq \frac{\sum_{e \in R'} g_e}{\max_{M \in \mathcal{M}} |M \cap R'|}.$$

- An immediate application of that observation with R' being all edges incident at a person $p \in P$ yields the BGT bound:
 - ▶ **Corollary 18** (Bamboo lower bound). *Given an OPS instance (P, R, g) and $p \in P$ with $g_1 \geq \dots \geq g_d$ the desire growth rates for edges incident at p . Set $G_p = g_1 + \dots + g_d$. Any periodic schedule for (P, R, g) has heat at least G_p .*

▶ **Remark 19** (Better general bounds?). For the general case, it seems challenging to obtain other such simple bounds. The bound of G/m is easy to justify without the linear programs by a “preservation-of-mass argument”: Assume a schedule S could achieve a heat $h < G/m$. Every day, the overall polycule’s desire grows by G , and S can schedule at most m pairs to meet, whose desire is reset to 0 from some value $\leq h$. Every day, S thus removes only a total of $\leq mh < G$ desire units from the polycule, whereas the overall growth is G , a contradiction to the heat remaining bounded.

Note that the bound of G/m is tight for some instances, so we cannot hope for a strictly lower bound. On the other hand, the example from Figure 4 shows demonstrates that it can also be arbitrarily far from h^* .

Figure 4 shows the tadpole family of instances demonstrating the power of the dual-LP approach and Theorem 7. All DPS tadpoles (as shown in the figure) are infeasible since already the triangle $A-B-C$ does not admit a schedule obeying the given frequencies. The corresponding OPS instances (as given by Lemma 11) with $g(e) = 1/f(e)$ thus have $h^* > 1$;



■ **Figure 4** The tadpole family of DPS instances, defined for parameters $k \geq 0$ (tail length) and $F \geq 3$ (tail frequency). The total growth rate is $G = \frac{1}{2} + \frac{2}{3} + k \cdot 1/F = \frac{7}{6} + \frac{k}{F}$ and the size of a maximum matching is $m = 1 + \lfloor (k+1)/2 \rfloor$.

indeed $h^* = 4/3$ if $F \geq 2$. However, the simple lower bounds or local arguments do not detect this: (a) All local Pinwheel Scheduling instances (any person plus their neighbours) are feasible. (b) The mass-preservation bound (Corollary 8) is $G/m < 1$ for $k \geq 1$. Indeed, setting $F = k$ and letting $k \rightarrow \infty$, $G/m = O(1/k)$, giving an arbitrarily large gap to h^* . By contrast, consider the LP fractional lower bound. One can show that $\bar{h}^* = \frac{7}{6} > 1$ for any $k \geq 1$ and $F \geq 2$, so Theorem 7 correctly detects the infeasibility in this example.

► **Remark 20** (Better Pinwheel density via dual LPs?). Since Poly Scheduling is a generalization of Pinwheel Scheduling resp. Bamboo Garden Trimming, we can apply Theorem 7 also to these problems. However, for this special case, the optimal objective value of the dual LPs is *always* $x^* = \ell^* = 1/G$ for G the sum of the growth rates, so we only obtain the trivial “biomass” lower bound of G for Bamboo Garden Trimming resp. the density ≤ 1 necessary condition for Pinwheel Scheduling. The more complicated structure of matchings in non-star graphs makes fractional lower bounds in Poly Scheduling much richer and more powerful.

7 Open Problems & Future Directions

This paper opens up several avenues for future work. The most obvious open problem concerns efficient approximation algorithms: We show that finding approximations with a better ratio than $4/3$ is NP-hard, and introduce an $O(\log n)$ polynomial-time approximation. Can the gap between these be reduced, or even eliminated?

In the appendix, available online, we conjecture that further analysis of the SAT reduction originally used to prove Theorem 3 may demonstrate better inapproximability results for OPS in the general case. The true lower bound may even be super-constant. However, in light of our Theorem 6, a super-constant hardness of approximation result would have to use Poly Scheduling instances with super-constant degrees. Open Problem 9 will also have clear implications for OPS, as well as being interesting in its own right.

There is also interesting work to be done looking at specific classes of polycules. Bipartite polycules are particularly interesting, both for the likelihood that they will permit better approximations than are possible in the general case and for their applications (e.g., modelling the users and providers of some service).

Polyamorous scheduling has several interesting generalizations including Fungible Polyamorous Scheduling, whose decision version we define as:

► **Definition 21** (Fungible Decision Polyamorous Scheduling (FDPS)). *An FDPS instance $\mathcal{P}_{fd} = (P, R, s, f)$ (a “(fungible decision) polycule”) consists of an undirected graph (P, R) where the vertices $P = \{p_1, \dots, p_n\}$ are n classes of fungible persons and the edges R are pairwise relationships between those classes. Classes have integer sizes $s : P \rightarrow \mathbb{N}$ and relationships have integer frequencies $f : R \rightarrow \mathbb{N}$.*

15:16 Polyamorous Scheduling

The goal is find an infinite schedule $S : \mathbb{N}_0 \rightarrow 2^R$, such that

- (1) (no overflows) for all days $t \in \mathbb{N}_0$, $S(t)$ is a multiset of elements from P such that each node $p \in P$ appears at most $s(p)$ times, and
- (2) (frequencies) for all $e \in R$ and $t \in \mathbb{N}_0$, we have $e \in S(t) \cup S(t+1) \cup \dots \cup S(t+f(e)-1)$; or to report that no such schedule exists.

FDPS also has an optimisation version, which again allows each class $p \in P$ to have at most $s(p)$ meetings each day. These problems have clear applications to the scheduling of staff, locations, resources etc. in real-world applications.

Another natural generalisation is Secure Polyamorous scheduling. Suppose that Adam is dating both Brady and Charlie, who are also dating each other. In a DPS or OPS polycule, on any day, Adam must choose to meet with either Brady or Charlie, who each face the same dilemma; but why can't he meet both?⁵ The Secure Decision Polyamorous scheduling problem allows this:

► **Definition 22** (Secure Decision Polyamorous Scheduling (SDPS)). *An SDPS instance $\mathcal{P}_{sd} = (P, R, f)$ (a “(secure decision) polycule”) consists of an undirected graph (P, R) where the vertices $P = \{p_1, \dots, p_n\}$ are n persons, and the edges R are pairwise relationships, with integer frequencies $f : R \rightarrow \mathbb{N}$ for each relationship.*

The goal is find an infinite schedule $S : \mathbb{N}_0 \rightarrow 2^R$, such that

- (1) (no third-wheels) for all days $t \in \mathbb{N}_0$, $S(t)$ is a set of meetings between cliques of people in P in which each person appears at most once, and
- (2) (frequencies) for all $e \in R$ and $t \in \mathbb{N}_0$, we have $e \in S(t) \cup S(t+1) \cup \dots \cup S(t+f(e)-1)$; or to report that no such schedule exists.

Again, this has a natural optimisation version.

Polyamorous Scheduling also motivates the study of several restricted versions of Pinwheel Scheduling and Bamboo Garden Trimming, including partial scheduling (wherein some portion of the schedule is fixed as part of the problem and the challenge is to find the remainder of the schedule), and fixed holidays (where the fixed part of the schedule consists of periodic gaps).

References

- 1 Peyman Afshani, Mark de Berg, Kevin Buchin, Jie Gao, Maarten Löffler, Amir Nayyeri, Benjamin Raichel, Rik Sarkar, Haotian Wang, and Hao-Tsung Yang. On cyclic solutions to the min-max latency multi-robot patrolling problem. In *International Symposium on Computational Geometry (SoCG)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SOCG.2022.2.
- 2 Amotz Bar-Noy, Richard E Ladner, and Tami Tamir. Windows scheduling as a restricted version of bin packing. *ACM Transactions on Algorithms*, 3(3):28–es, 2007. doi:10.1145/1273340.1273344.
- 3 Amotz Bar-Noy, Joseph (Seffi) Naor, and Baruch Schieber. Pushing dependent data in clients-providers-servers systems. *Wireless Networks*, 9(5):421–430, 2003. doi:10.1023/a:1024632031440.
- 4 Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti, and Giacomo Scornavacca. Cutting Bamboo down to Size. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *Fun with Algorithms (FUN)*, volume 157 of *Leibniz International Proceedings*

⁵ A key part of polyamory [23]!

- in Informatics (LIPIcs)*, pages 5:1–5:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FUN.2021.5.
- 5 Thomas Bosman, Martijn van Ee, Yang Jiao, Alberto Marchetti-Spaccamela, R. Ravi, and Leen Stougie. Approximation algorithms for replenishment problems with fixed turnover times. *Algorithmica*, 84(9):2597–2621, May 2022. doi:10.1007/s00453-022-00974-4.
 - 6 Mee Yee Chan and Francis Chin. Schedulers for larger classes of pinwheel instances. *Algorithmica*, 9(5):425–462, 1993. doi:10.1007/BF01187034.
 - 7 Mee Yee Chan and Francis Y. L. Chin. General schedulers for the pinwheel problem based on double-integer reduction. *IEEE Trans. Computers*, 41(6):755–768, 1992. doi:10.1109/12.144627.
 - 8 Wun-Tat Chan and Prudence W. H. Wong. On-line windows scheduling of temporary items. In *International Symposium on Algorithms and Computation (ISAAC)*, pages 259–270. Springer Berlin Heidelberg, 2004. doi:10.1007/978-3-540-30551-4_24.
 - 9 Serafino Cicerone, Gabriele Di Stefano, Leszek Gąsieniec, Tomasz Jurdzinski, Alfredo Navarra, Tomasz Radzik, and Grzegorz Stachowiak. Fair hitting sequence problem: Scheduling activities with varied frequency requirements. In *International Conference on Algorithms and Complexity (CIAC)*, pages 174–186. Springer, 2019. doi:10.1007/978-3-030-17402-6_15.
 - 10 Larry Clemmons, Ken Anderson, and Vance Gerry. Robin hood (movie). *Walt Disney Productions*, 1973.
 - 11 Wei Ding. A branch-and-cut approach to examining the maximum density guarantee for pinwheel schedulability of low-dimensional vectors. *Real-Time Systems*, 56(3):293–314, 2020. doi:10.1007/s11241-020-09349-w.
 - 12 Eugene A. Feinberg and Michael T. Curry. Generalized pinwheel problem. *Math. Methods Oper. Res.*, 62(1):99–122, 2005. doi:10.1007/s00186-005-0443-4.
 - 13 Peter C Fishburn and Jeffrey C Lagarias. Pinwheel scheduling: Achievable densities. *Algorithmica*, 34(1):14–38, 2002. doi:10.1007/s00453-002-0938-9.
 - 14 Leszek Gąsieniec, Benjamin Smith, and Sebastian Wild. Towards the 5/6-density conjecture of pinwheel scheduling. In C. A. Phillips and B. Speckmann, editors, *Symposium on Algorithm Engineering and Experiments (ALENEX)*, pages 91–103. SIAM, January 2022. doi:10.1137/1.9781611977042.8.
 - 15 Leszek Gąsieniec, Tomasz Jurdziński, Ralf Klasing, Christos Levcopoulos, Andrzej Lingas, Jie Min, and Tomasz Radzik. Perpetual maintenance of machines with different urgency requirements. *Journal of Computer and System Sciences*, 139:103476, February 2024. doi:10.1016/j.jcss.2023.103476.
 - 16 Leszek Gąsieniec, Ralf Klasing, Christos Levcopoulos, Andrzej Lingas, Min Jie, and Tomasz Radzik. *Bamboo Garden Trimming Problem*, volume 10139 of *Lecture Notes in Computer Science*. Springer, 2017. doi:10.1007/978-3-319-51963-0.
 - 17 C.-C. Han and K.-J. Lin. Scheduling distance-constrained real-time tasks. In *Proceedings Real-Time Systems Symposium*. IEEE Comput. Soc. Press, 1992. doi:10.1109/REAL.1992.242649.
 - 18 Felix Höhne and Rob van Stee. A 10/7-approximation for discrete bamboo garden trimming and continuous trimming on star graphs. In *Conference on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.APPROX/RANDOM.2023.16.
 - 19 Robert Holte, Al Mok, Al Rosier, Igor Tulchinsky, and Igor Varvel. The pinwheel: a real-time scheduling problem. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences. Volume II: Software Track*, volume 2, pages 693–702 vol.2, 1989. doi:10.1109/HICSS.1989.48075.
 - 20 Robert Holte, Louis E. Rosier, Igor Tulchinsky, and Donald A. Varvel. Pinwheel scheduling with two distinct numbers. *Theor. Comput. Sci.*, 100(1):105–135, 1992. doi:10.1016/0304-3975(92)90365-M.
 - 21 Ian Holyer. The np-completeness of edge-coloring. *SIAM Journal on computing*, 10(4):718–720, 1981.

15:18 Polyamorous Scheduling

- 22 Tobias Jacobs and Salvatore Longo. A new perspective on the windows scheduling problem. *coRR*, 2014. [arXiv:1410.7237](https://arxiv.org/abs/1410.7237).
- 23 John_Threepwood. Why not both? / Why don't we have both?, August 2011. URL: <https://knowyourmeme.com/memes/why-not-both-why-dont-we-have-both>.
- 24 Akitoshi Kawamura. Proof of the density threshold conjecture for pinwheel scheduling. In *Symposium on Theory of Computing (STOC)*, 2024. URL: https://www.kurims.kyoto-u.ac.jp/~kawamura/pinwheel/paper_e.pdf.
- 25 Akitoshi Kawamura and Makoto Soejima. Simple strategies versus optimal schedules in multi-agent patrolling. *Theoretical Computer Science*, 839:195–206, November 2020. doi:10.1016/j.tcs.2020.07.037.
- 26 Shun-Shii Lin and Kwei-Jay Lin. A pinwheel scheduler for three distinct numbers with a tight schedulability bound. *Algorithmica*, 19(4):411–426, 1997. doi:10.1007/PL00009181.
- 27 Jayadev Misra and David Gries. A constructive proof of vizing's theorem. In *Information Processing Letters*. Citeseer, 1992.
- 28 Martijn van Ee. A 12/7-approximation algorithm for the discrete bamboo garden trimming problem. *Operations Research Letters*, 49(5):645–649, September 2021. doi:10.1016/j.orl.2021.07.001.
- 29 Vadim G Vizing. The chromatic class of a multigraph. *Cybernetics*, 1(3):32–41, 1965.

Tetris Is Not Competitive

Matthias Gehnen   

RWTH Aachen University, Germany

Luca Venier  

RWTH Aachen University, Germany

Abstract

In the video game Tetris, a player has to decide how to place pieces on a board that are revealed by the game one after another. We show that the missing information about the upcoming pieces is indeed crucial to a player's success. We present a construction for piece sequences that force (online) players without or with a finite preview of upcoming pieces to lose while (offline) players who know the entire piece sequence can clear the board and continue to play.

From a competitive analysis perspective, it follows that there cannot be any c -competitive online algorithm for various optimization goals in the context of playing Tetris. Furthermore, we improve existing results by providing a construction for piece sequences which force every player to lose for every possible board size with at least two columns. With this construction, we are also able to show that an instance with just 435 pieces is sufficient to force every player to lose on a standard-size board with ten columns and twenty rows.








2012 ACM Subject Classification Theory of computation → Online algorithms

Keywords and phrases Online Algorithms, Tetris



Digital Object Identifier 10.4230/LIPIcs.FUN.2024.16

Acknowledgements This article bases on Luca Venier's Mastes's Thesis [16].

1 Introduction

Tetris is a classical video game in which four-celled pieces (, , , , , , ) fall onto a rectangular board. While a piece drops from the top, the player can rotate and move it horizontally until it lands on a filled cell or the bottom of the board. Whenever every cell of a row is occupied by pieces it disappears (lineclear) and all filled cells above drop down by one row. The player continues to play as long as pieces can be placed on the board and loses if a piece cannot be placed on the board.

Apart from its fame as a video game, the underlying concept also leads to interesting combinatorial questions, like: Given a board, and a sequence of pieces to place, is there a way to place the pieces without losing? Are there piece sequences that always lead to a loss or always have a loss-avoiding strategy? These questions have been studied in the past, often with the assumption that the player knows all upcoming pieces. However, in the classical video game, the player needs to make irrevocable decisions with just a few pieces of lookahead. Therefore, it seems reasonable to consider the underlying combinatorial concept as an online-problem.

For standard-size boards (ten columns and twenty rows) Brzustowski [6] gives an instance that forces an online player with a lookahead of just one piece to lose. Later Burgiel [7] showed that there is no chance to play for more than 69600 pieces before losing in an instance with alternating  and  pieces, even if the entire instance is known in advance.

If an instance forces an offline player to lose, then an online player with less information about the piece sequence is naturally also forced to lose. However, the opposite case is not trivial. If there is an instance that forces an online player with a finite lookahead to lose, must an offline player who knows the entire instance also lose?



© Matthias Gehnen and Luca Venier;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 16; pp. 16:1–16:16

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Related Work

After its development in the mid-1980s by Pajitnov, Pavlovsky, and Gerasimov [10], the history of analyzing the underlying theoretic concepts of Tetris started in 1992 with the already mentioned master's thesis by Brzustowski. Apart from showing that there are unwinnable instances for general Tetris, Brzustowski presents some winning strategies for restricted variants with only one or two different piece types. Even if there cannot be a general loss-avoiding strategy for classical Tetris, it is known that such strategies exist for some restricted piece-sets even on non-rectangular boards [15]. If the pieces of an instance cannot be chosen arbitrarily but have to be presented in a sequence of permutations of the seven different piece types (the so-called 7-bag randomizer) then some commercial game variants have loss-avoiding strategies [1]. For an overview of implemented piece selection algorithms see [12]. Here the different selection process has a crucial role since it is known that some specific sequences lead to a loss. If randomly selecting the next piece type determines the piece sequence then such a loss-forcing sequence appears for sure at some point in a resulting infinite random string. Burgiel therefore follows that every Tetris game (implying a random generation of the piece sequence) must end at some point [7].





Since it is known that the player has to lose for some piece sequences, it is natural to ask for the complexity to decide if a given instance is such a sequence. Demaine et al. found out in 2003 that this is indeed an NP-hard problem, and related optimization problems similar to the ones we investigate in Section 3.1 also cannot be approximated efficiently [9]. Later a simpler reduction was given [5, 4], and some modifications are shown to not be easier [2, 14].

For further related work about Tetris, also to non-theoretical results, we refer to the vast related work section of Dallant and Iacono's article [8].

In most of the mentioned results about Tetris, an algorithm or strategy is given the entire instance which must be solved in advance. In contrast, a Tetris player like an online algorithm has no advanced knowledge about the instance it needs to solve. Whenever a new element of the instance is given some irrevocable decision must be taken before the next piece is revealed. An online algorithm tries to optimize an objective function that is dependent on the solution set formed by its decisions. The *strict competitive ratio* of an algorithm, as defined by Sleator and Tarjan [13], is the worst-case ratio of the performance of an algorithm compared to that of an optimal solution computed by an offline algorithm for the given instance, over all instances. The competitive ratio of an online *problem* is then the best competitive ratio over all online algorithms. For a general introduction to online problems, we refer to the books of Borodin and Ran El-Yaniv [3] and of Komm [11].

1.2 Preliminaries and Notation

An instance of a Tetris game in the algorithmic sense consists of a board and a finite sequence of pieces (as piece types) that are played. The board has a width w , a height h , and consists of $w \times h$ many cells that are either filled or empty. We call each resulting pattern of filled and unfilled cells a board configuration. Each instance starts with an empty board configuration where all cells are empty. We divide the board in $\frac{w}{2}$ pairs of adjacent columns, which we call lanes like in [6].

The player immediately loses if a piece is placed in such a way that a cell outside the boundaries of the board would be filled. The player wins if each piece of the piece sequence has been successfully placed inside the board. A player can rotate a piece and select the column it is supposed to fall in. We call rotations of pieces where the piece is taller than wide (e.g., , , ...) vertical and the opposite rotations (e.g., , , ...) horizontal. Compared

to implemented games and some mathematical models the player cannot manipulate the piece while it is falling as the additional complexity of the model is not needed for the claims made in this work.

In an online setting a player has a lookahead $l \in \mathbb{N}$. The game proceeds with the player placing the i -th piece of the piece sequence, and then the adversary chooses and appends the $i + l + 1$ -th piece type to the piece sequence. In an offline setting the entire piece sequence is known before the game begins.

For some objective function, the ratio between the performance of the best possible online strategy and the optimal offline solution is called (*strict*) *competitive ratio*, usually denoted with c . Sometimes adding a constant α is sensible, making the analysis non-strict. Therefore, we call an online problem c -competitive if there is an online strategy that is not worse than $c \cdot \text{OPT} + \alpha$ than an optimal solution. Here OPT denotes the quality of the optimal solution with respect to the objective function [3, 11].

The following sections use state diagrams where nodes are board configurations to depict game instances similarly to [6]. Transitions have labels depicting which piece types is presented to the player. Furthermore, transitions can have an optional index (e.g., $\blacksquare, 2$) to indicate that the player places the piece with its leftmost cells in the 0-indexed column. An optional multiplier (e.g., $2 \times \blacksquare$) denotes that the same piece type is presented multiple times.

1.3 Results

In Section 2 we construct instances without any loss-avoiding strategies for players with arbitrary (but finite) lookahead, but for which loss-avoiding strategies exist if the full instance is known to the player. A common question for online-problems is to ask how much worse an online algorithm (in our case the player with a finite preview of upcoming pieces) will perform on the same instance due to the lack of information.

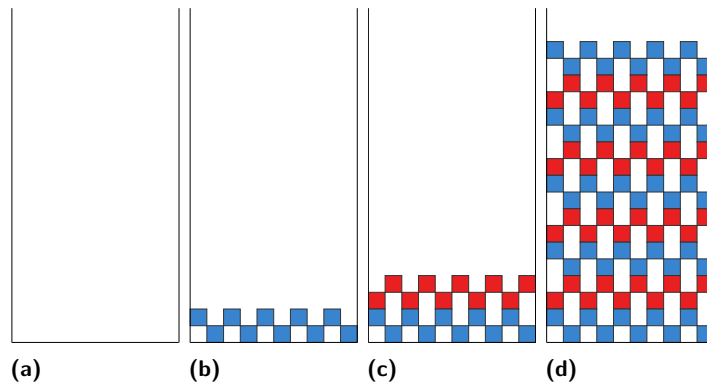
In Section 3.1 the aforementioned construction is used to show that there are instances where the online player performs arbitrarily badly against an offline player for several optimization goals. Hence, for these goals, there cannot be any c -competitive online algorithms for any c . In Section 3.2 we use parts of the construction to improve the upper bound for instances that force offline players to lose from 69600 down to 435 pieces on a standard-size board. This adversarial strategy is also used in Section 3.3 to show that such loss-forcing instances exist for every board that is at least two columns wide, no matter its size or the length of the player's preview.

In Section 4 we give strictly 1-competitive online algorithms for some optimization goals. For other optimization goals, we give instances where competitiveness cannot be strict. Interestingly online players cannot play optimally for some optimization goals such as clearing as many lines as possible without the necessary lookahead, even in instances with just a single piece type.

2 Construction

This section shows a way to construct sequences in which online players necessarily lose, while offline players end with an empty board, and can continue to play. The construction works for even height and width ($w \geq 4$) of a rectangular board and is divided into the following three parts.



First, a sequence of \blacksquare and \blacksquare pieces is presented to ensure that the board is almost filled. This technique is similar to the one used by Brzustowski and Brugiél and described in Section 2.1. Second, given an almost filled board, we present two strategies an adversary can use to make online players lose, if they only have finite lookahead. Finally, we will see that





■ **Figure 1** After the first iteration of the padding sequence a wave has been created (b). After another iteration a second wave has been placed on top (c). Repeating the procedure $\frac{h}{2} - 1$ times leads to a padded board (d).

the offline player (who chose the other option and therefore has not lost yet) can also clear the whole board. This ensures that the loss of an offline player is not just delayed by a few pieces.

2.1 Padding Sequence

In this part, we will see that playing with a sequence of  and  pieces leads to a board configuration, containing *waves* alternately.





► **Definition 1 (Wave).** A wave is a pattern of cells consisting of two rows where every second cell is filled horizontally and vertically.



► **Example 2 (Wave).** Placing a  piece vertically in each lane of a 10 columns wide board gives a board configuration wave as depicted in Figure 1b. Repeating the same with  pieces stacks a wave on top of the existing one as depicted in Figure 1c.

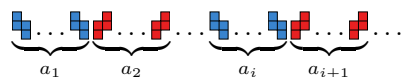
Two waves of the same orientation cannot be stacked on top of each other, as the resulting lineclears would remove one of them. Stacking several waves on each other ultimately leads to a padded board.



► **Definition 3 (Padded Board).** A padded board is an even-width even-height board configuration which consists of $\frac{h}{2} - 1$ stacked waves (of alternating orientation) and where the two topmost rows are completely empty.

► **Example 4 (Padded Board).** A padded standard-size board can be seen in Figure 1d.

Brzustowski [6] and Burgiel [7] use a similar approach, by presenting iterations of repeated  and  pieces as their piece sequence. We call piece sequences that switch between finite subsequences of  and  pieces SZ-Sequences.

► **Definition 5 (SZ-Sequence).** An SZ-Sequence is a piece sequence of alternating  and  pieces, with the amount of pieces in each iteration given by integers a_1, \dots, a_n .











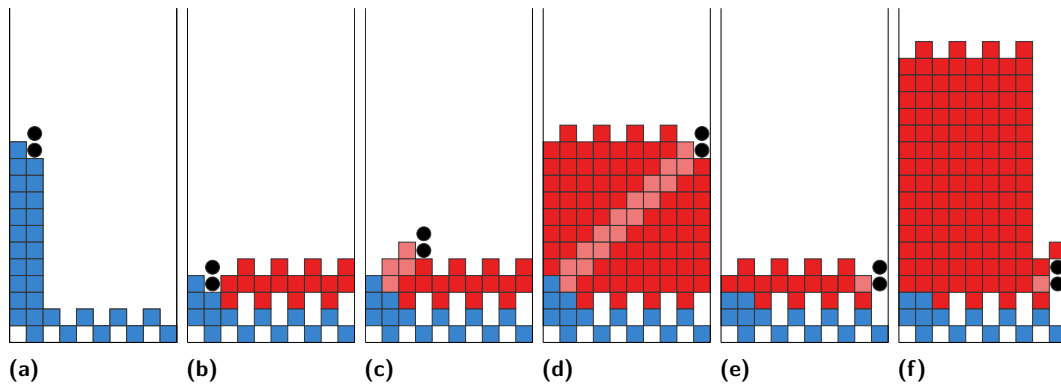
► **Example 6 (SZ-Sequence).** The sequence of 69.600 alternating  and  pieces that force an offline player to lose on a standard size board given by [7] can be characterized as an SZ-Sequence with $1 \leq i \leq 69.600$ and $a_i = 1$.

This section proposes an adapted construction that we call *padding sequence* intending to force the creation of a padded board by stacking waves on an empty board. The goal is that the competitive analysis performed in Section 3.1 is independent of the board size and does only have to regard a constant number of rows on top of the board. Thus, the goal of the padding sequence is not to force a player to lose, nonetheless Section 3.2 shows how it can be expanded to force offline players to lose and gives some results that follow. The following padding sequence is defined only for even-height boards, as this simplifies notation, and the bifurcation sequence presented in Section 2.2 is limited to even-height boards. The key to the following lemma is to prove that, given a padding sequence, a player has no other choice than to create a padded board as they would otherwise lose.

► **Lemma 7 (Padding Sequence).** *Given an empty even-width even-height board, there exists an SZ-Sequence with a set of integers $a_1, \dots, a_{\frac{h}{2}-1} \in \mathbb{N}$ such that, after all pieces have been played, the resulting board configuration is a padded board or the player has lost.*

Proof. The padding sequence consists of $\frac{h}{2} - 1$ iterations, each with a multiple of $\frac{w}{2}$ pieces and enough pieces to force the creation of a wave. Given these constraints, the player has four different options on how to play an iteration. The key of this proof is to show, that any other option than to create a wave (without residual pieces) in each iteration, will lead to a loss of the player.

1. **Playing all pieces vertically in lanes and distributed equally.** Given that the iteration consists of a multiple of $\frac{w}{2}$ pieces, this adds a wave to the board configuration. All following options will add additional unremovable rows to the board configuration and thus lead to a loss of the game.
2. **Playing some pieces horizontally (i.e., rotated as  / ).** According to [6] the horizontal piece can not be removed by additional  or  pieces and thus two more rows have been added to the board configuration that cannot be removed. Given that each iteration consists of at least $\frac{w}{2}$ pieces and there can only fit $\frac{w}{2} - 1$ horizontal pieces in two rows, the resulting board configuration after placing pieces horizontally must be at least two rows taller than if Option 1 had been chosen.
3. **Playing some pieces vertically in between lanes.** If a piece has been played vertically in between lanes, the adjacent cells to the right of the  piece (or left of the  piece) must be filled. If they are not filled, then two additional rows have been created that cannot be removed anymore compared to Option 1. To fill the adjacent cells another in-between piece must be played. This can be repeated until the adjacent cells to be filled reach the outermost column of the board. There they cannot be filled anymore and if enough pieces are played a piece must be placed to irrevocably cover the unfilled cells thus creating two additional unremovable rows compared to Option 1.
4. **Playing all pieces vertically in lanes, but not distributed equally.** The resulting board configuration has a wave and some residual pieces stacked in some lanes. When switching to the following iteration, the player has two options.
 - (a) Play all pieces of the following iteration vertically in lanes. For each piece in a lane that is not part of a wave two unremovable rows would be created by the switch from  to  or vice-versa.
 - (b) Play some pieces of the following iteration in-between lanes to avoid multiple pairs of unremovable rows like in Option 4a. However, playing in-between lanes will also lead to additional unremovable rows as described in Option 3 and depicted in Figure 2.



■ **Figure 2** Example that distributing pieces unequally during one iteration of the padding sequence leads to additional unremovable rows. If the player plays the \blacksquare pieces exclusively in lanes, the two black circles will not be filled thus creating two unremovable rows. Also filling them with a piece in-between lanes (marked with lighter shade in (c)) just moves the cells to be filled two columns to the right.

This can be repeated until the marked cells reach the side of the board. Latest here there is no way to fill the two black circles with \blacksquare pieces. This however must happen since the other lanes just allow a finite amount of pieces until they are full, thus creating two more unremovable rows.

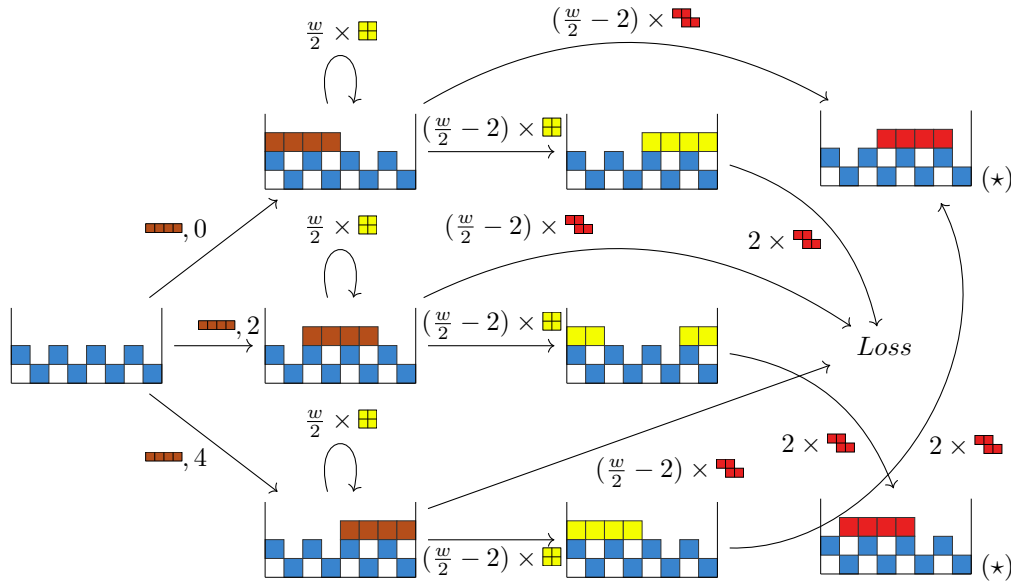
Playing all pieces vertically in lanes and distributing them equally creates a wave in each iteration. After $\frac{h}{2} - 1$ many iterations that yields $\frac{h}{2} - 1$ many stacked waves which equals a padded board. If however, the player chooses to deviate from the procedure and place some piece in another way, then at least two additional unremovable rows would be added to the board configuration as detailed in Options 2 - 4. This would lead to a loss of the player as there would not be enough space left on the board to place the remaining pieces of the padding sequence. Hence, given a padding sequence, a player has no other option than to play according to Option 1 and create a padded board. ◀

2.2 Bifurcation Sequence

We use the notion of an adversary to construct a piece sequence where an online player is forced to lose while an offline player can continue to play. Therefore, the piece sequence requires an element where offline and online players make different decisions. If the lookahead of an online player enables them to see a sufficient part of the piece sequence at the time of the decision, they would be able to avoid loss by choosing to play the item the same way as the offline player. Since we want to show that no finite amount of lookahead is sufficient, we construct the piece sequence in a way that the correct decision is dependent on pieces arbitrarily far in the future.

As such we construct two bifurcation sequences, both consisting of three parts: the decision, the consequence, and a looping part in between to push the two further away than the lookahead size. The first sequence can be used for an even width greater or equal than six, the second for a width of four.

► **Lemma 8 (Bifurcation Sequence).** *Given an even-width ($w \geq 6$) even-height ($h \geq 2$) padded board, then there is a piece sequence that forces an online player to lose while an offline player can continue to play.*



■ **Figure 3** Game graph for bifurcation sequence presented in Lemma 8 with board width $w = 8$. The board configurations marked with (\star) are only reachable for offline players.

Proof. Without loss of generality assume that the topmost wave is induced by \blacksquare pieces.

Given the player has a lookahead of l , we choose an integer k with $k \times \frac{w}{2} > l$ and construct the following piece sequence:

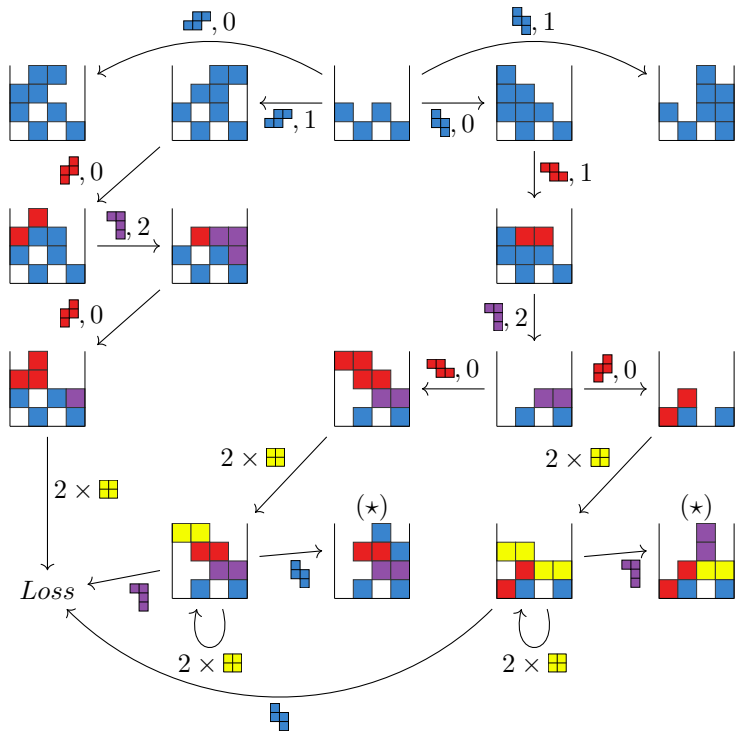
$$\underbrace{\blacksquare \dots \blacksquare}_{k \times \frac{w}{2}} \begin{cases} \underbrace{\blacksquare \dots \blacksquare}_{\frac{w}{2}-2} \underbrace{\blacksquare \blacksquare}_{2 \times 2} & \text{if first } \blacksquare \text{ is placed in the leftmost column} \\ \underbrace{\blacksquare \dots \blacksquare}_{\frac{w}{2}-2} & \text{otherwise} \end{cases}$$

The \blacksquare can only be placed horizontally in every second column such that it covers just two lanes. Otherwise, the following \blacksquare pieces would not fit on the board and the player would lose.

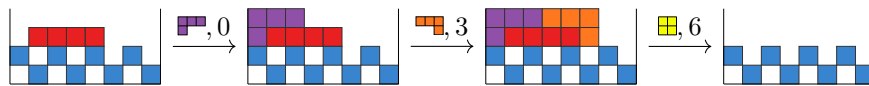
The following set of repeating $\frac{w}{2}$ many \blacksquare pieces always returns to the same board configuration. Once the set has been repeated often enough to surpass the online player’s lookahead the adversary can choose the appropriate response to force the online player to lose according to the case distinction above. Hence, the adversary forces the online player to lose with this sequence. An example for a board width of eight is given in Figure 3. As an offline player know the full piece sequence of the entire game beforehand, they can choose the correct way to place the initial \blacksquare piece, as they know the response that will come later. Thus, an offline player is not forced to lose with this sequence and can continue to play. ◀

► **Lemma 9.** *Given an even-width ($w \geq 4$) even-height ($h \geq 4$) padded board then there is a piece sequence that forces an online player with arbitrary but finite lookahead to lose while an offline player can continue to play.*

16:8 Tetris Is Not Competitive



■ **Figure 4** Game graph for bifurcation sequence described in Lemma 9 for boards with $w = 4$, even h , and $h \geq 4$. The board configurations marked with (*) are only reachable for offline players.

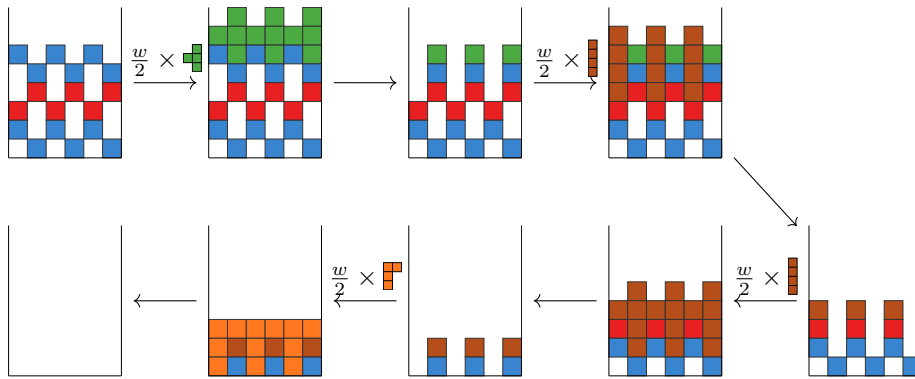


■ **Figure 5** Example of the first part of a clearing sequence for bifurcation sequence presented in Figure 3.

Proof. Without loss of generality assume that the topmost wave is induced by blue pieces. There is a bifurcation sequence for padded boards with width $w = 4$ consisting of the following pieces.

$$\begin{array}{c}
 \text{blue piece} \quad \text{red piece} \quad \text{purple piece} \quad \text{red piece} \quad \text{yellow piece} \quad \dots \quad \text{yellow piece} \\
 \uparrow \\
 2k, k > l
 \end{array}
 \left\{ \begin{array}{l}
 \text{purple piece} \quad \text{if marked piece } (\uparrow) \text{ has been placed horizontally} \\
 \text{blue piece} \quad \text{if marked piece } (\uparrow) \text{ has been placed vertically}
 \end{array} \right.$$

As can be seen from Figure 4 the online player has to choose how to place the second red piece and loses either way. An offline player however will choose the opposite way and therefore reaches one of two possible board states marked with (*) in Figure 4. ◀



■ **Figure 6** Example of the second part of a clearing sequence for a padded board with width $w = 6$ and height $h = 8$.

2.3 Clearing Sequence

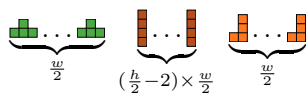
After the bifurcation sequence has been played, the online player must have lost, while there are still some board configurations the offline player can reach without having lost. It is important to show that the loss of the offline player has not been delayed by a finite number of pieces but instead that the offline player can continue to play forever if they are given an appropriate piece sequence. The following lemma shows that every board configuration resulting from the bifurcation sequence can be transformed into an empty board by a piece sequence we call *clearing sequence*.

► **Lemma 10** (Clearing Sequence). *There is a piece sequence that can be played after the bifurcation sequence from Lemma 8 or Lemma 9 to clear the board.*

Proof. Given the bifurcation sequence for a board with a width of four there are two board configurations only reachable for offline players (marked with \star) in Figure 4). For both cases, there is a piece sequence that removes the four topmost rows of the board given by $\begin{matrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{matrix}$ and $\begin{matrix} \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \\ \blacksquare & \blacksquare & \blacksquare & \blacksquare \end{matrix}$ respectively. This either leaves us with an empty or padded board, which can also be removed as shown below.

If the bifurcation sequence for a board of width six or more was used, there are four filled cells in the penultimate row in (0-indexed) columns $x, x+1, x+2, x+3$ for $x \in \{1, 3, \dots, w-5\}$. As the four filled cells on the penultimate row always start in an odd column, by placing a \blacksquare and a \blacksquare around it, the remaining space can be filled by $\frac{w}{2} - 3$ many \blacksquare pieces and the induced line clear removes everything from the top two rows. The result is a return to the padded board as can be seen in Figure 5.

Both cases leave us with a padded board, which can be deconstructed as seen in Figure 6 using the following piece sequence:



The rotation of the \blacksquare / \blacksquare piece should be chosen based on whether the topmost wave was induced by \blacksquare or \blacksquare pieces. The \blacksquare pieces are played in $\frac{h}{2} - 2$ iterations with each $\frac{w}{2}$ pieces to remove all but the last wave. The last wave is removed by $\frac{w}{2}$ many \blacksquare pieces and the result is an empty board. ◀

3 Consequences

The strategy presented in the previous chapter is useful for different results, that follow immediately as a corollary and are presented in this section. First, online players cannot perform competitively against offline players for different optimization goals. Second, we compute the number of pieces required for the padding sequence on a standard-size board and show that there is a piece sequence that forces any player to lose in just 435 pieces. Third, as the padding sequence applies to any even-width board we can generalize the known results by Brzustowski [6] and Burgiel [7], and show that there exists a piece sequence that forces an offline player to lose on any board that is at least two columns wide.

3.1 Competitive Analysis of Online Tetris

The padding, bifurcation, and clearing sequences of the previous section can be combined to show that an online player with any lookahead of $l \in \mathbb{N}$ pieces will perform arbitrarily badly in comparison to an offline player who knows the entire piece sequence from the beginning of the game. However, for Tetris, it is not immediately clear in which ways the performance is measured. This observation was already made by Demaine et al. when they analyzed approximation algorithms [9]. We are using similar optimization goals to [9] to show that the restriction to finite lookahead is crucial for any strategy. While the first three problems are identical to the ones analyzed by Demaine et al. [9] and known to be NP-hard, the fourth is a variation since we are interested in the highest filled cell at the end and not during the game.

► **Definition 11.** *Given an instance I consisting of an initial board configuration and a piece sequence $p_1 p_2 \dots p_n$ with $p_i \in \{\text{blue square}, \text{red square}, \text{orange square}, \text{yellow square}, \text{green square}, \text{purple square}, \text{brown square}\}$, we define the following four optimization goals:*



1. *SURVIVAL MAXIMIZATION: Number of pieces played in Instance I before losing*
2. *LINECLEAR MAXIMIZATION: Number of lineclears performed in Instance I before losing*
3. *TETRIS MAXIMIZATION: Number of tetrises (four lineclears at once) performed in Instance I before losing*
4. *HIGHEST FILLED CELL MINIMIZATION: Highest row with filled cells at the end of the game*

The first three problems can be understood as a proxy for the decision problem of survival (*Can a player place all given pieces without losing?*), and the fourth as a proxy for the decision problem of clearing (*Can a player place all piece in such a way that the resulting board configuration only contains empty cells?*). The bifurcation sequences presented in the previous chapter apply to even-width even-height boards with at least four columns. It follows as a corollary from the construction that for each of the four problems there cannot be any c -competitive online algorithm for any possible value of c .

► **Theorem 12.** *There cannot be any c -competitive online algorithm for ONLINE SURVIVAL MAXIMIZATION, ONLINE LINECLEAR MAXIMIZATION, ONLINE TETRIS MAXIMIZATION, and HIGHEST FILLED CELL MINIMIZATION for any c for even-width even-height boards with $w \geq 6, h \geq 2$ and $w = 4, h \geq 4$.*



It is easy to see that this result also stays true for the maximization goals when allowing non-strict competitiveness. Note that for Tetris Maximization the height of the board must be at least 4.





Proof. For a given size of the board, construct an instance I that starts with the padding, bifurcation, and clearing instance from the previous chapter. This ensures that the online player loses with some finite amount of played pieces and cleared lines, and without having performed a Tetris.

However, the offline player can continue playing on an empty board, thus the instance continues with the necessary pieces for the optimization goals. The number of lineclears and played pieces for the offline player can easily be increased by filling up the piece sequence with arbitrarily many  pieces, which can be played forever without difficulties. Analogously the number of tetrises can be increased arbitrarily by playing  pieces if the board has a height of at least four. Since the online player could only achieve a finite amount of played pieces, cleared lines, or performed tetrises, the ratio can get arbitrarily bad.

For HIGHEST FILLED CELL MINIMIZATION, it is sufficient to see that the online player leaves the game when the full height of the board is covered, while the offline player can decrease the size down to zero with the clearing sequence. ◀

3.2 A computed result for 10×20 boards

As seen in Lemma 7 and previously in the works by Brzustowski [6] and Burgiel [7], playing sequences of  and  alternately is an efficient way to force a player to fill the board. By adding an iteration to the padding sequence the player can be forced to lose.

► **Proposition 13.** *Given a padded board where the topmost wave has been induced by  pieces (or ), then $\frac{w}{2}$  pieces (or ) force the player to lose.*

Proof. The $\frac{w}{2}$ pieces do not fit vertically in lanes on top of a padded board. Horizontally or vertically in-between lanes there is only enough room left for $\frac{w}{2} - 1$ pieces. As the player cannot perform a lineclear by playing the pieces horizontally or vertically in between lanes, the last piece necessarily loses, since the next piece might not fit onto the board anymore. ◀

The goal of this section is to find lower bounds on how many pieces each iteration of the padding sequence must contain. As seen in Lemma 7, there are several ways a player can play the piece of an iteration, but finally at least two unremovable rows will be created. Similar to [6, 7], playing pieces horizontally or vertically in-between lanes immediately creates bumps that cannot be filled with subsequent pieces. This creates additional unremovable rows faster than Option 4 (playing all pieces in lanes but not distributing them equally) in the proof of Lemma 7. Therefore, we restrict the algorithm to play new pieces vertically, and compute how many it takes to create additional unremovable rows if the pieces of the previous iteration have not been distributed equally. If the previous iteration ended with a wave without any deviations, the amount of pieces required for the following iteration is given by completely stacking pieces in all lanes except of one, plus one piece. While it is obvious to see that distributing the pieces equally is a rather fast way to create a wave and the corresponding unremovable rows, it is not so clear which kind of deviation is able to prolong the construction of the wave most. Therefore, Algorithm 1 computes the number of pieces that suffices to force the creation of additional unremovable rows, given any possible way of unequally distributing the pieces in lanes of the previous iteration.



Since it is also possible to play pieces in-between lines like in Figure 2 to avoid new unremovable rows for some finite time, Algorithm 1 tries to prolong this for as many rows (and therefore pieces) as possible. In fact, the result pictured in Example 2 turns out to be the worst case. There, the rightmost piece in picture f is the first which covers the cells (with the black dots) irrevocably. The algorithm avoids covering the cells and instead loses the

16:12 Tetris Is Not Competitive

game, as there is not any other option. Hence, the number of pieces played by the algorithm until loss is the same as the maximum number of pieces that the player can play until they have to have created additional unremovable rows.

■ **Algorithm 1** Compute iteration sizes for padding sequence.

```

for all iterations  $a_2, \dots, a_{\frac{h}{2}-1}$  do
   $result \leftarrow 0$ 
  for all deviations playable in the previous iteration do
     $counter \leftarrow 0$ 
    for all lanes  $l$  without residual pieces do
      Play piece in lane  $l$ 
       $counter \leftarrow counter + 1$ 
    end for
    repeat
      Play piece in lowest leftmost (for , rightmost for ) possible placement which
      does not cover a cell,
       $counter \leftarrow counter + 1$ 
    until game over
    if  $counter > result$  then
       $result \leftarrow counter$ 
    end if
  end for
  ceil result to next multiple of  $\frac{w}{2}$ 
end for

```

Applying the algorithm to a board with ten columns and twenty rows leads to the results in Table 1. This yields the following result.

► **Theorem 14.** *Given an empty 10×20 board, there exists an SZ-Sequence with a set of integers $a_1, \dots, a_{10} \in \mathbb{N}$ that forces an offline player to lose that is 435 pieces long.*


Proof. The first nine iterations are the padding sequence as described in Lemma 7, and the final iteration in Proposition 13.

- The first iteration requires $\lceil (\frac{w}{2} - 1) \times (\frac{h}{2} - 1) + 1 \rceil \frac{w}{2} = 40$ pieces. The first $(\frac{w}{2} - 1) \times (\frac{h}{2} - 1)$ pieces may be filled into all but one lane, the next piece must necessarily be placed into the last lane to induce one line clear. The number of pieces must be ceiled to the next multiple of $\frac{w}{2}$ to enable the creation of a wave without residual pieces.
- Iterations a_2 to a_9 can be computed with Algorithm 1. See Table 1 for the results.
- In the final iteration $\frac{w}{2} = 5$ force the player to lose according to Proposition 13.

Adding the sizes of each iteration yields a sum of 435 pieces. ◀

The results of 435 pieces to force a player to lose on a standard-size board improves on the previously known upper bound of 69600 pieces by Burgiel [7]. Since ensuring that waves are created is not a necessary requirement for losing, it is plausible that even faster ways exist that do not enforce waves.

3.3 Offline players lose on every board of width at least two

We use the padding sequence with Proposition 13 together with existing results to show that every board that is at least two columns wide has a losing piece sequence. Since we expect every piece to physically fit on the empty board,  is the only allowed piece type for a single-column instance, and a player cannot lose.

■ **Table 1** Results for applying Algorithm 1 on a standard-size board. Deviations are given as a sequence of integers representing the number of pieces played per lane. The number 5,0,0,0,0 describes 5 pieces in the first lane and 0 pieces in the following ones. Iteration a_2 is depicted in Figure 2.

Iteration	Best Deviations	Unceiled Result	Ceiled Result
a_2	5,0,0,0,0	69	70
a_3	5,0,0,0,0	65	65
a_4	5,0,0,0,0	61	65
a_5	5,0,0,0,0	57	60
a_6	5,0,0,0,0	53	55
a_7	2,0,3,0,0 2,3,0,0,0	30	30
a_8	2,0,3,0,0	26	30
a_9	1,0,2,2,0 2,1,0,2,0 2,2,1,0,0	13	15

While it is known that there are losing piece sequences for offline players on odd-width boards ($w \geq 3$) [6] and for boards with a width of $2 \bmod 4$ [7], the case for boards with width $0 \bmod 4$ remained open. Brzustowski also proved that online players cannot win on boards of even width [6].

► **Theorem 15.** *For any empty board with a width of at least two, there is a piece sequence that forces an offline player to lose.*

Proof. For $w \geq 3$, $w \bmod 2 = 1$ Brzustowski gives loss-forcing piece sequences [6]. Combining the padding sequence (Lemma 7) and Proposition 13 gives a piece sequence that forces any player to lose on even-width boards. ◀

4 Competitiveness of Single Piece Tetris

This section looks at the competitiveness of a constrained version of Tetris. Brzustowski has shown that limiting the piece sequences of the game to a single piece type allows strategies to play forever, at least if the board size allows it [6]. We show that for each optimization goal analyzed in the previous section, there is an online algorithm that is 1-competitive if the piece sequence is constrained to contain only a single piece type. However, for some optimization goals, there are instances that do not allow strict 1-competitive algorithms. The constrained online problems are defined as follows.

► **Definition 16** (Single Piece Online Tetris). *The Online Tetris Problems defined in Definition 11 where every piece sequence can only contain elements of the same piece type are prefixed with SINGLE PIECE.*

4.1 Single Piece Survival Maximization and Single Piece Tetris Maximization have strictly optimal online algorithms




The following theorems show that there is a strategy to play optimally for SINGLE PIECE SURVIVAL MAXIMIZATION and SINGLE PIECE TETRIS MAXIMIZATION.

► **Theorem 17.** *There is a strictly 1-competitive online algorithm for SINGLE PIECE SURVIVAL MAXIMIZATION.*

16:14 Tetris Is Not Competitive

Proof. If there is no perpetual loss-avoiding strategy for the given board size and piece type, then online and offline players will play such that the highest achievable number of moves is played before losing. If there is a perpetual loss-avoiding strategy both players will place every single piece of the piece sequence without losing. In both cases, $gain_{\text{ALG}}(I) = gain_{\text{OPT}}(I)$ holds, and therefore, the strategy employed by the online player is strictly 1-competitive. ◀

► **Theorem 18.** *There is a strictly 1-competitive online algorithm for SINGLE PIECE TETRIS MAXIMIZATION.*


Proof. If the chosen piece type is not  or $h < 4$ then no tetrises can be played and hence $gain_{\text{ALG}}(I) = gain_{\text{OPT}}(I) = 0$. If the chosen piece type is  and the board has at least four rows then tetrises can be created by placing  pieces in columns next to each other until the last column is filled and a Tetris is performed. This leads to $gain_{\text{ALG}}(I) = gain_{\text{OPT}}(I) = \lfloor \frac{|P|}{w} \rfloor$ with $|P|$ the length of the piece sequence, which cannot be beaten by an offline algorithm. ◀

4.2 Online Players cannot perform optimally for Single Piece Lineclear Maximization and Single Piece Highest Filled Cell Minimization

Compared to the two optimization goals above we can construct instances for SINGLE PIECE LINECLEAR MAXIMIZATION and SINGLE PIECE HIGHEST FILLED CELL MINIMIZATION where online players are not able to perform optimally. Nonetheless, there are 1-competitive online algorithms for the problems.



The key to creating such instances is that an offline player can choose to play the final piece in such a way that improves the optimization function but would lead to a loss if the piece sequences did not end. As the offline player chooses to deviate from a perpetual loss-avoiding strategy at some point since they know when the instance ends, which an online player can not.


► **Theorem 19.** *There cannot be a strictly 1-competitive online algorithm for SINGLE PIECE LINECLEAR MAXIMIZATION without lookahead.*

Proof. Assume there is an online strategy to achieve optimality. On a 6×4 board and with  items, it, therefore, has to put the first two pieces horizontally to get the first lineclear to play optimal, since the instance could end after those two items and the optimal solution would also have a lineclear. Since an optimal algorithm can get two lineclears with four pieces, an online player has to place the next two pieces horizontally as well. This continues with the fifth piece and the sixth piece for the third lineclear with the next two horizontally placed pieces. Therefore, the online player is left with a configuration with three filled cells in the second and fifth column, where it is not able to make any further lineclears with the following items. Therefore, a strategy that guarantees optimality for the first items will necessarily lose after finitely many steps, while there is a strategy to play infinitely long. ◀

Algorithms with a non-strict competitive ratio of $c = 1$ exist, using the same strategies as in Theorem 17. The instance above requires an online player that does not have any lookahead. If there is an instance that is non-strictly competitive for SINGLE PIECE LINECLEAR MAXIMIZATION even if the online player is granted some lookahead remains an open question. However, we believe that an algorithm needs arbitrarily much lookahead if the instance gets wider.

► **Theorem 20.** *There cannot be a strictly 1-competitive online algorithm for SINGLE PIECE HIGHEST FILLED CELL MINIMIZATION.*

Proof. Given an instance I with a 4×4 board and a single  piece the optimal strategy is to place it horizontally, resulting in a maximum height of 2. However, an online player without lookahead cannot place the item horizontally, as an adversary could play additional  pieces which force a loss. Since it is possible to continue playing by placing the pieces vertically, it is crucial to know if more than one item is coming or not to decide whether placing the first item vertically or horizontally is optimal. ◀

Note that a single piece strategy needs at most eight rows [6], hence 1-competitive online algorithms with additive constant $\alpha \leq 8$ exist. Here, the instance given above can be extended to cover any available lookahead l by making the board $4 + 2 \times l$ columns wide. Again, the online player must either start by playing the  pieces vertically or horizontally, depending on their beliefs about what would be optimal. This however is still hidden behind the lookahead pieces.

5 Conclusion and Open Problems

In this article, we were able to see that for some instances knowing the whole instance is crucial if a player wants to avoid losing. However, this only works if the adversary is allowed to use the presented construction which leads to some open problems discussed in this section. Our construction works on any even-width, even-height board with $w = 4, h \geq 4$ and $w \geq 6, h \geq 2$ leaving out two classes of boards: boards with a width of just two columns and boards of odd width.

We analyzed a very restricted class of Tetris games limited to a single piece type, and have shown that online players (even without lookahead) can play optimally. Interestingly, for some optimization goals, the competitiveness is not strict. Another known restricted class of Tetris games can be found in some implemented video games with a standard-size board, a 7-bag randomizer, three pieces of lookahead, and one piece of hold. A perpetual loss-avoiding strategy for this class is given in [1].

Somewhere between those results must be the point, when knowing the future gets crucial. How can the game be restricted such that a preview of one, three, or infinite many pieces starts making a difference? One main feature of Tetris in favor of a player is the option to hold a piece, which can be taken at later steps. Similar to lookahead, this feature relaxes the need to play immediately in an online way. Therefore, the influence of the option to hold would be interesting: does Tetris get competitive if the player is allowed to hold one or more items?

On the other hand, lookahead and hold are two features to relax the strict online setting of Tetris: for online problems, many more relaxations such as advice, randomization, reservations, or predictions are known and part of current research. For future work it would be interesting to see how such relaxations applied to Tetris behave, and how large their impact is.

Last, we believe that 435 pieces are still just an upper bound for an instance size that is sufficient to force an (offline) player to lose a game of Tetris on a standard-size board. Since finding the actual number was neither a primary goal for Burgiel [7] nor for us, we believe that the bound can be decreased even further with a dedicated approach.

References

- 1 Playing forever - Hard Drop Tetris Wiki. https://harddrop.com/wiki/Playing_forever. Accessed: 2023-10-22.

16:16 Tetris Is Not Competitive

- 2 Sualeh Asif, Michael Coulombe, Erik D. Demaine, Martin L. Demaine, Adam Hesterberg, Jayson Lynch, and Mihir Singhal. Tetris is NP-hard even with $O(1)$ Rows or Columns. *Journal of Information Processing*, 28:942–958, 2020. doi:10.2197/ipsjjip.28.942.
- 3 Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.
- 4 Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. TETRIS IS HARD, EVEN TO APPROXIMATE. *International Journal of Computational Geometry & Applications*, 14(01n02):41–68, April 2004. doi:10.1142/S0218195904001354.
- 5 Ron Breukelaar, Hendrik Jan Hoogeboom, and Walter A Kosters. Tetris is Hard, Made Easy. *Technical Report 2003-9, Leiden Institute of Advanced Computer Science*, 2003.
- 6 John Brzustowski. Can You Win at Tetris? Master’s thesis, The University of British Columbia, March 1992. doi:10.14288/1.0079748.
- 7 Heidi Burgiel. How to lose at Tetris. *The Mathematical Gazette*, 81(491):194–200, July 1997. doi:10.2307/3619195.
- 8 Justin Dallant and John Iacono. How fast can we play tetris greedily with rectangular pieces? *Theoretical Computer Science*, 992:114405, 2024. doi:10.1016/j.tcs.2024.114405.
- 9 Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is Hard, Even to Approximate, October 2002. arXiv:cs/0210020.
- 10 Vadim Gerasimov. Original Tetris: Story and Download. <https://vadim.oversigma.com/Tetris.htm>. Accessed: 2023-10-23.
- 11 Dennis Komm. *An Introduction to Online Computation – Determinism, Randomization, Advice*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016. doi:10.1007/978-3-319-42749-2.
- 12 Simon Laroche. The history of Tetris randomizers. <https://simon.lc/the-history-of-tetris-randomizers>, November 2018. Accessed: 2023-10-01.
- 13 Daniel Dominic Sleator and Robert Endre Tarjan. Amortized efficiency of list update and paging rules. *Commun. ACM*, 28(2):202–208, 1985. doi:10.1145/2786.2793.
- 14 Oscar Temprano. Complexity of a Tetris variant. *CoRR*, June 2015. arXiv:1506.07204.
- 15 Kaitlyn Tsuruda. A closer look at TETRIS: Analysis of a variant game. *Atlantic Electronic Journal of Mathematics [electronic only]*, 4, January 2011.
- 16 Luca Venier. Online Tetris. Master’s thesis, RWTH Aachen University, November 2023.

Computational Complexity of Matching Match Puzzle

Yuki Iburi ✉

The Digital Value, LTD., Tokyo, Japan

Ryuhei Uehara ✉ 🏠 

School of Information Science, Japan Advanced Institute of Science and Technology, Tokyo, Japan

Abstract

Various forms of graph coloring problems have been studied over the years in the society of graph theory. Recently, some original puzzles are popularized in Japanese 100-yen shops, and one of them can be formalized as a graph coloring problem in a natural way. However, this natural graph coloring problem has not been investigated in the context of the graph theory. In this paper, we investigate this puzzle as a graph coloring problem. We first prove that this graph coloring problem is NP-complete even when the graph is restricted to a path or a spider. In these cases, diameter of the graphs seems to play an important role for its difficulty. We then show that the problem can be solved in polynomial time when the graph is restricted to some graph classes of constant diameter.

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

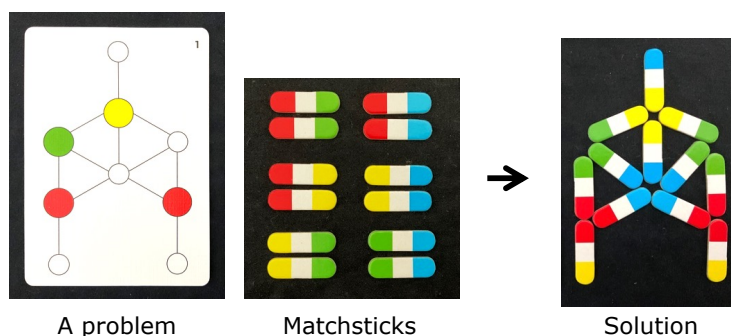
Keywords and phrases Graph coloring, Matching Match puzzle, NP-complete, polynomial-time solvable

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.17

Funding *Ryuhei Uehara*: JSPS KAKENHI Grant Numbers 20H05961, 20H05964, 24H00690.

1 Introduction

Research on computational complexity of puzzles and games has a long history. One of the reasons is that there exist some puzzles that characterize major computational complexity classes in natural and simple ways, and hence the features of these puzzles give us some understanding of such complexity classes [3, 6]. Especially, many NP-complete puzzles have helped us to obtain some intuitions for the class NP. They may lead us to the solution to the $P \neq NP$ conjecture, which is one of the millennium prize problems.



■ **Figure 1** The matching match puzzle sold at Daiso.

In this paper, we investigate a puzzle named “Matching Match” (Figure 1). This puzzle is designed by Rikachi, a Japanese puzzle designer, and it is a commercial product produced by Daiso, which is one of the major 100-yen shops in Japan.¹ As a commercial product, the rule of this puzzle is simple and understandable even for children. You are given a card and a set of “matchsticks”. On the card, a planar graph is drawn and some vertices of it are colored. Each matchstick has its own colors on both endpoints (the colors can be the same). The number of matchsticks is equal to the number of edges of the graph. That is, each matchstick corresponds to an edge of the graph and vice versa. The puzzle asks us to find an arrangement of the matchsticks on the edges so that every color matches at each vertex of the graph. The color of an endpoint should be matched to the color on the vertex if the vertex is colored. On the other hand, we have to assign a color to each vertex if it is not colored. That is, at a vertex without color, all the endpoints of the matchsticks sharing the vertex have to have the same color.

It is easy to see that the puzzle can be easily solved if every vertex of the graph on a card has its color. Namely, for a given graph, this puzzle asks us to find a proper coloring of the uncolored vertices in the graph for a given set of edges with pre-colored endpoints. As you can find in Figure 1, it is a natural problem not only in the context of the puzzle, but also as a variant of graph coloring problems. The graph coloring problem is one of classic problems which has been widely investigated in the context of theoretical computer science [4]. From the viewpoint of algorithmic technique, the color-coding is one of technique for solving a graph coloring problem efficiently [1]. However, as far as the authors know, this graph coloring problem corresponding to the matching-match puzzle has never been investigated in the context of graph coloring. One of the reasons may be that this graph coloring admits to color two neighbors with one color when we have an edge with endpoints with pre-colored by the same color. In fact, we will use such edges in our reduction.

We first show that the matching-match puzzle is NP-complete even if the graph is quite restricted. Precisely, this puzzle is NP-complete even if the graph is a spider, a path, or a cycle in general. On the other hand, when the graph is a complete graph or a star the matching-match puzzle is polynomial-time solvable. We note that a star is a spider with legs of length 1. That is, we have a constant B' such that the matching-match puzzle is NP-complete on spiders with legs of length at least B' , and it is polynomial-time solvable on spiders with legs of length at most B' . In this paper, we also show a polynomial-time algorithm on spiders with legs of length at most 2. That is, we prove that $B' \geq 2$. The keen threshold value of B' is open.

2 Preliminaries

In this paper, we only consider a simple graph $G = (V, E)$ with $|V| = n$ and $|E| = m$. A sequence of the vertices (v_0, v_1, \dots, v_k) is a *path* of length k in G when $\{v_i, v_{i+1}\} \in E$ for each $i = 0, \dots, k - 1$. It is a *cycle* of size k if $v_0 = v_k$ and $k > 2$. A graph is a *tree* if it is acyclic and connected. A tree is a *spider* if it has only one vertex of degree greater than 2. The unique vertex of degree greater than 2 of a spider is called the *body* of it. A spider consists of three or more paths sharing the body. Each path (including the body) is called a *leg* of the spider. A graph is a *star* if it is a spider with legs of length 1. A graph is *complete* if every pair of vertices are joined by an edge. A complete graph is denoted by K_n if it consists of n vertices. For a vertex v in V , its *neighbor set* in $G = (V, E)$ is defined by

¹ English instruction can be found at <https://www.daiso-syuppan.com/noutore/>.

$\{u \mid \{u, v\} \in E\}$ and denoted by $N_G(v)$ (or $N(v)$ if G is clear in the context). We also use a notation $N_G[v]$ and $N[v]$ defined by $N_G(v) \cup \{v\}$. The *distance* between two vertices u and v in G is the minimum length of a path joining u and v . The diameter of G is the maximum distance between all pairs of vertices in G . For a given graph $G = (V, E)$, an edge set M is a *matching* if no two edges share a common vertex. For a given graph $G = (V, E)$ and a vertex subset V' , we call the graph $G' = (V', E')$ with $E' = \{\{v, v'\} \mid v, v' \in V' \text{ and } \{v, v'\} \in E\}$ an *induced subgraph* by V' , and it is denoted by $G[V']$. A vertex set Q is called a *clique* if $G[Q]$ is a complete graph.

Now we turn to the definition of the matching-match puzzle. An instance of the matching-match puzzle consists of a graph $G = (V, E)$ with $V = \{v_0, v_1, \dots, v_{n-1}\}$ and a set of *sticks* $S = \{s_0, s_1, \dots, s_{m-1}\}$. We define the set of endpoints of sticks by U . That is, each stick s_i is a pair of distinct vertices $\{u_i, u'_{i,2}\}$, where $u_i, u'_{i,2} \in U$ and hence $|U| = 2m$. We will use two color sets $C_0 = \{0, 1, \dots, c\}$ and $C_1 = \{1, \dots, c\}$ for some positive integer c to distinguish colors in V and U . Each vertex v in V is colored by a function $C_0 : V \rightarrow C_0$ and each vertex u in U is colored by a function $C_1 : U \rightarrow C_1$. We say a vertex v in V is *not colored* if $C_0(v) = 0$. The other vertices in V and U are *colored*. Then the input of the matching-match puzzle is a 5-tuple (G, S, c, C_0, C_1) . An instance (G, S, c, C_0, C_1) is *feasible* if and only if there exists a mapping \mathcal{M} from U to V such that (1) for each $s_i = (u_i, u'_{i,2})$, $\{\mathcal{M}(u_i), \mathcal{M}(u'_{i,2})\}$ is in E , (2) for each $u \in U$, $C_1(u) = C_0(\mathcal{M}(u))$ or $C_0(\mathcal{M}(u)) = 0$, (3) for each $e \in E$, there exists a stick $s_i = \{u_i, u'_{i,2}\}$ with $e = \{\mathcal{M}(u_i), \mathcal{M}(u'_{i,2})\}$, and (4) for each $v \in V$ with $C_0(v) = 0$, there exists a color c' such that all vertices $u \in U$ with $\mathcal{M}(u) = v$ satisfy $C_1(u) = c'$. The *matching-match puzzle* asks us if there exists a feasible mapping \mathcal{M} for a given instance (G, S, c, C_0, C_1) . That is, we can formalize the matching-match puzzle as follows:

MATCHING-MATCH PUZZLE

Input: A graph $G = (V, E)$, a set S of sticks, an integer $c > 0$, and two functions C_0 and C_1 .

Output: Determine whether there exists a feasible mapping \mathcal{M} .²

We note that we consider general graphs and they are not necessarily planar. However, almost all graphs in this paper are planar except complete graphs (with at least 5 vertices). Except for complete graphs, all graphs in this paper can be drawn on a plane with edges of unit length without crossing. That is, they can be real problems in Matching Match puzzle.

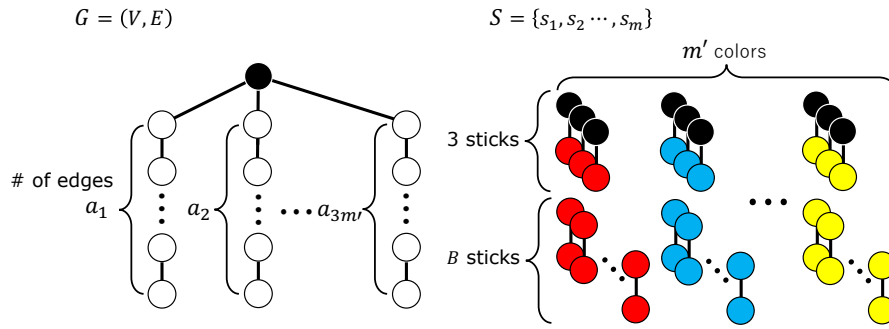
3 NP-completeness

In this section, we prove that the matching-match puzzle is intractable even if G is quite restricted.

► **Theorem 1.** *The matching-match puzzle is NP-complete in general even if G is (1) a spider, (2) a path, or (3) a cycle.*

Proof. Let (G, S, c, C_0, C_1) be an instance of the matching-match puzzle. If there is a feasible mapping \mathcal{M} , it is easy to confirm that \mathcal{M} is feasible. Thus the matching-match puzzle is in NP. Therefore we show NP-hardness. To show NP-hardness, we reduce the following 3-Partition problem to our problem:

² In the commercial puzzle, each of 40 instances has a unique solution. We do not assume it in this paper.



■ **Figure 2** Construction of a spider.

3-PARTITION

Input: Positive integers $a_1, a_2, a_3, \dots, a_{3m'}$ such that $\sum_{i=1}^{3m'} a_i = m'B$ for some positive integer B and $B/4 < a_i < B/2$ for $1 \leq i \leq 3m'$.

Output: Determine whether we can partition $\{1, 2, \dots, 3m'\}$ into m' subsets $A_1, A_2, \dots, A_{m'}$ so that $\sum_{i \in A_j} a_i = B$ for $1 \leq j \leq m'$.

It is well known that the 3-PARTITION problem is strongly NP-complete [2].

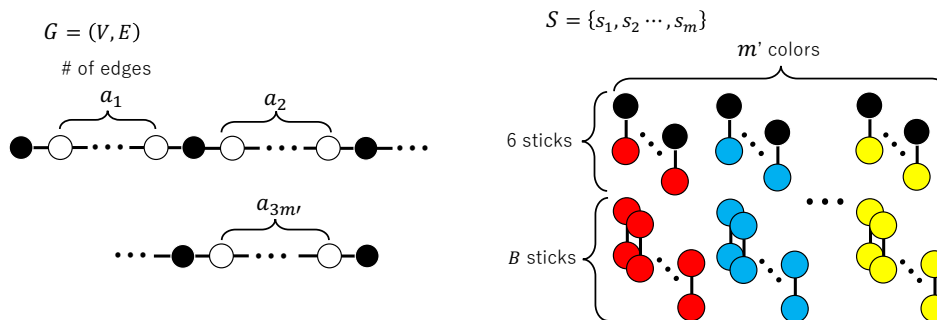
(1) We first show a reduction to a spider. For a given instance $a_1, a_2, a_3, \dots, a_{3m'}$ of the 3-PARTITION problem, we construct G and S as follows (Figure 2). We use $m' + 1$ colors, namely, $C_0 = \{0, 1, \dots, m' + 1\}$. The graph G is a spider with $3m'$ legs. For each i with $1 \leq i \leq 3m'$, the i th leg is a path of length $a_i + 1$. The body of the spider is colored by 1, and all the other vertices are not colored. That is, $C_0(b) = 1$ for the body vertex $b \in V$ and $C_0(v) = 0$ for each vertex $v \in V \setminus \{b\}$. The set S consists of $m = m'(3 + B)$ sticks. For each j with $1 \leq j \leq m'$, S contains B sticks $s = \{u, w\}$ with $C_1(u) = C_1(w) = j + 1$ and 3 sticks $s' = \{u', w'\}$ with $C_1(u') = 1$ and $C_1(w') = j + 1$.

Clearly, the reduction can be done in polynomial time. Thus we show that the instance $a_1, a_2, a_3, \dots, a_{3m'}$ of the 3-PARTITION problem has a solution if and only if G and S are feasible.

We first assume that the instance $a_1, a_2, a_3, \dots, a_{3m'}$ has a solution. Then they can be partitioned into m' subsets $A_1, A_2, \dots, A_{m'}$ such that $\sum_{i \in A_j} a_i = B$ for $1 \leq j \leq m'$. Without loss of generality, we assume that $A_1 = \{a_1, a_2, a_3\}$. Then we match three sticks $s_1 = s_2 = s_3 = \{1, 2\}$ to indicate three legs of lengths a_1, a_2 , and a_3 . That is, the function \mathcal{M} maps each 1 to the body of the spider, and 2 to the neighbors of the body corresponding to the three legs of lengths a_1, a_2 , and a_3 . Now we match the other sticks $\{2, 2\}$ to these three legs. Since $a_1 + a_2 + a_3 = B$, we can match all sticks to these legs and no sticks $\{2, 2\}$ remain. We repeat the same process for each of $A_2, \dots, A_{m'}$. Since they are a solution of the 3-PARTITION problem, we can match all the sticks.

We next assume that we can match all the sticks in S to the edges of the spider G . We first observe that all sticks that have an endpoint of color 1 to join legs to the body. Therefore, we cannot use two or more colors on a leg except the edge joining the leg to the body. Thus, each set of B sticks of the same color should be on three legs such that the total length is B . Hence we can construct a solution of the 3-PARTITION problem from the solution of the matching-match puzzle.

(2) We next show a reduction to a path. The basic idea is similar to the case (1) (Figure 3). The graph G is a path (v_0, v_1, \dots, v_m) of length $m = m'(B + 6)$. It has $3m' + 1$ vertices v with $C_0(v) = 1$, and all the other vertices are not colored. By the vertices v with



■ **Figure 3** Construction of a path.

$C_0(v) = 1$, the path is partitioned into $3m'$ subpaths since both endpoints v of the path satisfy $C_0(v) = 1$. The i th subpath has length a_i for each $i = 1, 2, \dots, 3m'$. The set S consists of $m = m'(6 + B)$ sticks. For each j with $1 \leq j \leq m'$, S contains B sticks $s = \{u, w\}$ with $C_1(u) = C_1(w) = j + 1$ and 6 sticks $s' = \{u', w'\}$ with $C_1(u') = 1$ and $C_1(w') = j + 1$. The reduction can be done in polynomial time. By using the same argument, we can show that the instance $a_1, a_2, a_3, \dots, a_{3m'}$ of the 3-PARTITION problem has a solution if and only if G and S are feasible.

(3) In the construction (2), by unifying the endpoints of the path G , we can obtain a cycle. On the cycle, the same argument works.

Therefore, the matching-match puzzle is NP-complete in general even if G is a spider, a path, or a cycle. ◀

By the proof of Theorem 1(1), we obtain the following corollary.

► **Corollary 2.** *The matching-match puzzle is NP-complete even if G is a spider and its body is the only colored vertex. Moreover, the matching-match puzzle is NP-complete even if G is a spider and no vertex is colored.*

Proof. The proof of Theorem 1(1) meets the first claim. Thus we focus on the case that the vertices in G are not colored. The construction of the graph G and the set S is the same as the proof of Theorem 1(1). The coloring of C_1 is also the same, and we define $C_0(v) = 0$ for all vertices in V . To derive a contradiction, we assume that the body b of the spider G is colored by the other color, say $C_0(b) = 2$, than 1 in a solution. In the original instance of the 3-PARTITION problem, we can assume that $m' > 6$ without loss of generality. Since we have at most B sticks $\{u, u'\}$ with $C_1(u) = C_1(u') = 2$ to cover the legs, we have to change the color on some legs from 2 to the other colors in the middle of the legs. To change the color, we have to use the vertices v with $C_0(v) = 1$. However, we have only three sticks $\{u, u'\}$ with $C_1(u) = 2$ and $C_1(u') = 1$. Thus, since $m' > 6$, there are at least 4 legs that should be totally colored by 2 from the body to their leaves. However, by the assumption that $B/4 < a_i < B/2$, we cannot cover 4 legs by B sticks, which is a contradiction. Therefore, we cannot color the body by any other color than 1. That is, we can assume that $C_0(b) = 1$ for the body vertex b without loss of generality. Thus the matching-match puzzle is NP-complete even if no vertex in G is colored. ◀

4 Polynomial time algorithms

In this section, we show polynomial-time algorithms for the matching-match puzzle on some graph classes. We first consider two simple cases that G is a complete graph and G is a star. In these two cases, we can solve the matching-match puzzle efficiently. Next we turn to the spider with legs of length 2. We give a polynomial-time algorithm for this case.

► **Theorem 3.** *For a given instance $(G, S, c, \mathcal{C}_0, \mathcal{C}_1)$, the matching-match puzzle can be solved in $O(n + m)$ time when G is a complete graph.*

Proof. We first check if G is a complete graph, and output “No” if it is not in $O(n + m)$ time. In the set of sticks in S , we can count the number of appearances of each color. We denote by $\#(i)$ the number of color i in S . Precisely, $\#(i) = \sum_{s=\{u,u'\} \in S} (\delta(\mathcal{C}_1(u) = i) + \delta(\mathcal{C}_1(u') = i))$ for each $i \in \{1, \dots, c\}$, where $\delta(\mathcal{C}_1(u) = i) = 1$ when $\mathcal{C}_1(u) = i$ and $\delta(\mathcal{C}_1(u) = i) = 0$ when $\mathcal{C}_1(u) \neq i$. Then, $\#(i)/(n - 1)$ gives the number of vertices v in G with $\mathcal{C}_0(v) = i$ after matching by a solution \mathcal{M} . If \mathcal{C}_0 cannot satisfy this condition by extension to \mathcal{C}_1 , the answer is “No”. Thus we assume that the given function \mathcal{C}_0 in the instance is consistent with the condition. (Precisely, for each color i , the number of vertices with $\mathcal{C}_0(v) = i$ is equal to or less than $\#(i)/(n - 1)$.) Now we assign the colors to the vertices v in V with $\mathcal{C}_0(v) = 0$ so that $\mathcal{C}_0(v) > 0$ and they are consistent with the condition. Since G is a complete graph, the assignment can be done according to the numbers $\#(i)/(n - 1)$. After the assignment, we check the consistency of the sticks in S . Precisely, for each edge $\{u, u'\}$, we decrease $\#(\mathcal{C}_0(u))$ and $\#(\mathcal{C}_0(u'))$ by 1, respectively. Then the sticks are consistent if and only if $\#(\mathcal{C}_0(v)) = 0$ after the decreasements. Since all the vertices in G are now colored, it can be done in $O(m)$ time. The correctness of the algorithm is trivial. Thus we can solve it in $O(m)$ time in total when G is a complete graph. ◀

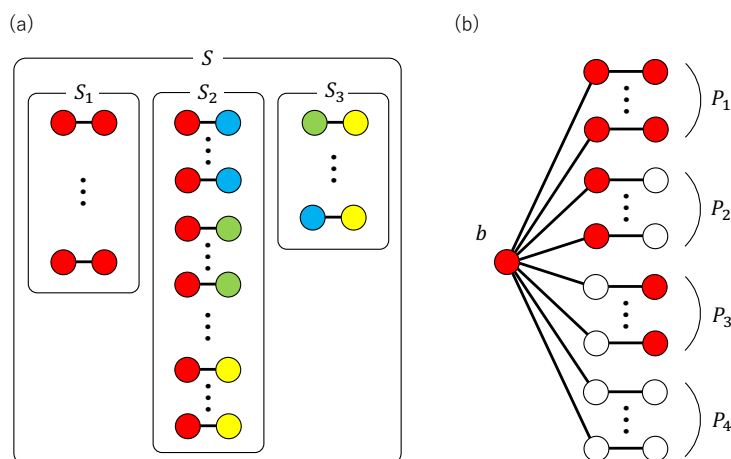
► **Theorem 4.** *For a given instance $(G, S, c, \mathcal{C}_0, \mathcal{C}_1)$, the matching-match puzzle can be solved in $O(n)$ time when G is a star.*

Proof. We first check if G is a star, and output “No” if it is not. It can be done in $O(n)$ time since G is a star if and only if it has one vertex of degree $n - 1$ and $n - 1$ vertices of degree 1. Assume $G = (V, E)$ is a star and $b \in V$ is the body vertex of degree $n - 1$. Now we pick any stick $s = \{u, u'\}$ in S . Then it should be either $\mathcal{M}(u) = b$ or $\mathcal{M}(u') = b$ if it is a yes-instance. We first check whether $\mathcal{M}(u) = b$. In this case, all sticks $s' = \{w, w'\}$ should satisfy $\mathcal{C}_1(u) = \mathcal{C}_1(w)$ or $\mathcal{C}_1(u) = \mathcal{C}_1(w')$. If all other sticks satisfy the condition, we output “Yes”. Otherwise, we check whether $\mathcal{M}(u') = b$ in the same way. If all other sticks satisfy the condition, we output “Yes”, otherwise, output “No”. The correctness of the algorithm is trivial, and it can be done in $O(n)$ time. ◀

Now we turn to the main theorem in this section:

► **Theorem 5.** *For a given instance $(G, S, c, \mathcal{C}_0, \mathcal{C}_1)$, the matching-match puzzle can be solved in polynomial time when G is a spider with legs of length 2, and $\mathcal{C}_0(v) = 0$ for all $v \in V$.*

Proof. Our algorithm checks all cases that $\mathcal{C}_0(b) = i$ with $i = 1, \dots, c$ for the body vertex b of G . Therefore, hereafter, we fix that $\mathcal{C}_0(b) = 1$ and $\mathcal{C}_0(v) = 0$ for all vertices $v \in V \setminus \{b\}$ without loss of generality. Then the set S of sticks can be partitioned into three subsets $S_1 = \{\{u, u'\} \mid \mathcal{C}_1(u) = \mathcal{C}_1(u') = 1\}$, $S_2 = \{\{u, u'\} \mid \mathcal{C}_1(u) = 1 \text{ and } \mathcal{C}_1(u') \neq 1\}$, and $S_3 = \{\{u, u'\} \mid \mathcal{C}_1(u) \neq 1 \text{ and } \mathcal{C}_1(u') \neq 1\}$ (Figure 4(a)).



■ **Figure 4** Classifications of S and V .

We suppose that there exists a feasible mapping \mathcal{M} from S to V , and consider conditions that \mathcal{M} has to satisfy. Here we color the graph G according to this mapping \mathcal{M} , and we suppose that each color of a vertex v in G can be referred as $\mathcal{C}(v)$ to simplify. Since G is a spider with legs of length 2, we have $n' = (n - 1)/2$ legs. Then, we partition these legs (b, v_1, v_2) into four subsets $P_1 = \{(b, v_1, v_2) \mid \mathcal{C}(v_1) = \mathcal{C}(v_2) = 1\}$, $P_2 = \{(b, v_1, v_2) \mid \mathcal{C}(v_1) = 1 \text{ and } \mathcal{C}(v_2) \neq 1\}$, $P_3 = \{(b, v_1, v_2) \mid \mathcal{C}(v_1) \neq 1 \text{ and } \mathcal{C}(v_2) = 1\}$, and $P_4 = \{(b, v_1, v_2) \mid \mathcal{C}(v_1) \neq 1 \text{ and } \mathcal{C}(v_2) \neq 1\}$ (Figure 4(b)).

Then we can observe that

$$\begin{aligned} 2|P_1| + |P_2| &= |S_1|, \\ |P_2| + 2|P_3| + |P_4| &= |S_2|, \\ |P_4| &= |S_3|. \end{aligned}$$

Thus, we can obtain $|P_4| = |S_3|$ first. Now the size $|P_3|$ is one of $0, 1, \dots, n'$. Therefore, our algorithm checks all cases for $|P_3| = 0, 1, \dots, n'$. Once $|P_4|$ and $|P_3|$ are fixed, $|P_2|$ and $|P_1|$ are also determined (when one of them is negative, the case is not feasible). Therefore, our algorithm checks if there is a feasible mapping \mathcal{M} from S to V for given $|P_1|, |P_2|, |P_3|$, and $|P_4|$. Intuitively, we have to make two matchings; one between S_2 and S_3 in P_4 and another one among S_2 in P_3 . The remaining edges in S_2 can be matched in P_2 in any way.

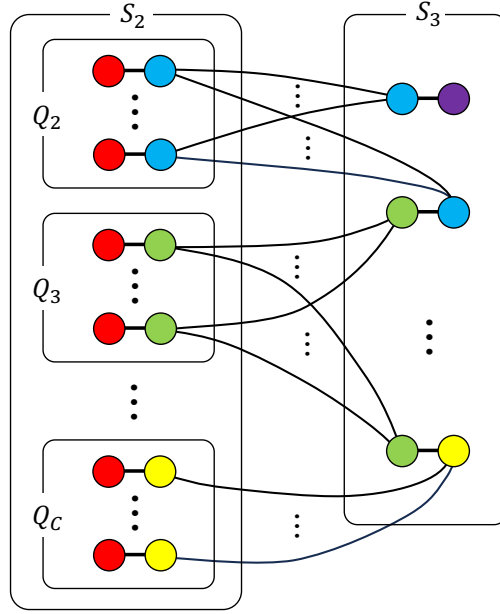
Now we construct an auxiliary graph $H = (S_2 \cup S_3, E')$, where

$$E' = \{\{u, v\} \mid \text{two sticks } u \text{ and } v \text{ share a color } i \text{ in } \{2, 3, \dots, c\}\}.$$

That is, two sticks in $S_2 \cup S_3$ are joined by an edge in E' if they have endpoints of the same color (under mapping \mathcal{M}) but not the color 1. Then we have the following observation:

► **Observation 6.** *A mapping \mathcal{M} is a feasible solution of the given instance of the matching-match puzzle if and only if H has a matching M such that (1) M contains $|P_4|$ edges joining a vertex in S_2 and another one in S_3 , (2) M contains $|P_3|$ edges joining two vertices in S_2 , and (3) M contains no any other edges.*

Proof. We can observe that each edge in M corresponding to the central vertex v_1 in each path (b, v_1, v_2) on G . That is, edges in M with the condition (1) correspond to the edges in H joining an edge in S_2 of G and an edge in S_3 on a leg in P_4 , and edges in M with the condition (2) correspond to the edges in H joining two edges in S_2 on a leg in P_3 . The remaining edges in S_2 are joined to edges in S_1 on legs in P_2 . ◀(of Observation 6)



■ **Figure 5** Graph $H' = (S_2 \cup S_3, E'')$.

By the observation, we can remove all edges in H that joins two vertices in S_3 , which are not necessary to construct the desired matching M in H . Let $H' = (S_2 \cup S_3, E'')$ be the graph obtained from H by removing them (Figure 5).

Now our goal is a construction of the matching M on H' that satisfies the conditions of Observation 6. We first note that $H'[S_2]$ induces a set of cliques Q_i for each color $i = 2, 3, \dots, c$. That is, S_2 can be partitioned into $c - 1$ subsets $S_2^i = \{\{u, u'\} \mid \mathcal{C}_1(u) = 1 \text{ and } \mathcal{C}_1(u') = i\}$ for each color $i = 2, 3, \dots, c$, and then $H'[S_2^i]$ is a clique Q_i . By the definition of H' , we can observe that $N_{H'}[q] = N_{H'}[q']$ for any $q, q' \in Q_i$. We also note that $|M| = |S_3|$ and $|S_2| - |S_3| = |P_2|$. That is, in the spider G , every stick in S_3 should be joined to a stick in S_2 in a proper way and other sticks in S_2 are connected to sticks in S_1 in any way.

To construct M , we first compute a maximum matching M' on the bipartite graph $H'' = (S_2, S_3, E''')$, where E''' is the set of edges joining vertices in S_2 and S_3 in H' . (In other words, we ignore the edges in the cliques Q_i .) If $|M'| < |S_3|$, the answer is clearly “No”. Otherwise, we choose $|S_3|$ edges in H' as the legs in P_4 . Let Q'_i be the set of cliques induced by the vertices not matched in M' . If $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor \geq |P_3|$, we have enough pairs to construct legs in P_3 . Then we choose any $|P_3|$ edges in $H'[\cup_i Q'_i]$ and add them to M' , which is the desired M . Once we can obtain M , we assign the remaining edges in S_2 as legs in P_2 . In this case, the answer is “Yes”.

The last remaining possible situation is that (1) the maximum matching M' has enough edges as $|M'| \geq |S_3|$ and (2) $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor < |P_3|$. In this case, some Q'_i may contain an odd number of vertices not matched in M' . When $|Q'_i|$ and $|Q'_j|$ with $i \neq j$ are odd, by changing the edges in M' , we may make both of $|Q'_i|$ and $|Q'_j|$ even and then we can increase $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor$ by one. If we can perform it repeatedly, we may achieve $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor = |P_3|$. This can be done if we have an *alternating path* P in H' with respect to M' between Q'_i and Q'_j . Here, an alternating path P is a path $(v_0, v_1, \dots, v_{2k})$ for some positive integer k such that the edges on P are in M alternately. Thus, by finding an alternating path P in H' with respect to M' and replacing M' by $M' \oplus P$ (swapping the members in M' according to P), we can increase $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor$ by one.

For the graph $H' = (S_2 \cup S_3, E'')$ and the maximum matching M' , we have the following lemma:

► **Lemma 7.** *We assume that $|Q'_i|$ and $|Q'_j|$ with $i \neq j$ are odd. Then there exists an alternating path between two vertices Q_i and Q_j if and only if a connected component of H' contains both of Q_i and Q_j .*

Proof (Outline). We first consider a vertex $s = \{u, w\}$ in S_3 with $\mathcal{C}_1(u) = \mathcal{C}_1(w)$. Then $N(s) = Q_{\mathcal{C}_1(u)}$. For such a vertex, M' contains an edge $\{s, q\}$ for some $q \in Q_{\mathcal{C}_1(u)}$ and we have nothing to do. Thus we focus on a vertex $s = \{u, w\}$ in S_3 with $\mathcal{C}_1(u) \neq \mathcal{C}_1(w)$. Then s has two clique neighbors Q_i and Q_j when $\mathcal{C}_1(u) = i$ and $\mathcal{C}_1(w) = j$. That is, $N(s) = Q_i \cup Q_j$.

Now we prove the claim of this lemma by induction on the length of the distance between Q_i and Q_j in a connected component of H' .

The base case is that the distance between Q_i and Q_j is 2. In this case, there exists a stick $s = \{u, w\}$ such that $\mathcal{C}_1(u) = i$ and $\mathcal{C}_1(w) = j$, and M' contains one of $\{s, q_i\}$ and $\{s, q_j\}$ for some vertices $q_i \in Q_i$ and $q_j \in Q_j$. Without loss of generality, we assume that M' contains $\{s, q_i\}$. Now, by assumption, we have an unmatched vertex q_j in Q_j . Then we can construct an alternating path is (q_i, s, q_j) . In this case, we can replace M' with $M' \cup \{\{s, q_j\}\} \setminus \{\{s, q_i\}\}$. After replacement, both of $|Q'_i|$ and $|Q'_j|$ are even, and a new pair $\{q_i, q'_i\}$ with $q_i, q'_i \in Q'_i$ appears and it can be used to add a leg into P_3 .

We turn to the inductive step: the distance between $q_i \in Q_i$ and $q_j \in Q_j$ is $2k$ for some $k > 1$ (and any pair of q_i and q_j). In a similar argument, we can assume that $\{s, q_i\}$ is in M' and $\{s, q_j\}$ is not in M' for some q_j in Q_j . Then, we can find a shortest path P of even length between q_i and q_j in H' when they are in the same connected component in H' . Since M' is a maximum matching, P is an alternating path and we can replace M' with $M' \oplus P$. After replacement, we can see that $|Q'_i|$ and $|Q'_j|$ are even, and any other clique Q_k appearing on P does not change the parity of $|Q_k|$. Therefore, a new pair $\{q_i, q'_i\}$ with $q_i, q'_i \in Q'_i$ appears again and it can be used to add a leg into P_3 .

When Q_i and Q_j are not in a connected component of H' , it is trivial to see that we cannot make any alternating path joining two vertices in Q_i and Q_j , which completes the proof. ◀(of Lemma 7)

By Lemma 7, repeating the process, we eventually maximize $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor$ for the maximum matching M' . When this maximum value $\sum_{i=2}^c \lfloor |Q'_i|/2 \rfloor$ is at least $|P_3|$, we can construct a desired matching M , which gives us a solution of the matching-match puzzle. On the other hand, when the maximum value is less than $|P_3|$, this case is not feasible.

The correctness of the algorithm follows with properties of the matroid. Using the standard technique for finding a maximum matching in a bipartite graph based on alternating paths (see, e.g., [5]), the algorithm can be performed in a polynomial time. ◀

Careful case analysis leads us to the following corollary:

► **Corollary 8.** *For a given instance $(G, S, c, \mathcal{C}_0, \mathcal{C}_1)$, the matching-match puzzle can be solved in polynomial time when G is a spider with legs of length at most 2, and $\mathcal{C}_0(v) = 0$ for all $v \in V$.*

5 Concluding Remarks

In this paper, we introduced and investigated the matching-match puzzle in general form. It is based on a commercial product puzzle which can be modeled as a variant of the graph coloring problems in a natural way.

17:10 Computational Complexity of Matching Match Puzzle

In the puzzle, the numbers of vertices and edges in the graph and the number of colors are variables and it is NP-complete even if the graph is quite restricted. In the proof of NP-completeness, the number of the colors is linear to the number of the vertices. It may be a reasonable assumption that the number of the colors is bounded above by a constant (in fact, the commercial product has 4 colors). For example, a natural brute force algorithm allows us to solve the matching-match puzzle on a k -partite complete graph $G = (V_1, V_2, \dots, V_k, E)$ in $O\left(\frac{c(c+n)^{k(c-1)}}{(c!)^k}\right)$ time. It is another open problem that the puzzle is fixed-parameter tractable with respect to the number of colors. That is, is there an algorithm that runs in $O(f(c) \cdot \text{poly}(n))$ for some graph classes of graphs of n vertices with c color, where $\text{poly}(n)$ is a polynomial function and $f(c)$ is a computable function?

A natural extension of Corollary 8 is the case that some vertices of a spider G of legs of length at most 2 are pre-colored additional to the body. We consider our polynomial-time algorithm can be extended to this case, however, case-analysis is quite complicated. (We note that it can be solved in $O\left(\frac{(c(c+n)^{c^2}}{(c!)^c}\right)$ time by a brute force algorithm.) It is unlikely that there exists a simple polynomial-time algorithm for solving this case. Based on this fact, we conjecture that the matching-match puzzle is NP-complete if G is a spider with legs of length at most 3, which is another open problem.

References

- 1 N. Alon, R. Yuster, and U. Zwick. Color-Coding. *Journal of the ACM*, 42(4):844–856, 1995.
- 2 Michael R. Garey and David S. Johnson. *Computers and Intractability — A Guide to the Theory of NP-Completeness*. Freeman, 1979.
- 3 R. A. Hearn and E. D. Demaine. *Games, Puzzles, and Computation*. A K Peters Ltd., 2009.
- 4 Tommy R. Jensen. *Graph Coloring Problems*. John Wiley & Sons, 1994.
- 5 C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization*. Dover, 1982.
- 6 Ryuhei UEHARA. Computational Complexity of Puzzles and Related Topics. *Interdisciplinary Information Sciences*, ID 2022.R.06:1–23, 2023. doi:10.4036/iis.2022.R.06.

Advanced Spikes ‘n’ Stuff: An NP-Hard Puzzle Game in Which All Tutorials Are Efficiently Solvable

Christian Ikenmeyer ✉ 🏠 

University of Warwick, UK

Dylan Khangure ✉

University of Warwick, UK

Abstract

We adjust Alan Hazelden’s 2017 polynomial time solvable puzzle game Spikes ‘n’ Stuff: We obtain the NP-complete puzzle game Advanced Spikes ‘n’ Stuff with 3 trap types so that each strict subset of the traps results in a polynomial time solvable puzzle game. We think of this as a “hard game in which all tutorial levels are easy”. The polynomial time algorithms for solving the tutorial games turn out to be quite different to each other.

While numerous papers analyze the complexity of games and which game objects make a game NP-hard, our paper is the first to study a game where the NP-hardness can only be achieved by a combination of all game objects, assuming P differs from NP.

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

Keywords and phrases computational complexity, P vs NP, motion planning, games

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.18

1 Motivation

By a *tutorial* for a game we mean the game with some of its parts removed. The exact notion of what constitutes a “removable part” of a game is arbitrary, but for many games there are natural choices. For games such as Alan Hazelden’s 2017 puzzle game *Spikes ‘n’ Stuff* [1], where the player navigates through a maze with different types of traps, the natural removable parts are the different types of traps. In a very similar manner, natural tutorials have been investigated for example in [9], where the removable parts are the interactive objects (which are not necessarily traps) in a level of the computer game Portal. One main observation in [9] is that many of the interactive objects in Portal are NP-hard on their own, i.e., in these cases the tutorial in which only one type of interactive object is present is already NP-hard. We list many more such examples from the literature in Section 3.

The original Spikes ‘n’ Stuff game is a turn-based game that is solvable in polynomial time (in the size of the level), because each level only has a state space of polynomially bounded size. We adjust this game slightly so that we get the NP-hard game Advanced Spikes ‘n’ Stuff, which is a game that is polynomial-time solvable if levels contain only two of the three traps, see Figure 1. The main technical difficulty that we overcome in this paper is that at least one trap type has to result in a superpolynomially large state space, but this trap type is not allowed to make the game NP-hard on its own or combined with any one of the other two trap types. This is the first NP-hard game that we know for which all tutorials are solvable in polynomial time.

Note that artificial NP-hard “games” in which all tutorials are easy can readily be created for example as follows (with two removable “game” parts instead of three): The game is to solve a 3SAT instance, and the two removable parts are *negation of variables* and *disjunction*.



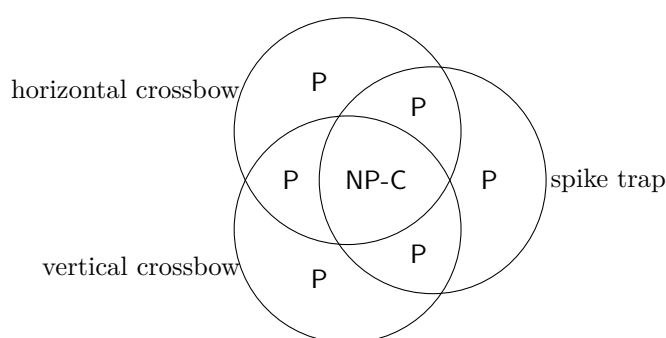
© Christian Ikenmeyer and Dylan Khangure;
licensed under Creative Commons License CC-BY 4.0
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 18; pp. 18:1–18:13

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** If all three trap types in Advanced Spikes ‘n’ Stuff are present, the game is NP-complete. Otherwise it is solvable in polynomial time.

While 3SAT is NP-complete, if we remove the negation of variables, the problem becomes trivial, whereas when we remove disjunctions, then all clauses have length 1, which also trivializes the problem.

Our point is that Advanced Spikes ‘n’ Stuff is only a slight variation of the fun game Spikes ‘n’ Stuff, and therefore our techniques (or refinements thereof) may have implications on actual game design approaches in the future. As a first insight, we can say that the fact that all tutorials are easier to solve than the complete game forces the player to rethink their strategies (that they have developed while playing through tutorial levels) when playing the complete game. Forcing the player to re-think established assumptions is usually a desirable property in puzzle games.

2 Advanced Spikes ‘n’ Stuff

Advanced Spikes ‘n’ Stuff is a turn-based single player puzzle game where the goal is to move an agent from the start position to the finish position through a maze without the agent getting killed by traps. The game is played on a square grid of tiles. Every tile is either walkable without a spike trap, or walkable with a spike trap, or a wall, or a horizontal crossbow trap (facing left or right), or a vertical crossbow trap (facing up or down). Each trap has an internal state, where spike traps have three states 0, 1, 2, vertical crossbows have two states 0, 1, and horizontal crossbows have six states 0, 1, . . . , 5. The state 0 is called the idle state. In addition to this state, every *vertical* crossbow is either *functional* or *deactivated*. Horizontal crossbows count as always *functional*. We define the *range* of a crossbow to be the set of tiles in a direct line in front of the crossbow up to the next unwalkable tile. Figure 2 depicts an example level of Advanced Spikes ‘n’ Stuff.

On each turn, first the agent must move either horizontally or vertically from its tile p to an adjacent walkable tile q . The direction is chosen by the player. If q is a state 1 spike trap or q is in the range of a functional state 1 crossbow, then the agent dies and the player loses the game. Traps in other states do not harm the agent. At this point, if the player hasn’t lost yet, all traps that are not in their idle state or deactivated advance their state by 1, coming back to their idle state after their maximal state is reached. If p is a spike trap, then now that spike trap’s state is set to 1. If q is in the range of an idle functional crossbow trap, the crossbow trap’s state is now set to 1. If q is left or right of a vertical crossbow trap, that crossbow trap is set to *deactivated* and its state is set to 0. At this point, the turn ends and the player must now make the next choice in which direction to move.



The light gray coloured tiles are the walls, and the darker coloured tiles are the walkable tiles. The start position is the tile at the top-left of the level, and the finish position is the tile at the bottom-right. The agent takes the form of a person and can be seen near the goal position. Each spike trap state is shown in this level. The spike trap immediately above the agent is in state 1 and the one immediately above that is in state 2. All other spike traps in the level are in state 0. Similarly all states of the horizontal crossbow trap are present. The crossbow adjacent to the player is in state 1. Going up from this position, the crossbows are in states 2,3,4 and 5 respectively, with the one at the very top being idle (in state 0). Two vertical crossbows are also in this level; the red one is *functional* and the blue one is *deactivated*. The notion of the *range* of a crossbow is also highlighted in this level using the horizontal crossbow towards the bottom. The walkable tiles with small gold squares are precisely those tiles which are in the range of this crossbow. The gold squares are hidden from the player during the game, and are merely used here for illustrative purposes.

■ **Figure 2** Sample level of Advanced Spikes ‘n’ Stuff.

A game state of this turn-based game consists of the start and finish positions and the tile arrangement (which do not change throughout the game), the agent’s position, the states of all traps, and the functional/deactivated flag of all vertical crossbow traps. The game starts with all traps in their idle state and all vertical crossbows functional. For the sake of simplicity, we assume that

the start tile is not in range of any crossbow trap and the tiles adjacent to the start tile are not spike traps, and (1)

the tiles left and right to of vertical crossbow traps and their adjacent tiles are not spike traps, and they are not in range of a horizontal crossbow trap. (2)

Formally, we consider the problem $\text{ASnS}(S)$ of deciding given an Advanced Spikes ‘n’ Stuff level satisfying (1) and (2) whether or not the player can move the agent from the start tile to the finish tile without the agent dying by traps. The parameter S is a subset of the set of symbols $\{h, s, v\}$, where $s \in S$ indicates that the levels can have spike traps, $h \in S$ indicates that the levels can have horizontal crossbow traps, and $v \in S$ indicates that the level can have vertical crossbow traps.

► **Theorem 2.1** (Main Theorem). $\text{ASnS}(\{h, s, v\})$ is NP-complete, but $\text{ASnS}(S) \in \text{P}$ for all strict subsets $S \subsetneq \{h, s, v\}$.

Theorem 2.1 is illustrated in Figure 1. Since a game cannot get harder if possible game pieces are removed, the results $\text{ASnS}(\{h\}) \in \text{P}$, $\text{ASnS}(\{s\}) \in \text{P}$, $\text{ASnS}(\{v\}) \in \text{P}$ are direct corollaries from $\text{ASnS}(\{h, s\}) \in \text{P}$, $\text{ASnS}(\{s, v\}) \in \text{P}$.

We prove $\text{ASnS}(\{h, s\}) \in \text{P}$ in Corollary 4.3, $\text{ASnS}(\{v, s\}) \in \text{P}$ in §5, and $\text{ASnS}(\{h, v\}) \in \text{P}$ in §6, interestingly with different proof strategies: While $\text{ASnS}(\{h, s\}) \in \text{P}$ follows from the polynomially bounded state space, the argument for $\text{ASnS}(\{v, s\}) \in \text{P}$ partitions the exponentially large state space into polynomially large pieces and greedily traverses polynomially many pieces, and $\text{ASnS}(\{h, v\}) \in \text{P}$ uses the same strategy, but with a more involved argument, because in the presence of horizontal crossbow traps, the agent cannot

safely traverse a safe walk in the reverse direction (see the discussion in §6). We prove the NP-completeness of $\text{ASnS}(\{h, s, v\})$ in Theorem 7.3. We use a standard framework for constructing a polynomial-time reduction from 3SAT to $\text{ASnS}(\{h, s, v\})$, closely based on [3]. Early versions of this framework appear in the hardness proofs for games such as SOKOBAN [10] and PushPush [8], but it has since been refined, as can be seen in the papers [2] and [7].

The original game Spikes ‘n’ Stuff

In the original Spikes ‘n’ Stuff the horizontal crossbow traps have only 4 states and the vertical crossbow traps work in the same way as the horizontal crossbows traps (and there are some other minor changes about arrows hitting each other mid-air and the player pulling a treasure, which add some twists to the game, but these are not essential, so we do not discuss them). Therefore, no crossbow trap can be deactivated, which implies that the state space has polynomially bounded size, see Lemma 4.2. Hence, the original Spikes ‘n’ Stuff game is solvable in time that is polynomially bounded in the size of the game board, for example by breadth-first-search in the state space.

3 Related Work

[11] proves various theorems about the complexity of two-dimensional platform games. For example, a level containing items for the player to collect is, on its own, polynomial-time solvable, but adding in a time-limit immediately makes it NP-hard. Moreover, platform games with drops longer than the maximum jump height are already NP-hard.

[13] proves that games where doors can be opened with pressure plates, but not closed, are in P, but adding the ability to close doors makes that game NP-hard. Similarly, it is shown that games with buttons that act on only a single door are in P, but games where a single button may act on two or more doors are NP-hard. Moreover, if a game contains a feature that forces the player to visit various locations, and there are single-use paths or “toll roads” (a certain number of a specific item must be collected to pass), that game is NP-hard.

In the game Lemmings, agents can be assigned skills by the player which alter their behaviour. For example, the *Builder* skill allows the Lemming to construct a bridge, and the *Digger* skill allows the Lemming to dig vertically downwards (see [6], [13] and [14]). One sensible way of defining a tutorial of Lemmings is to consider the game we get by restricting the skills we are allowed to give to the agents (and this is indeed how the actual game is set up). Some of these tutorials where only a single skill can be given to the agents are already NP-hard (even in simplified models of the game): in [6] hardness is proved using only *Digger* skills, and [13] presents a construction that only requires *Basher* or *Miner* skills to achieve hardness.

In [3], we see that even if we consider classic Nintendo games with a subset of their original features we may still get a game that is NP-hard. The original Super Mario Bros. is NP-hard with only the following game features: Super Mushrooms to turn Mario into Super Mario (who can break blocks but cannot fit into narrow horizontal corridors), Goombas to turn Super Mario back to normal Mario (who can fit into narrow horizontal corridors but cannot break blocks) vertical drops that are higher than the maximum jump height, and Stars which provide temporary invulnerability from Firebars. Pokémon is already NP-hard if it contains nothing other than pushable blocks, and also if pushable blocks are not present and the only feature used is battles with enemy Trainers (in this second case, the game is actually NP-complete). Similarly, the original Legend of Zelda is NP-hard even if block

pushing is the only feature that is kept. In later Zelda games, there are ice blocks. These are blocks that when pushed, slide until they hit an obstruction. It is shown that with just this feature, the game is in fact PSPACE-complete.

Another platform game is Celeste, where the player navigates through various levels containing obstacles. The character they control can move in eight directions, and has the ability to jump, dash and temporarily grab onto walls. The game contains many different mechanics and types of obstacles, but [2] proves that even if only the following are present, the game is already NP-hard: platforms that break when stood on, button-operated doors, and special types of blocks that teleport the player in a straight line. [5] solves the PSPACE-hardness for several of its tutorials.

[12] explores a tiling problem that is based on the game KPlumber. The input to the problem is a grid of tiles, with each tile being one of six possible types (O, C, D, S, T, X), and the goal is to rotate the tiles in order to reach a desired goal state. If the tiles C, D, S are used, the tutorial is already NP-complete. The same holds for C, S, T . For some subsets of $\{O, C, D, S, T, X\}$ the complexity has not been classified.

Another game that has been studied which contains many natural tutorials is Tetris. Simpler versions of the game can be obtained by, for example, restricting the size of the gameboard, or the pieces that are available. Some of these simpler versions are proven to be solvable in polynomial time in [4], some are NP-hard, where the tutorials are distinguished by board size, piece size, and empty/non-empty initial board state. We have not found any analysis in the literature about the complexity when the set of tetris pieces is restricted to a subset.

4 The game states in Advanced Spikes ‘n’ Stuff

In this section we prove $\text{ASnS}(\{h, s\}) \in \text{P}$.

► **Definition 4.1.** *A game state is called tame if at most 2 spike traps are not in the idle state and at most 12 crossbow traps are not in their idle state.*

Note that this definition poses no constraints on the *functional/deactivated* flags of the vertical crossbow traps, which means that the set of tame game states can be exponential in size.

► **Lemma 4.2.** *In a game of Advanced Spikes ‘n’ Stuff, only tame game states can be reached.*

Proof. During each move the agent can only set one spike trap to state 1. Hence, during a move only one spike trap can go from state 1 to state 2.

Similarly, during each move the agent can only trigger 2 crossbow traps, which have 6 or 2 states each, and $2 \max\{2, 6\} = 12$. ◀

► **Corollary 4.3.** $\text{ASnS}(\{h, s\}) \in \text{P}$.

Proof. If no vertical crossbow traps are present, then the size of the set of tame game states is polynomially bounded. By Lemma 4.2 it is sufficient to consider only this space. Hence, a breadth-first-search in the space of tame game states solves $\text{ASnS}(\{h, s\})$ in polynomial time. ◀

5 $\text{ASnS}(\{s, v\}) \in \text{P}$

We prove that $\text{ASnS}(\{s, v\}) \in \text{P}$. First, note that

- a spike trap kills an agent if and only if the agent moves from the spike trap to an adjacent tile and immediately back on the next turn.
 - a functional vertical crossbow trap kills the agent if and only if the agent makes a vertical move within its range.
- (3)

Let V denote the set of vertical crossbow traps. For any subset of $S \subseteq V$ of functional vertical crossbow traps, let $R_S \subseteq S$ denote the set of functional vertical crossbow traps the agent can deactivate when starting at the start tile and making game moves without dying and without deactivating any other functional vertical crossbow trap. For each fixed S , the set R_S can be determined in polynomial time using the observation in the above two bullet points, for example by a breadth-first-search in the subset T_S of the space of tame game states that have exactly the vertical crossbows S functional. Note that the size of T_S is polynomially bounded.

The polynomial-time algorithm to solve $\text{ASnS}(\{s, v\})$ initializes $S_0 := V$. Then it loops the following for $i = 0, \dots$:

1. If the game can be solved by just traversing game states in T_{S_i} , return **true**.
2. Determine R_{S_i} .
3. If $R_{S_i} = \emptyset$, return **false**. Else, pick an arbitrary $r_i \in R_{S_i}$ and define $S_{i+1} = S_i \setminus \{r_i\}$ and continue the loop with the next i .

If **true** is returned, then the solution to the level is by first moving the agent to deactivate r_0 , then moving the agent back to the start, then deactivating r_1 , then moving the agent back to the start, and so on, until finally traversing the game states in T_{S_i} to the finish tile. The crucial observation here is that the properties (3) and (2) imply that going back to the start by tracing the steps backwards will not kill the agent.

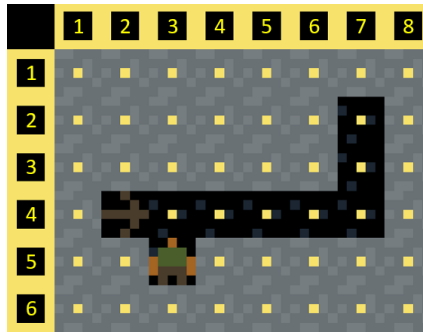
If **false** is returned, then S_i is the unique maximal set of functional vertical crossbow traps that can be reached from the start game state. But in T_{S_i} there is no traversal to a finish game state. This finishes the proof of $\text{ASnS}(\{s, v\}) \in \text{P}$.

6 $\text{ASnS}(\{h, v\}) \in \text{P}$

The proof outline is the same as in §5. The only difference is that once the agent deactivates a vertical crossbow trap, it might not be possible to trace the same walk back to the start without being killed, see Figure 3. First, note that the agent gets killed if it makes a vertical step in the range of a functioning vertical crossbow trap. Now, generalize this observation from 2 states (vertical crossbow trap) to 6 states (horizontal crossbow trap): the agent can only make at most 4 successive horizontal moves within the range of a horizontal crossbow trap without being killed. The following Theorem 6.1 states the “converse”.

► **Theorem 6.1.** *If a level has no spike traps, then for a walk w from tile p to tile q that does not make more than 4 successive horizontal moves within the range of a horizontal crossbow trap and that does not make a vertical move within the range of a functional vertical crossbow trap, there exists a walk w' from p to q such that the agent does not get killed when traversing w' .*

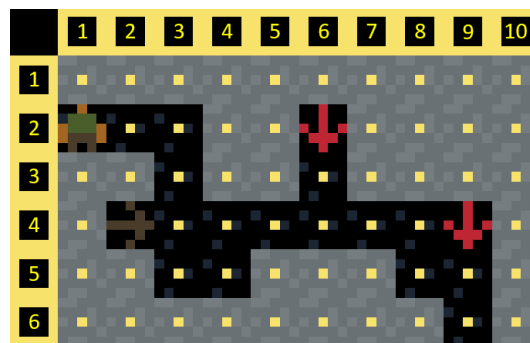
Theorem 6.1 implies $\text{ASnS}(\{h, v\}) \in \text{P}$ as follows. Since the property of w of avoiding 4 successive horizontal moves in ranges of horizontal crossbow traps and avoiding vertical moves in ranges of vertical crossbow traps is symmetric, traversing w back to the start can be done safely by replacing it by w' (where we use the safety assumptions (1) and (2)), which finishes the proof of $\text{ASnS}(\{h, v\}) \in \text{P}$.



■ **Figure 3** The walk $(3,4), (3,5), (3,4), (4,4), (5,4), (6,4), (7,4), (7,3), (7,2)$ can be safely traversed from the agent’s current position, but attempting to traverse this walk in reverse will cause the agent to be killed.

Proof of Theorem 6.1. First, observe that the agent cannot be killed by any horizontal crossbow trap when making a vertical step. The reason is that if a vertical step moves the agent into the range of a horizontal crossbow trap and the crossbow trap was in state 1 (which would kill the agent), then the agent would have set its state to 1 in the last turn. But in the last turn the agent was not in the range (otherwise, a vertical step would move the agent out of the range). Analogously, the agent cannot be killed by a vertical crossbow trap when making a horizontal step. Hence, in w the player is not killed by a vertical crossbow trap.

Now, take the walk w and adjust it as follows to obtain w' : Every move in which the agent enters and not immediately leaves the range of a crossbow trap (this must necessarily be a horizontal crossbow trap by assumption on w) that is in state $i \in \{0, 2, 3, 4, 5\}$, the agent repeatedly takes one step back and then one step forward: once if $i = 0$, 0 times if $i = 2$, 3 times if $i \in \{3, 4\}$, and 2 times if $i = 5$. The agent now proceeds traversing w and enters the range with the crossbow trap in state 2, which immediately moves to state 3. The agent can now safely make 4 horizontal steps in the range, while the state of the crossbow trap goes to 4, 5, 0, 1. As discussed before, the new vertical steps cannot get the agent killed by any horizontal crossbow traps, and clearly not by a vertical crossbow trap. An example of this construction of w' is given in Figure 4. ◀



■ **Figure 4** The walk $w = ((1, 2), (2, 2), (3, 2), (3, 3), (3, 4), (3, 5), (4, 5), (4, 4), (5, 4), (6, 4), (7, 4), (8, 4), (8, 5), (9, 5), (9, 6))$ will get the agent killed at $(7, 4)$, but the agent survives when using $w' = ((1, 2), (2, 2), (3, 2), (3, 3), (3, 4), (3, 5), (4, 5), (4, 4), (4, 5), (4, 4), (4, 5), (4, 4), (4, 5), (4, 4), (5, 4), (6, 4), (7, 4), (8, 4), (8, 5), (9, 5), (9, 6))$.

7 NP-completeness

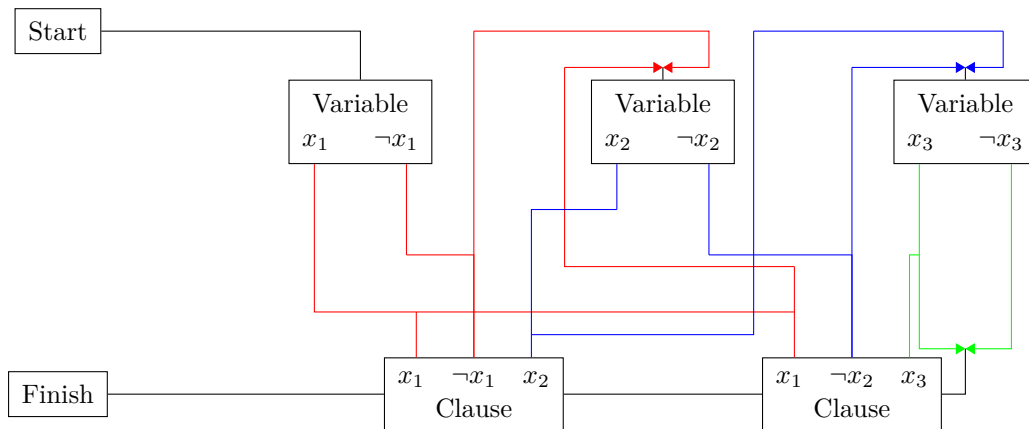
► **Theorem 7.1.** $ASnS(\{h, s, v\}) \in NP$.

Proof. We prove that every level can be solved in a number of steps that is polynomially bounded in the level size. This proves the theorem, because the step sequence gives a polynomially sized witness for the solvability.

Consider an arbitrary level that the player can complete. Since the level is completable, there exists some walk w that the agent is able to traverse from the start tile to the finish tile without being killed. Let (d_1, d_2, \dots, d_k) be the sequence of vertical crossbows that the agent deactivates during w , in chronological order of deactivation. Note that each crossbow can be deactivated at most once, so k cannot exceed the number of tiles of the level. For $i \in \{0, 1, \dots, k\}$ let D_i denote the set of tame states (see Definition 4.1) in which exactly the vertical crossbows $\{d_1, \dots, d_k\}$ are deactivated. By assumption and by Lemma 4.2 there exists a path p from the start game state through the space $D_0 \cup D_1 \cup \dots \cup D_k$ to a finish game state in which each edge is a game move. Since k is polynomially bounded, and the cardinality of each D_i is polynomially bounded, the length of a shortest path p is polynomially bounded. Such a path serves as the desired NP witness. ◀

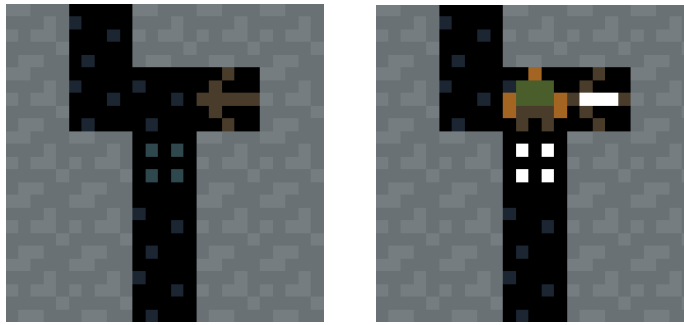
7.1 NP-hardness

The construction uses one-way gadgets (where the agent can only pass through in one direction), variable gadgets, clause gadgets, and crossover gadgets: one variable gadget for each variable in the 3SAT instance, and one clause gadget for each clause. Each variable gadget is connected to its clause gadgets by paths, and also to the next variable gadget, see Figure 5.



■ **Figure 5** NP-hardness framework: The agent starts at *Start* and tries to get to *Finish*, traversing the edges. Arrows stand for one-way gadgets. Each 4-way intersection must be crossed in a straight manner. This is ensured by crossover gadgets. Our crossover gadget is *not* symmetrical and care must be taken with its placement. Colored edges of the same color belong to the same *section*.

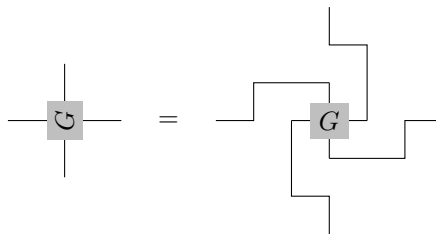
The one-way gadget is constructed as depicted on the left side in Figure 6. It is easy to see that a player can traverse this gadget from north to south by entering the range of the crossbow to set its state to 1, backtracking one step to avoid being hit by the arrow, then proceeding in the obvious way to the exit, which is at the south. However, if the player tries to traverse this gadget from the south to the north, they will find themselves in the state



■ **Figure 6** Left: One-way gadget, only traversable from north to south, not from south to north. Right: Traversing one-way gadget from south to north results in agent being killed.

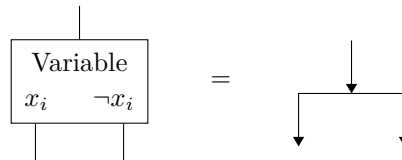
depicted in Figure 6 on the right side. This is clearly a losing position: moving west causes the agent to be killed by the horizontal crossbow trap, and moving south causes them to be killed by the spike trap.

The solvability of levels is *not* invariant under rotations, but for each gadget we also have the gadgets in all four rotations, which is illustrated in Figure 7.



■ **Figure 7** Constructing a 90° rotated version of a gadget G . The depicted gadget G has 1 input on each side, but this construction obviously works for any number of inputs on any side.

The variable gadget is constructed from one-way gadgets as depicted in Figure 8.



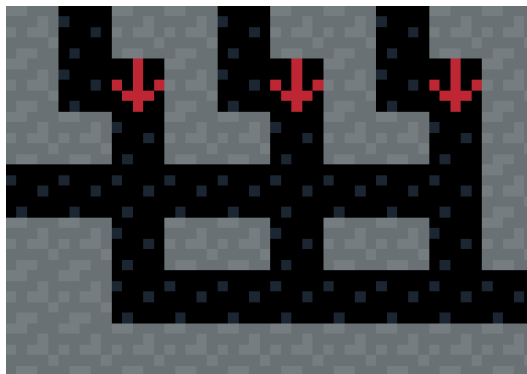
■ **Figure 8** The variable gadget consists of three one-way gadgets.

The clause gadget is also easy to build, see Figure 9. There are three entrances at the north side of the gadget (that correspond to the three literals in the clause that this gadget is representing). When the agent uses one of these entrances, they can unlock the gadget by setting a vertical crossbow trap to *deactivated*. When the agent enters the gadget from the east, they can traverse it to the west if and only if it is unlocked, i.e., it has been unlocked in at least one of the northern entrances.

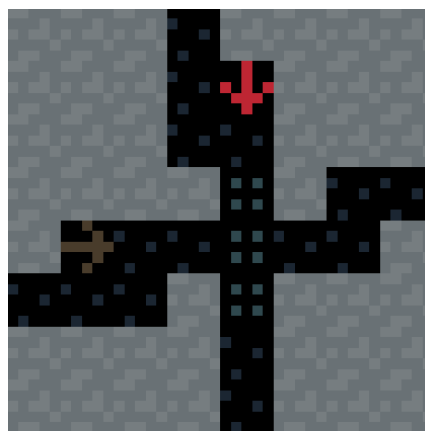
7.2 The Crossover gadget and its placement

The final gadget required for the construction is the crossover gadget, see Figure 10. There are two paths, which we refer to as the *north-south* path and the *east-west* path, which cross

18:10 An NP-Hard Puzzle Game in Which All Tutorials Are Efficiently Solvable



■ **Figure 9** The clause gadget.



■ **Figure 10** The crossover gadget in its closed state, in the *can be opened at north* rotation.

at the spike trap in the center. This gadget comes equipped with a *state*, which is *closed* if the vertical crossbow is functional, and *open* if it is deactivated. The key properties of the gadget are explained in the following lemma.

► **Lemma 7.2.** *If the crossover gadget is closed, then the agent can traverse east to west and vice versa, and the agent can open the gadget from the north. If the crossover gadget is open, then the agent can traverse from east or west to any direction, while from the north and south the agent can only go to the north and south.*

Proof. If the gadget is closed and the agent comes from the south, the vertical crossbow trap will kill the agent. Obviously, the agent can open the gadget from the north by deactivating the vertical crossbow trap. If the gadget is open and the agent comes from the north or south, then the horizontal crossbow trap together with the spike traps prevents the agent to go east or west, but north and south are possible.

Independent of the gadget's state, the gadget can be traversed east to west or vice versa by the agent entering the horizontal crossbow's range, taking one step back, and then moving across. This also works for moving to the north or south, but only if the gadget is open, i.e., the vertical crossbow is *deactivated*. If the gadget is closed, the vertical crossbow is *functional*, which prohibits any turning onto the *north-south* path. ◀

Recall that the crossover gadget comes in all 4 rotations, see Figure 7. We now show how this gadget is used in our construction. Recall Figure 5. Where two paths cross, we insert a crossover gadget in one of the four rotations, but the choice of rotation is not arbitrary.

As seen in Figure 5, we have an ordering x_1, x_2, \dots, x_n on the set of variables in the 3SAT instance. The edge colors in Figure 5 indicate so-called *sections*: The i -th section consists of the two walks $w_{i,0}, w_{i,1}$ originating from the variable gadget for x_i , one corresponding to setting x_i to 0 (**false**) and the other to setting it to 1 (**true**): These walks touch all of the clause gadgets containing the literal they correspond to, and then go to the next variable gadget. For each intersection of two walks, it will be important from which direction the walks enter the intersection for the first time when traversed. To place the crossover gadgets, first consider intersections of walks in different sections: w_{i,b_i}, w_{j,b_j} for $i \neq j$, w.l.o.g. $i < j$. We place the crossover gadget so that the vertical crossbow trap is at the place where w_{j,b_j} enters the intersection for the first time when traversed. We claim that the agent can only first enter the variable gadget 1, then section 1, then the variable gadget 2, then section 2, and so on, until the agent has to traverse all clause gadgets and reaches the finish tile. This can be seen as follows. Clearly, the agent has to first enter variable gadget 1 and then section 1. The placement of the crossover gadgets guarantees that the agent cannot open any crossover gadget of section 1 with any section $j \neq 1$. Therefore, the agent must traverse to variable gadget 2 and enter section 2. By the same argument, in section 2 the agent cannot open any crossover gadget of section 2 with any section $j > 2$. However, the agent can open the crossover gadgets of section 2 with section 1. But the placement guarantees that the agent cannot enter section 1 from these crossover gadgets. Therefore, the agent has to enter variable gadget 3 and section 3, and so on.

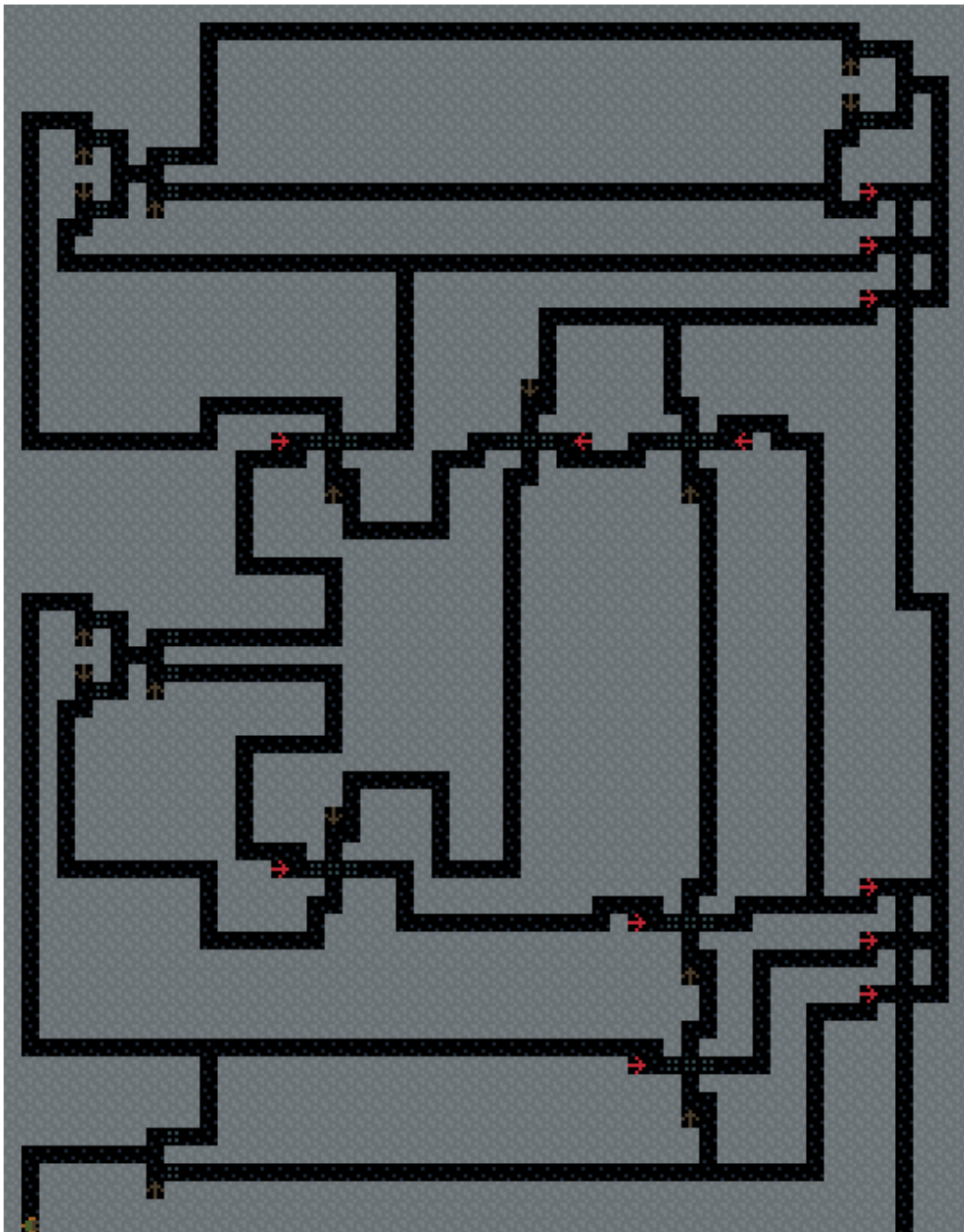
When placing crossover gadgets at the intersection of $w_{i,0}$ and $w_{i,1}$, we take the orientation so that the vertical crossbow trap is at the place where $w_{i,0}$ enters the intersection for the first time when traversed. This makes sure that if the player chooses $x_i = 1$, then the agent only ever encounters east-west crossings of crossover gadgets, which means that the agent cannot open these gadgets and hence cannot traverse east or west, and if the player chooses $x_i = 0$, then the agent only ever encounters north-south crossings of crossover gadgets, and hence can open the gadgets, but not traverse east or west.

► **Theorem 7.3.** $\text{ASnS}\{h, s, v\}$ is NP-complete.

Proof. The proof is now obvious. We have seen that in order to reach the finish tile, the agent must traverse section 1, then section 2, and so on, and choose exactly one Boolean value for each variable. And after choosing the variable values, the agent can traverse the clause gadgets if and only if the agent touched each clause gadget. ◀

7.3 Full example

Figure 11 depicts the Advanced Spikes ‘n’ Stuff level built from the 3SAT instance $\phi = (x_1 \vee \neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3)$. Setting each of the variables x_1, x_2 and x_3 to **true** makes ϕ evaluate to **true**. It can be easily verified that if the player makes these choices at the variable gadgets, they are able to unlock both clause gadgets during their traversal, and are hence able to complete the level.



■ **Figure 11** Advanced Spikes ‘n’ Stuff level built from $\phi = (x_1 \vee \neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2 \vee x_3)$. The situation is rotated by 90° from Figure 5. The variable gadgets are on the west, x_1 to x_3 from south to north. A southern traversal sets the variable to **true**, a northern traversal sets it to **false**. The clause gadget for $(x_1 \vee \neg x_1 \vee x_2)$ is at the south east, and the clause gadget for $(x_1 \vee \neg x_2 \vee x_3)$ is at the north east.

References

- 1 Spikes ‘n’ Stuff. <https://draknek.it.ch.io/spikes-n-stuff>, <https://alan.draknek.org/games/puzzlescript/spikes-n-stuff.php>. Accessed: 25/02/2024.
- 2 Zeeshan Ahmed, Alapan Chaudhuri, Kunwar Shaanjeet Singh Grover, Ashwin Rao, Kushagra Garg, and Pulak Malhotra. Classifying Celeste as NP Complete. In *International Conference on Foundations of Computer Science & Technology, Chennai, India*, November 2022.
- 3 Greg Aloupis, Erik D Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo Games are (Computationally) Hard. *Theoretical Computer Science*, 586:135–160, 2015.
- 4 Sualeh Asif, Michael Coulombe, Erik D Demaine, Martin L Demaine, Adam Hesterberg, Jayson Lynch, and Mihir Singhal. Tetris is NP-hard even with $O(1)$ rows or columns. *Journal of Information Processing*, 28:942–958, 2020.
- 5 Lily Chung and Erik D. Demaine. Celeste is PSPACE-hard. *Thai Journal of Mathematics*, 21(4):671–686, December 2023.
- 6 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.
- 7 Diogo M Costa, Alexandre P Francisco, and Luís Russo. Hardness of Modern Games. arXiv:2005.10506, 2020.
- 8 Erik D Demaine, Martin L Demaine, and Joseph O’Rourke. PushPush and Push-1 are NP-hard in 2D. In *Proceedings of the 12th Annual Canadian Conference on Computational Geometry (CCCG 2000)*, pages 211–219, August 2000.
- 9 Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The Computational Complexity of Portal and other 3D Video Games. In *9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100, pages 19:1–19:22, 2018.
- 10 Dorit Dor and Uri Zwick. SOKOBAN and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.
- 11 Michal Forišek. Computational complexity of two-dimensional platform games. In *Fun with Algorithms: 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings 5*, pages 214–227. Springer, 2010.
- 12 Daniel Král, Vladan Majerech, Jiří Sgall, Tomáš Tichý, and Gerhard Woeginger. It is tough to be a plumber. *Theoretical computer science*, 313:473–484, 2004.
- 13 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54:595–621, 2014.
- 14 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

Anarchy in the APSP: Algorithm and Hardness for Incorrect Implementation of Floyd-Warshall

Jaehyun Koo   

MIT EECS and CSAIL, Cambridge, MA, USA

Abstract

The celebrated Floyd-Warshall algorithm efficiently computes the all-pairs shortest path, and its simplicity made it a staple in computer science classes. Frequently, students discover a variant of this Floyd-Warshall algorithm by mixing up the loop order, ending up with the incorrect APSP matrix. This paper considers a computational problem of computing this *incorrect* APSP matrix. We will propose efficient algorithms for this problem and prove that this incorrect variant is APSP-complete.

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

Keywords and phrases fine-grained complexity, recreational algorithms

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.19

1 Introduction

The Floyd-Warshall algorithm computes the all-pairs shortest path (APSP) in a directed weighted graph [4, 9, 2]. Known in the computer science community for over 60 years, Floyd-Warshall is still one of the most efficient algorithms for the APSP problem, where it has a runtime of $O(n^3)$ for a graph with n vertices. No algorithms have improved this runtime by a polynomial factor in general graphs, which motivates the APSP Conjecture by [10].

Another remarkable characteristic of this algorithm is its simplicity - the standard implementation of this algorithm is a short triply nested loop, as shown below:

■ **Algorithm 1** The Floyd-Warshall Algorithm (KIJ Algorithm).

```
A ← Adjacency matrix of the graph
Ensure:  $A[i, i] = 0$  for all  $1 \leq i \leq n$ 
Ensure:  $G$  has no negative cycles
for  $k \leftarrow 1, 2, \dots, n$  do
  for  $i \leftarrow 1, 2, \dots, n$  do
    for  $j \leftarrow 1, 2, \dots, n$  do
       $A[i, j] \leftarrow \min(A[i, j], A[i, k] + A[k, j])$ 
```

Unfortunately, due to the algorithm being too simple, some students write a variant of the Floyd-Warshall algorithm, either by mistake or as a deliberate attempt to rectify the loop order into a *natural* lexicographic order.¹

¹ We are unaware of any publication over this variant except [7], but some Internet forums and even the lecture material discuss it. Examples are: <https://www.quora.com/Why-is-the-order-of-the-loops-in-Floyd-Warshall-algorithm-important-to-its-correctness>, <https://cs.stackexchange.com/questions/9636/why-doesnt-the-floyd-warshall-algorithm-work-if-i-put-k-in-the-innermost-loop>, <https://codeforces.com/blog/entry/20882>, <https://cs.nyu.edu/~siegel/JJ10.pdf>.



© Jaehyun Koo;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 19; pp. 19:1–19:11

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

19:2 Algorithm for Incorrect Implementation of Floyd-Warshall

■ **Algorithm 2** The Variant of Floyd-Warshall Algorithm (IJK Algorithm).

$A \leftarrow$ Adjacency matrix of the graph G
Ensure: $A[i, i] = 0$ for all $1 \leq i \leq n$
Ensure: G has no negative cycles
for $i \leftarrow 1, 2, \dots, n$ **do**
 for $j \leftarrow 1, 2, \dots, n$ **do**
 for $k \leftarrow 1, 2, \dots, n$ **do**
 $A[i, j] \leftarrow \min(A[i, j], A[i, k] + A[k, j])$

Algorithm 2 do not compute the correct APSP matrix. For example, let A be the adjacency matrix, and the M_1 and M_2 be the resulting matrix computed by Algorithm 1 and Algorithm 2 on A . The following choice of A makes $M_1[2, 1]$ and $M_2[2, 1]$ different:

$$A = \begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & \infty & 1 \\ 1 & \infty & 0 & \infty \\ \infty & \infty & 1 & 0 \end{bmatrix}, M_1 = \begin{bmatrix} 0 & \infty & \infty & \infty \\ 3 & 0 & 2 & 1 \\ 1 & \infty & 0 & \infty \\ 2 & \infty & 1 & 0 \end{bmatrix}, M_2 = \begin{bmatrix} 0 & \infty & \infty & \infty \\ \infty & 0 & 2 & 1 \\ 1 & \infty & 0 & \infty \\ 2 & \infty & 1 & 0 \end{bmatrix}$$

Since Algorithm 2 cannot compute the APSP correctly, it is natural to regard it as a novice mistake and move on. But we encounter surprisingly nontrivial questions when we try to understand what's happening. For example, in a sparse graph with no negative edges, we can compute the APSP problem using Dijkstra's algorithm [3] for computing a single-source shortest path in each vertex. Using the Fibonacci heap from [5] yields an $O(nm + n^2 \log n)$ time algorithm for n -vertex, m -edge graphs. Hence, we have an efficient way to obtain the resulting matrix by Algorithm 1, but for Algorithm 2, despite being a seemingly *novice* version, it is unclear how to obtain such matrix efficiently. To this end, we formally define the INCORRECT-APSP problem as follows:

► **Definition 1** (INCORRECT-APSP). *Given a weighted graph with n vertices without negative cycles, compute the matrix returned by Algorithm 2.*

We note that calling Algorithm 2 as an *incorrect* algorithm might be unfair. Indeed, a recurring theme in art and fashion is to take a seemingly *correct* version of a piece and twist it cleverly so that people can find something new from a familiar composition. Even in theoretical computer science, such attempts are not new: For example, a famous *Bogosort* [8] is a sorting algorithm deliberately engineered to perform worse. A more similar example where people consider a variant of a well-known algorithm also exists [7, 6]. In this way, depending on the inspiration we drew, we can call the Algorithm 2 as either IMPROVISED-APSP (Jazz music reference), PUNK-APSP (Rock music reference), or BOGO-APSP (Bogosort reference). However, we will (unfortunately) call the problem INCORRECT-APSP to avoid possible confusion.

A significant inspiration for this work is an arXiv preprint by [7], which considers the exact problem of INCORRECT-APSP. In the preprint, the authors proved that running Algorithm 2 three times on the given adjacency matrix will compute a correct APSP distance matrix. In other words, the INCORRECT-APSP problem is not entirely incorrect - even if the student does not know the correct loop order, they can run the algorithm three times and obtain a correct APSP matrix. While this result itself is funny (who cares about the loop order if you can just run three times?), it raises an intriguing question about the

hardness of the INCORRECT-APSP problem, as their result implies that INCORRECT-APSP is at least as hard as the APSP problem. In other words, a subcubic implementation of INCORRECT-APSP can imply a breakthrough! So, is this seemingly *novice* mistake of APSP made the problem strictly harder? Or can we find a subcubic reduction to prove that the INCORRECT-APSP problem is equivalent to the APSP problem?

1.1 Our Results

Our results in this paper are the following:

► **Theorem 2.** *Given a weighted graph with n vertices and m edges, we can solve the INCORRECT-APSP in $O(nT_{SSSP}(n, m))$ time, where $T_{SSSP}(n, m)$ is the time to execute a single-source shortest path (SSSP) algorithm in a graph with n vertices and m edges.*

► **Theorem 3.** *INCORRECT-APSP is subcubic equivalent to APSP.*

Theorem 2 concerns a notable special case where n iteration of single-source shortest path (SSSP) algorithm is known to outperform the Floyd-Warshall algorithm on the APSP problem. For the INCORRECT-APSP problem, it is unclear whether it can be optimized using the SSSP-based approach. We show this is possible and present an algorithm matching the APSP problem's corresponding bounds. Note that the current record for the SSSP algorithm is $T_{SSSP}(n, m) = O(m \log^2 n \log(nM) \log \log n)$ ([1]). Under the assumption where G do not contain negative weight edges, $T_{SSSP}(n, m) = O(n \log n + m)$ ([5]).

Theorem 3 implies that both INCORRECT-APSP and APSP have a subcubic algorithm or both do not, meaning that we have a new member in the list of APSP-complete problems where the subcubic solution to any of them implies a breakthrough.

1.2 Organization of our paper

In Section 2, we list the definitions and carefully formalize Algorithm 2 in graph theoretic notions. In Section 3, we prove that the INCORRECT-APSP problem is equivalent to the path minimization problem with certain constraints. In Section 4, we prove Theorem 2 by solving the path minimization problem by combining single-source shortest path problem and dynamic programming. In Section 5, we prove Theorem 3 by devising an algorithm that solves the INCORRECT-APSP in subcubic time under the subcubic APSP oracle.

2 Preliminaries

2.1 Notations

For integers i and j , we use $[i, j]$ to denote the set $\{i, i + 1, \dots, j\}$. Let $G = (V, E, w)$ be a weighted directed graph, where $V = [1, n]$ is the set of vertices, $E \subseteq V \times V$ is a set of edges where an element $(i, j) \in E$ represents a directed edge from vertex i to vertex j , and $w : E \rightarrow [-M, M]$ is the weight function of edges. As we represent vertices as integers, we can say that a vertex is *smaller* or *larger* than the other vertex by comparing the integer. We heavily rely on such notation, as our algorithm iterates V in the order of these integers.

We can use another representation for the directed weighted graphs if we do not rely on the graph's sparsity. For a graph $G = (V, E, w)$ with $V = [1, n]$, an *adjacency matrix* $A(G)$ of G is a $n \times n$ matrix such that:

$$A(G)[i, j] = \begin{cases} 0 & \text{if } i = j \\ w(i, j) & \text{if } i \neq j, (i, j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Note that $A(G)[i, j]$ may not accurately represent G if it has negative-weight loops, but this is not a problem since we will invoke Algorithm 1 or Algorithm 2 only if G has no negative cycles. Here, ∞ is an element where $\infty + x = \infty$ holds for all $x \in [-M, M]$.

2.2 Definitions

Algorithm 1 and Algorithm 2 both try to recognize certain kinds of paths in a graph: Starting from the paths with at most one edge, it tries to build up a new paths that is a concatenation of two paths. Here, we will formalize it so that we can view it as a minimization problem over certain types of paths. In Section 3, we will argue that there is a simple characterization of paths realized by Algorithm 2.

We formally define a concept of *path realization*, a generic description of paths realized by the algorithms above.

► **Definition 4.** A path $P = \{p_0, p_1, \dots, p_m\}$ is **realized** by a sequence of 3-tuple $T = \{(i_1, j_1, k_1), (i_2, j_2, k_2), \dots, (i_l, j_l, k_l)\}$, if and only if $m \leq l$, or there exists a pair of integer d, x such that:

- $1 \leq d \leq l$
- $1 \leq x \leq m - 1$
- $(i_d, j_d, k_d) = (p_0, p_m, p_x)$
- $\{p_0, p_1, \dots, p_x\}$ is realized by $\{(i_1, j_1, k_1), (i_2, j_2, k_2), \dots, (i_{d-1}, j_{d-1}, k_{d-1})\}$.
- $\{p_x, p_{x+1}, \dots, p_m\}$ is realized by $\{(i_1, j_1, k_1), (i_2, j_2, k_2), \dots, (i_{d-1}, j_{d-1}, k_{d-1})\}$.

We list some immediate corollaries that can be shown by structural induction on Definition 4. While the corollaries themselves are somewhat trivial, they help us to organize our theorems in a purely mathematical way.

► **Corollary 5.** Let $T_{kij}(n)$ be a sequence of length n^3 consisting of 3-tuples, where the $an^2 + bn + c + 1$ -th element in the sequence is $(b + 1, c + 1, a + 1)$ for all $0 \leq a, b, c \leq n - 1$. Given a $n \times n$ adjacency matrix of G , where G has no negative cycle, Algorithm 1 will return a matrix M , where $M[i, j]$ is the minimum total weight of all simple paths from i to j that is realized by $T_{kij}(n)$.

Proof. It suffices to prove the statement for all paths, as the graph does not contain any negative cycles, and we can turn any non-simple paths into simple paths without increasing their total length.

For any path from i to j realized by $T(n)$, $M[i, j]$ is at most the weight of such paths. Note that the triple nested loop in Algorithm 1 exactly iterates the list $T_{kij}(n)$ and performs a *relaxation* operation of $A[i, j] = \min(A[i, j], A[i, k] + A[k, j])$ for each 3-tuple $(i, j, k) \in T_{kij}(n)$. Hence, we can show this by induction over Definition 4. Specifically, we can prove the following: For all path P of length k realized by a sequence of 3-tuples of length l , $M[i, j]$ is always at most the weight of P after applying the relaxation operation for first l elements of $T_{kij}(n)$. Then, we can apply induction over (k, l) .

Conversely, we can also prove that, for any $1 \leq i, j \leq n$, there exists a path of weight at most $M[i, j]$, which is realized by $T(n)$, which also follows the identical induction as above. ◀

► **Corollary 6.** Let $T_{ijk}(n)$ be a sequence of length n^3 consisting of 3-tuples, where the $an^2 + bn + c + 1$ -th element in the sequence is $(a + 1, b + 1, c + 1)$ for all $0 \leq a, b, c \leq n - 1$. Given a $n \times n$ adjacency matrix of G , where G has no negative cycle, Algorithm 2 will return a matrix M , where $M[i, j]$ is the minimum total weight of all simple paths from i to j that is realized by $T_{ijk}(n)$.

Proof. You can proceed identically as in Corollary 5. ◀

The following provides some characterizations of paths that we will use in the later stage of the paper.

► **Definition 7.** A path $P = \{p_0, p_1, \dots, p_k\}$ with $k \geq 1$ is **increasing** if for all $1 \leq i \leq k$ it holds that $p_{i-1} < p_i$.

► **Definition 8.** A path $P = \{p_0, p_1, \dots, p_k\}$ with $k \geq 1$ is **decreasing** if for all $1 \leq i \leq k$ it holds that $p_{i-1} > p_i$.

► **Definition 9.** A path $P = \{p_0, p_1, \dots, p_k\}$ with $k \geq 1$ is **valley** if for all $1 \leq i \leq k-1$ it holds that $p_i \leq \min(p_0, p_k)$.

► **Definition 10.** A path $P = \{p_0, p_1, \dots, p_k\}$ with $k \geq 1$ is **proper** if there is no index $1 \leq i \leq k-2$ such that $p_i > \min(p_0, p_k)$ and $p_{i+1} > \min(p_0, p_k)$.

3 Main Theorem

The main theorem of our paper is as follows:

► **Theorem 11.** In a graph with n vertices, a nonempty simple path $P = \{p_0, p_1, \dots, p_k\}$ is realized by $T_{ijk}(n)$ if and only if one of the following holds:

- $p_0 < p_k$, and there exists an index $0 \leq i \leq k$ such that $\{p_0, p_1, \dots, p_i\}$ is proper, $\{p_i, p_{i+1}, \dots, p_k\}$ is increasing, and $p_i \geq p_0$.
- $p_0 > p_k$, and there exists an index $0 \leq i \leq k$ such that $\{p_0, p_1, \dots, p_i\}$ is decreasing, $\{p_i, p_{i+1}, \dots, p_k\}$ is proper, and $p_i \geq p_k$.

By combining Theorem 11 with Corollary 6, we can yield an explicit characterization for the output of Algorithm 2.

► **Corollary 12.** Given a $n \times n$ adjacency matrix of G where G has no negative cycle, Algorithm 2 will return a matrix M , where $M[i, j]$ is:

- If $i < j$, the minimum possible total weight of a path $P = \{p_0 = i, p_1, \dots, p_k = j\}$ such that there exists an index $0 \leq x \leq k$ such that $\{p_0, p_1, \dots, p_x\}$ is proper, $\{p_x, p_{x+1}, \dots, p_k\}$ is increasing, and $p_x \geq p_0$.
- If $i = j$, 0.
- If $i > j$, the minimum possible total weight of a path $P = \{p_0 = i, p_1, \dots, p_k = j\}$ such that $\{p_0, p_1, \dots, p_x\}$ is decreasing, $\{p_x, p_{x+1}, \dots, p_k\}$ is proper, and $p_x \geq p_0$.

Proof. By Corollary 6, Algorithm 2 returns a matrix where $M[i, j]$ is the minimum total weight of all simple paths from i to j realized by $T_{ijk}(n)$, which is of above form by Theorem 11. ◀

3.1 High-level ideas

Before proceeding to the proof of Theorem 11, let's play with several examples to motivate intuition. Recall the standard proof of Algorithm 1, where one proves the following lemma by the induction on t :

► **Lemma 13** (Key lemma of [4]). For all $0 \leq t \leq n$, a path $P = \{p_0, p_1, \dots, p_k\}$ is realized by the tn^2 -length prefix of $T_{kij}(n)$ if and only if $p_i \leq t$ for all $1 \leq i \leq k-1$.

19:6 Algorithm for Incorrect Implementation of Floyd-Warshall

This proof strategy surely does not work for Algorithm 2, but it does work identically for the specific type of path, which we define as a valley path:

► **Definition 9.** A path $P = \{p_0, p_1, \dots, p_k\}$ with $k \geq 1$ is **valley** if for all $1 \leq i \leq k - 1$ it holds that $p_i \leq \min(p_0, p_k)$.

However, while all valley paths are realized by $T_{ijk}(n)$, this is still not an exhaustive characterization. Consider the path $P = \{3, 101, 1, 102, 2\}$, which is not a valley path but is realized by $T_{ijk}(n)$ as the subsequence $\{(1, 2, 102), (3, 1, 101), (3, 2, 1)\}$ of $T_{ijk}(n)$ can realize P . The reason is that the algorithm can glue some large vertices into P before it starts to realize the valley path. For this specific counterexample, one can observe that these large vertices should be adjacent to small vertices to get glued inside a valley path, which motivates the definition of a proper path.

► **Definition 10.** A path $P = \{p_0, p_1, \dots, p_k\}$ with $k \geq 1$ is **proper** if there is no index $1 \leq i \leq k - 2$ such that $p_i > \min(p_0, p_k)$ and $p_{i+1} > \min(p_0, p_k)$.

With some modification of proofs in Lemma 13, we can see that all proper paths are realized by $T_{ijk}(n)$. However, this definition is slightly strict: For example, the path $P = \{1, 2, 3, 4\}$ is realized by $\{(1, 3, 2), (1, 4, 3)\}$ but does not fit in a definition of valley path. More generally, we can append any increasing paths in the back of the proper path and prepend any decreasing paths in the front of the proper path. By addressing these cases, we do reach an appropriate characterization of paths realized by $T_{ijk}(n)$, that fits nicely to the inductive argument given in the proof of Theorem 11.

3.2 Proof of the Main Theorem

► **Theorem 11.** In a graph with n vertices, a nonempty simple path $P = \{p_0, p_1, \dots, p_k\}$ is realized by $T_{ijk}(n)$ if and only if one of the following holds:

- $p_0 < p_k$, and there exists an index $0 \leq i \leq k$ such that $\{p_0, p_1, \dots, p_i\}$ is proper, $\{p_i, p_{i+1}, \dots, p_k\}$ is increasing, and $p_i \geq p_0$.
- $p_0 > p_k$, and there exists an index $0 \leq i \leq k$ such that $\{p_0, p_1, \dots, p_i\}$ is decreasing, $\{p_i, p_{i+1}, \dots, p_k\}$ is proper, and $p_i \geq p_k$.

Proof (\Leftarrow). Let n be the number of vertices, and let $T_{ijk}(n, i, j)$ be the prefix of $T_{ijk}(n)$ up to and including the element (i, j, n) .

We first show that all proper paths from i to j are realized by $T_{ijk}(n, i, j)$. We perform an induction over (i, j) . Consider a proper path $P = \{p_0, p_1, \dots, p_k\}$, and assume that there is no index $1 \leq i \leq k - 1$ such that $p_i \leq \min(p_0, p_k)$. Then, since there could be no two entries with $p_i > \min(p_0, p_k)$, it holds that $k \leq 2$, and we are done. Otherwise, take the maximum p_x such that $p_x < \min(p_0, p_k)$. By maximality, both $\{p_0, p_1, \dots, p_x\}$ and $\{p_x, p_{x+1}, \dots, p_k\}$ are proper paths, and by inductive hypothesis, they are realized in $T_{ijk}(p_0, p_x)$ and $T_{ijk}(p_x, p_k)$. Hence, by Definition 4, P is realized.

As $P = \{p_0, p_1, \dots, p_k\}$ from p_0 to p_k are realized by $T_{ijk}(n, p_0, p_k)$, we can see $\{p_0, p_1, \dots, p_k, x\}$ are realized by $T_{ijk}(n, p_0, x)$ for any $x > p_k$, and similarly, $\{x, p_0, p_1, \dots, p_k\}$ are realized by $T_{ijk}(n, x, p_k)$ for any $x > p_0$. Using this fact, we can use a similar induction to prove that we can append an increasing path in the back or a decreasing path in the front. ◀

Proof (\rightarrow). We use the induction on the length l of the prefix of $T_{ijk}(n)$ to show that the path realized by a length- l prefix of $T(n)$ is always in one of such patterns. This claim is true for $l = 0$. Assume $l \geq 1$ and let (i, j, k) be the last element of $T(n)$. Note that we can

assume $i \neq j, j \neq k, k \neq i$, since if one of them holds, we have no new realized paths or a non-simple path.

We list all possible cases and show we can always find such patterns. Here, we use $*$ to denote that any index from 1 to n can be there.

- $i < j, k < i$: We consider a path in a form of $i \rightarrow$ (decreasing) \rightarrow (proper) $\rightarrow k \rightarrow$ (proper) \rightarrow (increasing) $\rightarrow j$. Let z be the first vertex in the increasing part of this path, where $i \leq z$ (as $i < j$ such vertex exists). The path between i to z is proper, and the path between z to j is increasing.
- $i < j, i < k < j$: All entries of $(k, j, *)$ are not in the current prefix, so the only path from k to j realized now is the trivial path $\{k, j\}$. We can append this in the increasing part of our path.
- $i < j, j < k$: All entries of $(i, k, *)$ and $(k, j, *)$ are not in the current prefix, so the path we consider is exactly $\{i, k, j\}$ which is proper.
- $j < i, k < j$: We consider a path in a form of $i \rightarrow$ (decreasing) \rightarrow (proper) $\rightarrow k \rightarrow$ (proper) \rightarrow (increasing) $\rightarrow j$. Let z be the last vertex in the decreasing part of this path, where $j \leq z$ (as $j < i$ such vertex exists). The path between i to z is decreasing, and the path between z to j is proper.
- $j < i, j < k < i$: All entries of $(i, k, *)$ are not in the current prefix, so the only path from i to k realized now is the trivial path $\{i, k\}$. We can prepend this in the decreasing part of our path.
- $j < i, i < k$: All entries of $(i, k, *)$ and $(k, j, *)$ are not in the current prefix, so the path we consider is exactly $\{i, k, j\}$ which is proper. ◀

4 Algorithm for Incorrect-APSP Problem

In this section, we prove the following theorem:

► **Theorem 2.** *Given a weighted graph with n vertices and m edges, we can solve the INCORRECT-APSP in $O(nT_{SSSP}(n, m))$ time, where $T_{SSSP}(n, m)$ is the time to execute a single-source shortest path (SSSP) algorithm in a graph with n vertices and m edges.*

For this, we will try to solve the optimization problem described in Corollary 12. Here, it is useful to observe the following:

► **Observation 14.** *Let $rev(P)$ be the reverse of the path. A path P is realized by $T_{ijk}(n)$ if and only if $rev(P)$ is realized by $T_{ij^k}(n)$.*

Proof. If P satisfies the property from Theorem 11, the reverse also satisfies the property from Theorem 11. ◀

By Observation 14, it suffices to find $M[i, j]$ for all $i < j$, as the other case can be solved by reversing all edges and repeating the same algorithm.

We will fix i and devise an algorithm that computes $M[i, j]$ for all $j \geq i$. To start, we find a minimum-length proper path to all j . By the description in Corollary 12, we only need to compute the proper path ending in $j \geq i$. By Definition 10, a path is proper if it does not contain two adjacent vertices p_x, p_{x+1} of index greater than $\min(i, j)$ unless one of the vertices is the last vertex of the path.

Since we have $\min(i, j) = i$, this is equivalent to not using an edge where both endpoints have an index greater than i , except where the edge is the last on the path. As we fixed i , we can apply SSSP algorithm on G where edges with endpoints greater than i are removed. Let $PROPEREXCEPTLAST[j]$ be the distance from i to j on such graph.

19:8 Algorithm for Incorrect Implementation of Floyd-Warshall

Then, for the last move, we simulate the move for all vertex. Specifically, we initialize $\text{PROPER}[j] = \text{PROPEREXCEPTLAST}[j]$ for all $1 \leq j \leq n$, and for all edge $(u, v) \in E$, we set $\text{PROPER}[v] = \min(\text{PROPER}[u], \text{PROPEREXCEPTLAST}[u] + w(u \rightarrow v))$. In the end, $\text{PROPER}[j]$ holds the shortest proper path from i to j for all $j \geq i$.

Finally, we may need to append the increasing path at the back of each proper path. We can use dynamic programming as the increasing path consists of edges headed to the larger indexed vertex. For computing the $M[i, j]$, we have two choices:

- Assume $p_x = p_k$ and simply take the shortest proper path from i to j .
- Otherwise, we pick an edge $(k \rightarrow j)$ and use the edge as a final edge in the optimal path. Here, $k < j$ should hold, and there should be an edge $(k, j) \in E$. The remaining path from i to k is a proper path appended with an increasing path computed by $M[i, k]$.

In Algorithm 3, we present a pseudocode of the above algorithm.

■ **Algorithm 3** Computing $M[i, j]$ for fixed i , all $j \geq i$.

```
 $G' = (G \text{ without edges where both vertices have index greater than } i)$ 
 $\text{PROPER} = \text{SSSP}(G', i)$ 
 $\text{PROPEREXCEPTLAST} = \text{PROPER}$ 
for  $(u, v) \in E$  do
     $\text{PROPER}[v] = \min(\text{PROPER}[u], \text{PROPEREXCEPTLAST}[u] + w(u \rightarrow v))$ .
for  $j \leftarrow i, i + 1, \dots, n$  do
     $M[i, j] = \text{PROPER}[j]$ 
    for  $k$  incident to  $j$  do
        if  $i \leq k < j$  then
             $M[i, j] = \min(M[i, j], M[i, k] + w(k \rightarrow j))$ 
```

As all procedures except the SSSP use $O(n + m)$ computation, the algorithm's running time is dominated by the SSSP, which leads to the runtime of $O(nT_{\text{SSSP}}(n, m))$.

5 Subcubic equivalence between APSP and Incorrect-APSP

In this section, we will prove Theorem 3. For a computational problem A, B , we say $A \leq_3 B$ if there is a subcubic reduction from A to B as in [10]. The following result is known:

► **Theorem 15** (Theorem 1 of [7]). $APSP \leq_3 \text{INCORRECT-APSP}$.

In the first subsection, we will recite the proof of Theorem 15 from [7]. We emphasize that this is not our original contribution: Our goal is to make this paper self-contained and to rephrase the statement of [7] in terms of subcubic reduction.

In the second subsection, we complement Theorem 15 by proving the following theorem, hence closing the gap:

► **Theorem 16.** $\text{INCORRECT-APSP} \leq_3 APSP$.

Given that Theorem 16 and Theorem 15 are true, the proof of Theorem 3 follows by definition of subcubic equivalence in [10].

► **Theorem 3.** INCORRECT-APSP is subcubic equivalent to $APSP$.

5.1 Proof of Theorem 15

► **Lemma 17.** *Let T^i be a sequence T repeated for i times and P be any simple path in a graph with n vertices. $(T_{ijk}(n))^3$ realizes P .*

Proof. Given a path $P = \{p_0, p_1, \dots, p_k\}$, let m be an integer in $[0, k]$ where $p_m = \max_{i=0}^k p_i$. We call p_i a *skyscraper*, if it satisfies the following:

- $i < m$, and there exists no $j < i$ such that $p_j > p_i$.
- $i = m$.
- $i > m$, and there exists no $j > i$ such that $p_j > p_i$.

It is helpful to observe that p_0 and p_k are always a skyscraper.

Let $0 = i_0 < i_1 < \dots < i_j = m < i_{j+1} < \dots < i_l = k$ be a list of indices where p_{i_s} is a skyscraper for all $0 \leq s \leq l$. For all $1 \leq s \leq l$, it holds that the subpath $p_{i_{s-1}}, p_{i_{s-1}+1}, \dots, p_{i_s}$ is proper - in fact, there can't be even a single vertex with value larger than $\min(p_{i_{s-1}}, p_{i_s})$ since that will add such vertex into a list of skyscraper, contradicting our assumption that i_{s-1} and i_s are adjacent. By Theorem 11, we can see that the subpath $p_{i_{s-1}}, p_{i_{s-1}+1}, \dots, p_{i_s}$ is realized by $(T_{ijk}(n))^1$.

Next, we will prove that both subpath p_0, p_1, \dots, p_m and p_m, p_{m+1}, \dots, p_k are realized in $(T_{ijk}(n))^2$. Given that all subpaths between $p_{i_{s-1}}$ and p_{i_s} are realized in $(T_{ijk}(n))^1$, the remaining skyscrapers are either increasing or decreasing according to its relative location per p_m . The statement holds as we can realize the increasing and decreasing path by Theorem 11.

Finally, to realize P in $(T_{ijk}(n))^3$, we need (p_0, p_k, p_m) in $T_{ijk}(n)$ which we definitely have. ◀

► **Theorem 15** (Theorem 1 of [7]). $APSP \leq_3 INCORRECT-APSP$.

Proof. By Definition 3.1 in [10], it suffices to design a subcubic algorithm for APSP which calls the oracle to compute INCORRECT-APSP for an $n \times n$ matrix, at most a polylogarithmic times.

We take an input graph, construct an adjacency matrix, call INCORRECT-APSP, call INCORRECT-APSP again in the returned matrix, call INCORRECT-APSP again in the returned matrix, and return the output of INCORRECT-APSP. This algorithm calls an oracle for 3 times and runs in quadratic time, which satisfies all requirements to be a subcubic reduction. The correctness follows by Lemma 17. ◀

5.2 Proof of Theorem 16

Let $A \odot B$ be a min-plus matrix product of two $n \times n$ matrix A and B . The following lemma shows that we can convert a proper path minimization into a valley path minimization problem. Note that the definition of G^2 is equivalent to the usual definition of graph powers, defined as a power of adjacency matrix (here, the multiplication operator is \odot).

► **Lemma 18.** *Given a graph G , A proper path of G from i to j with weight w exists if and only if a valley path of G^2 from i to j with weight w exists. Here, G^2 is a complete directed graph on the same set of vertices with G , where the weight of the edge from i to j is the minimum weight path from i to j using at most 2 edges.*

Proof. Every proper path of G translates to a valley path of G^2 , as all vertices that are not i, j and have an index at most $\min(i, j)$ are either adjacent in the path or have at most one intermediate vertex in the path. Conversely, given a valley path of G^2 , we can turn it into a path in G by replacing an edge with two edges and a vertex. Those new are the only vertex that can violate the $p_i \leq \min(p_0, p_k)$ condition, but they are not adjacent by construction. ◀

19:10 Algorithm for Incorrect Implementation of Floyd-Warshall

We will devise an algorithm that uses subcubic oracle for APSP-complete problems to compute the resulting matrix $M[i, j]$. Note that we will only demonstrate how to compute $M[i, j]$ for $i \leq j$ - as in Observation 14, we can compute the opposite part by computing the transpose of $A(G)$ in quadratic time and running the same algorithm. We use the result from [10] that APSP and min-plus matrix multiplication are subcubic equivalent.

Let's first find a matrix for minimum proper paths in G , which by Lemma 18 is equivalent to minimum valley paths in G^2 . Let $V_l[i, j]$ be a minimum weight of the valley path from i to j , which includes all valley paths with at most 2^l edges. Note that this definition includes all valley paths with $\leq 2^l$ edges, but it is not limited to such valley paths - it is, however, limited to valley paths. Here, $V_0 = A(G^2) = A(G)^2$, and we can compute this using min-plus matrix multiplication.

To compute the entry $V_l[i, j]$, we fix the vertex k in the middle of the (imaginary) path P . As k is in the middle of P , and since P has at most 2^l edges, the path to the left and the right of k has at most 2^{l-1} edges.

Let's say a valley path from i to j is *ascending* if $i < j$ and *descending* otherwise. The path to the left of k can be divided into a block of descending paths, and the path to the right of k can be divided into a block of ascending paths. To see this, consider all vertex v such that the subpath from v to k does not have any vertex greater than v , which we refer to as a *skyscraper*. Then, for each adjacent skyscraper, the subpath they form is a valley, as otherwise, we find a skyscraper in between them.

Conversely, it's easy to see that a concatenation of ascending valley paths, followed by descending valley paths, forms a valley path. Hence, a valley path of at most 2^l edges is equivalent to a sequence of descending and ascending valley paths, each with at most 2^{l-1} edges.

The minimum path from i to j that is a sequence of descending valley paths of at most 2^{l-1} edges, can be computed with the APSP oracle: We can provide an adjacency matrix of $V_{l-1}[i, j]$, where all entries with $i < j$ are overwritten to ∞ . Conversely, we can compute the sequence of ascending paths with the APSP oracle by overwriting $V_{l-1}[i, j]$ with $I > j$ to ∞ . Then, we can combine those two patterns by a single min-plus matrix multiplication.

Finally, we need to append an increasing path in the back of the path. By supplementing the adjacency matrix of G where we overwrite all entries with $i > j$ to ∞ , we can compute the increasing path of minimum cost for all pairs using an APSP oracle. We can obtain the desired answer by multiplying this with the valley path matrix.

In Algorithm 4, we present a pseudocode of the above algorithm.

As we make at most $O(\log n)$ calls to the subcubic oracles, Algorithm 4 is subcubic, which proves Theorem 16.

■ **Algorithm 4** Computing $M[i, j]$ for all $1 \leq i \leq j \leq n$.

Ensure: $A \odot B$ is subcubic

Ensure: $APSP(G)$ is subcubic

$V_0 = A(G) \odot A(G)$

for $l \leftarrow 1, 2, \dots, \lceil \log_2 n \rceil$ **do**

 DESCENDINGVALLEY = V_{l-1}

 ASCENDINGVALLEY = V_{l-1}

for $i \leftarrow 1, 2, \dots, n$ **do**

for $j \leftarrow 1, 2, \dots, n$ **do**

if $i < j$ **then**

 DESCENDINGVALLEY[i, j] = ∞

else if $i > j$ **then**

 ASCENDINGVALLEY[i, j] = ∞

$V_l = APSP(DESCENDINGVALLEY) \odot APSP(ASCENDINGVALLEY)$

 VALLEY = $V_{\lceil \log_2 n \rceil}$

$G' = (G \text{ with edges } u \rightarrow v \text{ such that } u < v)$

 ANSWER = VALLEY $\odot APSP(A(G'))$

for $i \leftarrow 1, 2, \dots, n$ **do**

for $j \leftarrow 1, 2, \dots, n$ **do**

if $i \leq j$ **then**

$M[i, j] = \text{ANSWER}[i, j]$

References

- 1 K. Bringmann, A. Cassis, and N. Fischer. Negative-weight single-source shortest paths in near-linear time: Now faster! In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 515–538, Los Alamitos, CA, USA, November 2023. IEEE Computer Society. doi:10.1109/FOCS57990.2023.00038.
- 2 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- 3 Edsger W Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: His Life, Work, and Legacy*, pages 287–290. 2022.
- 4 Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.
- 5 Michael L. Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, July 1987. doi:10.1145/28869.28874.
- 6 Stanley P. Y. Fung. Is this the simplest (and most surprising) sorting algorithm ever?, 2021. arXiv:2110.01111.
- 7 Ikumi Hide, Soh Kumabe, and Takanori Maehara. Incorrect implementations of the floyd-warshall algorithm give correct solutions after three repeats, 2019. arXiv:1904.01210.
- 8 Eric S Raymond. *The new hacker's dictionary*. Mit Press, 1996.
- 9 Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, January 1962. doi:10.1145/321105.321107.
- 10 Virginia Vassilevska Williams and R. Ryan Williams. Subcubic equivalences between path, matrix, and triangle problems. *J. ACM*, 65(5), August 2018. doi:10.1145/3186893.

Variations on the Tournament Problem

Fabrizio Luccio ✉

University of Pisa, Italy

Linda Pagli ✉

University of Pisa, Italy

Nicola Santoro ✉

Carleton University, Ottawa, Canada

Abstract

In 1883, Lewis Carrol wrote a newspaper article to criticize how the second best player was determined in a tennis tournament, and to suggest how such a task could be done correctly. This article has been taken by Donald Knuth as the inspiration for efficiently determining the smallest t elements of a totally ordered set of size n using k -comparisons. In the ensuing research, optimal algorithms for some low values of k and t have been established, by Knuth and Aigner; for $k = 2$ and $t \leq 3$, a few new bounds have been established for special values of n . Surprisingly, very little else is known on this problem, in spite of its illustrious pedigree and its relationship to other classical problems (e.g., selection and sorting with k -sorters). Enticed by the undeniable beauty of the problem, and the obvious promise of fun, we have joined the investigative quest. The purpose of this paper is to share some new results obtained so far. We are glad to report advances in two directions.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms; Computing methodologies → Parallel algorithms; Mathematics of computing → Discrete mathematics

Keywords and phrases algorithms, parallel algorithms, tournament, selection, ranking

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.20

Funding This research has been supported in part by the Natural Sciences and Engineering Research Council of Canada under the Discovery Grant program.

1 Introduction

Lewis Carrol, the poet, mathematician, photographer, and beloved author of *Alice's Adventures in Wonderland*, was also known to be a tennis enthusiast and to have followed many important matches. He did observe that, in the single-elimination tournaments usually adopted for tennis matches, the selection of the winner was fair while that of the runner-up was not [4]. Indeed, in this type of tournaments, the player defeated in the final match is declared second, with the implicit meaning of “second best in the tournament”, while s/he is the true second only with a probability close to half. If, for instance, the true second belongs to the same part of the board as the champion, s/he does not have any possibility to emerge. Lewis Carrol also explained with an example how to determine the true second and third, without however specifying a real algorithm [4]; furthermore, his proposal required a high number of matches, most of which are actually not needed for this determination. A well known presentation and discussion of Lewis Carroll’s proposal has been provided by Donald Knuth [5].

In the game of tennis a match involves two players; in general, the problem of determining the ranking of the players in tournaments can be formulated as the following combinatorial problem: given a set of n elements and a comparison operation on k elements (k -comparison) that returns the linear ordering of them, we want to calculate the number of k -comparisons needed to find the top t elements. We shall call this problem **t-TopFinding in k-TrackRacing** (t - T - k - T); see Figure 1.



© Fabrizio Luccio, Linda Pagli, and Nicola Santoro;
licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 20; pp. 20:1–20:11

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

20:2 Variations on the Tournament Problem

Let $C(n, k, t)$, with $t < n$, be the number of k -comparisons of an algorithm that determines the first t elements among n (*upper bound*), and let $S(n, k, t)$ be the minimum number of k -comparisons required to solve the problem (*lower bound*). For $k = 2$ and $t = n$ the problem corresponds to traditional sorting with binary comparisons. Optimal algorithms for some low values of k and t have been established by Knuth [5] and, a decade later, by Aigner [1], namely:

► **Fact 1** ([5]).

$$\begin{aligned} C(n, 2, 1) &= S(n, 2, 1) = n - 1 \\ C(n, 2, 2) &= S(n, 2, 2) = n + \lceil \log_2 n \rceil - 2 \\ S(n, k, 1) &= \lceil \frac{n-1}{k-1} \rceil \end{aligned}$$

► **Fact 2** ([1]). Let $n = 2^s + r$, where $0 \leq r \leq 2^{s-1}$. Then

$$C(n, 2, 3) = S(n, 2, 3) = \begin{cases} (n-3) + \lceil \log_2 n \rceil & \text{if } r = 0, \\ (n-3) + \lceil \log_2 n \rceil + 1 & \text{if } 1 \leq r \leq 2^{s-2}, \\ (n-3) + \lceil \log_2 n \rceil + 2 & \text{otherwise.} \end{cases}$$

For $k = 2$ and $t \leq 3$, a few new bounds have been established for special values of n [6, 3]. Surprisingly, very little else is known on this problem, in spite of its relationship to classical problems of selection and sorting (see the nice review by Iványi and Fogarasi [8]) especially using k -comparators (called k -sorters, e.g. [2, 9])

Enticed by the noble pedigree of the problem, the undeniable beauty of the challenge, and the obvious promise of fun, we have started to examine t - T - k - T . The purpose of this paper is to share some new results obtained so far. We are glad to report advances in two directions.

In Section 2 of this paper we discuss our study of the problem, for any value of n and k , when $t = 2$. We show that a standard two-phases tournament can be improved when $(n-1)/(k-1)$ is not an integer, and find a new lower bound $S(n, k, 2)$ that matches the upper bound $C(n, k, 2)$ for all values of n .

In Section 3 we take a different approach which has so far received little attention. We consider the k -comparison problem in a parallel setting, where the most significant function to consider is the number $R(n, k, t, p)$ of *rounds* played in parallel by a certain number of groups of k competitors, where p is now the maximum number of available facilities (racetracks, or tennis courts, or football fields, etc.). For minimum R we then minimize $C(n, k, t, p)$. We study the problem thoroughly for k and t up to three, conjecturing what happens for larger values.



■ **Figure 1** A horse race: a real life instance of t - T - k - T .

2 Winner and runner-up using k comparisons

2.1 An introductory example

Consider n horses running on a track of k lanes, among which we want to establish the first $t = 2$ with minimal number of races. After a race, a linear ordering is established among the k horses on the track, based only on the finishing order (race time is not considered). For $n = 25$ and $k = 5$ lanes in a track, we can find the winner and runner-up by applying an immediate generalization of the 2-phases known tournament algorithm for $k = 2$. In the first phase, the horses are divided into $n/k = 5$ groups of five horses each, and each group runs in a race corresponding to a 5-comparison. Then the five winners run in a final race to establish the champion, with a total of six races. The second phase takes place with a second tournament between all the horses who were beaten directly (i.e. in the same race) by the champion and ranked second in that race. Since there are five such horses, one additional race is sufficient to award the second place, for a total of seven races.

2.2 The upper bound

Let us review what is known on $C(n, k, 2)$ for arbitrary values of n and k , and propose a non-standard tournament organization by which the value of $C(n, k, 2)$ can be lowered for particular pairs n, k .

As stated in Fact 1, $\lceil \frac{n-1}{k-1} \rceil$ k -comparisons are needed in general to establish the top element. In a standard two-phases tournament for arbitrary k , the top-element appears in $\lceil \log_k n \rceil$ k -comparisons, from which $\lceil \log_k n \rceil$ elements are ranked as a possible second and additional $\lceil (\lceil \log_k n \rceil - 1)/(k - 1) \rceil$ k -comparisons are used to establish the real second one. As we have seen in the introductory example above, for $n = 25$ and $k = 5$ we have $\lceil \log_k n \rceil = 2$, $\lceil (\lceil \log_k n \rceil - 1)/(k - 1) \rceil = 1$, and a further k -comparison is sufficient to award second place. Therefore for a standard two-phases tournament we have:

► **Fact 3.** $C(n, k, 2) = \lceil (n - 1)/(k - 1) \rceil + \lceil (\lceil \log_k n \rceil - 1)/(k - 1) \rceil$.

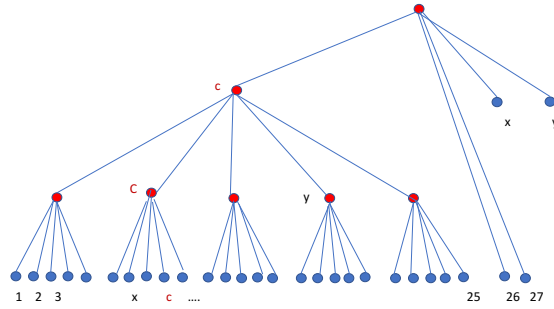
This upper bound is tight if $(n - 1) \bmod (k - 1) = 0$. However, should $(n - 1)$ and $(k - 1)$ be coprime, in a k -comparison of the first phase, not all k entry points are needed; these “free” entry points could be used to partially contribute to the second phase where the second element is determined.

For example, for $n = 27$ and $k = 5$ we perform $\lceil n - 1/(k - 1) \rceil = 7$ comparisons for the top-element, but one of them has two not exploited entry points that can be used in the search for the second element. As shown in Figure 2, we can re-organize the initial phase of a tournament on $n = 27$ elements, postponing the last of the seven 5-comparisons where the winner of the first twenty-five elements is compared to elements number 26 and 27, to include the elements x and y that *must* compete for the second place. So the total number of k -comparisons is seven instead of eight, as we would get from Fact 3 if a standard two-phases algorithm is applied.

We now transform the above considerations into a precise algorithmic form, and we establish an upper bound for $C(n, 2, 2)$ and any value of n and k with $n > k$.

Let $r = (n - 1) \bmod (k - 1)$; then the number p of entry points of a k -comparison left free in the first tournament is $p = 0$ if $r = 0$, $p = k - r - 1$ otherwise. Let $n' = \lceil \log_k n \rceil$; let $n'' = n' - p - 1$ if $n' > p$, $n'' = 0$ otherwise.

If $r \neq 0$ and $n' \leq p$, the free entry points of the first phase can absorb all the remaining elements among which the second element has to be determined. This is the case of the latter example where $n = 27$, $k = 5$, $\lceil \log_k n \rceil = 3$, $r = 2$, and the selection of the first and second element can be performed in a single phase. We have:



■ **Figure 2** Tournament for $n = 27$ and $k = 5$. The selection of the first and second element can be accomplished in the first phase. The last comparison includes elements 26, 27, x and y . x is the second in the first k -comparison of the champion c , and y is the second in the second comparison of c .

► **Theorem 1.** *Let $n > k \geq 2$. Then*

$$C(n, k, 2) = \begin{cases} \lceil (n-1)/(k-1) \rceil + \lceil (n'-1)/(k-1) \rceil & \text{if } r = 0, \\ \lceil (n-1)/(k-1) \rceil + \lceil n''/(k-1) \rceil & \text{otherwise.} \end{cases}$$

Proof. Case $r = 0$ is known. If $r \neq 0$ run the first tournament with $n + p$ elements with $\lceil (n-1)/(k-1) \rceil$ comparisons. If $n' \leq p + 1$, then $n'' = 0$ and no other comparisons is needed. Otherwise, $n'' = n' - p - 1$ and a second tournament is run with $n - p$ elements. ◀

2.3 The lower bound

We now show a matching lower bound $S(n, k, 2)$ for determining the first and second elements, recalling the definition of r and p of the previous section. .

► **Theorem 2.** *Let $n > k \geq 2$. Then $S(n, k, 2) = C(n, k, 2)$.*

Proof. First of all, note that for $k = 2$ the lower bound is the known formula $S(n, 2, 2) = n + \lceil \log_2 n \rceil - 2$ (see Fact 1). Note that this formula has been proved by [5] selecting the second, once the first has been found. We follow the same approach.

Further note that, for $r \neq 0$ and $p = 0$, upper and lower bounds match.

Let $k > 2$; we shall consider the cases $r = 0$ and $0 < r < k$, separately.

Case $r = 0$.

First observe that $\lceil (n-1)/(k-1) \rceil$ is the necessary number of k -comparisons for the selection of the top element. Let us then determine the minimal number of additional k -comparisons required to determine the second element. In order to do so, we need not to include any comparison among elements whose relation can be derived, by transitivity, from previous k -comparisons. In order to minimize the number of elements ordered by transitivity, it is sufficient to impose the following *balancing rule*: if, in a k -comparison, it results that x_j is less than x_l , $1 \leq j \neq l \leq k$, then the number of elements already known to be less than x_j ,

are less or equal than the number of elements already known to be less or equal than x_l . Let us now define as $L(i)$ the maximum number of elements established to be less than the champion after i k -comparisons; this knowledge can be acquired either directly, because the elements are compared with the top one in a k -comparisons, or by the transitive property. From the balancing rule, we have the following recursive definition:

$$L(i) = \begin{cases} k - 1 & \text{if } i = 1 \\ kL(i - 1) + k - 1 & \text{otherwise} \end{cases}$$

whose closed formula is $L(i) = k^i - 1$.

Since the elements less than the champion are $n - 1$, there exists a \hat{i} such that $L(\hat{i}) = k^{\hat{i}} - 1 = n - 1$. The value $\hat{i} = \lceil \log_k n \rceil$ represents the number of k -comparisons done by the champion to win the competition. The second element has to be established among all the elements classified at the second place in a direct k -comparison with the champion, whose number is precisely \hat{i} . Therefore the second element cannot be determined in less than $\lceil [(\log_k n) - 1]/(k - 1) \rceil$ additional comparisons, for a total of $\lceil (n - 1)/(k - 1) \rceil + \lceil (n' - 1)/(k - 1) \rceil$.

Case $0 < r < k$.

In this case, exactly $p = k - r - 1$ entry points are empty and can be exploited to compare elements to determine the second element. If $n' - 1 \leq p$, all these elements can be accommodated in the empty entry points and the second element can be selected directly in the block of comparisons needed to establish the first element. However, if $n' > p$, at least $n'' = n' - p - 1$ elements require a second phase, requiring an additional $\lceil (n'' - 1)/(k - 1) \rceil$ k -comparisons, for a total of $\lceil (n - 1)/(k - 1) \rceil + \lceil n''/(k - 1) \rceil$. ◀

Let consider, for instance, the case $n = 47$ $k = 4$. We have $n' = 3$. According to the theorem we have $S(47, 4, 2) = \lceil 46/3 \rceil + \lceil (3 - 2 - 1)/3 \rceil = 16$. Computing the two phases separately we would have been obtained a number of comparisons equal to: $\lceil 47/3 \rceil + \lceil (3 - 1)/3 \rceil = 16 + 1 = 17$.

It is very easy to modify the tournament algorithm of Sec.2 in a way that the case that the approximation $\lceil (n - 1)/(k - 1) \rceil$ is not exact is also considered, in order to obtain a matching upper bound.

3 t-T-k-T in parallel

Everything we have discussed so far reflects a standard computing approach. Let us now look at the problem from a slightly different point of view, that is, the various operations required may take very different times to be executed. In particular this is the case of sports competitions such as horse racing, tennis tournaments, final stages of football championships, where comparisons of indexes such as those appearing in **while** or **do** instructions needed for deciding the order of the events take incomparably less time than the comparison in **if** $A[i] > A[j]$, where $A[i]$ and $A[j]$ are two players competing in the game.

In fact the complexity of most sorting and searching algorithms is evaluated in order of magnitude by counting the number of comparisons between elements of the dataset, and this is due to the observation that the number of all other operations is of the same order as those, and each of them requires time comparable to the others. Instead we will now evaluate the number of single sports matches because they dominate the overall time. In particular we consider the *k-comparison problem* in a parallel setting, where a new function to consider

20:6 Variations on the Tournament Problem

is the number $R(n, k, t, p)$ of rounds played in parallel by a certain number of groups of k competitors, where p is a new parameter representing the maximum number of available facilities (racetracks, or tennis courts, or football fields). Our primary goal is minimizing the number R of rounds (e.g. days) the tournament lasts, imagining that each competitor can play a maximum of one game per day. For minimum R we then minimize the total number $C(n, k, t)$ of comparisons made (e.g., for better conservation of the facilities). As you shall see, we study the problem in full for k and t up to three, leaving as a conjecturing what happens for larger values.

Note that the k -comparison problem was discussed in depth in the literature without any real attention to parallel processing. This aspect was considered in [10] in a much wider context, evaluating the time complexity in order of magnitude as a function of n and p instead of counting the exact number comparisons and rounds as we do. And then it was discussed in [7], with focus on the overhead communication time for different kinds of interconnection networks. Our approach is completely different. Let's start with some clarifications on what was said in the previous sections.

The results of Fact 1 are valid for any value $p \geq 1$. Another result implicitly discussed in the previous section must be better specified, namely:

► **Fact 4.** $R(n, k, 1, p) = \lceil \log_k n \rceil$ is an upper and a lower bound of R for $p \geq n/k$.

Note that the upper bound in Fact 4 is demonstrated directly by referring to the standard tournament algorithm (see the proof of Theorem 2), and the lower bound is also proved by equivalent reasoning although this is generally not underlined. In fact, for $k = 2$ the adversary model used in the proof implies that in an optimal algorithm with $n - 1$ comparisons, when the final winner f is engaged in a new comparison with another competitor g the following must happen. If f and g have already been recognised as superior to other participants of sets F and G respectively, f wins the comparison with g only if $|F| \geq |G|$ and $F \cap G = \Phi$. Then the computation must proceed essentially as in a tournament or equivalent, where f and g are the temporary winners of two independent sub-problems solved on disjoint subsets of participants. This reasoning can be immediately extended to $k > 2$ where an optimal algorithm requires $\lceil \frac{n-1}{k-1} \rceil$ k -comparisons to find the winner f , and when this is engaged in any k -comparison with $k - 1$ competitors all of them must be top elements of disjoint subsets of cardinality less or equal the one of f .

From now on we will assume that n is a power of k without affecting the core of the problem. In fact we can add fictitious competitors who, by default, would lose all matches with the others, until reaching a number of competitors equal to the next power of k . This does not affect Fact 4 where the integer approximation is no more needed. For $p < n/k$, instead, the bound of Fact 4 must be increased because it will not be possible to execute in parallel all the comparisons due in certain rounds. As an example we limit the calculation of these bounds to $k = 2$ as it is not particularly interesting.

► **Fact 5.** For $n = 2^q$, and for $2^i \leq p < 2^{i+1}$ with $0 \leq i \leq q - 1$, $R(n, 2, 1, p) = 2^{q-i} + i - 1$ is an upper bound of R .

Proof. Consider the tournament algorithm, where some rounds are divided into several consecutive sub-rounds if needed.

(1) The two limit cases $i = 0$ and $i = q - 1$ are immediate. For $i = 0$ we have $2^0 \leq p < 2^1$ and $R(n, 2, 1, 1) = 2^q - 1$, in fact only one comparison can be made in each round. For $i = q - 1$ we have $2^{q-1} \leq p < 2^q$ and $R(n, 2, 1, 1) = q$, so that the $\log_2 n$ rounds indicated in Fact 4 are executed.

(2) For a generic value of i with $0 < i < q - 1$, some rounds of the tournament must be performed as a sequence of sub-rounds. In fact, the $2^{q-1}, 2^{q-2}, \dots, 2^{i+1}$ comparisons needed in rounds $1, 2, \dots, q - i - 1$, must be divided into $2^{q-1-i}, 2^{q-2-i}, \dots, 2^1$ sub-rounds played in the $p = 2^i$ facilities. The total number of these sub-rounds is $2^{q-i} - 2$, to which the number $q - (q - i - 1) = i + 1$ of undivided rounds must be added, and the bound $2^{q-i} + i - 1$ follows. ◀

For example, for $n = 2^5$ and $2^2 \leq p < 2^3$ we have $i = 2$ hence $R(2^5, 2, 1, 2^2) = 2^3 + 2 - 1 = 9$. Giving a matching lower bound is open.

We now study the best way to determine in parallel the first elements of the game for the values $n = 2^q$, $k = 2$, $p \geq n/2$, and $t=2$ (for $t = 1$ the problem is solved in Fact 3), and for $n = 3^q$, $k = 3$, $p \geq n/3$, and $t = 3$. We give bounds on the function R , and then on C when the former are satisfied. Then the solution for $k > 3$ and $t > 3$ is discussed as a conjecture.

3.1 The case $k=2$

We start computing $R(n, 2, 2, n/2)$ and then $C(n, 2, 2)$, for $n = 2^q$. Based on the considerations made above on the proof of Fact 4 we will use a parallel tournament to calculate the winner, adding new comparisons in different rounds to calculate the second too. Recall that in a binary tree representing a tournament, the vertices in each round (i.e. level of the tree) are labelled with match winners who are known at the end of the matches, so the results cannot be used before the next round. We have:

▶ **Theorem 3.** $R(n, 2, 2, n/2) = \log_2 n + 1$ is an upper and a lower bound of R , for n power of two.

Proof. (i) *Upper bound.* Based on the inductive application of the following reasoning. Consider the section of a tournament shown in figure 3-above (solid edges) where s is possible final winner, as it will be known after the comparison between p and s is made in round r_{i+2} . At round r_{i+1} , n, m are the possible candidates for the second position, where m is the maximum element among the ones in the sub-tournament lead by p in round r_i . A symmetric situation holds for the pair t, q in the sub-tournament lead by s in the round r_{i+1} . Comparisons n vs m and t vs q can be made in round r_{i+2} along with p vs s , and these three results will be usable from round r_{i+3} onwards.

The same situation now occurs for the element s which is the winner of the round r_{i+2} , and will compete in the round r_{i+3} for the final victory with the winner of another section of the tournament not shown. The looser p at level r_{i+2} , and the maximum element in the sub-tournament lead by s in round r_{i+1} , will compete for second position. In the example provided, such a maximum element is q , as determined with the comparison q vs t in the round r_{i+2} . Note that the comparison n vs m in the round r_{i+2} is actually useless, but this was not predictable in that round where the outcome of s vs p was not known.

To see how our algorithm works, refer to figure 3-below. Rounds r_1 and r_2 are like the ones of a standard tournament for the winner ($t = 1$). Starting to round r_3 a subset of competitors are compared to determine the final winner, and for each of these comparisons, two more comparisons are created to determine the final runner-up as shown in the figure. The basis of the induction is shown in the first three rounds, where the runner-up competitors in each sub-tournament of round r_3 are the pairs of elements defeated by the current winner in rounds r_1 and r_2 .

As usual the winner is decided in round $r_{\log_2 n}$ (r_4 in the example). In the same round two more comparisons are made to decide the two candidates for second place which, is decided in the next round $r_{\log_2 n + 1}$. Note that in each round the number of comparisons is less than or equal to $p = n/2$ (see the next corollary).

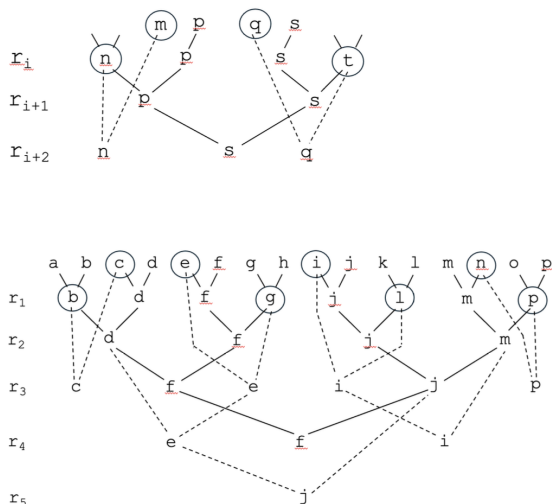
20:8 Variations on the Tournament Problem

(ii) *Lower bound.* To prove that $\log_2 + 1$ rounds are needed note that, as we said before, the winner can be determined in $\log_2 n$ rounds only using a standard tournament or equivalent algorithm. Two candidates x, y for the first position are compared in the last round, and all others are divided into two subsets X, Y where so far no element of one has been compared directly or indirectly with an element of the other. Clearly the loser between x and y , say x , can be the runner-up, but to make this decision x must be compared to at least one element of Y , say z , since no element of Y except y has ever been compared to the elements of X . However the x vs z comparison cannot be done before knowing that x lost to y , so an additional round is required after the comparison x vs y . ◀

From the algorithm reported in Theorem 3 (i) we have with simple calculations:

► **Corollary 4.** *If the first and the second elements are determined in $\log_2 n + 1$ rounds, $C(n, 2, 2) = 3n/2 - 2$ is an upper bound of C , for n power of two.*

Note that the upper bound of Corollary 4 is higher than that of Fact 1, due to the requirement to proceed with a minimum number of parallel rounds. Finding a corresponding lower bound is an open problem.



■ **Figure 3** Above: Section of a scoreboard (solid edges). s is a possible winner. n, m and q, t are the pairs for a possible runner-up before the comparison between p and s is made. Below: Scoreboard of a tournament of 5 rounds for 2^4 competitors. f is the winner, and j is the runner-up.

For $p = n/2$ the comparisons required in each round of the algorithm in Theorem 3 are done in the round itself. Since it is not possible to select some participants to engage in further comparisons before the results of the current round are known, it is not necessary to allow $p > n/2$ parallel comparisons. For $p < n/2$, however, the initial rounds must be divided into parts and a new value for R must be determined as done in Fact 3. Let's leave out the boring and uninteresting calculations involved.

3.2 The case $k=3$ and beyond

We now compute $R(n, 3, 3, n/3)$ and then $C(n, 3, 3)$, for $n = 3^q$. Again we use a parallel tournament to calculate the winner, adding new comparisons in different rounds to calculate the second and the third too. For better understanding, in the scoreboard the vertices in

each round will be labeled $x - y - z$ according to the resulting order in the comparison among x, y, z (in particular the winner is shown in bold). Again this order is known at the end of the comparison, so the result cannot be used before the next round. The proof of the following theorem is an extension of the one given for Theorem 3. We have:

► **Theorem 5.** $R(n, 3, 3, n/3) = \log_3 n + 2$ is an upper and a lower bound of R , for n power of three.

Proof. (i) *Upper bound.* Consider the section of a tournament shown in figure 3-above. In each round the comparisons for the possible first, second, and third final elements are shown with thick-solid lines, thin-solid lines, and dashed lines, respectively. First turn your attention to the participants a to i in the left section of the figure. Once the result $d - a - g$ of round r_{i+1} is known, the candidates for the final second and third positions are limited to $e - a$ and $f - b - g$ respectively, and these two comparisons are scheduled for round r_{i+2} , where they will be carried out together with the comparison among d, z, j for the first place. This last comparison produces the ordering $z - d - j$, so now d can be at most the second finisher and will be considered for this position in the next round r_{i+3} , competing with v which emerges as possible second from the section of the tournament led by z in the previous round r_{i+1} .

The second position occupied by d in round r_{i+2} also has an important consequence on the competition for the third place. Indeed, the possible second e now becomes a possible third, and will compete with x and j which similarly emerge from the section of the tournament led by z in the round r_{i+1} . Note that no element in the tournament section led by j in round r_{i+1} , except j itself, can now compete for second or third position because j was third in the comparison $z - d - j$ in round r_{i+2} . Furthermore, the possible third f that emerged in round r_{i+2} in the comparison $f - b - g$ is now abandoned, and that comparison has no further consequences.

All in all, in each round $r_{j \geq i+1}$, for each comparison for the first position, there is one comparison for the second position and one for the third position in the round r_{j+1} . The winner is decided in round $r_{\log_3 n}$. In the same round four more comparisons are made, two of which to decide the two candidates for second place, and the other two to decide the three candidates for the third place. The holders of the second and third place will be respectively determined in the rounds $r_{\log_3 n+1}$ and $r_{\log_3 n+2}$ as shown in figure 4-below. Note that in each round the number of comparisons is less than or equal to $p = n/3$ (see the next corollary).

(ii) *Lower bound.* As in the proof of theorem 3, note that $\log_3 n$ rounds are needed for determining the winner (Fact 4), and this may happen only using a tournament or equivalent algorithm. Refer to figure 3. In the round $r_{\log_3 n}$ three candidates a, b, c remain for the first position and are compared in this round. All the others are divided into three subsets A, B, C where so far no element of one has been compared directly or indirectly with an element of the others. Suppose the comparison for first place results in $b - a - c$. Then b is the final winner, and the second place can be assigned to a or to at least one element, say v , belonging to B . Note that v cannot belong to C , as c was third in the comparison with b and a and all elements of C are inferior to c . So at least one comparison, say $a - v$, must be made in the round $r_{\log_3 n} + 1$ to assign second place. Clearly the winner a in this comparison is the second of the tournament and the loser v may be the third, but in this round there is at least another competitor for third place, coming from one of the three sections of the tournament led by a, b , and c in the round $r_{\log_3 n}$. So another round $r_{\log_3 n+2}$ is needed to award the third place. ◀

20:10 Variations on the Tournament Problem

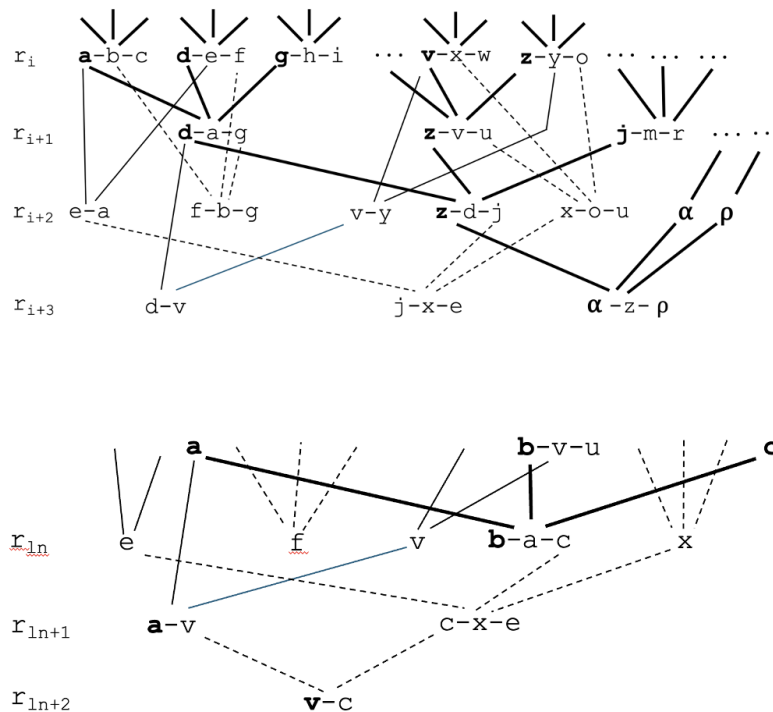
From the algorithm reported in Theorem 5 we have:

► **Corollary 6.** *If the first, second, and third elements are determined in $\log_3 n + 1$ rounds, $C(n, 3, 3) = 13n/18 + 1/2$ is an upper bound of C , for n power of three.*

Finding a corresponding lower bound for $C(n, 3, 3)$ is an open problem.

We can now speculate on competitions with k -ary comparisons, with $k > 3$. We believe that the algorithm presented in the upper-bound of the Theorems 3 and 5 can be immediately extended, adding comparisons from the third round onwards. In particular, adding binary, ternary, \dots k -ary comparisons in each round to determine the candidates for the second, third, \dots k -th place. Since the proof would probably be long and tedious, we pose the following:

Strong Conjecture. $R(n, t, t, n/t) = \log_t n + t - 1$ is an upper and a lower bound of R , for n power of t .



■ **Figure 4** Above: Section of a scoreboard for 3^q competitors. Thick-solid, thin-solid, and dashed lines indicate comparisons for the first, second, and third place respectively. Below: The final three rounds.

References

- 1 Martin Aigner. Selecting the top three elements. *Discrete Applied Math.*, 4:247–267, 1982.
- 2 Richard Beigel and John Gill. Sorting n objects with a k -sorter. *IEEE Transactions on Computers*, 39:714–716, 1990.
- 3 D.Kirkpatrick. *Closing a Long-standing Complexity Gap for Selection*, volume 8066. LNTCS Springer, 2013.
- 4 Charles L. Dodgson. The fallacies of lown tennis tournaments. *St. James Gazette*, 1:5–6, 1883.
- 5 Donald E.Knuth. *The Art of Computer Programming, Vol 3*. Addison Wesley, 1973.

- 6 Jutta Eusterbrock. Errata to "selecting the top three elements by m. aigner: A result of a computer-assisted proof search". *Discrete Applied Math.*, 41:131–137, 1993.
- 7 Susumu Horiguci and Willard L. Miranker. On parallel algorithm for finding the maximum value. *Parallel Computing*, 10:101–108, 1989.
- 8 Andras Ivanyi and Norbert Fogarasi. On partial sorting in restricted rounds. *Acta Univ. Sapiientiae, Informatica*, 9:17–34, 2017.
- 9 Bruce Parker and Ian Parberry. Costructinfg sorting networks from k-sorters. *Information Processing Letteers*, 33:157–162, 1989.
- 10 Yo Shiloach and Uzi Vishkin. Finding the maximum, merging, and sorting in a parallel computational model. *Journal of Algorithms*, 2(88):88–102, 1981.

PSPACE-Hard 2D Super Mario Games: Thirteen Doors

MIT Hardness Group¹

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Hayashi Ani ✉

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Erik D. Demaine ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Holden Hall ✉

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Matias Korman ✉

Siemens Electronic Design Automation, Wilsonville, OR, USA

Abstract

We prove PSPACE-hardness for fifteen games in the Super Mario Bros. 2D platforming video game series. Previously, only the original Super Mario Bros. was known to be PSPACE-hard (FUN 2016), though several of the games we study were known to be NP-hard (FUN 2014). Our reductions build door gadgets with open, close, and traverse traversals, in each case using mechanics unique to the game. While some of our door constructions are similar to those from FUN 2016, those for Super Mario Bros. 2, Super Mario Land 2, Super Mario World 2, and the New Super Mario Bros. series are quite different; notably, the Super Mario Bros. 2 door is extremely difficult. Doors remain elusive for just two 2D Mario games (Super Mario Land and Super Mario Run); we prove that these games are at least NP-hard.

2012 ACM Subject Classification Theory of computation → Complexity classes

Keywords and phrases video games, computational complexity, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.21

Supplementary Material

Software (ROM Files): <https://github.com/65440-2023/mario-hardness-gadgets>

Acknowledgements This paper was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.5440), taught by Erik Demaine in Fall 2023; and during the 33rd Bellairs Winter Workshop on Computational Geometry, co-organized by Erik Demaine and Godfried Toussaint in March 2018 in Holetown, Barbados. We thank the other participants of that class and workshop for helpful discussions and providing an inspiring atmosphere. Much of the work in this paper would not have been possible without the wealth of tools created by the communities surrounding the Super Mario games. In particular, the following level editing and emulation tools were of immense importance to testing various gadget ideas and mechanics:

- *The NES Super Mario Brothers 2 Level Editor* by loginsinex – <https://github.com/loginsinex/smb2>
- *SMB3 Foundry* by mchlnix – <https://github.com/mchlnix/SMB3-Foundry>
- *MarCas* by Coolman – <https://www.romhacking.net/utilities/518/>
- *Lunar Magic* by FuSoYa – <https://fusoya.eludevisibility.org/lm/>

¹ Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as “MIT Hardness Group” (without “et al.”).



- *Golden Egg* by Romi – <https://www.smwcentral.net/?p=section&a=details&id=4645>
- *Reggie!* by the NSMBW Community – <https://github.com/NSMBW-Community/Reggie-Updated>
- *CoinKiller* by Arisotura – <https://github.com/Arisotura/CoinKiller>
- *Miyamoto* by abood40091 – <https://github.com/abood40091/Miyamoto>
- *BizHawk* by TASEmulators – <https://github.com/TASEmulators/BizHawk>
- *Mesenrta* by threecreepio – <https://github.com/threecreepio/mesenrta>
- *Mesenrta-s* by threecreepio – <https://github.com/threecreepio/mesenrta-s>
- *mGBA* by endrift – <https://mgba.io/>
- *Dolphin* by the Dolphin Emulator Project – <https://dolphin-emu.org/>
- *Citra* by Citra Team – <https://citra-emu.org/>
- *Cemu* by Team Cemu – <https://cemu.info/>

1 Introduction

At FUN 2016, Demaine, Viglietta, and Williams [3] proved that it is PSPACE-hard to complete a level in Super Mario Bros., when the game is generalized to an arbitrary level size, screen size, number of on-screen enemies, and (exponentially large) time limit. (They also considered versions with bounded screen size, where off-screen enemies reset, but this makes the game substantially easier.) But Super Mario Bros. is just the first game in a venerable series of Super Mario platforming video games. Consequently, that paper ended with an open problem about other games:

Finally, we suspect that our proofs can be adapted to the many Super Mario Bros. sequels, but this remains to be explored. [3]

In this paper, we explore these sequels, analyzing the complexity of all fifteen 2D Super Mario platform video games released to date. Table 1 summarizes our results, most of which are PSPACE-hardness. Previously, no other PSPACE-hardness results were known, though four of the games we prove PSPACE-hard were known to be NP-hard from an earlier FUN 2014 paper [1].

Our PSPACE-hardness reductions all involve building “door gadgets”, a technique first used to prove PSPACE-completeness of Lemmings [6] and then Super Mario Bros. [3]. An *open-close door gadget* is a constant-size piece of a level that can be in two states, open or closed, and has three possible traversal paths: the *open* path allows the player to change the state to open, the *close* path forces the player to change the state to closed, and the *traverse* path can be traversed only when the door is open.

These original applications also required a “crossover gadget” to enable non-interacting crossing tunnels for the player to traverse. At FUN 2020, however, Ani et al. [2] showed that (in most cases) just a door gadget suffices, and crossovers are unnecessary. They also introduced two other types of doors – self-closing doors and symmetric self-closing doors – each of which alone suffices to prove PSPACE-completeness. They also applied this doors framework to prove that all 3D Mario games released to date are PSPACE-hard.²

In this paper, we apply the doors framework of [2] to prove PSPACE-hardness of thirteen more 2D Mario games. Several of these doors (presented in Section 3) are variations of the open-close door gadget from Super Mario Bros. [3], but even so, they require careful

² Since the paper appeared, one more 3D Mario game has been released: Bowser’s Fury (as part of Super Mario 3D World + Bowser’s Fury). But this game has the same mechanics as Super Mario 3D World, in particular switchboards, so their PSPACE-hardness proof applies.

■ **Table 1** New and known results for all sixteen 2D Mario platform games, in order of release date.

Year	Game	Lower Bound	Ref	Previous Bound
1985	Super Mario Bros.	PSPACE-hard	[3]	NP-hard [1]
1986	Super Mario Bros.: The Lost Levels	PSPACE-hard	Thm. 1	NP-hard [1]
1988	Super Mario Bros. 2	PSPACE-hard	Thm. 12	NP-hard [1]
1988	Super Mario Bros. 3	PSPACE-hard	Thm. 2	NP-hard [1]
1989	Super Mario Land	NP-hard	Thm. 13	
1990	Super Mario World	PSPACE-hard	Thm. 4	NP-hard [1]
1992	Super Mario Land 2: 6 Golden Coins	PSPACE-hard	Thm. 9	
1995	Super Mario World 2: Yoshi's Island	PSPACE-hard	Thm. 10	
2006	New Super Mario Bros.	PSPACE-hard	Thm. 11	
2009	New Super Mario Bros. Wii	PSPACE-hard	Thm. 11	
2012	New Super Mario Bros. 2	PSPACE-hard	Thm. 11	
2012	New Super Mario Bros. U	PSPACE-hard	Thm. 11	
2015	Super Mario Maker (<i>all four styles</i>)	PSPACE-hard	Thm. 5	
2016	Super Mario Run	NP-hard	Thm. 14	
2019	Super Mario Maker 2 (<i>all five styles</i>)	PSPACE-hard	Thm. 7	
2023	Super Mario Bros. Wonder	PSPACE-hard	Thm. 8	

adjustments and checking because each game (except one) adds some mechanics while removing other mechanics from Super Mario Bros. In one case, Super Mario Bros. 2, the open-close door we construct (in Section 6) is completely different and quite complicated. For other games, we build self-closing doors (in Section 4) or symmetric self-closing doors (in Section 5).

For two 2D Mario games, Super Mario Land and Super Mario Run, we have not yet succeeded in building any door gadget. But we can at least prove only NP-hardness of these games (in Section 7), following the SAT framework first used to prove Super Mario Bros. NP-hard [1].

Our PSPACE-hardness results leave open which Mario games are in PSPACE and which are harder. Specifically, membership in PSPACE would hold if we polynomially bounded the maximum number of on-screen enemies or the maximum number of enemies at each screen position. This claim was made for Super Mario Bros. in [3]. But even Super Mario Bros. has an infinite source of enemies (if we remove the bound on enemies): Lakitu periodically spawns spinies. Many other Mario games have pipes that periodically spawn items or enemies. In some cases, these mechanics can be used to prove RE-completeness and thus undecidability; we explore this direction in a companion paper [4].

2 Generalized Mario

For each Mario game that we analyze, we make sure to only use blocks, enemies, objects, and other elements that appear in that game as released. We also make no changes to the physics or other interactions between the player and game elements.

However, actual Mario video games place several constraints on level sizes, number of onscreen enemies, and other parameters. For the purposes of analyzing complexity, we define generalized versions of each game with the following properties:

- No arbitrary limits on the level width, level height, and numbers of objects and events.
- Exponentially long time limits, or no time limit whatsoever (as in Super Mario Bros. 2 and Super Mario Bros. Wonder).

- Arbitrarily large screen size, as large as the entire level. One exception is Super Mario Bros. 2, which remembers necessary offscreen state, so we do not generalize in this case; see Section 6.

For simplicity, we use the original name of each game to refer to the generalized version. In all of these games, we restrict to a single player using a single input device unless otherwise stated.

2.1 Forbidding Powerups

A key defining feature of Mario games is powerups, such as the mushroom which makes Mario grow in size and allows him to take damage once without dying. Powerups can break many of the gadgets presented in this paper. For this reason, we want to assume that Mario comes into each level without a powerup and never collects one, unless we explicitly say otherwise. One way of doing this is to simply build a Mario game which features no powerups. However, in the interest of only caring about solvability of a single level, we should assume that Mario might be able to come into the level with a powerup. We can handle this by starting each level with a powerup and forcing Mario to take damage, e.g., by having to walk over a long row of spikes/munchers or through an enemy. Henceforth, in all of our gadgets, we will assume Mario begins in the non-powered state.

2.2 Playtesting

The majority of the gadgets featured here have been tested in the physics of the original games, by modifying those games with community-built level editors (see the Acknowledgments for details). You can watch videos of the gadgets in action, under both correct use and attempted misuse, on YouTube.³ To try these levels out yourself, you can download playable level files from GitHub.⁴ Given game hardware constraints and software limits in the released versions of these games, the gadgets may be modified slightly from what we present in the paper.

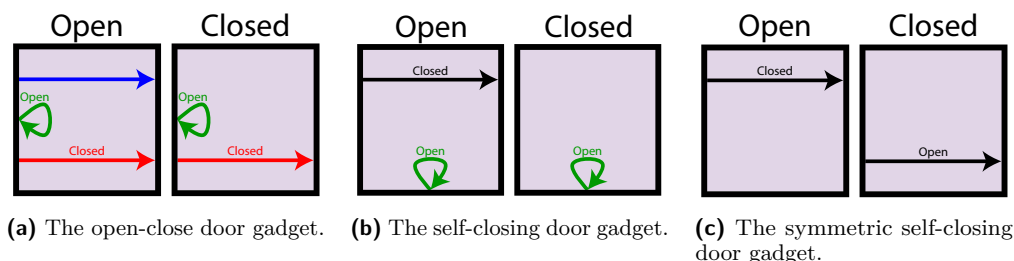
2.3 Reachability with Doors

In this paper, we reduce from a known PSPACE-complete problem called “reachability with planar door gadgets” [2], which we now describe.

The specific door gadgets used in this paper are the open-close door, self-closing door, and symmetric self-closing door, as shown in Figure 1. Each of these door gadgets has two states: open and closed. The (optional-open) *open-close door* consists of a traversal path (blue), a close path (red), and an optional open path (green). Traversing the close path forces the door into the closed state. Traversing the open path puts the door into the open state, but as the entrance and exit location of the open path are the same, the player can freely decide whether to open the door or not. The traversal path can be traversed by the player only when the door is in the open state. The (optional-open) *self-closing door* consists of a traversal path, which forces the door into the closed state when traversed, and an optional open path which opens the door. A *symmetric self-closing door* consists of two traversal paths. Traversing the top path is possible only when the door is open, and it forces the door to become closed; while traversing the bottom path is possible only when the door is closed, and it forces the door to open.

³ <https://www.youtube.com/playlist?list=PLCZQ5yzonfsaxrs9jZ41pgMvK4nRHSTXh>

⁴ <https://github.com/65440-2023/mario-hardness-gadgets>



■ **Figure 1** State diagrams for the PSPACE-complete gadgets used in this paper. Each box denotes a state (labeled Open or Closed), and each arrow denotes a possible transition in that state, labeled with the new state that the gadget enters upon such traversal.

Now we are given a *system* of door gadgets, consisting of several instances of the same type of door gadget, an initial state for each gadget, and a graph defining connections between locations (entrances and exits) of the doors. In a *planar* system, these connections do not cross each other or the door gadgets themselves. The *reachability* problem asks, given a system and two locations, whether it is possible for the player to start at the first location and reach the second location, by a sequence of traversals of door gadgets and connection edges. In *planar reachability*, we restrict to planar systems of gadgets.

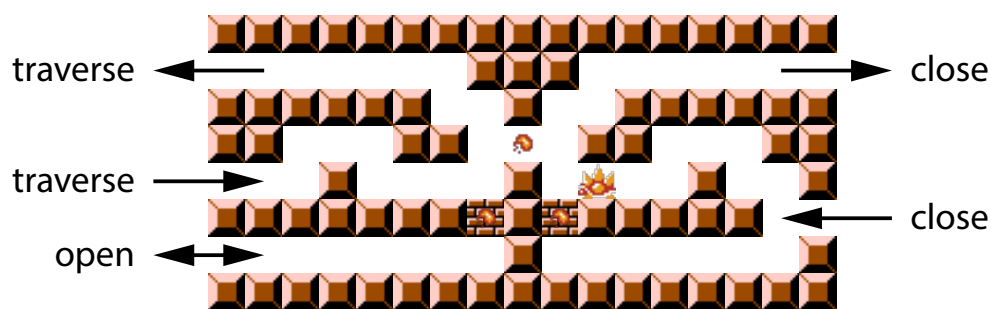
For all three types of door gadgets described above, planar reachability is PSPACE-complete [2]. (That paper defines one open-close door gadget which it does not prove PSPACE-complete, but it has a non-optional open path.) If we can build any door gadget in a Super Mario platform game, then we can instantiate this gadget multiple times and connect them together via tunnels made of blocks in such a way that the only way for Mario to reach the flagpole is via suitable traversal through these gadgets, and thereby prove PSPACE-hardness of the Mario game.

3 Standard Open-Close Doors

We will begin by examining games which can build doors with a similar structure to that featured in [3]. All of these doors are open-close doors. These doors have traverse and open on their left side, and close on the right. An enemy is hit from below while on a certain type of block such as a brick block to cause it bounce up and move between the two sides of the door. While on the left, the enemy blocks access to the traverse tunnel, and while on the right, the enemy blocks access to the close tunnel. There is some object which prevents Mario from crossing between the traverse and close tunnels but which allows for the enemy to pass through. In most of these games, the enemy in question becomes stunned for a constant amount of time when hit from below, and if the player is able to hit it and traverse quickly enough to pick it up, this could break the door. For this reason, we assume all tunnels in these gadgets are significantly longer than pictured, such that they take longer to traverse than it does for the enemy to wake up. This is indicated in our gadget figures with ellipses (\dots).




3.1 Super Mario Bros.: The Lost Levels

This game was originally released in Japan as スーパーマリオブラザーズ2 (Super Mario Bros. 2), but was renamed to Super Mario Bros.: The Lost Levels when it was finally released in North America, as part of the 1993 compilation Super Mario All-Stars. We follow the latter naming convention, to avoid confusion with the other game named (uniquely) Super Mario Bros. 2 (covered in Section 6).








■ **Figure 2** An open-close door in Super Mario Bros.: The Lost Levels.

► **Theorem 1.** *Super Mario Bros.: The Lost Levels is PSPACE-hard.*

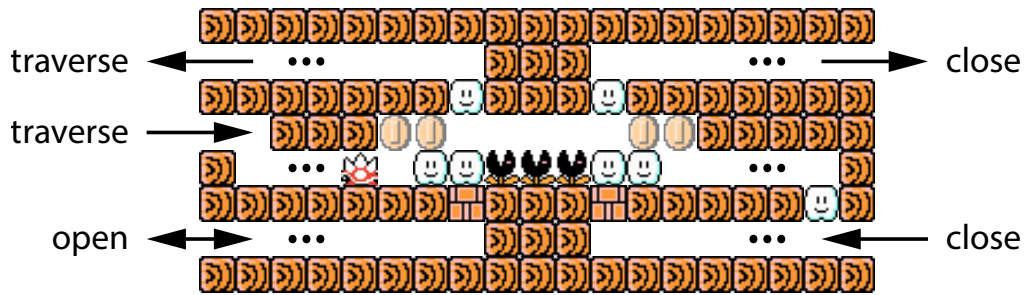
Proof. Super Mario Bros.: The Lost Levels is almost exactly the same as Super Mario Bros. but with different levels, graphics, and only a few minor tweaks. In particular, the physics of Mario’s movement is the same, and spinies and firebars behave in the same way. Hence, we can build the same door used in [3] to show that Super Mario Bros.: The Lost Levels is hard. For reference, Figure 2 shows such a gadget built in Super Mario Bros.: The Lost Levels. The idea is that the door is open whenever the spiny  is on the right side of the central fire  (as in the figure), in which case Mario can freely follow the traverse path. To traverse the close path, Mario must jump to hit the brick block  with careful timing so that the spiny above gets bumped to the other side of the gadget, closing the gadget and enabling Mario to reach the close exit. Similarly, visiting the open path allows Mario to hit the spiny to the other side, opening the gadget. ◀

3.2 Super Mario Bros. 3

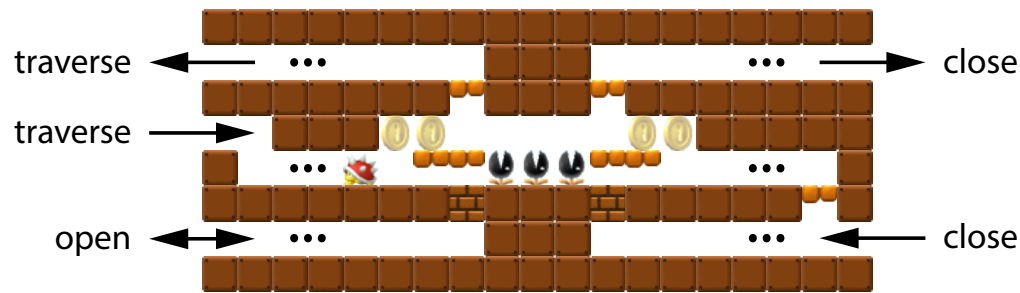
► **Theorem 2.** *Super Mario Bros. 3 is PSPACE-hard.*

Proof. Figure 3 shows our door gadget for Super Mario Bros. 3. Notably, we make use of *clouds* , which are a semi-solid platform. Objects (including both Mario and enemies) can pass through clouds from below and the sides, but not from above. When Mario hits a brick block  with a spiny on top of it, the spiny is bounced on top of the cloud. After a brief period of waking up, the spiny walks to the opposite side of the gadget. The black plants are munchers , an enemy which will damage Mario if he touches them. Mario cannot pass above the munchers in the center of the gadget without touching them and dying, but the spiny is invulnerable to their effects and can pass through without issue. We also make use of invisible coin blocks . When Mario hits these from below, they become solid blocks, but before that point, they are completely intangible, and the spiny can pass through them from the side with no issues. Their purpose is to restrict Mario from jumping over the spiny  and onto the cloud. Because of them, Mario can only pass through the traverse or close tunnel by jumping next to the munchers, and this can only happen if the spiny is on the opposite side of the gadget.

One might worry that Mario can hit the coin blocks on traversal, permanently breaking the gadget. This is true. If Mario hits a coin block, the gadget will enter a broken state where it is stuck open or closed, depending on the location of the spiny. However, this is not an issue, as doing so as it can only make the reachability problem more restrictive. If Mario hits a coin block while traversing on the left side of the gadget, the spiny will never fall down



■ **Figure 3** An open-close door in Super Mario Bros. 3.



■ **Figure 4** An open-close door in New Super Mario Bros. Wii.

the left side, and the door will be permanently open. This means that if Mario attempts to close the door, he will become stuck and forced to backtrack or die. A similar argument shows that it is never in Mario’s interest to force the door into a stuck closed state. ◀

3.3 New Super Mario Bros. Series

► **Corollary 3.** *New Super Mario Bros., New Super Mario Bros. Wii, New Super Mario Bros. 2, and New Super Mario Bros. U are all PSPACE-hard.*

Proof. The door from Theorem 3 works in all of the New Super Mario Bros. games for the same reasons. In place of clouds, we use a different semisolid platform 🍄, and some of the New Super Mario Bros. games feature spikes 📍 instead of munchers 🐉, but the objects behind these graphical changes function in the same way. For reference, Figure 4 shows a door built in New Super Mario Bros. Wii. ◀

We will give another proof of this result in Section 5, which uses different mechanics and is furthermore robust against the mechanics of Wii U gamepads.

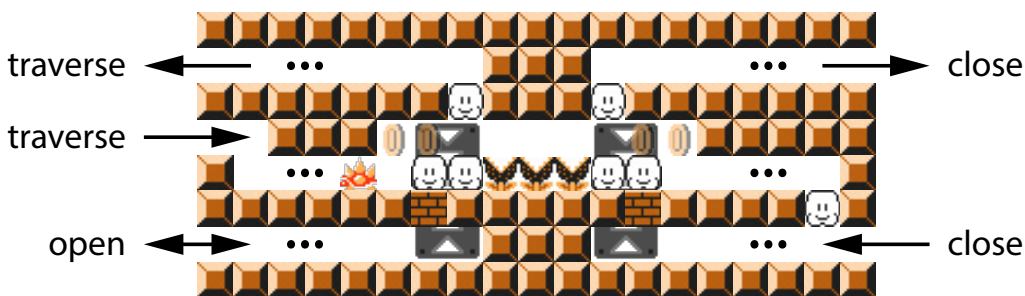
3.4 Super Mario World

► **Theorem 4.** *Super Mario World is PSPACE-hard.*


Proof. In Super Mario World, we cannot use spinies because they die instantly upon being hit from below. Instead, we make use of the goomba 🐉 enemy, which behaves differently in Super Mario World in that it becomes stunned instead of killed when hit. Figure 5 shows the resulting door gadget. Unlike with spinies, Mario can safely jump on goombas. To ensure Mario does not jump over the goomba or step on it to stun it and pass through a blocked tunnel, we place munchers 🐉 at the top of the traversal tunnels. If Mario would jump on



■ **Figure 5** An open-close door in Super Mario World.





■ **Figure 6** An open-close door in Super Mario Maker.

the goomba, it will bounce him into the munchers, killing him instantly. The only valid way to traverse the gadget is to first open it by hitting the on/off block  from below when a goomba is on it. Doing so will cause the goomba to bounce onto the munchers in the center of the gadget and walk across to the other side, allowing safe traversal. The close traversal works analogously. As a remark, the on/off switches contain extra functionality which we completely ignore here; we only care about them in terms of their ability to bounce goombas when hit from below. ◀

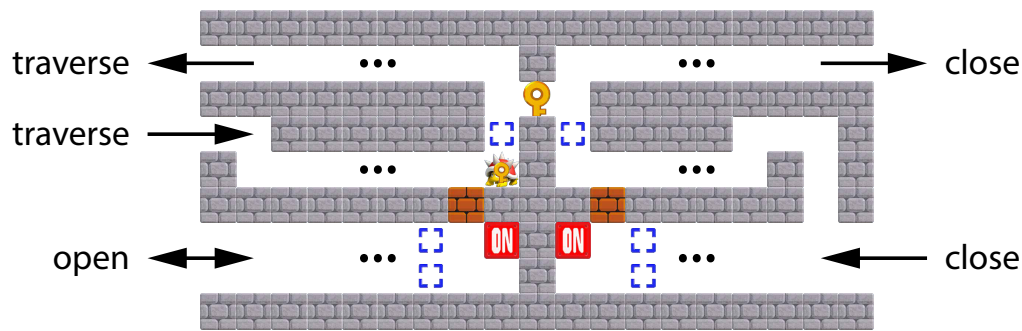
3.5 Super Mario Maker

► **Theorem 5.** *All styles of Super Mario Maker are PSPACE-hard.*

Proof. Our door gadget for Super Mario Maker is almost identical to the door gadget for Super Mario Bros. 3. The main difference is that we add *one-ways* , which only allow one-directional traversal, immediately below and above the brick blocks . This is because, in the Super Mario World style, the blocks corresponding to brick blocks temporarily become intangible when hit, allowing a spiny or Mario to pass through. The one-ways ensure that Mario cannot go up and the spinies cannot go down through these blocks, but Mario maintains the ability to bounce the spiny from below. All features present in this gadget exist in all four styles of Super Mario Maker, so this gadget works in all of them. Figure 6 shows a construction of the gadget. ◀

► **Corollary 6.** *The four 2D styles Super Mario Maker 2 are PSPACE-hard.*

Proof. Each of the four 2D styles of Super Mario Maker 2 features all elements of the door gadget pictured in Figure 6, behaving in the same way, so we use the same gadget gadget from Super Mario Maker. As a small detail, if we choose, we can simplify the construction



■ **Figure 7** An open-close door in the Super Mario 3D World style of Super Mario Maker 2.

involving 2 one-ways and a brick block by using the on/off switches present in Super Mario Maker 2 since they cannot be passed through in any game style and bounce spinies in the same way (these are the same type of blocks we used in the reduction to show that Super Mario World is PSPACE-hard). ◀

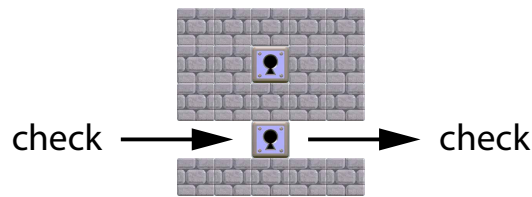
► **Theorem 7.** *All styles of Super Mario Maker 2 are PSPACE-hard.*

Proof. Corollary 6 proves hardness for all 2D styles. There is one remaining style present in Super Mario Maker 2, the Super Mario 3D World style. This style works very differently from the other four styles, so we construct a different door gadget, shown in Figure 7. The issue in the Super Mario 3D World style is that the spiny 🐉 changes direction when it wakes up and is unable to cross the gadget on its own. Fortunately, there are on/off switches **ON** which toggle the state of the blue squares **□** pictured. When the switch is on, as pictured, the blue squares are empty. When the switch is off, they are replaced by solid blue blocks. To cause the spiny to change sides, the player hits the brown brick block **■**, bounding the spiny into the center of the gadget. The player can then toggle the state of the switched blocks to allow the spiny to walk across to the other side of the gadget. Then, toggling the switch again will drop the spiny into the desired side. Hitting the switch toggles the state of all blue blocks in the level, including those in other gadgets, but these do not interact with the other gadgets because the blue blocks are above the spinies, so only the door where the player hits the spiny is affected.

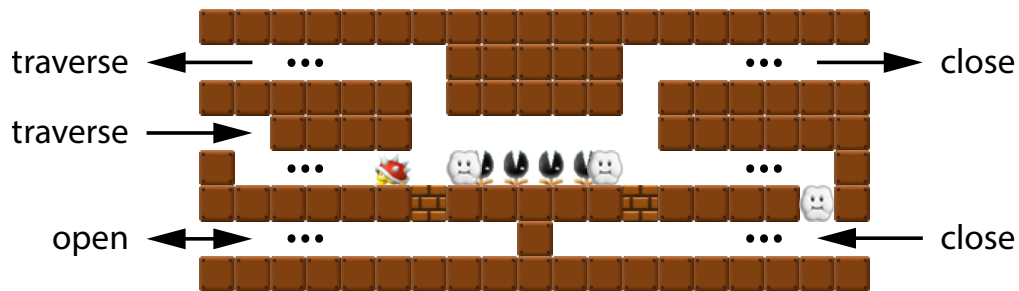
With the addition of the switched blocks, it is possible for Mario to hit the switch at such a time that the spiny is crushed by the blocks. If this happens, the door can no longer be closed, which is bad. In Super Mario Maker, enemies can be created holding keys 🗝️ which Mario will collect if the spiny dies. Mario will keep this key and any others he collects until they are used on locked doors or warp blocks **🗝️**. We give the spiny in our gadget a key, and at the very end of the level we force Mario through a “check” gadget which consists of a 1 tall path with a locked warp box (see Figure 8). If the player has a key, they get warped and become completely stuck. If they have no key, they pass through with no issue. Taking advantage of the check gadget, we use a second key to prevent the player from switching between the two sides of the gadget, taking the place of the munchers in the previous reduction. ◀

3.6 Super Mario Bros. Wonder

► **Theorem 8.** *Super Mario Bros. Wonder is PSPACE-hard.*



■ **Figure 8** The check gadget for the Super Mario 3D World style of Super Mario Maker 2.





■ **Figure 9** An open-close door in Super Mario Bros. Wonder.

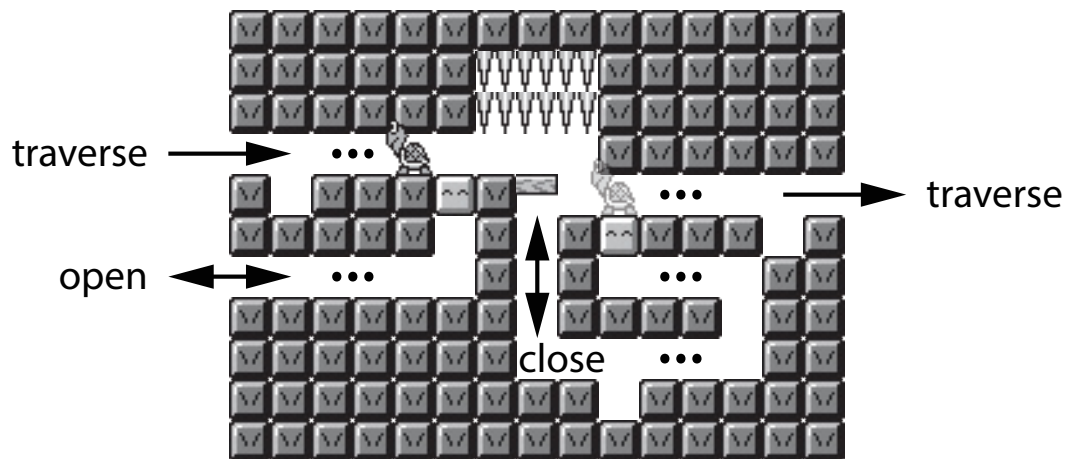
Proof. Our door gadget for Super Mario Bros. Wonder is similar to the door gadget for Super Mario Bros. 3, but we take advantage of the fact that Super Mario Bros. Wonder can have non-grid-aligned objects to simplify the gadget by removing the need for invisible coin blocks. Figure 9 shows a construction of the gadget. ◀

4 Self-Closing Doors

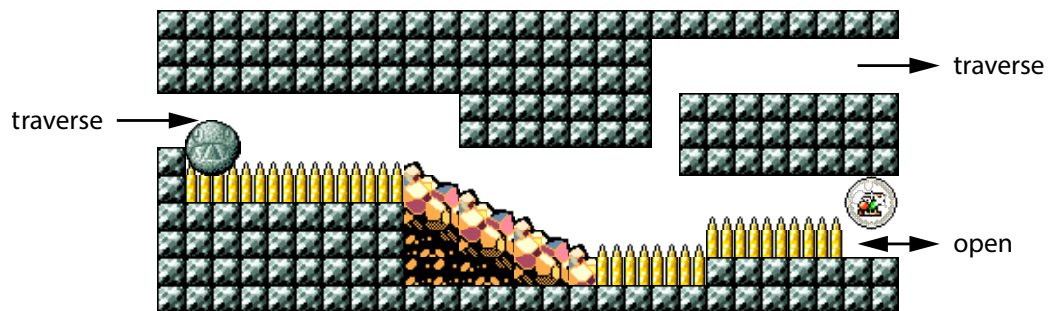
4.1 Super Mario Land 2

► **Theorem 9.** *Super Mario Land 2: 6 Golden Coins is PSPACE-hard*

Proof. To show Super Mario Land 2: 6 Golden Coins is PSPACE-hard, we reduce from planar reachability with self-closing doors [2], using the self-closing door gadget shown in Figure 10. Super Mario Land 2 does not have a spiny enemy, so we instead use a koopa . In this game, koopas do not walk off of ledges (like red koopas in other games), which complicates matters. When the door is closed (the koopas is in the top left, pictured solid in Figure 10), it blocks the traversal path because Mario is too tall to jump past it in the 1-block corridor. To open the door, Mario enters the open tunnel, and with a well-timed hit, it is possible to bounce the koopa into the right of the gadget (as pictured partially transparent in Figure 10). This opens up the beginning of the traversal tunnel, but blocks the right half. To continue with traversal, Mario must head down the vertical tunnel, below the semisolid , and reach the other light gray block to bounce the koopa back to the left. Mario can then backtrack and finish traversal. However, when Mario bounces the koopa to the left, it is possible for him to instead time the hit such that the koopa falls down the same tunnel Mario entered. If this happens, the koopa will block Mario's path out, which prevents Mario from finishing the traversal. Therefore, an optimal player must choose not to do this, and we can safely assume it does not happen. ◀






■ **Figure 10** A self-closing door in Super Mario Land 2.



■ **Figure 11** A self-closing door in Super Mario World 2.

4.2 Super Mario World 2: Yoshi's Island

► **Theorem 10.** *Super Mario World 2: Yoshi's Island is PSPACE-hard*

Proof. To show PSPACE-hardness for Super Mario World 2: Yoshi's Island, we reduce from planar reachability with self-closing doors [2], using the self-closing door gadget shown in Figure 11. To traverse, the player must ride the chomp rock , a spherical enemy which rolls when stood on, to the right side of the gadget. Attempts to cross without using the chomp rock will fail as the player does not have enough height to reach the traversal exit without the chomp rock and cannot stand on spikes  without dying. The chomp rock rolls down a slope and loses momentum upon reaching the bottom due to partially hitting the spikes. The player can then use it to traverse, but is unable to push the chomp rock back up the slope at the same time. To open the gadget, the player gains access to a reusable helicopter powerup . This powerup briefly turns Yoshi into a helicopter, allowing the player to safely push the the chomp rock up the slope and reset the door while hovering safely above the spikes. When the timer on the helicopter powerup runs out, Yoshi is instantly transported back to the open tunnel. Because of the powerup timer, by making all tunnels between gadgets sufficiently long, we can ensure that the player cannot reach and open any gadgets other than the intended one. ◀

5 Event-Based Symmetric Self-Closing Doors

Corollary 3 already showed that all four New Super Mario Bros. games are PSPACE-hard. But these games also implement a system called *events*, which allow the player to interact with blocks via switches and event controllers. In this section, we develop event-based symmetric self-closing door gadget, for two main reasons:

1. These doors are incredibly simple and small compared to the previous ones.
2. Unlike enemies, events persist regardless of the location of the screen. This means that we do not need to generalize screen size, which shows hardness for something that is much closer to the original game. This is also of some importance for New Super Mario Bros. U, as discussed in Section 5.2.

That being said, there are some drawbacks to using events. For one, they work behind the scenes and their behavior is not transparent to the player. It could be seen as not in the spirit of the game to build doors out of objects that are not true visible game elements. Furthermore, while these games do not impose arbitrary limits on the number of enemies (outside of memory constraints), they do impose a limit of 255 events based on the encoding of object data in binary. To allow for arbitrary events, we would have to generalize to a system which allows for more events. For these reasons, this door does not completely supplant the door of Corollary 3.

5.1 Event Mechanics

The relevant event mechanics used in this paper are as follows:

- There are numbered events, each of which can be either on or off
- There are numbered locations, which are sections of the level bounded by some rectangle
- Location controllers toggle an event based on the status of enemies or players in a given location
- There are blocks which appear or disappear according to the status of an event

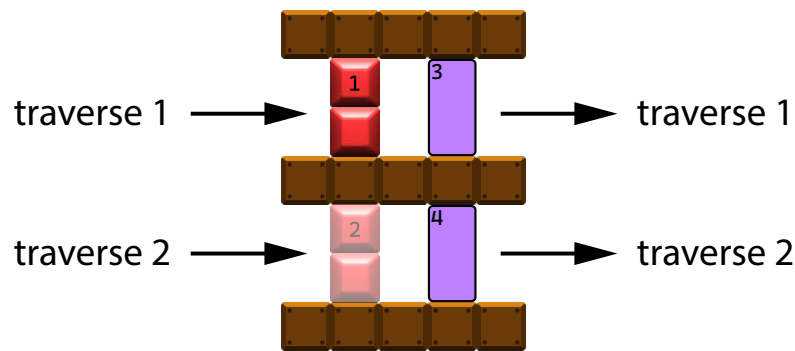
5.2 New Super Mario Bros. U with Gamepad

In New Super Mario Bros. U, the Wii U gamepad serves a special purpose. If the player is playing on a non-gamepad controller, they (or a friend) can use the gamepad in “boost mode” to help Mario. The player with the gamepad can create platforms on-screen to help the player cross dangerous areas, and if Mario steps on 10 of these, the gamepad player gains access to a boost star which allows them to kill enemies by tapping on them. It is very reasonable to disallow boost mode in our generalized New Super Mario Bros. U as it somewhat breaks the restriction of only one player, and this must be done for the door in Figure 4 to work properly, but with events, we can allow for the gamepad as the gamepad player has no control over any event behavior.

5.3 Self-Closing Door

► **Theorem 11.** *New Super Mario Bros., New Super Mario Bros. Wii, New Super Mario Bros. 2, and New Super Mario Bros. U are all PSPACE-hard, even when restricted to just blocks and events.*

Proof. The gadget pictured in Figure 12 is an event-based symmetric self-closing door. It uses one event. The blocks labeled 1 are on if and only if the event is on, and the blocks labeled 2 are on if and only if the event is off. When the player enters location 3, the event



■ **Figure 12** A symmetric self-closing door in New Super Mario Bros. U.








turns on, and when the player enters location 4, the event turns off. In this way, traversal through the top path is only possible when the event is off, and doing so turns the event on, and traversal through the bottom path is only possible when the event is on, and doing so turns the event off. The door in Figure 12 is pictured in New Super Mario Bros. U, but the same gadget works in all four New Super Mario Bros. games. ◀

6 Super Mario Bros. 2 Open-Close Door

In this section, we prove PSPACE-hardness of Super Mario Bros 2 (as named in North America). This reduction consists of our most complicated door gadget by far.

6.1 Mechanics

Super Mario Bros 2 has significantly different mechanics from the rest of the games in this paper, and our reduction for it relies on many of these mechanics, so we take some extra time to describe the behavior of various objects. Refer to Figure 13 to see images of relevant objects.

- The blue blocks with an X  pictured in our gadgets are semi-solids or diodes. They can be passed through from below, but not from above.
- The spikes  will kill the player if they land on them from above.
- Mario cannot naturally pass through a 1 block tall tunnel. However, by crouching, holding in the direction of desired movement, and repeatedly jumping, Mario can make very slow progress through the tunnel. This mechanic is used to pass through the tunnel at the bottom right of the gadget.
- The mushrooms  are objects which the player can pick up and place down. When placed, they fall and then snap to the grid in the level. When the player is holding any object, they cannot hold any other object or enemy. They also cannot pass through 1 block tall tunnels.
- The Birdo enemy , pictured in bottom right of the gadget will shoot eggs  when on screen. Eggs move across the screen horizontally at a constant speed until they hit a solid object. The player can jump off of them and can also pick them up and throw them to kill enemies.
- There are two types of shy guy enemies. They walk back and forth and can be picked up and thrown. When thrown, they return to walking. The red shy guy  will walk off of ledges (analogous to green koopas in other Mario games), and the pink shy guy  will turn when it reaches a ledge (analogous to red koopas in other Mario games). Mario can also stand on shy guys and ride them across spikes.

- By crouching for a brief period of time to charge his jump, Mario can perform a super jump, which is an especially tall jump.
- The game features 4 playable characters: Mario, Luigi, Toad, and Peach. The relevant differences for this gadget are that Luigi jumps higher than the others and that Peach can float for a brief period of time. Our gadget is robust to use of all four characters.
- The game remembers the position of mushrooms that are moved when they become off screen, so we do not need to generalize screen size. In fact, we will take advantage of screen size constraints.

6.2 Glitches

Super Mario Bros. 2 has many glitches and unintended behavior. Several of these are documented in [5]. As far as we are aware, there are two which are relevant for our gadget.

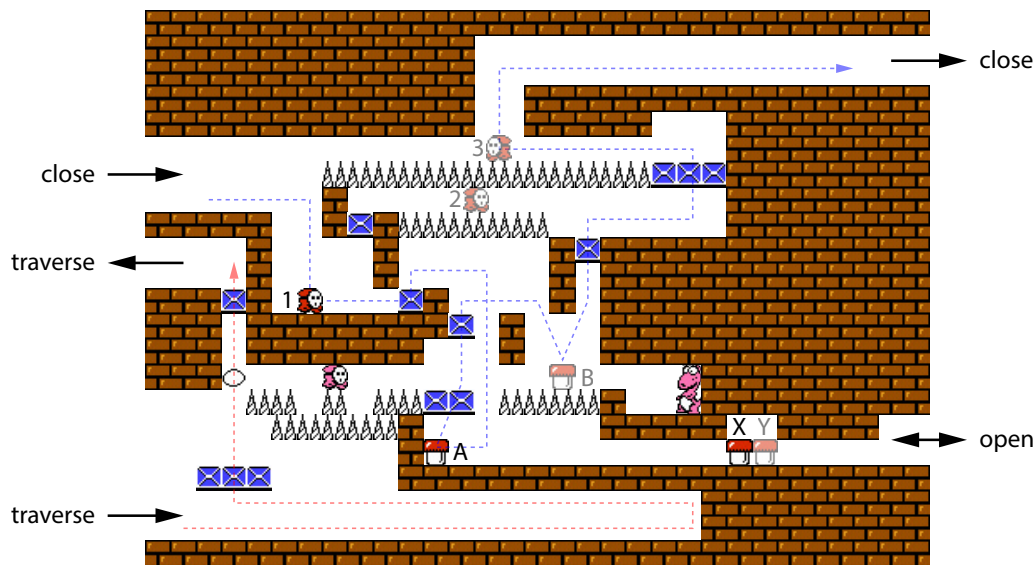
- If the player jumps and makes contact with the corner of an enemy, they can perform a second jump midair, resulting in a much higher jump than should be allowed. We avoid abuse of this by not giving the player opportunities to jump off of the corners of enemies where important.
- If the player is crouching on top of an enemy which walks into a one block tall tunnel, the player will not collide with the wall, despite doing so visually. This does not happen if the player is holding an item. If the roof of the tunnel is only one block thick, the player can then jump through the ceiling. We make tunnel ceilings two blocks thick and make use of shy guys to prevent abuse of this.

6.3 Open-Close Door

► **Theorem 12.** *Super Mario Bros. 2 is PSPACE-hard.*

Proof. We reduce from planar reachability with open-close doors [2]. A door is pictured in Figure 13. The state of the gadget is determined by a mushroom object, which the player can pick up and place in different locations. Specifically, the mushroom in position A, representing the open state, can instead be in position B, representing the closed state. When the door is open, the player can run through the bottom tunnel until Birdo is onscreen. Birdo will then shoot an egg to the left which the player can jump on to reach the exit of the traverse tunnel. This path is pictured in red. When the door is closed, the mushroom is in position B, and the egg's path is blocked, preventing traversal. Even Luigi with a super jump is unable to reach the platform without the egg.

Closing the door begins with the player throwing the shy guy in position 1 onto the spikes above, as shown in position 2. The shy guy walks to the right for use later. The player cannot cross this same gap because of the spikes. The glitch which allows for the player to ride an enemy across does not apply because the ceiling prevents Mario from getting on the enemy in the first place. Then, the player proceeds to the right and moves the mushroom from position A to B. Once the mushroom is in position B, they can use it to super jump up to the right. If they do not place the mushroom, they will be unable to exit, as a regular jump off of an egg does not give enough height, even for Luigi, to exit, and it takes too long to charge a super jump for the player to perform one off of the egg before it moves too far to the left. Hence, for the player to continue through the close tunnel, they must place the mushroom in location B, blocking the path of the eggs, and closing the traverse tunnel. Finally, the player can move the shy guy that was thrown to position 2 to the row of spikes above, and jump off of it in position 3 to cross an otherwise impassable spike tunnel. The shy guy continues walking to the left and falls back down to position 1 in preparation for the



■ **Figure 13** An open-close door in Super Mario Bros. 2.

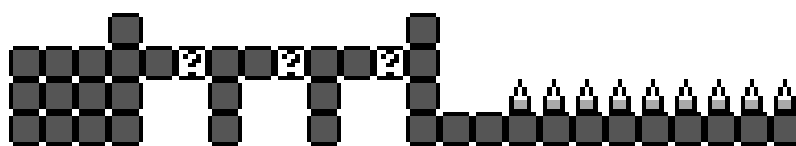
next use. An outline of this path is pictured in blue. Note that the player can choose not to jump off the shy guy and instead ride it back to the entrance of the close gadget, but this does not accomplish anything productive as they are simply forced to close the gadget again, so we can assume this does not happen.

Opening the door begins on the bottom right of the gadget with a 1-toggle sub-gadget. The mushroom can be placed in either position X or Y, but can only be picked up from the top, not the side. The effect of this is that the 1-toggle can only be traversed from the right if the mushroom is in position X and from the left if in position Y. This prevents the player from exiting through the open tunnel when closing the door. For the player to traverse from right to left in the open path, the mushroom is forced into position Y. Then, the player can (optionally) move the other mushroom from position B to position A, opening the door. They cannot exit through the close tunnel as they do not have the shy guy required to cross the spikes, and are forced to return through the 1-toggle, returning it to position X as they leave.

One might worry that the player can ride an egg through the gadget to the traverse path when opening or closing the door. This would be an issue, but there is a pink shy guy in the egg's path. The egg will pass through with no issue, but if a player is riding an egg, the taller hitbox of the shy guy will cause them to become stuck in the middle of a row of spikes, and they are unable to jump across the shy guy fast enough to continue riding the egg. The shy guy is placed far enough to the left such that Birdo is off screen and will not shoot any more eggs while the player is stranded. As pictured, the player could grab an egg and throw it at the shy guy before performing this shortcut, thus clearing the way, but making the tunnel sufficiently long and placing the shy guy far enough from the ends will prevent this. We simply show the smaller gadget here for simplicity. ◀

7 NP-Hardness




We initially set out to prove all 2D Mario Games PSPACE-complete. Sadly, we have not yet succeeded for two of these games we considered: Super Mario Land and Super Mario Run. Nonetheless, we can at least prove NP-hardness for both of these games. We will be using the framework developed in [1].



■ **Figure 14** A clause gadget in Super Mario Land.


► **Theorem 13.** *Super Mario Land is NP-hard.*

Proof. The framework for proving NP-hardness requires us to create the following gadgets:




- **Start and Finish.** We use the trivial start and end gadgets.
- **Variable.** We use the same variable gadget as used in [1].
- **Clause and Check.** We use the clause gadget pictured in Figure 14. Each of the 3 question blocks  in the left side of the gadget creates a mushroom which can be used to damage boost through the spikes  at the right of the gadget. However, if there is no mushroom, the player is unable to cross.
- **Crossover.** We use the crossover gadget pictured in Figure 15. This is a unidirectional crossover in which left to right traversal happens before top to bottom. When coming from the left, the player gains access to a star in a question block. This allows them to pass to the right over the row of spikes, which is much longer than pictured such that the player can only barely make it across with the star. They are unable to go up as they do not have a mushroom. To traverse from bottom to top, the player gains access to a mushroom in a question block which allows them to break the breakable blocks  and move up, where they then take forced damage and are returned to a powered-down state. However, they are unable to cross the spikes to the right because the maximum distance Mario can run on spikes with a star is much longer than the distance he damage boost with a mushroom, and the player is unable to reach the star block on the left side of the gadget. After the player completes a vertical traversal, there is leakage from horizontal to vertical, but, as discussed in [1], this is not an issue.

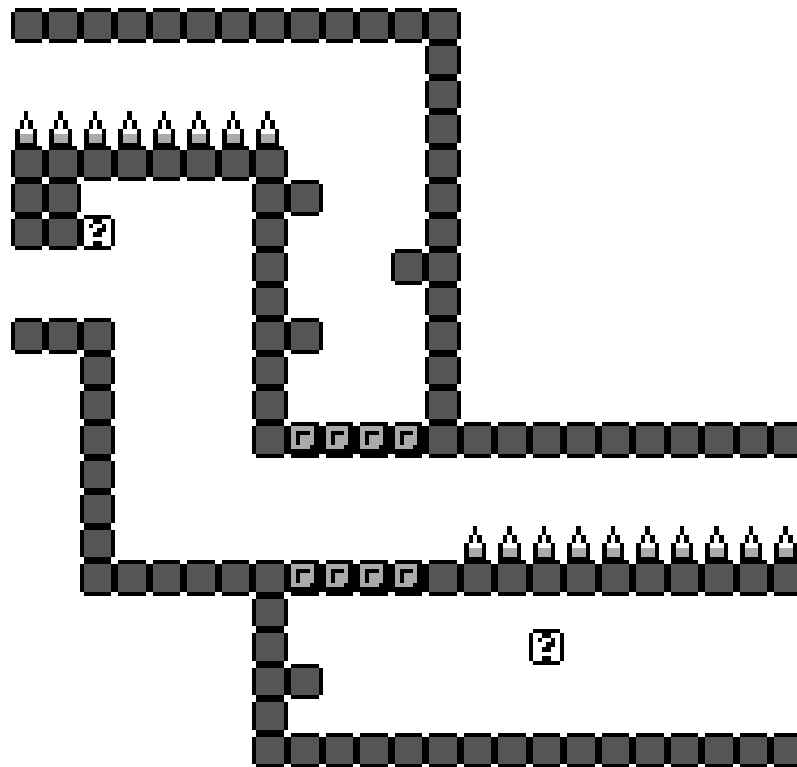
The combination of these gadgets is sufficient to show NP-hardness. ◀

► **Theorem 14.** *Super Mario Run is NP-hard.*


Proof. First, we note that Super Mario Run is unique compared with other Mario games in that the player can, under normal circumstances, only move right (and is pulled to the right by default). Fortunately, the game provides backflip  blocks which will cause the player to move left when jumping off of them, which allows for easy creation of wires which allow for moving left, as pictured in Figure 16a. Vertical wires are traversed via wall jumping or falling.

The framework for proving NP-hardness requires us to create the following gadgets:

- **Start and Finish.** We use the trivial start and end gadgets.
- **Variable.** We use the variable gadget pictured in Figure 16b. The forced rightward movement inherently creates a diode, ensuring that once the player chooses either the top or bottom path, they cannot choose the other variable.
- **Clause and Check.** We use the clause gadget pictured in Figure 16c. This works almost identically to the clause gadget in [1], but with grinders  in place of fire bars to enforce that the player must have a star.
- **Crossover.** We use the clause gadget pictured in Figure 16d. Vertical traversal is accomplished either via falling or wall jumping, and is bidirectional. The traversal from left to right requires hitting a switch  to briefly toggle the states of the red  and



■ **Figure 15** A crossover gadget in Super Mario Land.

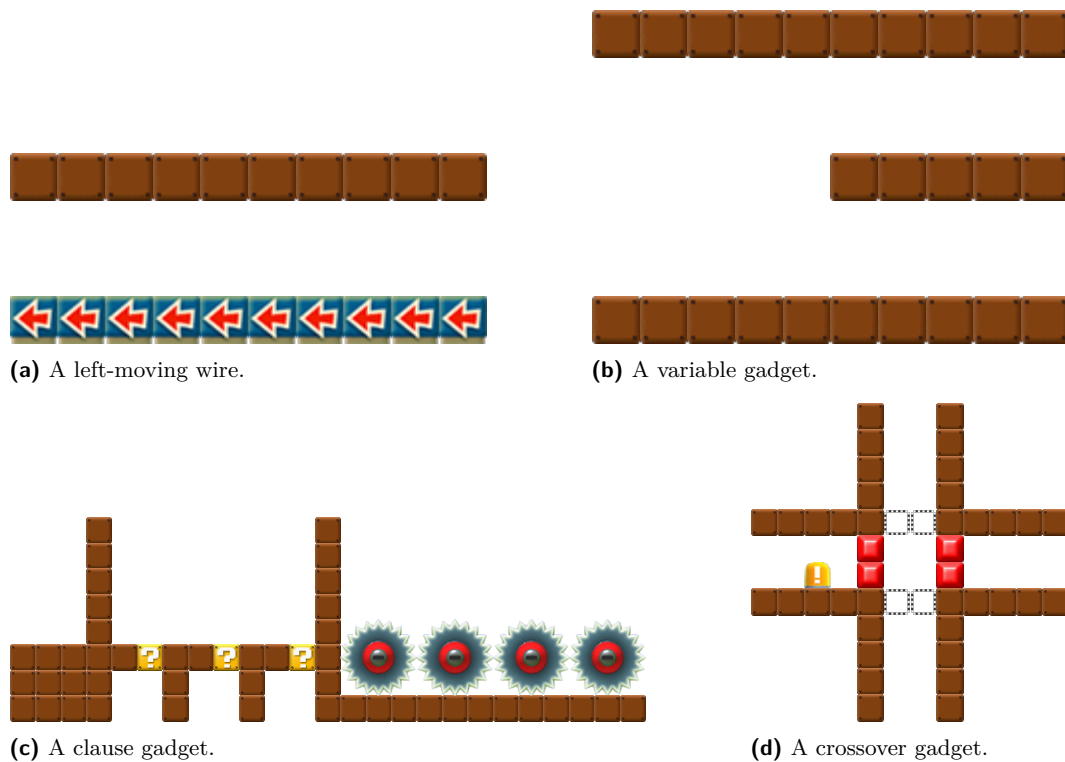
outlined  blocks, creating a tunnel from left to right. Because of the forced right condition, the player moves through the tunnel before the switch state can revert, and is unable to stall for leakage.

The combination of these gadgets is sufficient to show NP-hardness. ◀

8 Open Problems

As discussed in the previous section, while we initially set out to prove all 2D Mario Games PSPACE-complete, Super Mario Land and Super Mario Run have only been shown to be NP-hard. Super Mario Land features a very different set of objects from the other Mario games, and none that stand out to us as being able to toggle the state of a door an unbounded number of times. Super Mario Run poses a similar issue, with enemies behaving somewhat differently from the other games. It also suffers from the issue of access: unlike all other games considered in this paper, there are no known tools for creating custom Super Mario Run levels.

In this paper, we did not place any restrictions on generating new enemies, although none of our doors take advantage of this. If we do enforce that no new objects can be spawned, then we get containment in NPSpace as the size of the level cannot grow, and hence PSPACE by Savitch's Theorem. In this case, we have PSPACE-completeness for every PSPACE-hard game examined in this paper. However, most of these games have some mechanism by which the level size could grow without bound, with the removal of arbitrary object limits. For example, the original paper on Super Mario Bros. PSPACE-hardness, [3], makes claims that Super Mario Bros. is contained in NPSpace based on its levels only taking up a polynomial



■ **Figure 16** Super Mario Run Gadgets.

amount of space. However, this is not entirely true as Super Mario Bros. also has the Lakitu enemy which can create additional spinies. With the exponentially long timer, a player could stand near a Lakitu for exponentially long and generate exponentially many spinies which walk off-screen. These will require an exponential amount of memory, since we assume in SMB-General that the game remembers the positions of all off-screen enemies. With an exponentially large timer, all of these games are clearly contained in NEXPTIME, but this is a very arbitrary limit, and in the version without a timer, there are no obvious upper bounds other than RE for any of these games.⁵

References

- 1 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015. doi:10.1016/j.tcs.2015.02.037.
- 2 Hayashi Ani, Jeffrey Bosboom, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, pages 3:1–3:23, 2020. Full paper available as arXiv:2006.01256.
- 3 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 (FUN 2016)*, pages 13:1–13:14, La Maddalena, Italy, June 2016.

⁵ In fact, we explore RE-completeness for some of these games in [4].

- 4 MIT Hardness Group, Hayashi Ani, Erik D. Demaine, Holden Hall, Ricardo Ruiz, and Naveen Venkat. You can't solve these Super Mario Bros. levels: Undecidable Mario games. In *Proceedings of the 12th International Conference on Fun with Algorithms (FUN 2024)*, pages 29:1–29:20, La Maddalena, Italy, June 2024.
- 5 Super Mario Wiki. List of Super Mario Bros. 2 glitches. https://www.mariowiki.com/List_of_Super_Mario_Bros._2_glitches. Accessed December 2023.
- 6 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015. doi:10.1016/J.TCS.2015.01.055.

You Can't Solve These Super Mario Bros. Levels: Undecidable Mario Games

MIT Hardness Group¹

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Hayashi Ani ✉

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Erik D. Demaine ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Holden Hall ✉

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Ricardo Ruiz ✉

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Naveen Venkat ✉

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We prove RE-completeness (and thus undecidability) of several 2D games in the Super Mario Bros. platform video game series: the New Super Mario Bros. series (original, Wii, U, and 2), and both Super Mario Maker games in all five game styles (Super Mario Bros. 1 and 3, Super Mario World, New Super Mario Bros. U, and Super Mario 3D World). These results hold even when we restrict to constant-size levels and screens, but they do require generalizing to allow arbitrarily many enemies at each location and onscreen, as well as allowing for exponentially large (or no) timer.

In our Super Mario Maker reductions, we work within the standard screen size and use the property that the game engine remembers offscreen objects that are global because they are supported by “global ground”. To prove these Mario results, we build a new theory of counter gadgets in the motion-planning-through-gadgets framework, and provide a suite of simple gadgets for which reachability is RE-complete.

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

Keywords and phrases video games, computational complexity, undecidability

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.22

Related Version *Full Version*: <https://arxiv.org/abs/2405.10546>

Supplementary Material

Software (ROM Files): <https://github.com/65440-2023/mario-hardness-gadgets>

Acknowledgements This paper was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.5440) taught by Erik Demaine in Fall 2023. We thank the other participants of that class for helpful discussions and providing an inspiring atmosphere. Portions of this paper originally appeared in Ani’s master’s thesis [2]. Several community level editors and emulators were very helpful in building and testing counters:

- *Reggie!* by the NSMBW Community – <https://github.com/NSMBW-Community/Reggie-Updated>
- *CoinKiller* by Arisotura – <https://github.com/Arisotura/CoinKiller>
- *Miyamoto* by abood40091 – <https://github.com/abood40091/Miyamoto>

¹ Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as “MIT Hardness Group” (without “et al.”).



© MIT Hardness Group, Hayashi Ani, Erik D. Demaine, Holden Hall, Ricardo Ruiz, and Naveen Venkat;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 22; pp. 22:1–22:20

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- *Dolphin* by the Dolphin Emulator Project – <https://dolphin-emu.org/>
- *Cemu* by Team Cemu – <https://cemu.info/>
- *Citra* by Citra Emu – <https://github.com/citra-emu/citra>

1 Introduction

In 2014, Hamilton [10] proved that solving a (bounded-size) level in the 2008 video game *Braid* is **RE-complete** (the same difficulty as the halting problem, and thus undecidable), even without its famous time-travel mechanic. The reduction is from **counter machines** [15, 16, 13]: a finite-state machine with a constant number of natural-number counters equipped with increment, decrement, and jump-if-zero instructions. The central idea was to represent the value of each counter in a *Braid* level by the number of enemies occupying a particular location in the level, exploiting that this number can be arbitrarily large even in a bounded-size level. The same paper conjectured that many other video games were amenable to this approach.

In this paper, we prove that this approach extends to several 2D games in the Super Mario Bros. platform video game series,² when we generalize them to remove any limits on time, the total number of enemies, or the number of enemies at each location. Most of these games (even the original Super Mario Bros. from 1985) have mechanics for spawning arbitrarily many enemies over time, but it is difficult to find the right combination of mechanics to implement the specific functionality of an increment/decrement/jump-if-zero counter. Specifically, we show that the following games are RE-complete by building computer machines where the number of enemies in a particular location represents the value of each counter:

1. The *New Super Mario Bros.* series (Section 3): New Super Mario Bros., New Super Mario Bros. Wii, New Super Mario Bros. U, and New Super Mario Bros. 2. One reduction covers all four games, using their powerful “event” game mechanic, where Mario’s location can toggle the existence of blocks. We build an entire universal counter machine within a single screen of the the Wii version, so solving even a single-screen level is RE-complete.
2. The *Super Mario Maker* series (Section 4): Super Mario Maker 1 in all four game styles (Super Mario Bros. 1, Super Mario Bros. 3, Super Mario World, and New Super Mario Bros. U) and Super Mario Maker 2 in all five game styles (the same four, plus Super Mario 3D World).

All of these games are in RE: if a level is solvable, then there is a finite algorithm to solve them, by trying all possible sequences of inputs to the game, and simulating the result. Because any solvable level is (by definition) solvable in finite time, and the state in the game after any finite time is finite, this algorithm is finite. Thus, to prove RE-completeness, it remains to prove RE-hardness.

To simplify the process of proving such games RE-hard using counter machines, we develop in Section 2 a new theory of “counter gadgets” which shows that a *single* gadget diagram suffices to prove RE-hardness in a one-player game like Super Mario Bros. This framework builds on the ***motion-planning-through-gadgets framework*** developed in recent years [8, 3, 6, 5, 4, 5, 14, 11], starting with FUN 2018 [7]. In the single-player version of this framework, one agent (Mario) traverses an environment consisting of local “gadgets”, with the locations of the gadgets connected together in a graph (representing freely traversable connections). Each ***gadget*** has a finite set of states and a finite set of possible transitions,

² After all, “*Braid* is a postmodern Super Mario Bros.” [12].

where a transition $(q, a) \rightarrow (r, b)$ means that, when the gadget is in state q , the agent can enter at location a and leave at location b , while changing the state of the gadget to r . We generalize this framework to allow for *infinite-state* gadgets called **counter gadgets**, where the states are the natural numbers and the traversals are among common operations such as increment, decrement, traversable-if-zero, and traversable-if-nonzero. Specifically, we show that any one of the following counter gadgets suffices to prove RE-completeness of the **reachability** problem (can the agent get from one location to another?):

1. **Inc-Dec-JZ**: One traversal path that increments the counter value, another that decrements the counter value unless it is already zero, and a third that leads the agent to two different locations depending on whether the counter value is zero.
2. **Inc-JZDec**: One traversal path that increments the counter value, and another that leads the agent to two different locations depending on whether the counter value is zero, and if it is not, decrements the counter.
3. **Inc-DecNZ-PZ**: One traversal path that increments the counter value, another that decrements the counter value but which can be traversed only if the counter is nonzero, and a third that is traversable only if the counter value is zero.

Notably, this RE-hardness result holds even when the connections between gadgets form a planar graph, so there is no need for a crossover gadget.

4. **Inc $[a, b]$ -DecNZ $[c, d]$ -PZ**: A generalization of the previous gadget where, when the agent traverses an increment or decrement path, it gets to choose how much to increment or decrement, within a range of $[a, b]$ or $[c, d]$, respectively, where $a, c > 0$. This robustness to unknown gadget behavior is helpful for building gadgets in video games, which can require very careful timing and alignment to force a specific number of increments or decrements, but forcing at least one and at most some constant is relatively easy.

Specifically, we build an Inc $[1, 2]$ -DecNZ $[1, 2]$ -PZ gadget in the New Super Mario Bros. series, and we build three different Inc-DecNZ-PZ gadgets for different variants of the Super Mario Maker series. Some of our gadgets are available for download and play [17]. On the plus side, we need to build only one main gadget for each game, together with a crossover gadget in some cases (such as New Super Mario Bros.). On the negative side, as we will see, that one main gadget is quite complex.

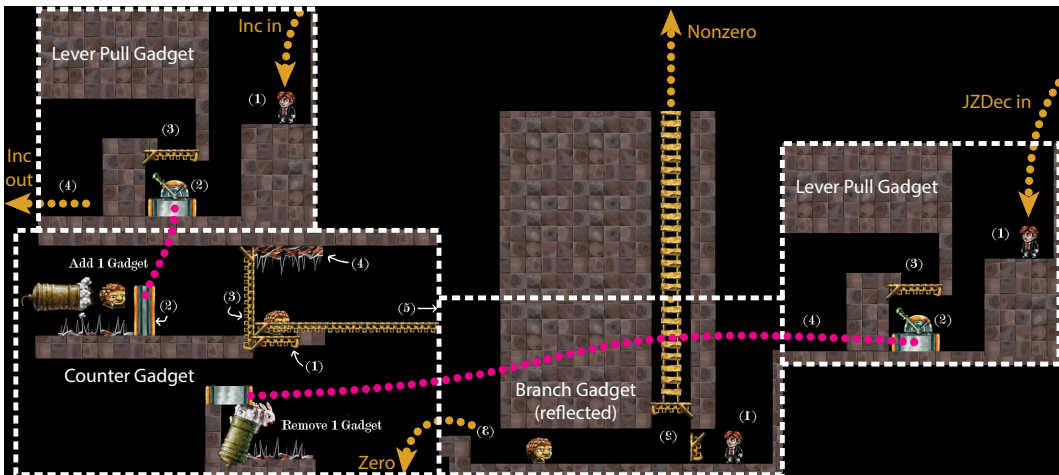
By contrast, the Braid proof [10] had six individual gadgets. We can instead combine three of them (Lever Pull, Counter, and Branch) in a straightforward way to build an Inc-JZDec gadget; see Figure 1. Then we only need a crossover gadget for the agent (Tim), which is one of the three crossover gadgets in [10]; the other two are no longer necessary.

Like Hamilton, we conjecture that our counter-gadget framework can be applied to prove RE-hardness of other video games as well. We discuss further possibilities for Super Mario Bros. games in Section 5.

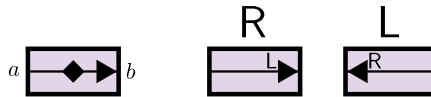
2 Counter Gadgets

In this paper, we reduce from “reachability with gadgets”, first explored in [7, 8]. We define a **gadget** $G = (Q, L, T)$ to consist of a set Q of **states** (not necessarily finite), a finite set L of **locations**, and a set $T \subseteq (Q \times L)^2$ of allowed **transitions** on pairs of locations and states, each written in the form $(q, a) \rightarrow (r, b)$ where $q, r \in Q$ and $a, b \in L$.

An example of a gadget is the **1-toggle**: it has a single path that the player can cross in only one direction, and every time they do, the allowed direction flips. Figure 2 gives a graphical representation. In this case, there are two locations $L = \{a, b\}$, two states $Q = \{R, L\}$, and $T = \{(0, a) \rightarrow (1, b), (1, b) \rightarrow (0, a)\}$.



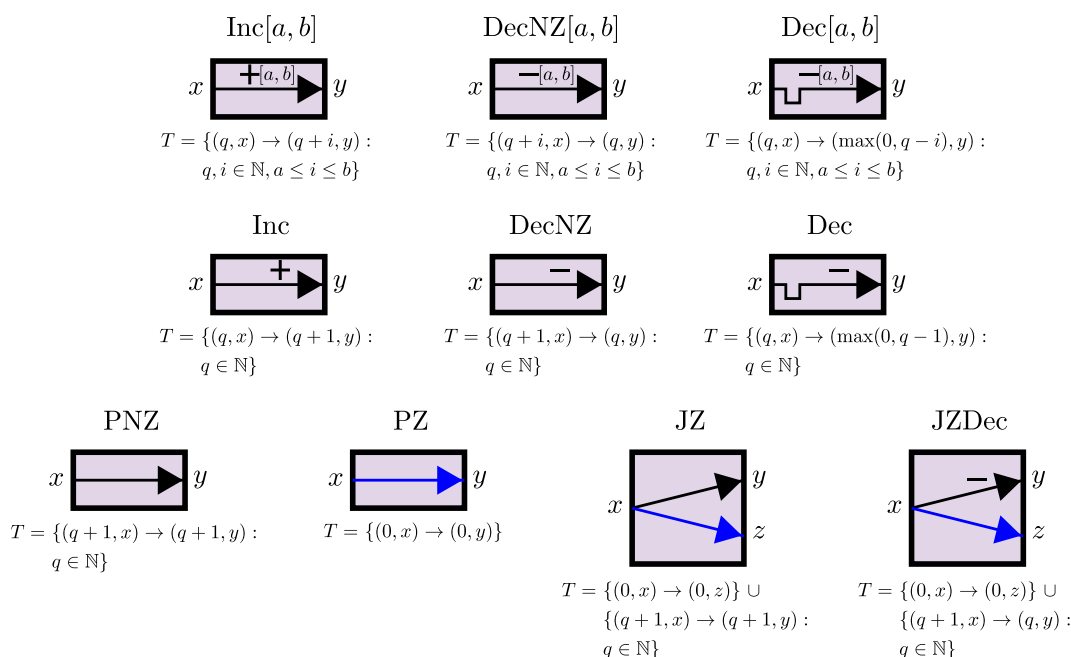
■ **Figure 1** An Inc-JZDec counter gadget in Braid, built from the Level Pull, Counter, and Branch gadgets of [10].



■ **Figure 2** A 1-toggle, along with its state diagram on the right. When the player goes from a to b , the arrow flips.

Here, we consider gadgets with an infinite number of states, namely, one for each natural number. We further restrict a *counter gadget* to consist of *counter components* (see Figure 3) that interact with each other when put in the same gadget:

- **Inc** $[a, b]$. This is a directed tunnel that is always traversable. When the player traverses it, they choose a natural number i such that $a \leq i \leq b$. The gadget's state increments by i .
- **DecNZ** $[a, b]$. This is a directed tunnel that is traversable if and only if the gadget's state is at least a . When the player traverses it, they choose a natural number i such that $a \leq i \leq \min(s, b)$, where s is the gadget's state. The gadget's state decrements by i .
- **Dec** $[a, b]$. This is like DecNZ $[a, b]$, except that it is always traversable, and if the gadget's state would become negative, it instead becomes 0.
- **Inc, DecNZ, Dec**. These are short for Inc $[1, 1]$, DecNZ $[1, 1]$, and Dec $[1, 1]$, respectively.
- **PZ**. This is a directed tunnel that is traversable if and only if the gadget's state is 0. It does not change the state.
- **PNZ**. This is a directed tunnel that is traversable if and only if the gadget's state is not 0. It does not change the state. This is only defined for convenience of defining the JZ switch below.
- **JZ**. This is a switch formed by putting a PZ tunnel and a PNZ tunnel in the same gadget and combining their entrances.
- **JZDec**. This is a switch formed by putting a PZ tunnel and a DecNZ tunnel in the same gadget and combining their entrances.



■ **Figure 3** Counter components that are allowed in counter gadgets, along with their sets T of allowed traversals.

2.1 Undecidable Gadgets via Counter Machines

A *system* of gadgets consists of instances of gadgets, an initial state for each gadget, and a graph connecting the gadgets' locations together. In the *reachability* problem, we are given a system of gadgets, a start location s , and a goal location t , and we want to know whether the player can get from s to t by making a sequence of gadget traversals and following edges of the connection graph.

With infinite-state gadgets, sometimes reachability is undecidable, since we can simulate counter machines. We use the fact that the counter-machine halting problem is RE-hard [15] as a starting point. First, we give a brief introduction to counter machines.

A *counter machine* is a set of counters and a sequence of instructions that run on those counters. The instructions are:

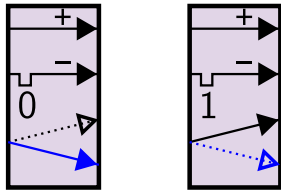
- **Inc(c):** Add 1 to counter c and move on to the next instruction
- **Dec(c):** Subtract 1 from counter c , leaving its value alone if it was 0, and move on to the next instruction.
- **JZ(c, i):** Check if counter c is 0. If so, jump to instruction i . Otherwise, move on to the next instruction.
- **Halt:** Stop the machine.

It is RE-hard to determine whether a counter machine reaches a **Halt** instruction, even with just two counters [15], [16, pp. 255–258].

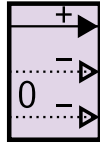
2.2 Inc-Dec-JZ is RE-complete

The Inc-Dec-JZ gadget (Figure 4) consists of, as the name indicates, an Inc tunnel, a Dec tunnel, and a JZ switch.

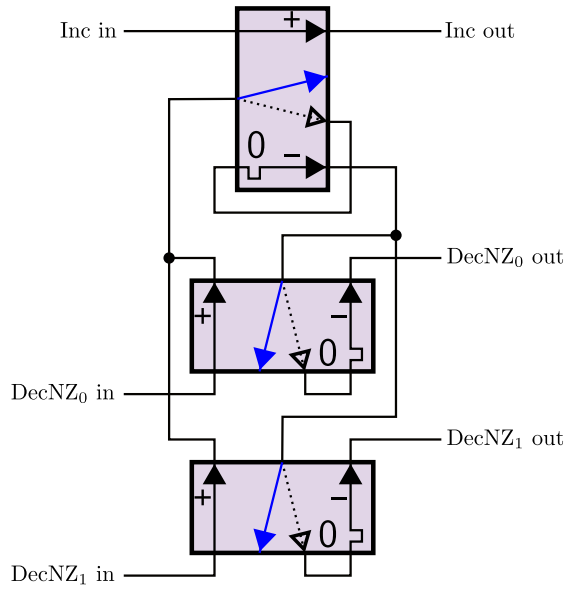
We prove that reachability with this gadget is NP-hard. The proof involves duplicating the Inc and Dec tunnels a bunch, then duplicating the JZ switch a bunch, then simulating a counter machine using multiple of the gadget, using each set of Inc, Dec, and JZ components



■ **Figure 4** The Inc-Dec-JZ gadget, shown in states 0 and 1.



■ **Figure 5** The Inc-DecNZ-DecNZ gadget, along with its simulation by Inc-Dec-JZ gadgets.



as an instruction. We will show RE-hardness again, but with a different method where we build a gadget made for flow control, and use multiple copies of that gadget along with unaltered Inc-Dec-JZ gadgets.

► **Theorem 1.** *Reachability with Inc-Dec-JZ is RE-complete.*

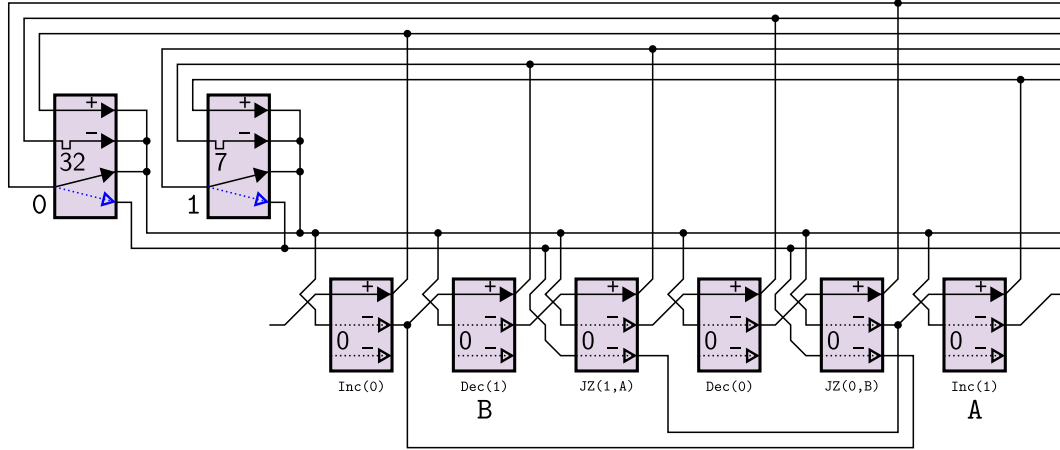
Proof. We show this by simulating a counter machine with $\text{Inc}(c)$, $\text{Dec}(c)$, and $\text{JZ}(c, i)$ instructions, which increment counter c , decrement counter c (saturated at 0), and jump to instruction i if $c = 0$, respectively.

First, we build an **Inc-DecNZ-DecNZ** gadget to help with flow control. We simulate one as shown in Figure 5. If the agent enters via Inc in , they increment the top gadget and leave. If the agent enters via $\text{DecNZ}_i \text{ in}$, they eventually get stuck if the top counter is 0. Otherwise, they increment the middle/bottom gadget, decrement the top gadget, and decrement the gadget that they incremented before, leaving via $\text{DecNZ}_i \text{ out}$.

To construct the counter machine, we use an Inc-Dec-JZ gadget c for each counter c , and an Inc-DecNZ-DecNZ gadget i for each instruction $i: \dots$. At a high level, we connect gadgets so that the agent flow works as follows:

- For an instruction $i: \text{Inc}(c)$, the agent increments instruction gadget i , increments the counter gadget c , finds the incremented instruction gadget i and decrements it, and then moves on to the next instruction gadget $i + 1$.
- For an instruction $i: \text{Dec}(c)$, the agent does the same as above, except that they decrement the counter gadget c .
- For an instruction $i: \text{JZ}(c, i')$, the agent increments instruction gadget i and goes to check whether counter gadget c is in state 0. If it is, they decrement gadget i and branch to instruction gadget i' . Otherwise, they decrement gadget i using the other decrement path and move on to the next instruction gadget $i + 1$.

Figure 6 shows an example.



■ **Figure 6** The Inc-Dec-JZ gadget simulating a counter machine.

More concretely, we connect gadgets in the following ways, where $a[b]$ denotes location b of gadget a , I denotes the counter machine program, $I[i]$ denotes the i th instruction, and C denotes the set of counter gadgets:

- $i[\text{Inc out}] \sim c[\text{Inc in}]$ for all i, c where $I[i] = \text{Inc}(c)$.
- $i[\text{Inc out}] \sim c[\text{Dec in}]$ for all i, c where $I[i] = \text{Dec}(c)$.
- $i[\text{Inc out}] \sim c[\text{JZ in}]$ for all i, c, i' where $I[i] = \text{JZ}(c, i')$.
- $c[\text{Inc out}] \sim i[\text{DecNZ}_0 \text{ in}]$ for all i, c where $0 \leq i < |I|$ and $c \in C$.
- $c[\text{Dec out}] \sim i[\text{DecNZ}_0 \text{ in}]$ for all i, c where $0 \leq i < |I|$ and $c \in C$.
- $c[\text{JZ out} (\neq 0)] \sim i[\text{DecNZ}_0 \text{ in}]$ for all i, c where $0 \leq i < |I|$ and $c \in C$.
- $c[\text{JZ out} (= 0)] \sim i[\text{DecNZ}_1 \text{ in}]$ for all i, c where $0 \leq i < |I|$ and $c \in C$.
- $i[\text{DecNZ}_0 \text{ out}] \sim (i + 1)[\text{Inc in}]$ for all i where $0 \leq i < |I| - 1$.
- $i[\text{DecNZ}_1 \text{ out}] \sim i'[\text{Inc in}]$ for all i, c, i' where $I[i] = \text{JZ}(c, i')$.

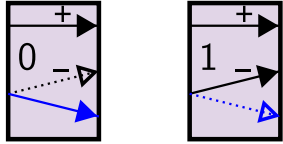
The agent starts just in front of the first instruction gadget, and any instruction gadget that corresponds to a `halt` instruction is replaced with a goal. Then the agent can win if and only if the counter machine halts, reducing the counter machine halting problem to reachability. ◀

2.3 Inc-JZDec is RE-complete

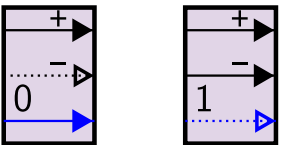
The Inc-JZDec gadget (Figure 7) replaces the Dec and JZ components with a single JZDec switch. We prove that reachability with this gadget is RE-hard by simulating the Inc-Dec-JZ gadget.

► **Theorem 2.** *Reachability with Inc-JZDec is RE-complete.*

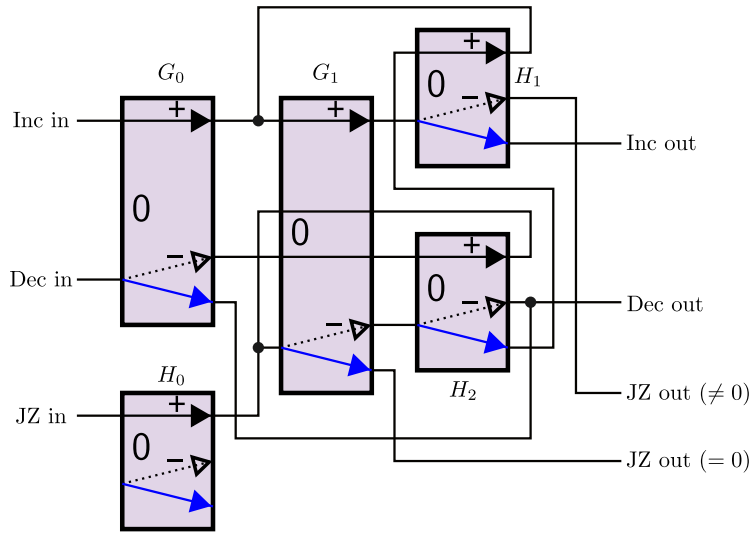
Proof. We reduce from reachability with Inc-Dec-JZ by simulating the Inc-Dec-JZ gadget, as shown in Figure 8. We use G_0 and G_1 to store the counter's value. If the agent enters via Inc in, they increment G_0 and G_1 and leave via Inc out. If the agent enters via Dec in, then if the counter is 0, they nearly immediately leave via Dec out. Otherwise, they decrement G_0 , increment H_2 , decrement G_1 , decrement H_2 , and leave via Dec out. Note that H_2 has no net change. If the agent enters via JZ in, if the counter is 0, they nearly immediately leave



■ **Figure 7** The Inc-JZDec gadget, shown in states 0 and 1.



■ **Figure 9** The Inc-DecNZ-PZ gadget, shown in states 0 and 1.



■ **Figure 8** Simulation of Inc-Dec-JZ with Inc-JZDec. Gadgets G_0 and G_1 store the counter's value, while H_0 , H_1 and H_2 are used for flow control.

via $JZ\ out = 0$. Otherwise, they decrement G_1 , increment H_1 , increment G_1 , decrement H_1 , and leave via $JZ\ out \neq 0$, leaving no net change (except in H_0 , but that gadget is just used as a diode). So this simulation works. ◀

2.4 Inc-DecNZ-PZ is RE-complete

The Inc-DecNZ-PZ gadget (Figure 9) replaces the JZDec switch with separate DecNZ and PZ tunnels. This gadget can easily simulate the Inc-JZDec gadget, by combining the entrances of DecNZ and PZ.

► **Corollary 3.** *Reachability with Inc-DecNZ-PZ is RE-complete.*

In addition, we have a planar result. In **planar reachability**, we restrict to **planar** systems of gadgets where the graph of connections between gadgets' locations do not cross gadgets or each other (except at common endpoints).

► **Theorem 4.** *Planar reachability with any planar system of Inc-DecNZ-PZ gadgets is RE-complete.*

Proof. Ani et al. [3, Theorems 3.1 and 3.2] show that a crossover can be built from a **symmetric self-closing door**: a gadget with two states $Q = \{1, 2\}$ and two possible traversal paths $L_1 \rightarrow R_1$ and $L_2 \rightarrow R_2$, where $L_1 \rightarrow R_1$ is possible only in state 1, $L_2 \rightarrow R_2$ is possible only in state 2, and every traversal switches the state. Figure 11 shows how to build a symmetric self-closing door from Inc-DecNZ (and thus Inc-DecNZ-PZ) in all cases. ◀

Thus, we do not need to build a crossover when reducing from reachability with the Inc-DecNZ-PZ gadget, even in a planar application.

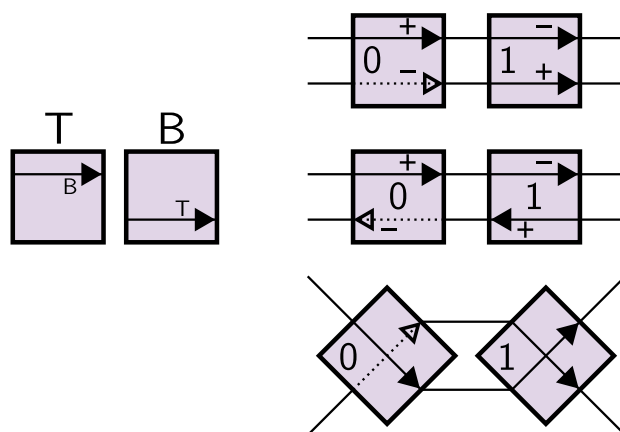


Figure 11 Left: State diagram of the symmetric self-closing door. Right: Simulation of a symmetric self-closing door, no matter the planar arrangement of tunnels in Inc-DecNZ.

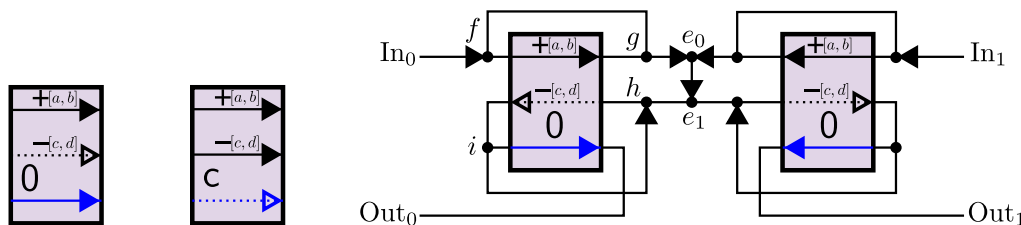


Figure 12 The Inc[a, b]-DecNZ[c, d]-PZ gadget, shown in states 0 and c .

Figure 13 An edge duplicator built with the Inc[a, b]-DecNZ[c, d]-PZ gadget. The player can go from e_0 to e_1 along two different paths without leaking between them.

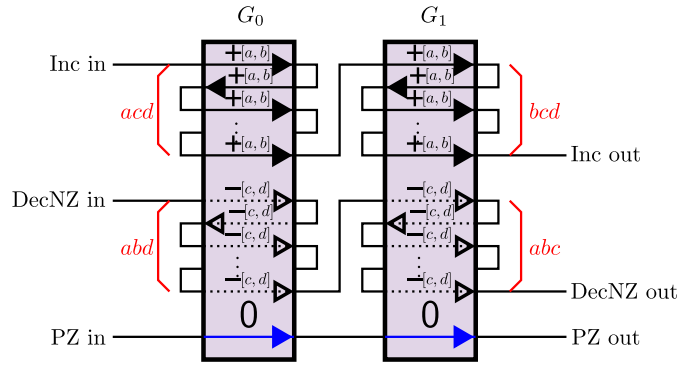
2.5 Inc[a, b]-DecNZ[c, d]-PZ, $a > 0, c > 0$ is RE-complete

The Inc[a, b]-DecNZ[c, d]-PZ gadget (Figure 12) replaces the Inc and DecNZ tunnels with Inc[a, b] and DecNZ[c, d] tunnels, respectively. Recall that an Inc[a, b] tunnel allows the player to choose an integer between a and b inclusive, and increment the gadget’s state by that amount. A DecNZ[c, d] tunnel allows the player to choose an integer between c and d inclusive and decrement the state by that integer, but only if the result is nonnegative.

► **Theorem 5.** *Reachability with Inc[a, b]-DecNZ[c, d]-PZ is RE-hard if $a > 0$ and $c > 0$.*

Proof. We simulate the Inc-DecNZ-PZ gadget. First, we build an *edge duplicator* (Figure 13), which allows us to duplicate Inc[a, b] and DecNZ[c, d] tunnels as many times as we want. In the edge duplicator shown, $e_0 \rightarrow e_1$ is being duplicated. If the player enters via In_0 , they go from f to g c times and go through e_0 to reach e_1 . Then since the gadget on the right does not allow passage, they must go from h to i until the path to Out_0 opens up (that is, a times). Then they leave, and the left gadget is reset. This works symmetrically for $In_1 \rightarrow Out_1$.

Then we use two gadgets with as many Inc[a, b] tunnels, as many DecNZ[c, d] tunnels, and as many PZ lines as we want to simulate the Inc-DecNZ-PZ gadget (Figure 14). In fact, we use acd Inc[a, b] tunnels and abd DecNZ[c, d] tunnels in G_0 , and bcd Inc[a, b] tunnels and abc DecNZ[c, d] tunnels in G_1 .



■ **Figure 14** Simulation of Inc-DecNZ-PZ with Inc $[a, b]$ -DecNZ $[c, d]$ -PZ gadget. The numbers of copies of repeated tunnels are shown in red.

Let $s(G)$ be the state of gadget G . We say that $G \sqsubset [i, j]$ if $s(G)$ is between i and j , and it is possible that $s(G) = i$, and also possible that $s(G) = j$. For example, if G starts at state 0, then $G \sqsubset [0, 0]$. If the player goes through the Inc $[a, b]$ tunnel, then $G \sqsubset [a, b]$. After going through the DecNZ $[c, d]$ tunnel, which is only possible if $b - c \geq 0$, $G \sqsubset [\max(a - d, 0), b - c]$. If $G \sqsubset [i, j]$, we define $\min(G) = i$ and $\max(G) = j$.

We maintain the invariant that $\max(G_0) = \min(G_1) = abcdn$, where n is state of the simulated Inc-DecNZ-PZ gadget.

- If the player crosses the simulated Inc tunnel, then $\max(G_0) := \max(G_0) + b \cdot acd = abcd(n + 1)$ and $\min(G_1) := \min(G_1) + a \cdot bcd = abcd(n + 1)$.
- If the player successfully crosses the simulated DecNZ tunnel, then because of G_0 , it was the case that $abcdn \geq c \cdot abd$, meaning that $n > 0$, which is what we want. Then $\max(G_0) := \max(G_0) - c \cdot abd = abcd(n - 1)$ and $\min(G_1) := \min(G_1) - d \cdot abc = abcd(n - 1)$. Note that since $\max(G_0) = \min(G_1)$, if G_0 's portion was crossable, then so is G_1 's portion.
- If the player successfully crosses the simulated PZ tunnel, then because of G_1 , it was the case that $abcdn = 0$, meaning that $n = 0$. Then $\max(G_0) := 0$ and $\min(G_1) := 0$. If G_1 's portion was crossable, then so is G_0 's portion, because $\max(G_0) = \min(G_1)$.

So this simulation works. ◀

2.6 Constant-Size Levels

Universal Turing or counter machines let us strengthen our results to need just a constant number of gadgets. For example, Korec [13] designs a 2-counter machine U_{32} (consisting of 32 instructions over Inc, Dec, and JZ) that is **strongly universal** in the sense that there is a computable function f such that, given any 2-counter machine M , M applied to x and y produces the same result as U_{32} applies to counter values $f(x)$ and y . By applying the theorems above to U_{32} , we obtain a system of a constant number of counter gadgets that simulates U_{32} and thus any 2-counter machine and thus any Turing machine. Crucially, this system must start with arbitrary specified initial states, to represent $f(x)$ and y . This framework and its implications are described in more depth in [1].

Applied to Mario, this means that we can prove RE-hardness of constant-size levels, provided they can start with arbitrarily many enemies at each location.

Alternatively, if we must start with all counters in state 0, or Mario levels without any enemies, then we need a linear number of instructions to build up the initial counter values (by repeated addition and multiplication implemented by repeated addition).

3 New Super Mario Bros. Series

An earlier version of this work [2] showed that Generalized New Super Mario Bros. is undecidable by a reduction from reachability with the Inc-DecNZ-PZ gadget and a crossover. In this section, we will prove undecidability for all games in the New Super Mario Bros. series by building a similar counter with the Inc[1,2]-DecNZ[1,2]-PZ gadget. Specifically, this gadget can be used in New Super Mario Bros., New Super Mario Bros. Wii, New Super Mario Bros. U, and New Super Mario Bros. 2. We use the number of enemies called Goombas to keep track of the counter state, and since we can simulate the Inc-DecNZ-PZ gadget by Theorem 14 we can obtain undecidability in constant size as in Section 2.6. However, this result is not necessarily planar, so we provide a simple crossover gadget.

In this section, we make heavy use of a mechanic in the New Super Mario Bros. games called *events*, which allows the player to interact with blocks via switches and event controllers. The functionality of events that we will use can be summarized as follows, although exact implementations vary per game:

- Each event can be either on or off.
- Event controllers toggle one event based on the status of another.
- Location controllers toggle an event based on the status of enemies or players in a pre-defined location.
- There are blocks which appear or disappear according to the status of an event.

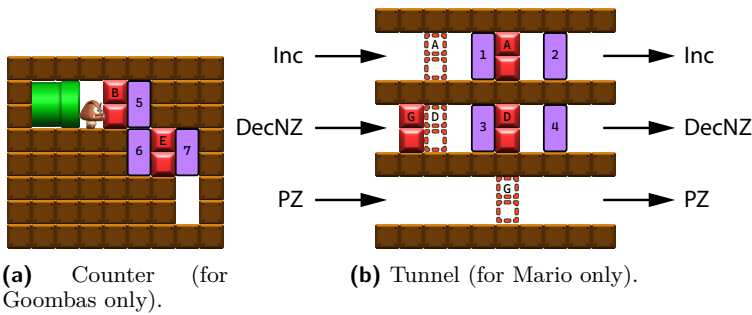
► **Theorem 6.** *The New Super Mario Bros. games are RE-complete.*

Proof. Our counter gadget is composed of two parts: the counter, pictured in Figure 15a, and the tunnel, pictured in Figure 15b. Each purple area represents a defined location. The groups of blocks associated with a letter are controlled by the event with the given letter³. When the event with the corresponding letter changes state, the blocks also change state between visible and invisible (where invisible blocks are also intangible). Invisible blocks are depicted as outlined instead of filled. There are also several invisible control objects:

- An enemy spawner spawns Goombas 🍄 from the pipe 🟩.
- A location controller activates event A when a player enters location 1.
- An event controller activates event B when event A is activated.
- A location controller deactivates event A when a player enters location 2.
- A location controller activates event D when a player enters location 3.
- An event controller activates event E when event D is activated.
- A location controller deactivates event D when a player enters location 4.
- A location controller activates event C when an enemy enters location 5.
- An event controller deactivates event B when event C is activated.
- A location controller activates event F when an enemy enters location 7.
- An event controller deactivates event E when event F is activated.
- An event controller activates event G if and only if an enemy is in location 6.

Inc[1,2]. When the player enters the Inc tunnel, they encounter location 1 which enables event A, preventing backtracking and opening the path through the tunnel. They then enter location 2, which reverses this change, preventing backtracking. At the same time, event A triggers event B which allows a Goomba 🍄 to pass through the blocks tied to event B. That same Goomba 🍄 enters location 5 which indirectly deactivates event B. Because of the way Goombas 🍄 spread out when moving, only one or two Goombas 🍄 will pass through during this time. The incremented Goomba(s) 🍄 end up in location 6.

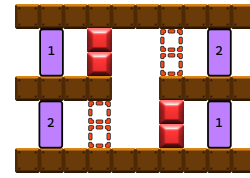
³ In memory, events are numbered, not lettered, but we use letters to disambiguate with locations



(a) Counter (for Goombas only).

(b) Tunnel (for Mario only).

■ **Figure 15** The New Super Mario Bros. counter gadget in state 0.



■ **Figure 16** The New Super Mario Bros. crossover gadget.

DecNZ[1, 2]. The DecNZ tunnel works analogously to the Inc tunnel, instead allowing one Goomba 🍄 to pass through the blocks tied to event E. In addition, if a Goomba 🍄 is not in location 6, i.e. the counter value is zero, event G will be disabled, and blocks tied to G will block traversal through the decrement tunnel, enforcing the NZ condition.

PZ. Similarly to how the NZ condition is enforced, if a Goomba 🍄 is in location 6, event G is active and blocks tied to G block the path through PZ.

To complete the proof, we provide a crossover gadget, as pictured in Figure 16. This is another event-based gadget, where all blocks are controlled by one event, which is off as pictured. Entering either location labeled 1 will enable the event, toggling the states of all blocks and allowing traversal only between the top left and bottom right, and entering either location labeled 2 will disable the event, returning it to the state pictured and allowing traversal between the top right and bottom left. ◀

3.1 Constant-Size New Super Mario Bros. Wii

As described in Section 2.6, the above result implies undecidability for a level of constant size. In fact, we have explicitly built a universal counter machine in New Super Mario Bros. Wii within a *single screen*. Figure 17 depicts such a counter, as shown in the *Reggie* level editor. Our level file is available to download and play [17].

Specifically, this level builds the strongly universal counter machine U_{32} of Korec [13], following an approach taken for the video game *Baba Is You* [1]. The construction of this level is somewhat different from the reductions we have described, making many simplifications to fit the level on one screen. The left side of the level is devoted to the 8 registers, while the majority of the level is devoted to traversal tunnels which control the states of the gadget. Effectively, this is the same as the two components of our gadget featured in Theorem 6, but with extra space removed. Branches are achieved easily by having an event controller to check whether a Goomba is in the counter location. As pictured, the gadget has values of 0, 0, 1, 1, 0, 0, 1, 0 for registers 0, 1, 2, 3, 4, 5, 6, 7 respectively. The goal of the level is to traverse from the starting area (in the bottom left) to the pipe (bottom right) which leads to the flagpole. A mini-mushroom is provided in the starting area to help the player traverse more easily through tight tunnels, but is not essential to the functioning of the gadget.

Because this entire counter fits on one screen, it is unnecessary to generalize level size as we have with other games. However, we still need to generalize the game in a few ways. Specifically, the actual game features a timer, and the actual game engine will spawn a Goomba from each pipe only if there are not already eight Goombas on screen. Both of these limitations need to be removed to make the level fully work.



■ **Figure 17** U_{32} in New Super Mario Bros. Wii.

Because the functionality of events is effectively the same across the New Super Mario Bros. series, this same counter should be able to be built in the other games, although smaller Nintendo DS screen sizes seem to prevent making such a large screen size in the DS games.

4 Super Mario Maker

First we describe how the screen size (which we do not generalize from the implemented size) affects local vs. global behaviors in Super Mario Maker 1 and 2. As in most Mario games, memory and effects are typically limited to the extent of the screen – technically, the *relevant screen* which extends four blocks beyond the visible screen. For example, activating a P-switch temporarily turns coins into blocks, but only within the current relevant screen; as Mario and this screen moves, which coins are transformed changes, which can impact nearby enemies etc. Similarly, enemies typically *spawn* when the relevant screen first overlaps their start location, and then *despawn* when they leave that relevant screen.

















But the Super Mario Maker game engines also define a notion of *global* objects [9] whose state is remembered even when it outside the relevant screen. Only a handful of objects are inherently global; for example, One-ways spawn at the beginning of the level and never despawn, while Yoshi spawns when reaching the screen but then never despawns. Crucially, an object becomes global if it is on Tracks, or is on top of another global object; this principle is called *global ground*. We use this property to make global any objects we want to be remembered, such as the enemies representing the state of a counter.


The Super Mario Maker games are also unusual in that they allow level creation in multiple (4–5) game styles, which each offer some slightly unique mechanics. Our constructions that apply to four styles make use of only mechanics that are present in all four styles, and all four of the styles have the same physics.




4.1 Super Mario Maker 1



► **Theorem 7.** *All four game styles (Super Mario Bros. 1, Super Mario Bros. 3, Super Mario World, and New Super Mario Bros. U) of Super Mario Maker 1 are RE-complete.*





Proof. We reduce from planar reachability with the Inc-DecNZ-PZ gadget (Theorem 4). Our gadget uses the following elements:

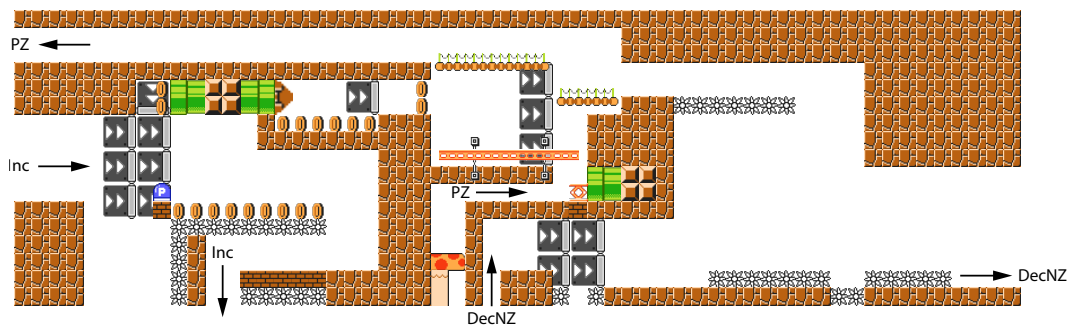
- **Solid ground:**   Solid ground with no special effects.
- **Semisolid platforms:**    Platforms which can be jumped through from below but are solid from above.
- **One-ways:**  Walls which allow entities (Mario, enemies, etc.) to pass the white bar only in the direction of the arrows.
- **Brick block:**  Solid block which can be hit from below to defeat enemies or bounce trampolines above it,
- **Coin:**  Transparent course element which allows Mario and enemies to pass through freely. If Mario touches a coin, it is collected and disappears from the course.
- **P-switches:**  Switches which turn coins into brick blocks and vice versa for the duration of a timer.
- **Tracks:**  Rails specifying periodic movement of attached entities; platforms  on tracks are global ground, meaning that they and entities on them do not despawn when offscreen.
- **Spikes:**  Blocks which damage Mario upon contact.
- **Goombas:**  Enemies which damage Mario when he contacts from any direction but above. Can safely walk on spikes and are defeated by being jumped on, giving Mario a vertical boost similar to a jump. We use big (2×2) Goombas in this construction.
- **Trampoline:**  Item which bounces entities on it up into the air.
- **Pipes:**  Elements which periodically spawn a particular item or enemy (drawn next to the pipe) into the course. If the pipe spawns items, it will only do so if the last element that the pipe spawned is no longer loaded or exists. Enemies such as Goombas  spawn indefinitely. Their spawn frequency can be adjusted to one of four speeds.

The key idea in this gadget is similar to other Mario Inc-DecNZ-PZ reductions: use the number of Goombas  in a particular location to represent the value of a counter. The gadget has infinitely many enumerated states such that, for any integer $n \geq 0$, there exists a state with a collection of n Goombas. The particular construction of the gadget determines how to increase, decrease, and check for zero in the gadget.


The counter element of the gadget is a semisolid platform  on a track . The track makes the platform global ground, preventing the Goombas  from despawning as Mario moves away from the gadget.






Another critical element to the gadget construction is Goomba pushing. If two Goombas  are on the same y level and walk into each other, they both bounce off and move in the opposite direction. This property is useful for building single-Goomba chambers: if there is a one-way  into a chamber with the width of a single Goomba that can only be accessed from the side, at most one Goomba can occupy it. This is because, if any other Goombas attempt to walk into the space, they will bounce off and be unable to pass through the one-way.












Inc. On traversal of the increment path, the player is forced to add exactly one Goomba  to the gadget counter. Above the semisolid platform  counter, there is a long horizontal chamber with coins  for a floor. At one end of the corridor, a pipe  spawns Goombas




















■ **Figure 18** The counter gadget for Super Mario Maker.

which immediately fall through the coin floor. At the other end of the pipe, a one-way  leads to a space wide enough for exactly one Goomba bounded by coins: a single-Goomba chamber as described above. This chamber is level with the pipe and has a solid floor, unlike the coin floor of the corridor.






On activation of a P-switch , the coins  turn to brick blocks . Any Goombas  already in the gadget are defeated instantly, and future spawned Goombas can freely walk towards the chamber. The coins bordering the chamber on the other side will be brick blocks, trapping exactly one Goomba in the chamber. Once the P-switch timer expires, the floor will transform back into coins and all the Goombas currently in the corridor will be too low to enter the chamber. The brick blocks at the end of the chamber will turn back into coins, allowing the Goomba to pass through and fall onto the semisolid platforms  to increment the counter. As long as the corridor length and pipe spawn frequency guarantee that at least one Goomba reaches the chamber by the end of a P-switch cycle, every P-switch activation guarantees a counter increment by exactly one.




An Inc traversal path can be built such that Mario is forced to hit a P-switch exactly once. We accomplish this by spawning P-switches  from a pipe  in the ceiling, which fall onto a brick block  behind a one-way . Beyond the brick block, a row of coins  above a row of spikes  leads to solid ground . After the solid ground, another row of spikes  below a row of brick blocks  lead out to the exit port. The gadget is constructed such that Mario must jump through a set of one-ways : one before hitting the P-switch  and another immediately after.



To traverse, Mario must jump past the one-way  and land on the P-switch , then run across the transformed brick blocks  over the spikes  and onto the solid ground . Once through the one-way, Mario cannot go back and activate another P-switch. Since the exit port is blocked by a spike row, Mario is forced to wait out the full P-switch timer, guaranteeing that the Goomba  remains loaded until it can fall onto the counter and reach global ground.

DecNZ. At the end of the semisolid counter platform  farthest from the chamber described above, another Goomba chamber is formed by a one-way  and solid ground  tiles. This guarantees that, in any state $n > 0$, exactly one Goomba  is in the chambered section of the counter. Below the chamber, a pipe  spawns trampolines  onto a brick block  which is accessible from below. Directly above the chamber, a semisolid platform  leads toward a long fall. The platform is low enough such that a Goomba bounced up from the counter by a trampoline would land on the upper semisolid platform , no longer in the counter. The upper semisolid platform is bounded on one side by one-ways , and on the other it has a long fall onto a row of spikes .

22:16 Undecidable Mario Games

The decrement works by having small Mario hit the brick block  from below, bouncing the trampoline  upwards and allowing it to bounce a Goomba  up into the air onto the upper semisolid platform . Mario must then traverse the row of spikes , which is too long to clear in a single jump. If the counter had more than one Goomba and the bounce succeeded, a Goomba will fall onto the spikes, allowing Mario to jump off it to gain extra height and clear the row, exiting the gadget. If the counter was at zero, Mario would be unable to clear the spikes and lose a life instead. This checks that the decrement removes at least one Goomba.










We enforce the condition that Mario can only decrement once per traversal by using one-ways. Mario is forced to jump through a one-way , hit the brick block , then fall through another one-way . Using this construction, Mario can only hit the brick block once, making a tight lower bound and guaranteeing decrement of exactly one enemy.

























PZ. The traverse path requires that Mario jumps through the semisolid counter platform  from below, in particular through the bottom of the single-Goomba chamber. If the gadget is in any nonzero state, there is guaranteed to be one Goomba  in the chamber. Mario will take damage when making contact from below and lose a life. If the gadget is in a zero state instead, the Goomba chamber will be empty and Mario will be free to traverse. ◀

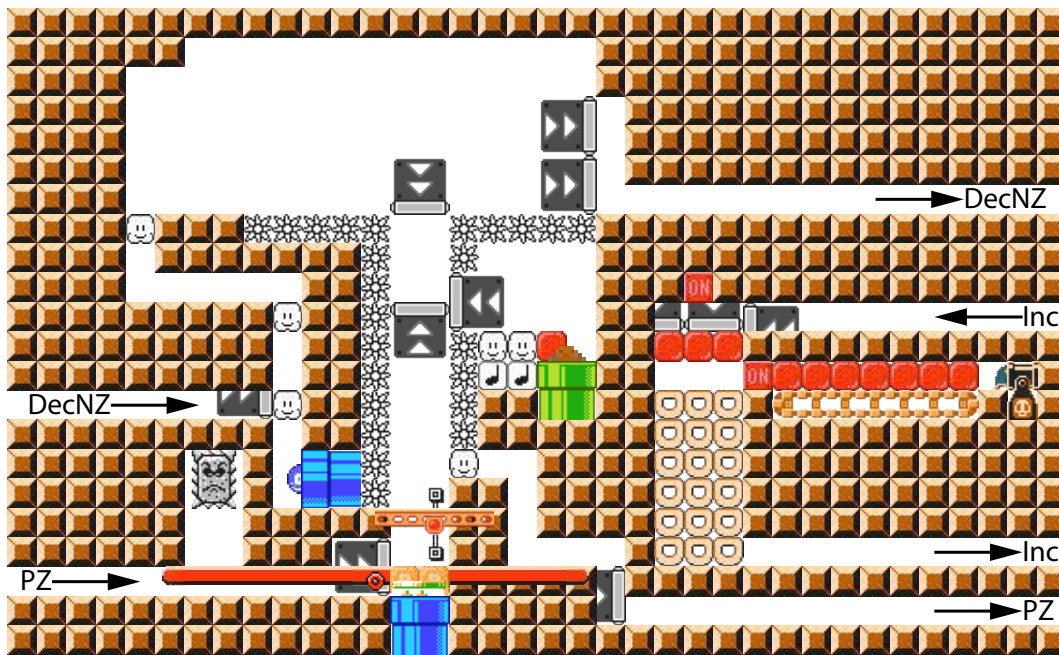
4.2 Super Mario Maker 2

► **Theorem 8.** *All four normal game styles (Super Mario Bros. 1, Super Mario Bros. 3, Super Mario World, and New Super Mario Bros. U) of Super Mario Maker 2 are RE-complete.*

The reduction for Super Mario Maker 1 from Section 4.1 might be adaptable to Super Mario Maker 2, but differing mechanics in the latter game (which allows picking up items from behind one-ways) would mean that the gadget would have to be altered for some of the game styles in order to prevent breaking the gadget. Thus, we demonstrate here a different gadget which works equally well for all four styles, making use of some mechanics exclusive to Super Mario Maker 2.

Proof. We reduce from planar reachability with the Inc-DecNZ-PZ gadget (Theorem 4). Figure 19 shows the gadget, and Figure 20 shows portions of a playthrough. We use some entities already described in Section 4.1: solid ground , semisolid platforms , one-ways , P-switches , tracks , spikes , platforms , goombas , and pipes . In addition, we use the following entities:

- **Donut blocks:**  Semisolid blocks that begin to fall after Mario stands on them for a short amount of time. Placing many in a vertical drop effectively slows Mario's downward traversal.
- **Note blocks:**  Bouncy blocks that cause any entity that walks on them to be bounced upward.
- **P-blocks:**   These blocks flip from being outlines to being solid (or vice versa) while the P-switch  is active. In this construction, they prevent the clown car  from spawning out of the blue pipe unless the P-switch is active.
- **On/off switches:**   When hit from below, these switches flip a global on/off state, toggling the solidity of on/off blocks:   vs.  .
- **On/off blocks:**     These blocks come in and out of existence based on the state of the on/off switches  :   in the “on” red state , and   in the “off” blue state .



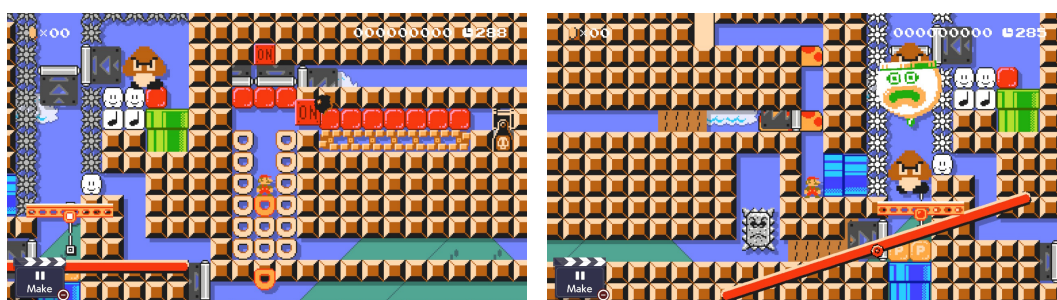
■ **Figure 19** Complete Inc-DecNZ-PZ gadget for the normal game styles of Super Mario Maker 2.

- **Blaster:** 🚬 These blasters shoot a shell 🐢 in the direction of Mario if he is sufficiently close and the corresponding barrel of the blaster is not blocked. These shells can activate on/off switches ON OFF.
- **Clown cars:** 🤡 These vehicles allow exactly one falling enemy to enter and ride them. If a loaded clown car touches spikes ❄️, then it will panic and fly upward.
- **Thwomps:** 🐼 An enemy that charges downward if Mario is sufficiently close and at or below the level of the Thwomp, and then resets to its original location.
- **Seesaws:** 🛖 A tilting platform that balances based on the weight on both sides of the center. If a Thwomp 🐼 pounds on one side of the seesaw, it will launch up any enemies on the other side of the platform. This is used to get the Goombas 🐵 into a falling state so that they can be picked up by the clown car 🤡.
- **Conveyors:** 🚶 A moving platform that can slow down a shell 🐢 that is moving in the opposite direction. In this construction, it allows us to use less space to achieve the timing for the shell that deactivates an activated on/off switch ON.

As before, the state of the counter gadget is the number of Goombas 🐵 on the platform ON on the track ON.

PZ. When the gadget is in state 0, there are no Goombas 🐵 on the platform ON, allowing the player to traverse the PZ path. When the gadget is in any state > 0 , there is at least one Goomba on the platform, preventing the player from traversing the PZ path without taking damage (and thus dying).

Inc. When the player enters the increment path, the first tunnel is long enough that a shell 🐢 will be launched left from the blaster 🚬 and bounce within the single adjacent space. At the end of this tunnel, the leftward one-way ▶▶ ensures that the player cannot hit the on/off switch ON without fully activating the increment, and the downward facing one-ways and on/off blocks below ensure that the player cannot hit the on/off switch more than once.



(a) Inc path: The shell 🐚 resets the on/off switch 🔌 after exactly one Goomba 👹 has spawned. (b) DecNZ path: A single Goomba 👹 rides the panicked clown car 🤡 upward.

■ **Figure 20** Screenshots from playing the gadget from Figure 19 in Super Mario Maker 2.

Once the on/off switch 🔌 is triggered, the green pipe 🌱, being no longer blocked from spawning, immediately starts spawning Goombas 👹. At the same time, the shell 🐚, no longer confined to a single space, starts moving left toward an on/off switch 🔌. The shell's distance to the switch, along with the speed of the conveyor 🚚 and the spawning speed of the green pipe 🌱, ensures that exactly one Goomba 👹 can spawn before the pipe is again blocked from spawning by the shell hitting the on/off switch, as shown in Figure 20a. This single Goomba 👹 moves left, bouncing up from walking on note blocks 🎵, and falls onto the platform 🟭 on the track 🛤️.

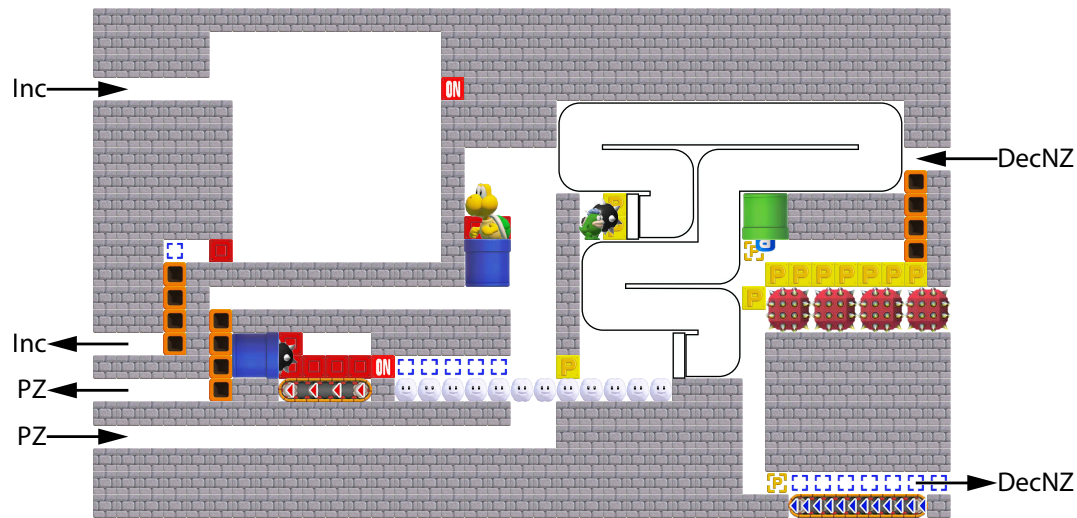
The rows of donut blocks 🍩 that Mario must endure at the end of this path ensures that the player keeps the Goomba 👹 loaded on screen until it is on the global ground of the platform 🟭, so that the counter value is correct.

DecNZ. Meanwhile, when the player enters the decrement path, they fall onto a P-switch 🟩 spawned from the the left blue pipe 🌊, activating the P-switch. (Note that Mario now blocks spawns from the pipe, so no new P-switches will spawn. And even if the player managed to fall into the hole exactly when the P-switch was spawning, since it is a 1-wide hole, they can never pick up the P-switch, and only fall onto it or block spawns from the pipe.)

This causes the bottom blue pipe 🌊, no longer being blocked from spawning, to spawn a clown car 🤡. At the same time, falling into this hole is just far enough down to trigger the Thwomp 🐼, which repeatedly charges downward onto the seesaw 🛖. This, in turn, causes the opposite end of the seesaw to shoot up, launching the Goombas 👹.

Though it usually takes a few seconds for all of this to line up, eventually this results in the Goombas 👹 being shot up while the clown car 🤡 is under them, causing one Goomba to enter the clown car while the rest fall back onto the platform 🟭. Now that it is entered, the clown car panics from touching spikes 🌵, and flies upward until it gets stuck between the two one-ways 🚪 at the top of the spike column. Figure 20b shows the separation of a single Goomba and panicking of the clown car.

Now the player is free to jump up to the top tunnel of the decrement path, where they can clear the long jump over the spikes 🌵 by bouncing on the Goomba 👹 in the clown car 🤡 in the middle of the jump. If the player performs an illegal decrement from state 0, then the clown car will never fly up because it never gets entered, and Mario will not be able to clear the long jump over the spikes. ◀



■ **Figure 21** The complete Inc-DecNZ-PZ gadget for the 3D World game style of Super Mario Maker 2.

The 3D world style of Super Mario Maker 2 has many different mechanics from that of the four regular styles, and so requires a different construction to show that it is RE-hard. See the full paper for details.

► **Theorem 9.** *The Super Mario 3D World style of Super Mario Maker 2 is RE-complete.*

Proof. Our gadget is presented in Figure 21. The details of this gadget are covered in the full version of our paper. ◀

5 Open Problems

Of the 2D Mario Games released since New Super Mario Bros., we have shown that all except for Super Mario Wonder are undecidable, and a natural open question is whether it is too. There is evidence which suggests that it might be based on the presence of events and infinitely spawning Goombas, but the game is still very new, and more research is needed to understand the mechanics of the game well enough to make further claims about undecidability.

There are also several older 2D Mario Games which have evidence that they might be able to build counters. In particular, any game with a Lakitu has a way of generating unlimited numbers of Spinies, Super Mario Bros. 3 and Super Mario World 2: Yoshi's Island both have enemies that can be generated from pipes, and Super Mario World has enemies generated by falling from the sky. Can we use any of the mechanics in those games to build counters? If any of these other games are not undecidable, it would also be noteworthy if we can obtain any other upper bounds on their complexity.

Finally, we showed that all games considered here are hard in constant-size levels, but it is not certain exactly what that constant is. We provided an explicit example of a single-screen counter in New Super Mario Bros. Wii, but we can definitely compact it further if we want to build the smallest possible universal counter machine. Furthermore, we can consider the Super Mario Maker games and whether it is possible to build a universal counter machine that fits inside of the standard constraint on level size.

References

- 1 Zachary Abel and Della Hendrickson. Baba is universal. In *Proceedings of the 12th International Conference on Fun with Algorithms (FUN 2024)*, pages 31:1–31:16, La Maddalena, Italy, June 2024.
- 2 Hayashi Ani. Unsimulability, universality, and undecidability in the gizmo framework. Master’s thesis, Massachusetts Institute of Technology, 2023.
- 3 Hayashi Ani, Jeffrey Bosboom, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, pages 3:1–3:23, La Maddalena, Italy, September 2020.
- 4 Hayashi Ani, Lily Chung, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Pushing blocks via checkable gadgets: PSPACE-completeness of Push-1F and Block/Box Dude. In *Proceedings of the 11th International Conference on Fun with Algorithms (FUN 2022)*, pages 2:1–2:30, Favignana, Italy, May–June 2022.
- 5 Hayashi Ani, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Traversability, reconfiguration, and reachability in the gadget framework. In *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation (WALCOM 2022)*, volume 13174 of *Lecture Notes in Computer Science*, pages 47–58, Jember, Indonesia, March 2022.
- 6 Hayashi Ani, Erik D. Demaine, Della Hendrickson, and Jayson Lynch. Trains, games, and complexity: 0/1/2-player motion planning through input/output gadgets. In *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation (WALCOM 2022)*, volume 13174 of *Lecture Notes in Computer Science*, pages 187–198, Jember, Indonesia, March 2022.
- 7 Erik D. Demaine, Isaac Grosf, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, pages 18:1–18:21, La Maddalena, Italy, June 2018.
- 8 Erik D. Demaine, Della Hendrickson, and Jayson Lynch. Toward a general theory of motion planning complexity: Characterizing which gadgets make games hard. In *Proceedings of the 11th Conference on Innovations in Theoretical Computer Science (ITCS 2020)*, pages 62:1–62:42, Seattle, Washington, January 12–14 2020.
- 9 flamewizzy21. “Don’t leave me!” — Guide to global loading. Reddit post, January 2020. URL: https://www.reddit.com/r/MarioMaker/comments/eobdmj/dont_leave_me_guide_to_global_loading/.
- 10 Linus Hamilton. Braid is undecidable. arXiv:1412.0784, 2014. URL: <http://arxiv.org/abs/1412.0784>, arXiv:1412.0784.
- 11 Della Hendrickson. Gadgets and gizmos: A formal model of simulation in the gadget framework for motion planning. Master’s thesis, Massachusetts Institute of Technology, 2021.
- 12 Erik Kain. ‘Braid’ is a postmodern Super Mario Bros. Forbes Games blog, June 18 2012. URL: <https://www.forbes.com/sites/erikkain/2012/06/18/braid-is-a-postmodern-super-mario-bros/>.
- 13 Ivan Korec. Small universal register machines. *Theoretical Computer Science*, 168(2):267–301, 1996. doi:10.1016/S0304-3975(96)00080-1.
- 14 Jayson Lynch. *A framework for proving the computational intractability of motion planning problems*. PhD thesis, Massachusetts Institute of Technology, 2020.
- 15 Marvin L. Minsky. Recursive unsolvability of Post’s problem of “Tag” and other topics in theory of Turing machines. *Annals of Mathematics*, 74(3):437–455, 1961. URL: <http://www.jstor.org/stable/1970290>.
- 16 Marvin L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.
- 17 MIT Hardness Group. Mario hardness gadgets. GitHub repository. URL: <https://github.com/65440-2023/mario-hardness-gadgets>.


ASP-Completeness of Hamiltonicity in Grid Graphs, with Applications to Loop Puzzles

MIT Hardness Group¹

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Josh Brunner ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Lily Chung ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Erik D. Demaine ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Della Hendrickson ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Andy Tockman ✉ 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We prove that Hamiltonicity in maximum-degree-3 grid graphs (directed or undirected) is ASP-complete, i.e., it has a parsimonious reduction from every NP search problem (including a polynomial-time bijection between solutions). As a consequence, given k Hamiltonian cycles, it is NP-complete to find another; and counting Hamiltonian cycles is #P-complete. If we require the grid graph's vertices to form a full $m \times n$ rectangle, then we show that Hamiltonicity remains ASP-complete if the edges are directed or if we allow removing some edges (whereas including all undirected edges is known to be easy). These results enable us to develop a stronger “T-metacell” framework for proving ASP-completeness of rectangular puzzles, which requires building just a single gadget representing a degree-3 grid-graph vertex. We apply this general theory to prove ASP-completeness of 37 pencil-and-paper puzzles where the goal is to draw a loop subject to given constraints: Slalom, Onsen-meguri, Mejilink, Detour, Tapa-Like Loop, Kouchoku, Icelom; Masyu, Yajilin, Nagareru, Castle Wall, Moon or Sun, Country Road, Geradeweg, Maxi Loop, Mid-loop, Balance Loop, Simple Loop, Haisu, Reflect Link, Linesweeper; Vertex/Touch Slitherlink, Dotchi-Loop, Ovotovata, Building Walk, Rail Pool, Disorderly Loop, Ant Mill, Koburin, Mukkonn Enn, Rassi Silai, (Crossing) Ichimaga, Tapa, Canal View, and Aqre. The last 13 of these puzzles were not even known to be NP-hard. Along the way, we prove ASP-completeness of some simple forms of Tree-Residue Vertex-Breaking (TRVB), including planar multigraphs with degree-6 breakable vertices, or with degree-4 breakable and degree-1 unbreakable vertices.

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

Keywords and phrases pencil-and-paper puzzles, computational complexity, parsimony

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.23

Related Version *Full Version:* <https://arxiv.org/abs/2405.08377>

Acknowledgements This paper was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.5440) taught by Erik Demaine in Fall 2023. We thank the other participants of that class for helpful discussions and providing an inspiring atmosphere. Some figures drawn using SVG Tiler [<https://github.com/edemaine/svgtiler>].

¹ Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as “MIT Hardness Group” (without “et al.”).



© MIT Hardness Group, Josh Brunner, Lily Chung, Erik D. Demaine, Della Hendrickson, and Andy Tockman;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 23; pp. 23:1–23:20

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Hamiltonicity is one of the core NP-complete problems, used as the basis for countless NP-hardness reductions. It accounts for two of Karp’s 21 NP-complete problems [22]: directed and undirected Hamiltonian cycle. It has been shown to remain NP-complete for many restricted graph classes: undirected maximum-degree-3 graphs [15], undirected bipartite graphs [24], undirected 3-connected 3-regular bipartite graphs [2], undirected 2-connected 3-regular bipartite planar graphs [2], undirected 3-connected 3-regular planar graphs of minimum face degree 5 [16], directed planar graphs with indegree and outdegree at most 2 and total degree at most 3 [29], and so on.

One of the most useful special cases of Hamiltonicity is (square) *grid graphs*: graphs whose vertices are a subset of the 2D integer lattice, with an edge connecting two vertices exactly when they have distance 1. Itai, Papadimitriou, and Szwarcfiter [19] proved that Hamiltonicity is NP-complete in grid graphs. Papadimitriou and Vazirani [28] improved this result by proving Hamiltonicity NP-complete in grid graphs of maximum degree 3. Together, these results strengthen most of the special graph classes mentioned above (as grid graphs are necessarily planar and bipartite), with a stronger geometric guarantee. Other papers extend these results to other 2D grids [6, 10, 17]. Hamiltonicity in grid graphs is the foundation for NP-hardness proofs of countless games and puzzles, from video games [13, 9, 1] to pencil-and-paper puzzles [36, 3], as well as practical problems such as lawn mowing and milling [5, 4].

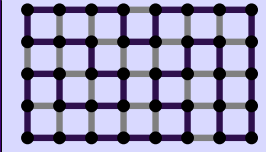
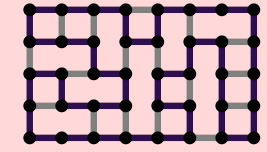
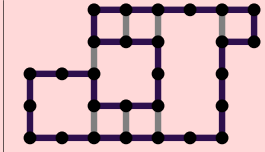
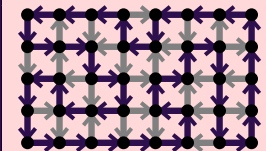
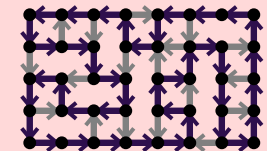
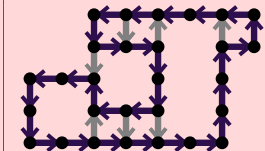
But what about *parsimonious* reductions that preserve the number of solutions? A particularly strong form of this notion is ASP-completeness: an NP search problem P is *ASP-complete* [37] if there is a polynomial-time reduction from every NP search problem Q to P along with a polynomial-time bijection converting every solution of P to a unique solution of Q and vice versa. If P is ASP-complete, then the decision version of P is NP-complete, counting solutions to P is #P-complete, and the k -ASP P problem – given an instance of P and k solutions, find another solution – is NP-complete for any $k \geq 0$ [37].

Only a few versions of Hamiltonicity are known to be ASP-complete, or weaker, #P-complete. Liśkiewicz, Ogihara, and Toda [25] proved #P-completeness of Hamiltonicity in undirected 3-regular planar graphs (based on [16]). Seta [30] proved ASP-completeness of Hamiltonicity in undirected maximum-degree-3 planar graphs (based on [29]). Bosboom et al. [8] proved ASP-completeness of Hamiltonicity in *directed* 3-regular (indegree 2 and outdegree 1 or vice versa) planar graphs (based on [29]). But what about grid graphs?

1.1 Our Results

In this paper, we prove that Hamiltonicity in maximum-degree-3 grid graphs is ASP-complete. Thus this popular problem can serve as a foundation for ASP-completeness proofs as well. The same result holds for Hamiltonicity in *directed* maximum-degree-3 grid graphs, where each edge has a specified direction. As mentioned above, grid graphs are bipartite and planar, so these results roughly strengthen the ASP-completeness results mentioned above, except that we can guarantee “maximum-degree-3” but not “3-regular”. (No grid graphs are 3-regular; consider the top-left corner. Furthermore, undirected 3-regular graphs have an even number of Hamiltonian cycles by Smith’s Theorem [34], so we cannot hope for ASP-completeness in this case: the 1-ASP decision problem is trivial, while the 1-ASP construction problem is in PPA [27].)

The basis for this result is another form of Hamiltonicity called *Tree-Residue Vertex-Breaking (TRVB)* [11], previously used to analyze Hamiltonicity in grid graphs [10]. In TRVB, we are given a graph where some vertices are breakable, and the goal is to *break*

	Rectangular	Max-degree-3 spanning subgraph of rectangular	Max-degree-3
Undirected	P [19] 	ASP-complete [§4.2] 	ASP-complete [§4.3] 
Directed	ASP-complete [§4.1] 	ASP-complete [§4.2] 	ASP-complete [§4.3] 

■ **Table 1** Complexity of Hamiltonicity in various types of grid graphs. Each cell shows an example of a Hamiltonian graph of the specified type, with a darkened Hamiltonian cycle. The first and third column concern true grid graphs, where there is an edge between each pair of vertices at distance 1. In the first and second columns, the vertices form exactly an $m \times n$ rectangle, whereas the third column allows an induced subgraph of a rectangular grid graph. The middle column concerns graphs constructed from a rectangular grid graph by removing some edges (but no vertices) so that each vertex has degree at most 3. The second and third columns have maximum degree 3.

a subset of the breakable vertices – replacing each broken degree- k vertex with k degree-1 vertices – to make the graph into a tree. This problem has a known characterization of what degrees of breakable or unbreakable vertices make the problem polynomial vs. NP-complete [11]. We prove that several forms of TRVB are in fact ASP-complete, including planar multigraphs with degree-6 breakable vertices, and planar multigraphs with degree-4 breakable and degree-1 unbreakable vertices.

We also study even more geometric forms of grid-graph Hamiltonicity. Suppose instead of allowing an arbitrary set of vertices on the square grid, we require the vertex set to be an entire $m \times n$ rectangle of integer points. Such graphs are known as *rectangular grid graphs* [19]. In this case, undirected Hamiltonicity is known to be easy [19]. But we show that *directed* Hamiltonicity in rectangular grid graphs is ASP-complete. Alternatively, if the graph is undirected but we allow removing some edges (but not vertices) from the rectangular grid – a spanning subgraph of a rectangular grid graph – then Hamiltonicity is also ASP-complete. Table 1 summarizes these results.

Rectangular grid graphs are useful because many (if not most) pencil-and-paper puzzles take place on a full rectangular grid. In particular, the *T-metacell framework* of Tang [32] shows how NP-hardness for a pencil-and-paper puzzle often follows from building a single gadget, essentially representing a degree-3 vertex that must be visited at least once. In Section 5, we extend this framework to prove ASP-completeness as well. We also extend the framework to allow for T-metacells where some exits are directed (usable in only one direction) and up to one exit is forced (must be used). In some cases, we need to build more than one T-metacell to handle different orientations of directions and/or forced edges.

Finally, in Section 6, we apply this framework to prove ASP-completeness of 37 pencil-and-paper puzzles, listed in Table 2. Five of these results use the same reduction from [32], while the remainder involve creating new T-metacell gadget(s). For thirteen of the analyzed puzzles, even our NP-hardness result is new.

■ **Table 2** Our results on pencil-and-paper puzzles. All ASP-completeness results are new; some are via an existing reduction [32] and some are via a new reduction; and some puzzles were not even known to be NP-hard. (Puzzles known to be NP-hard have corresponding citations).

Games	#	New ASP-Hardness	New Reduction	New NP-Hardness
Slalom/Suraromu [21, 32], Onsen-meguri [32], Mejilink [32], Detour [31, 32], Tapa-Like Loop [32], Kouchoku [32], Icelom [32]	7	yes	no	no
Masyu [14, 32], Yajilin [18, 32], Nagareru [20, 32], Castle Wall [32], Moon or Sun [20, 32], Country Road [18, 32], Geradeweg [32], Maxi Loop [32], Mid-loop [32], Balance Loop [32], Simple Loop [19, 32], Haisu [31, 32], Reflect Link [32], Linesweeper [26]	14	yes	yes	no
Vertex/Touch Slitherlink, Dotchi-Loop, Ovotovata, Building Walk, Rail Pool, Disorderly Loop, Ant Mill, Koburin, Mukkonn Enn, Rassi Silai, (Crossing) Ichimaga, Tapa, Canal View, Aqre	16	yes	yes	yes

2 Connections Between Problems

We collect together some useful equivalences between problems on plane graphs, which are variously present in the literature [12, 11].

► **Definition 1** ([11]). *The **Tree-Residue Vertex-Breaking** (TRVB) problem takes place on an undirected multigraph with vertices marked as either “breakable” or “unbreakable”. The goal is to break a subset S of the breakable vertices to leave a tree – to break a vertex of degree d , replace it with d new leaves attached to its incident edges. In other words, the graph obtained from G by subdividing every edge and deleting the vertices in S must be a tree.*

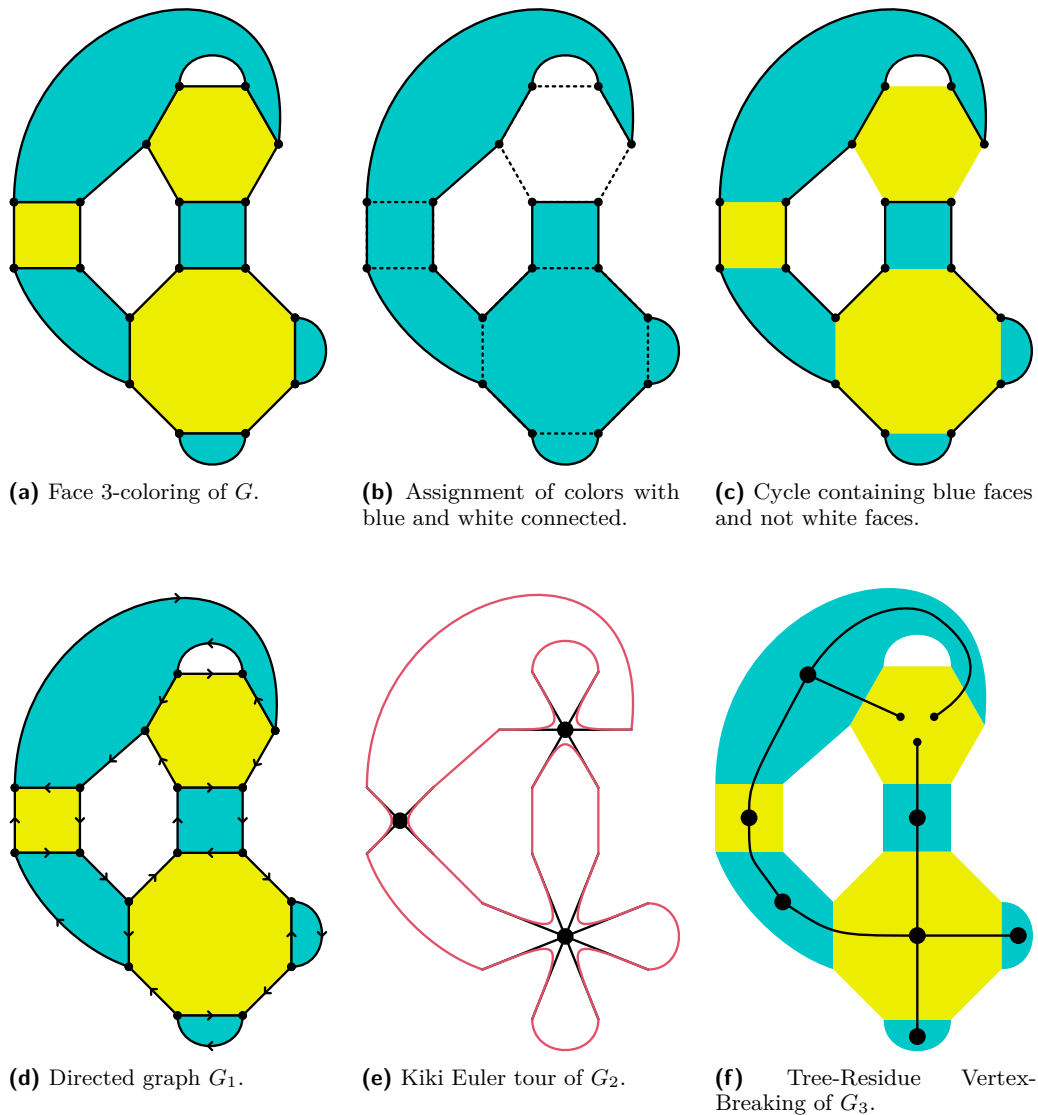
► **Definition 2** ([7, 12]). *Given a plane multigraph, a **kiki Euler tour** is a cycle which traverses every edge exactly once, such that any time the cycle enters a vertex via an edge e , it leaves by an edge adjacent to e in the cyclic order.²*

The following is a well-known result with a long history; see [33].

► **Theorem 3.** *Every Eulerian plane graph where every face is a triangle, except possibly the exterior face (a “near-triangulation”), has a proper vertex 3-coloring.*

Let G be a connected 3-regular bipartite plane multigraph, and let \tilde{G} be its plane dual. By Theorem 3, \tilde{G} is 3-colorable; equivalently it is possible to 3-color the faces of G so that adjacent faces have different colors, where faces are regarded as adjacent if they share an edge. Note that in such a 3-coloring, the three faces around a single vertex contain each color exactly once.

² This notion is one of two definitions of “nonintersecting” or “noncrossing Euler tour”. We avoid this term to avoid confusion with the other definition, where an Euler tour is has a **crossing** if there are four edges e, e', f, f' adjacent to a single vertex so that e' follows e and f' follows f in the tour, and $\{e, e'\}$ alternates with $\{f, f'\}$ in the cyclic order [33]. Noncrossing Euler tours in this sense always exist, whereas kiki is a stricter condition.



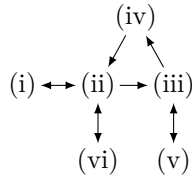
■ **Figure 1** Illustration of Lemma 4.

Let us fix such a coloring using the colors {white, blue, yellow} such that the exterior face is colored white. Define the following graphs:

- G_1 is the directed plane multigraph obtained from G by orienting every blue face clockwise and every white face counterclockwise. This fully determines the orientation.
- G_2 is the plane multigraph obtained from G by contracting every yellow face to a single vertex.
- G_3 is the subgraph of \tilde{G} induced by the non-white vertices.

► **Lemma 4.** *There are bijections between the following sets:*

- (i) *Assignments of colors {white, blue} to each yellow vertex of \tilde{G} such that the white induced subgraph is connected and the blue induced subgraph is also connected.*
- (ii) *Hamiltonian cycles of G which contain all blue faces and no white faces.*
- (iii) *Hamiltonian cycles of G which use every edge separating white faces from blue faces.*



■ **Figure 2** The bijections we define for Lemma 4.

- (iv) *Directed Hamiltonian cycles of G_1 .*
- (v) *Kiki Euler tours of G_2 .*
- (vi) *Tree-Residue Vertex-Breakings of G_3 , where yellow vertices are breakable and blue vertices are unbreakable.*

Proof. Refer to Figure 1. We give explicit transformations between the sets; it can be checked that these transformations invert each other as needed. Figure 2 summarizes the transformations we describe, which form a strongly connected graph.

- (i) \rightarrow (ii): Consider an assignment of colors to faces of G . For each vertex, two of the faces around it are one color and the third is the other color, so exactly two edges incident to it separate blue from white. The set of all edges separating blue from white thus forms a collection of cycles visiting each vertex once.
We claim that this is actually a single cycle. If it were multiple cycles, they would divide the plane into more than two regions. Two of those regions must be the same color (blue or white), violating the assumption that each color is connected.
So we have a Hamiltonian cycle separating blue from white, and since the exterior face is white, it contains all blue faces and no white faces of G .
- (ii) \rightarrow (i): Given a cycle, assign blue to exactly the faces it contains. Since the cycle is Hamiltonian, it does not intersect itself, so the blue faces are connected and the white faces are connected.
- (ii) \rightarrow (iii): If C contains all blue faces and no white faces, then it must use every edge separating white from blue.
- (iii) \rightarrow (iv): If C is a cycle on G_1 which uses every edge separating white from blue, then at each individual vertex it is impossible for C to reverse directions; thus it is always consistent with the orientations, so it is a directed Hamiltonian cycle.
- (iv) \rightarrow (ii): Suppose C is a directed Hamiltonian cycle of G_1 . Since C visits every vertex, it contains at least one edge of every face. Because C contains an edge of the exterior face its orientation must be consistently clockwise. Therefore C it encounters every blue face on its right side and every white face on the left, meaning it contains every blue face and does not contain any white faces.
- (iii) \rightarrow (v): The edges separating white and blue faces are exactly the edges of G_3 remaining after contracting the yellow faces. Let C be a Hamiltonian cycle of G containing every white-blue edge, and let C' be the Euler tour of G_3 obtained from C by the contraction. It must be the case that C contains exactly half of the edges incident to each yellow face, each of which connects two adjacent white-blue edges; so C' is kiki.
- (v) \rightarrow (iii): Suppose C' is a kiki Euler tour of G_3 . Let C be the set of edges of G consisting of all white-blue edges, together with those that connect consecutive edges in C' ; then C is a Hamiltonian cycle of G containing every white-blue edge.

(ii) \rightarrow (vi): Note that G_3 does not have any edges between two breakable vertices, so breaking a vertex is equivalent to removing it and all incident edges. Thus TRVB becomes “find an induced subgraph of G_3 containing all unbreakable vertices which is a tree”.

Given a cycle C , break all yellow vertices which are outside C , or equivalently take the induced subgraph on vertices inside C . This subgraph is clearly connected. If it has a cycle, there is a face of \tilde{G} inside that cycle, which corresponds to a vertex v of G . Then v is strictly inside C . But v must touch a white face, contradicting the fact that all white faces are outside C . Hence the induced subgraph on vertices inside C is a tree.

(vi) \rightarrow (ii): Take C to be the boundary of the tree containing blue faces and nonbroken yellow faces. Then C is a cycle because it bounds a tree, its interior contains all blue faces (which cannot be broken) and no white faces (which are not present in G_3). Finally, C is Hamiltonian because every vertex is incident to an edge separating blue from white, which must be in C . \blacktriangleleft

Furthermore, given any of the graphs G_i , equivalents to the others can be obtained by analogous transformations. So these various problems can be regarded as equivalent.

An important special case of TRVB is when every breakable vertex has degree at most 3. For planar graphs this condition is equivalent to requiring that every yellow face of the graph G in the preceding discussion is a digon or triangle; it is also equivalent to kiki Euler tour with vertices of degree at most 6. In this case, the problem can be solved in polynomial time by reducing it to a matroid parity problem.[12][11] In the next section we will discuss breakable vertices with higher degrees, with which the problem turns out to be ASP-complete.

3 ASP-Completeness of Tree-Residue Vertex-Breaking

Demaine and Rudoy [11] prove several NP-hardness results for TRVB using reductions from finding Hamiltonian cycles on a max-degree-3 planar directed graph. At the time, this Hamiltonian cycle problem was not known ASP-complete, so they did not consider ASP-completeness.

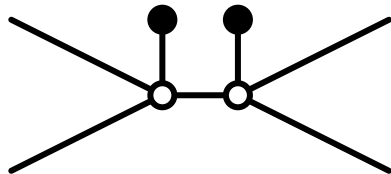
More recently, Bosboom et al. [8] showed that finding Hamiltonian cycles on a directed max-degree-3 planar graph is ASP-complete, using a reduction from positive 1-in-3SAT.

Several of the reductions used by Demaine and Rudoy [11] are easily verified to be parsimonious, proving ASP-completeness. We are specifically interested in the results of Section 4, on planar $(\{k\}, \{4\})$ -TRVB.

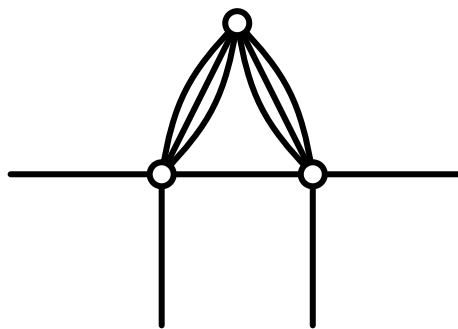
They first reduce finding Hamiltonian cycles on a max-degree-3 planar directed graph to finding Hamiltonian cycles on a planar graph where all vertices have indegree and outdegree 2 and vertices have their two in-edges and their two out-edges adjacent in the planar embedding. This last condition is called *non-alternating*, because vertices are not allowed to alternate in-edges and out-edges. The reduction is by contracting forced edges, and is straightforwardly parsimonious.

► **Theorem 5.** *Finding Hamiltonian cycles on non-alternating indegree-2 outdegree-2 planar graphs is ASP-complete.*

Next, Demaine and Rudoy reduce this problem to a version of Tree-Residue Vertex-Breaking. Specifically, Demaine and Rudoy [11] prove NP-hardness of TRVB on a planar graph where each unbreakable vertex has degree 4 and each breakable vertex has degree k , for any constant $k \geq 4$. This is *planar $(\{k\}, \{4\})$ -Tree-Residue Vertex-Breaking*. This reduction is a bit more complicated (see Section 4.2 and in particular Figures 11 through 13



■ **Figure 3** Simulating a degree-4 unbreakable vertex using degree-4 breakable vertices (white) and degree-1 unbreakable vertices (black).



■ **Figure 4** Simulating a degree-4 unbreakable vertex using degree-6 breakable vertices.

of [11]) but it is again parsimonious; indeed, [11, Lemmas 4.14 and 4.15] show that there is a bijection between Hamiltonian cycles in the input problem and solutions to the TRVB instance.

► **Theorem 6.** *Planar $(\{k\}, \{4\})$ -TRVB is ASP-complete, for each $k \geq 4$.*

To further simplify our reductions, we will use a slightly simpler version of TRVB: degree-4 breakable vertices and degree-1 unbreakable vertices.

► **Theorem 7.** *Planar $(\{4\}, \{1\})$ -TRVB is ASP-complete.*

Proof. It suffices to parsimoniously simulate a degree-4 unbreakable vertex. Such a simulation is shown in Figure 3. No vertex in the simulation can be broken in a solution to TRVB. ◀

► **Theorem 8.** *Planar $(\{6\}, \emptyset)$ -TRVB is ASP-complete.*

Proof. It again suffices to simulate a degree-4 breakable vertex. Such a simulation is shown in Figure 4. If the top vertex is not broken, both others must be broken, disconnecting the middle edge. So the top vertex must be broken, and then the other two vertices must not be. ◀

4 Hamiltonian Cycles in Grid Graphs

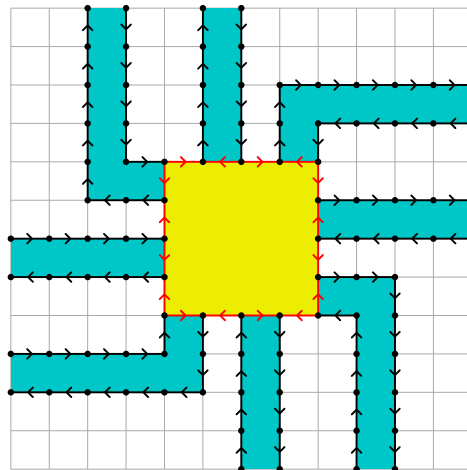
In this section, we prove ASP-completeness of finding Hamiltonian cycles in several natural classes of grid graphs. We begin by defining the types of graph that appear in our results.

► **Definition 9.** A *grid graph* is an induced subgraph of the square lattice. That is, its vertices are a subset of \mathbb{Z}^2 , and it has an edge between each pair of vertices at distance 1. In a *directed grid graph*, each edge has a direction, so there is exactly one edge between each pair of vertices at distance 1.

► **Definition 10.** A *rectangular grid graph* is one whose vertex set consists of all lattice points within a rectangle.

► **Definition 11.** A graph is *max-degree-3* if each of its vertices have degree at most 3.

► **Definition 12.** A *spanning subgraph* of G is a subgraph of G which contains all of the vertices (and some subset of the edges) of G .



■ **Figure 5** An example showing how reductions from TRVB to Hamiltonian cycle work.

Note that grid graphs contain all possible edges: graphs that contain only some of the edges are (spanning) subgraphs of grid graphs.

We consider three types of graph for each of undirected and directed. Our results are summarized in Table 1.

Most of our ASP-completeness results are by reductions from planar $(\{4\}, \{1\})$ -TRVB, and use the same core idea illustrated in Figure 5. This is a breakable degree-8 vertex, with the yellow square in the middle representing the vertex itself and the blue tentacles representing edges. We replace every vertex in the TRVB instance with a vertex like the one shown, and connect the tentacles of adjacent vertices. By Lemma 4, Hamiltonian cycles of the resulting graph correspond to solutions of the original TRVB instance.

This idea works equally well for directed and undirected graphs. To apply this idea to each of the five types of graph we prove ASP-completeness for, we need to show how to draw gadgets for degree-4 breakable and degree-1 unbreakable vertices in that type of graph, while ensuring that the tentacles representing edges do not interfere with each other.

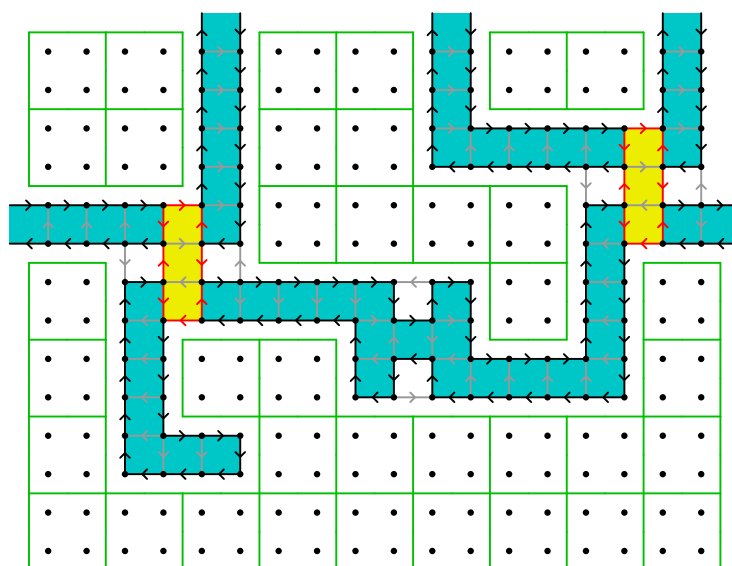
4.1 Rectangular Grid Graphs

► **Theorem 13** ([19]). *Finding Hamiltonian cycles on an undirected rectangular grid graph is in P.*

► **Theorem 14.** *Finding Hamiltonian cycles on a directed rectangular grid graph is ASP-complete.*

Proof. We first consider directed grid graphs, and later fill in holes to make them rectangular. Everything we need for this is shown in Figure 6. The yellow rectangles are degree-4 breakable vertices with exactly two local solutions, and the dead end in the bottom left is a degree-1 unbreakable vertex. As before, blue is inside the loop and yellow might be inside the loop depending on the choice made for a vertex gadget. If we ignore the gray edges, this is essentially the same as Figure 5.

We just need to ensure that gray edges cannot be used, which we can do by orienting them carefully. Ignoring the H-shaped construction in the center for the moment, each black edge is either the only edge pointing towards or the only edge pointing away from some vertex (depending on which side of the tentacle it's on), and thus must be used in a Hamiltonian



■ **Figure 6** TRVB gadgets for directed grid graphs, showing two breakable degree-4 vertices connected by an edge and an unbreakable degree-1 vertex.

cycle. We call such an edge *forced*. Each gray edge (still ignoring the H) shares either its source or its target with a black edge, and thus cannot be used. We call such an edge *unusable*.

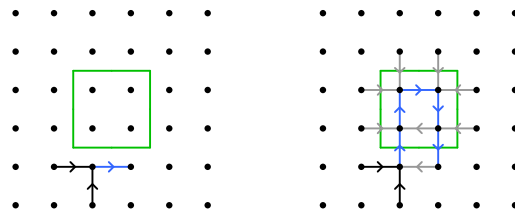
This requires the orientation of the gray edges relative to a tentacle to be different on the two ends of the tentacle, which is what the H achieves: one can verify by repeatedly finding forced edges and deleting unusable edges that any Hamiltonian cycle must use all black edges and no gray edges in the H. Each tentacle representing an edge between two degree-4 breakable vertices will have such an H.

This reduction proves a weaker version of the theorem: Finding Hamiltonian cycles on a directed grid graph is ASP-complete. It remains to fill all of the unused space to make a rectangular grid graph.

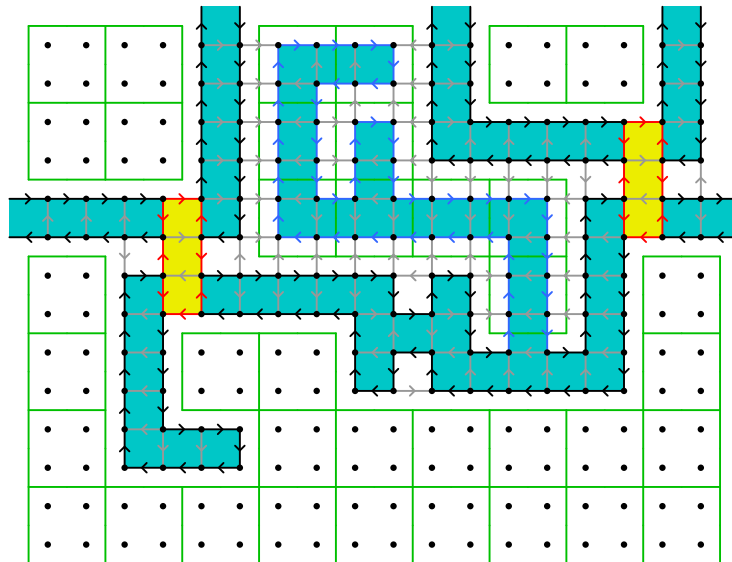
If we place each vertex gadget, H, and turn on the same parity, the construction lies neatly on a 2×2 grid, and in particular the holes are made of 2×2 squares. Figure 6 indicates these squares in green. In addition, in each hole at least one of these squares is adjacent to a forced edge: all black edges except a few in each H are forced,³ and each hole is adjacent to a non-H section of tentacle provided we do not use any extremely short tentacles.

Pick one such 2×2 square, and add four new vertices to fill it. Assume that the adjacent forced edge is the only outgoing edge from its source; the case where it is the only edge pointing towards its target is similar but with directions reversed. This situation is illustrated in Figure 7 (left), with the forced edge in blue. Now reverse the forced edge, and add new edges as shown on the right of Figure 7 (omitting any edges between a vertex in the square and a vertex outside it which doesn't yet exist). It is straightforward to check that all gray edges are unusable, so any Hamiltonian cycle must follow the blue path, which is equivalent to the original forced edge but consumes the added vertices.

³ They all become forced after deleting some unusable edges, but it's simpler to argue that hole filling works with directly forced edges.



■ **Figure 7** Filling holes in a directed rectangular grid graph.



■ **Figure 8** Figure 6 after some hole filling.

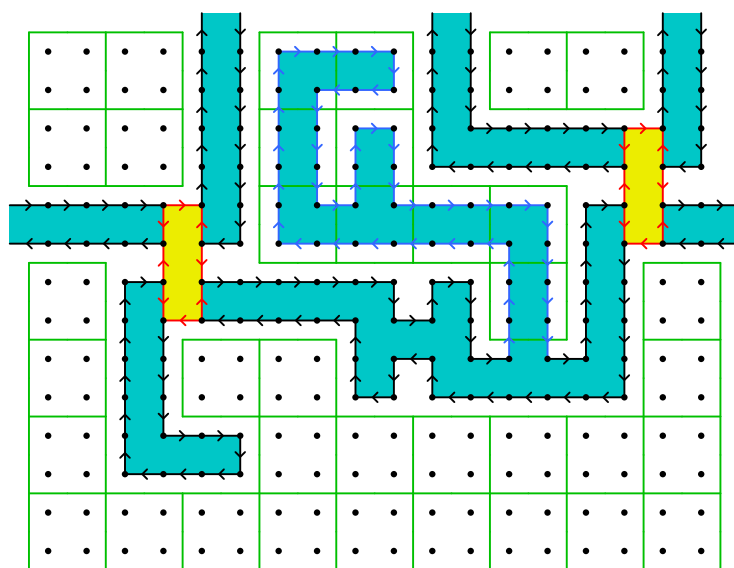
Filling this small portion of hole preserves the fact that every hole has a 2×2 square adjacent to a forced edge, since the three relevant blue edges are forced. Thus we can repeat this process until all holes are filled, ultimately filling each hole with paths that outline a spanning forest of the 2×2 squares. Figure 8 shows what this looks like after filling (the visible portion of) the top middle hole in Figure 6.

The result is a directed rectangular grid graph which is equivalent to the original directed grid graph for the purposes of Hamiltonian cycles. Hence Hamiltonian cycles in the final graph correspond to solutions to the instance of TRVB. ◀

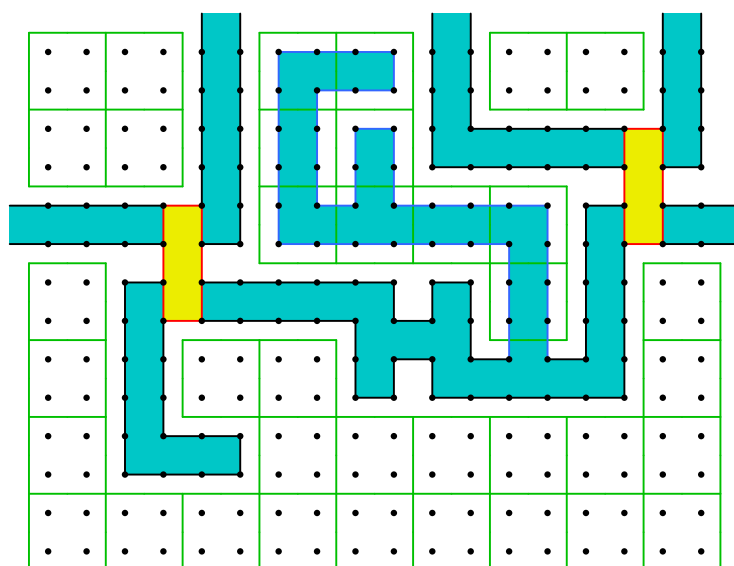
4.2 Max-Degree-3 Spanning Subgraphs of Rectangular Grid Graphs

► **Theorem 15.** *Let G be a directed max-degree-3 spanning subgraph of a rectangular grid graph. Consider the promise problem of finding an undirected Hamiltonian cycle on G , subject to the promise that all such cycles respect the given edge directions; that is, they would also be valid directed Hamiltonian cycles of G . This promise problem is ASP-complete.*

Proof. We modify the construction from Theorem 14 by simply removing all of the gray edges. Inspection of Figure 8 reveals that every vertex is incident to at most three non-gray edges: vertices along tentacles have two forced edges, and vertices in degree-4 vertex gadgets have one forced edge and two optional red edges. Filling holes preserves the non-gray degree of existing vertices and adds vertices with two non-gray edges.



■ **Figure 9** Figure 8 after removing gray edges.

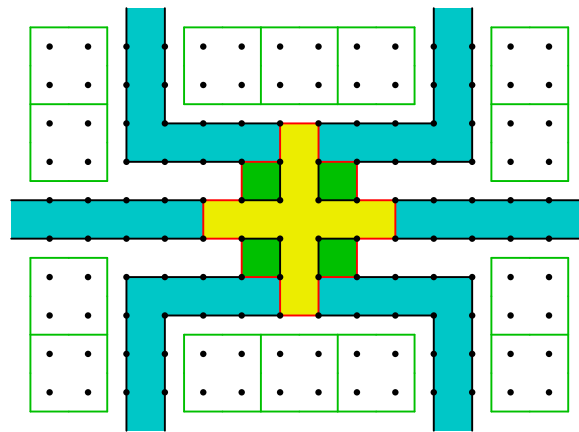


■ **Figure 10** Figure 9 after forgetting directions of edges.

In the previous proofs, all of the possible solutions only used non-gray edges. Thus, we can adapt the previous reduction by simply deleting all gray edges, obtaining a directed max-degree-3 spanning subgraph of a rectangular grid graph. For instance, doing this to Figure 8 yields Figure 9, which also has the advantage of being easier to read.

By the proof of Lemma 4, directed Hamiltonian cycles on G are the same as undirected Hamiltonian cycles on G , and the set of such cycles is in bijection with solutions of the original TRVB instance. ◀

► **Corollary 16.** *Finding Hamiltonian cycles on a directed max-degree-3 spanning subgraph of a rectangular grid graph is ASP-complete.*



■ **Figure 11** A breakable degree-6 TRVB vertex gadget for undirected max-degree-3 spanning subgraphs of rectangular grid graphs.

Proof. This is a special case of Theorem 15. ◀

In the undirected case, we can strengthen the assumption about forced edges. For undirected graphs, an edge is *forced* if it is incident to a degree-2 vertex, since both edges incident to such a vertex must be used in any Hamiltonian cycle. A degree-3 vertex in a subgraph of a grid graph has two edges in opposite directions, which we call *side* edges, and a third edge between them, which we call the *center* edge. In this case, we can assume not only that each degree-3 vertex has a forced edge, but that this forced edge is a side edge, further reducing the number of distinct vertices we need to simulate for an application.

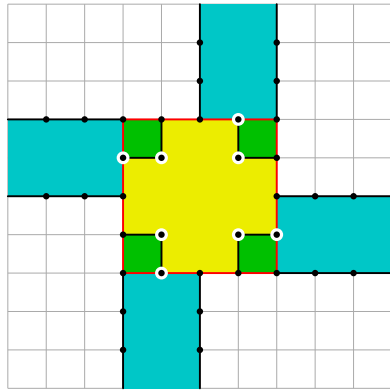
► **Theorem 17.** *Finding Hamiltonian cycles on an undirected max-degree-3 spanning subgraph of a rectangular grid graph is ASP-complete, even when every degree-3 vertex has a forced side edge.*

Proof. We are not able to directly build breakable degree-4 TRVB vertices under these constraints. However, we are able to build a breakable degree-6 vertex, so we reduce from planar $(\{6\}, \emptyset)$ -TRVB, which was shown ASP-complete in Theorem 8.

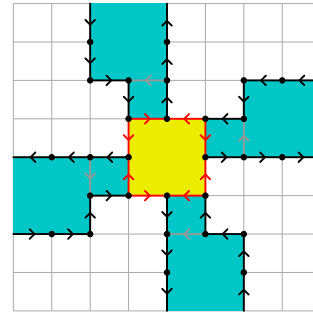
Our breakable degree-6 vertex gadget is shown in Figure 11. Black edges are forced, and red edges are optional. Note that vertices in tentacles all have degree 2, and each degree-3 vertex inside the vertex gadget has a forced side edge. This is equivalent to the cycle of red edges turning at every vertex. The vertex gadget has exactly two local solutions, which each use alternating red edges.

As before, blue tentacles are inside the cycle, and the yellow region is inside the cycle in one of the local solutions, corresponding to not breaking the TRVB vertex. We have new color as well: the green squares are inside the cycle in the other solution, when the TRVB vertex is broken. It is clear by inspection that the yellow local solution connects all six tentacles, and the green local solution disconnects them all.

Finally, we connect vertex gadgets along tentacles and fill holes in exactly the same way as before. Filling holes uses only degree-2 vertices, so it does not introduce degree-3 vertices without forced side edges. ◀



■ **Figure 12** A breakable degree-4 TRVB vertex gadget for undirected max-degree-3 grid graphs. Removing the vertices highlighted in white gives an unbreakable degree-4 vertex gadget.



■ **Figure 13** A breakable degree-4 TRVB vertex gadget for directed max-degree-3 grid graphs.

4.3 Max-Degree-3 Grid Graphs

► **Theorem 18.** *Finding Hamiltonian cycles on an undirected max-degree-3 grid graph is ASP-complete, even when every vertex has a forced edge.*

Proof. This proof is sketched, and its key gadget is shown, by Demaine and Rudoy [11], but at the time TRVB was not known to be ASP-complete, so it was purely a simpler proof of NP-hardness used to motivate the usefulness of TRVB.

Like most of our other proofs, we reduce from planar $(\{4\}, \{1\})$ -TRVB. Our breakable degree-4 vertex gadget is shown in Figure 12. The main difficulty in this case is that we need the paths on each side of a tentacle to be separated by distance at least 2, so that the cycle cannot cross between the two sides (and all tentacle edges are forced). As usual, black edges are forced, and there are exactly two solutions which each use alternating red edges. One solution puts the green region inside the cycle, and one puts the yellow region inside the cycle, corresponding to breaking and not breaking the vertex, respectively.

A degree-1 unbreakable vertex can be made by simply “capping off” a tentacle. Alternatively, we could reduce from $(\{4\}, \{4\})$ -TRVB, and construct a degree-4 unbreakable vertex gadget by removing the vertices highlighted in white from Figure 12. ◀

► **Theorem 19.** *Finding Hamiltonian cycles on a directed max-degree-3 grid graph is ASP-complete, even when every vertex has a forced edge.*

Proof. The proof is extremely similar to the previous proof. We again reduce from $(\{4\}, \{1\})$ -TRVB. Our degree-4 breakable vertex gadget is shown in Figure 13, and a degree-1 unbreakable vertex can again be made by capping off a tentacle. Black edges are forced and gray edges are unusable. We again keep the sides of a tentacle apart from each other (away from vertex gadgets) so that a cycle cannot leak between them.

As before, there are exactly two solutions to the vertex gadget, one of which put the yellow square inside the cycle corresponding to leaving the TRVB vertex unbroken. ◀

5 T-Metacells

Many puzzle genres which involve drawing a single loop are proven hard using reductions from various forms of grid graph Hamiltonicity. Tang [32] described a simple “T-metacell” framework for proving NP-hardness of these puzzles using grid graph Hamiltonicity. A

T-metacell is a gadget which represents a single degree-3 vertex in a grid graph. Each T-metacell is a (usually square) tile with 3 exits (on 3 of the 4 sides) such that the loop may traverse the gadget between any pair of exits. The gadget should be reflectable and rotatable, and the loop may travel between adjacent T-metacells only when both have exits along their shared border. Finally, the loop must be required to visit every T-metacell.

It's straightforward to see how T-metacells can simulate degree-3 vertices in a Hamiltonicity reduction; Tang showed that they can also simulate degree-2 vertices. Let G be a subgraph of a grid graph in which every vertex has degree 2 or 3. Degree-3 vertices of G can be replaced directly with T-metacells. To handle degree-2 vertices, consider the graph H on the same vertex set as G which has an edge between two lattice-adjacent vertices precisely when G is missing that edge. Then H consists of degree-1 and degree-2 vertices. Orient the edges of H into directed paths and cycles such that each vertex has a maximum indegree and outdegree of 1. Each degree-2 vertex of G can now be replaced by a T-metacell with its extra edge facing in the direction of the outward-pointing edge from that vertex in H . This ensures that this extra exit will always be facing a non-exit in the adjacent cell, so only the intended edges of G may be used by the loop.

We apply our results from Section 4 to show that solving T-metacell problems is ASP-complete, instead of just NP-hard. We extend the framework to allow for some exits of a T-metacell to be *directed*, meaning that the loop must have a consistent orientation which agree with the directions of the exits it uses. We also allow for T-metacells to have one *forced exit* through which the loop must pass. Note that when all three exits are directed, these necessarily create a forced exit: there must be either a lone exit directed inwards or a lone exit directed outwards, which in either case must be chosen. T-metacells with forced edges can be classified into two categories: symmetric and asymmetric. A symmetric T-metacell has its two unforced edges directly opposite each other, while an asymmetric T-metacell has its two unforced edges adjacent. We use this classification to reduce the number of distinct gadgets which need to be constructed to apply the framework.

► **Corollary 20.** *Finding Hamiltonian cycles on a rectangular grid of undirected T-metacells is ASP-complete.*

Proof. We reduce from finding Hamiltonian cycles on max-degree-3 spanning subgraphs of rectangular grid graphs (Theorem 17). Replace each vertex with a undirected T-metacell, handling degree-2 vertices as described above. ◀

► **Corollary 21.** *Finding Hamiltonian cycles on a rectangular grid of required-edge directed T-metacells is ASP-complete.*

Proof. We reduce from finding Hamiltonian cycles on directed max-degree-3 spanning subgraphs of rectangular grid graphs (Corollary 16). Place a T-metacell for each degree-3 vertex, and handle degree-2 vertices in the same way as above. The direction of the unusable edge on a T-metacell at a degree-2 vertex can be arbitrary. ◀

► **Corollary 22.** *Finding Hamiltonian cycles on a rectangular grid of asymmetric required-edge undirected T-metacells is ASP-complete.*

Proof. In the proof of Theorem 17, every degree-3 vertex conveniently has a forced side edge, which is equivalent to being a asymmetric undirected T-metacell. Degree-2 vertices require a bit more care, but are not an obstruction: after deciding how to orient T-metacells as described above, note that for each degree-2 vertex, at least one of its edges is a side edge of the T-metacell. So we can simply place a T-metacell with that side edge forced. ◀

► **Corollary 23.** *Finding Hamiltonian cycles on a rectangular grid of required-edge directed asymmetric T-metacells and required-edge undirected symmetric T-metacells is ASP-complete.*

Proof. We reduce from the promise problem of finding a Hamiltonian cycle of a directed max-degree-3 spanning subgraph of a rectangular grid graph, with the promise that every undirected Hamiltonian cycle is a valid directed Hamiltonian cycle (Theorem 15). We perform the same replacement of vertices with T-metacells as in Corollary 21, except that the symmetric T-metacells are undirected. We claim that Hamiltonian cycles of the original graph are in bijection with solutions to the T-metacell instance. A directed Hamiltonian cycle of the original graph clearly solves the T-metacell instance, since it correctly passes through the directions on the directed T-metacells. On the other hand, a solution to the T-metacell instance is necessarily an undirected Hamiltonian cycle of the original graph; by the promise, directed Hamiltonian cycles and undirected Hamiltonian cycles are the same. ◀

6 Applications

We apply our improved T-metacell framework to a variety of pencil-and-paper logic puzzles implemented by the online puzzle-solving interface “puzz.link” [23]. This web resource implements more than 240 different logic puzzles. It includes most genres published by the Japanese publisher Nikoli, whose puzzles have a long history of analysis from a computational complexity perspective [30] [37] [3] [35] [26] [32], as well as many others in a similar style.

We improve existing NP-hardness results for pencil-and-paper logic puzzles to ASP-completeness, and give new ASP-completeness results. Many of the ASP-completeness proofs consist of just a single T-metacell, demonstrating the ease of applying the framework for proving ASP-completeness. The main additional requirement when designing a T-metacell gadget for ASP-completeness proofs is that it be “parsimonious”: for each pair of exits, there must be a *unique* local solution where the loop passes through those exits.

Full explanations for each proof can be found in the full version of this paper; due to space constraints, we present an abridged gallery of reductions here.

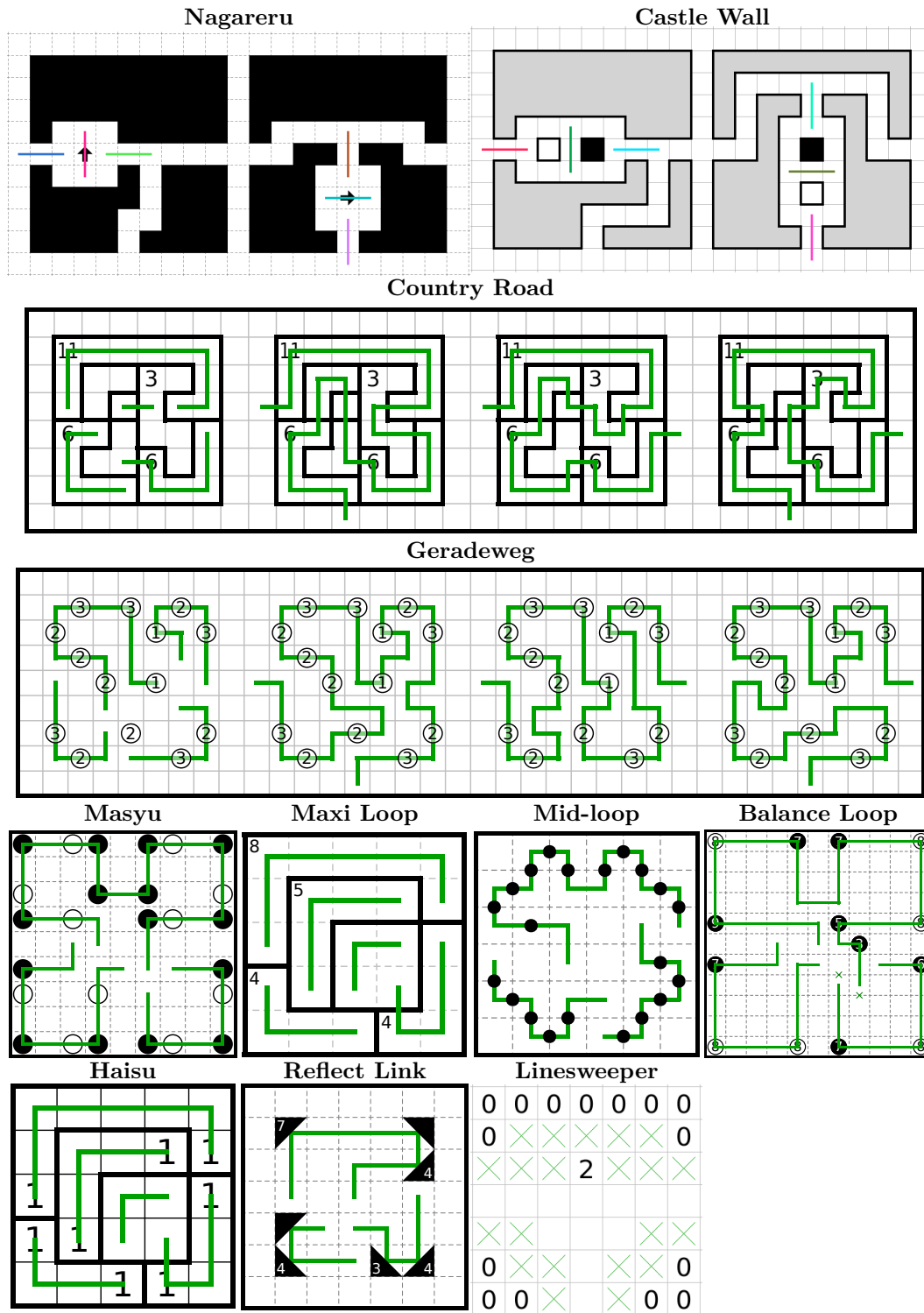
Figure 14 shows the gadgets for improving prior NP-hardness results to ASP-completeness, most of which consist of minor adjustments to existing T-metacells in [32] to ensure parsimony. We also make similar improvements for Yajilin, Moon and Sun, and Simple Loop via direct reductions from Hamiltonicity.

Figure 15 shows the gadgets for new NP- and ASP-completeness results. We also give similar results for Dotchi Loop, Ovotovata, and Koburin via direct reductions from Hamiltonicity.

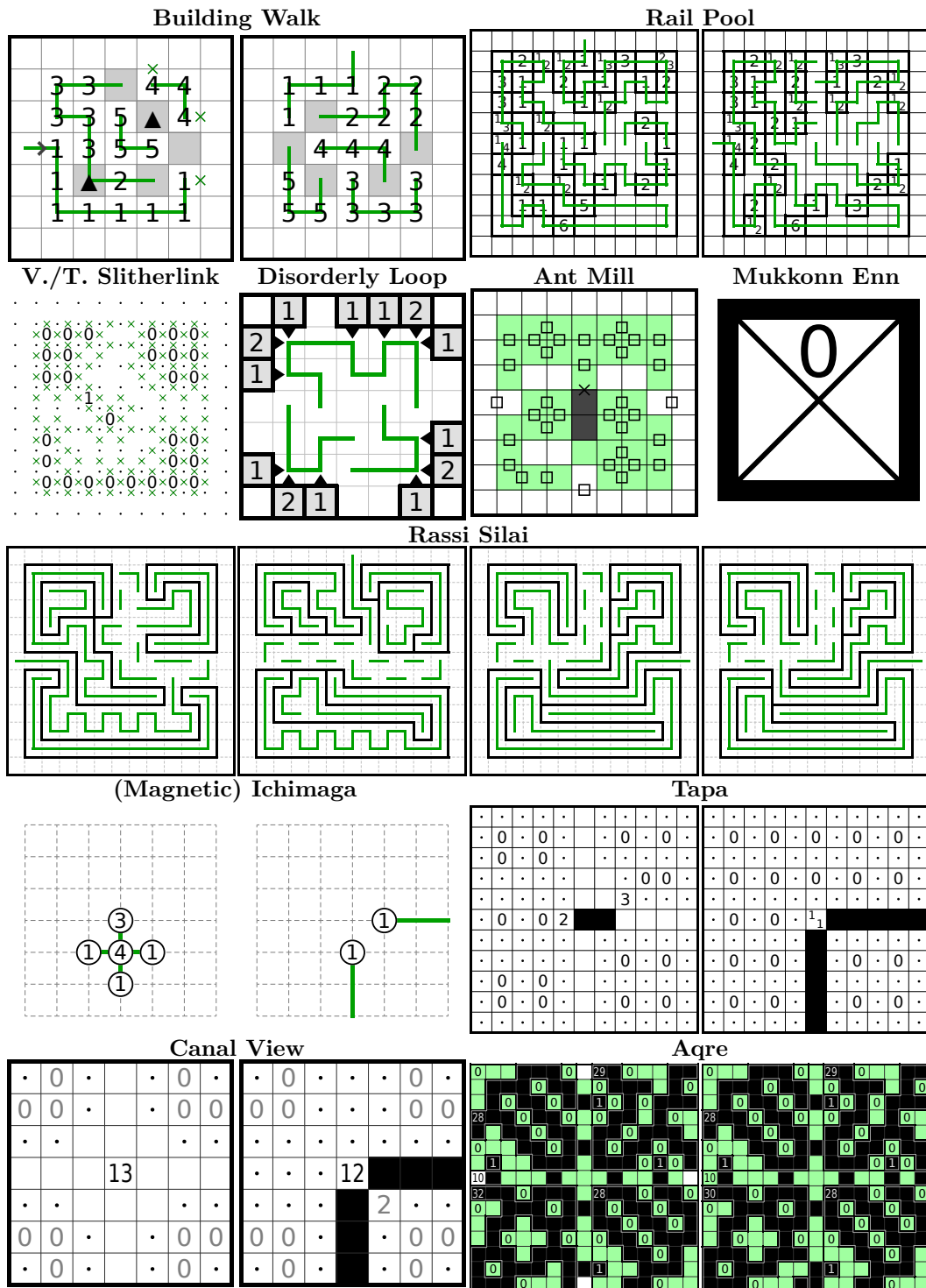
Finally, some puzzle genres were proved NP-complete by Tang, but we have not yet found parsimonious adaptations of the corresponding T-metacells. These genres are Angle Loop, Double Back, Scrin, Icebarn, and Icelom 2.

References

- 1 Zachary Abel, Jeffrey Bosboom, Michael Coulombe, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, Mikhail Rudoy, and Clemens Thielen. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. *Theoretical Computer Science*, 839:41–102, November 2020.
- 2 Takanori Akiyama, Takao Nishizeki, and Nobuji Saito. NP-completeness of the Hamiltonian cycle problem for bipartite graphs. *Journal of Information Processing*, 3(2):73–76, 1980. URL: https://ipsj.ixsq.nii.ac.jp/ej/?action=repository_action_common_download&item_id=59994&item_no=1&attribute_id=1&file_no=1.



■ **Figure 14** T-metacells proving ASP-completeness of puzzles previously only known to be NP-complete.



■ **Figure 15** T-metacells proving ASP-completeness about puzzles whose complexity had not been studied.

- 3 Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109(19):1145–1146, 2009. doi:10.1016/j.ipl.2009.07.017.
- 4 Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Sándor P. Fekete, Joseph S. B. Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. *SIAM Journal on Computing*, 35(3):531–566, 2005.
- 5 Esther M. Arkin, Sándor P. Fekete, and Joseph S.B. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry: Theory and Applications*, 17(1–2):25–50, 2000. doi:10.1016/S0925-7721(00)00015-8.
- 6 Esther M. Arkin, Sándor P. Fekete, Kamrul Islam, Henk Meijer, Joseph S. B. Mitchell, Yurái Núñez-Rodríguez, Valentin Polishchuk, David Rappaport, and Henry Xiao. Not being (super)thin or solid is hard: A study of grid hamiltonicity. *Computational Geometry: Theory and Applications*, 42(6–7):582–605, 2009. doi:10.1016/j.comgeo.2008.11.004.
- 7 Samuel W. Bent and Udi Manber. On non-intersecting Eulerian circuits. *Discrete Applied Mathematics*, 18(1):87–94, 1987. doi:10.1016/0166-218X(87)90045-X.
- 8 Jeffrey Bosboom, Charlotte Chen, Lily Chung, Spencer Compton, Michael Coulombe, Erik D. Demaine, Martin L. Demaine, Ivan Tadeu Ferreira Antunes Filho, Dylan Hendrickson, Adam Hesterberg, Calvin Hsu, William Hu, Oliver Kortén, Zhezhen Luo, and Lillian Zhang. Edge matching with inequalities, triangles, unknown shape, and two players. *Journal of Information Processing*, 28:987–1007, 2020.
- 9 Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The computational complexity of Portal and other 3D video games. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, pages 19:1–19:22, La Maddalena, Italy, June 2018.
- 10 Erik D. Demaine and Mikhail Rudoy. Hamiltonicity is hard in thin or polygonal grid graphs, but easy in thin polygonal grid graphs. arXiv:1706.10046, 2017. URL: <https://arXiv.org/abs/1706.10046>.
- 11 Erik D. Demaine and Mikhail Rudoy. Tree-Residue Vertex-Breaking: a new tool for proving hardness. In *Proceedings of the 20th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2018)*, pages 32:1–32:14, Malmö, Sweden, June 2018. Full paper at arXiv:1706.07900.
- 12 Tomás Feder and Carlos Subi. On Barnette’s conjecture. *Electronic Colloquium on Computational Complexity (ECCC)*, January 2006.
- 13 Michal Forišek. Computational complexity of two-dimensional platform games. In *Proceedings of the 5th International Conference on Fun with Algorithms*, pages 214–227, Ischia, Italy, June 2010. doi:10.1007/978-3-642-13122-6_22.
- 14 Erich Friedman. Pearl puzzles are NP-complete. Manuscript, August 2002. URL: <https://erich-friedman.github.io/papers/pearl.pdf>.
- 15 M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 47–63, Seattle, Washington, 1974. doi:10.1145/800119.803884.
- 16 M. R. Garey, D. S. Johnson, and R. Endre Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976. doi:10.1137/0205049.
- 17 Kaiying Hou and Jayson Lynch. The computational complexity of finding Hamiltonian cycles in grid graphs of semiregular tessellations. In Stephane Durocher and Shahin Kamali, editors, *Proceedings of the 30th Canadian Conference on Computational Geometry (CCCG 2018)*, pages 114–128, Winnipeg, Canada, August 2018. URL: <http://www.cs.umanitoba.ca/~cccg2018/papers/session3B-p1.pdf>.
- 18 Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. NP-completeness of two pencil puzzles: Yajilin and Country Road. *Utilitas Mathematica*, 88, June 2012. URL: <https://utilitasmathematica.com/index.php/Index/article/view/863>.
- 19 Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982. doi:10.1137/0211056.

- 20 Chuzo Iwamoto and Tatsuya Ide. Moon-or-Sun, Nagareru, and Nurimeizu are NP-complete. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 105(9):1187–1194, 2022.
- 21 Shohei Kanehiro and Yasuhiko Takenaga. Satogaeri, Hebi, and Suraromu are NP-complete. In *Proceedings of the 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*, pages 46–51, 2015.
- 22 Richard M. Karp. Reducibility among combinatorial problems. In *Proceedings of a Symposium on the Complexity of Computer Computations*, pages 85–103, Yorktown Heights, New York, March 1972. doi:10.1007/978-1-4684-2001-2_9.
- 23 Daisuke Kobayashi, Robert Vollmert, Lennard Sprong, et al. puzz.link. URL: <https://puzz.link>.
- 24 M. S. Krishnamoorthy. An NP-hard problem in bipartite graphs. *SIGACT News*, 7(1):26, January 1975. doi:10.1145/990518.990521.
- 25 Maciej Liśkiewicz, Mitsunori Ogiwara, and Seinosuke Toda. The complexity of counting self-avoiding walks in subgraphs of two-dimensional grids and hypercubes. *Theoretical Computer Science*, 304(1):129–156, 2003. doi:10.1016/S0304-3975(03)00080-X.
- 26 Mieke Maarse. The NP-completeness of some lesser known logic puzzles. Bachelor’s thesis, Utrecht University, June 2019. URL: <https://studenttheses.uu.nl/handle/20.500.12932/33867>.
- 27 Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994. doi:10.1016/S0022-0000(05)80063-7.
- 28 Christos H. Papadimitriou and Umesh V. Vazirani. On two geometric problems related to the travelling salesman problem. *Journal of Algorithms*, 5(2):231–246, 1984. doi:10.1016/0196-6774(84)90029-4.
- 29 Ján Plesník. The NP-completeness of the Hamiltonian cycle problem in planar graphs with degree bound two. *Information Processing Letters*, 8(4):199–201, April 1979. doi:10.1016/0020-0190(79)90023-1.
- 30 Takahiro Seta. The complexities of puzzles, Cross Sum, and their Another Solution Problems (ASP). Senior thesis, University of Tokyo, 2002. URL: https://web.archive.org/web/20221007013910/www-imai.is.s.u-tokyo.ac.jp/~seta/paper/senior_thesis/seniorthesis.pdf.
- 31 Hadyn Tang. On the NP-completeness of satisfying certain path and loop puzzles. arXiv:2004.12849, 2020. URL: <https://arXiv.org/abs/2004.12849>.
- 32 Hadyn Tang. A framework for loop and path puzzle satisfiability NP-hardness results. arXiv:2202.02046, 2022. URL: <https://arXiv.org/abs/2202.02046>.
- 33 Mu-Tsun Tsai and Douglas B. West. A new proof of 3-colorability of Eulerian triangulations. *Ars Mathematica Contemporanea*, 4(1):73–77, 2011. doi:10.26493/1855-3974.193.8E7.
- 34 W. T. Tutte. On Hamiltonian circuits. *Journal of the London Mathematical Society, Series 1*, 21(2):98–101, 1946. doi:10.1112/jlms/s1-21.2.98.
- 35 Akihiro Uejima, Hiroaki Suzuki, and Atsuki Okada. The complexity of generalized pipe link puzzles. *Journal of Information Processing*, 25:724–729, 2017.
- 36 Takayuki Yato. On the NP-completeness of the Slither Link puzzle. *IPSJ SIG Notes*, AL-74:25–32, 2000.
- 37 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*, E86-A(5):1052–1060, 2003. Also IPSJ SIG Notes 2002-AL-87-2, 2002. URL: <http://ci.nii.ac.jp/naid/110003221178/en/>.


Tetris with Few Piece Types

MIT Hardness Group¹

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Erik D. Demaine  

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Holden Hall 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Jeffery Li 

CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We prove NP-hardness and #P-hardness of Tetris clearing (clearing an initial board using a given sequence of pieces) with the Super Rotation System (SRS), even when the pieces are limited to *any two* of the seven Tetris piece types. This result is the first advance on a question posed twenty years ago: which piece sets are easy vs. hard? All previous Tetris NP-hardness proofs used five of the seven piece types. We also prove ASP-completeness of Tetris clearing, using three piece types, as well as versions of 3-Partition and Numerical 3-Dimensional Matching where all input integers are distinct. Finally, we prove NP-hardness of Tetris survival and clearing under the “hard drops only” and “20G” modes, using two piece types, improving on a previous “hard drops only” result that used five piece types.

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

Keywords and phrases complexity, hardness, video games, counting








Digital Object Identifier 10.4230/LIPIcs.FUN.2024.24

Related Version *Full Version:* <https://arxiv.org/abs/2404.10712>

Acknowledgements This paper was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.5440) taught by Erik Demaine in Fall 2023. We thank the other participants of that class for helpful discussions and providing an inspiring atmosphere. Figures drawn with SVG Tiler (<https://github.com/edemaine/svgtiler>).



1 Introduction















Tetris is one of the oldest and most popular puzzle video games, originally created by Alexey Pajitnov in 1984. Tetris has reached mainstream media many times, most recently in the biopic *Tetris* [1] and with the news of 13-year-old Willis Gibson being the first person to “beat” the NES version of Tetris by reaching a killscreen [6].

The rules of Tetris are simple. In each round, a tetromino piece (one of , , , , , , ) spawns at the top of a grid and periodically moves down one unit, assuming the squares below the piece are empty. The player can repeatedly move this piece one unit left, one unit right, or one unit down, or rotate the piece by $\pm 90^\circ$. When any part of the piece rests on top of a filled square for long enough that it triggers an automatic downward move, the piece “locks” in place, and stops moving. If a piece stops above a certain height or where the next piece would spawn, the player loses; otherwise, the next

¹ Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as “MIT Hardness Group” (without “et al.”).



■ **Table 1** Our NP-hardness results for Tetris clearing assuming SRS. Each entry in a specific row and column corresponds to the proposition for the hardness of the two-element subset consisting of the row piece and column piece (for example, the entry “Prop. 6” in row  and column ) indicates that Proposition 6 proves hardness for the subset $\{\text{cyan 1x4}, \text{yellow 2x2}\}$. Letters in parentheses denote additional models (“H” for “hard drop only”, “G” for “20G”); question mark indicates a conjecture for hardness under that additional model.

							
	–	Prop. 6 (H)	Prop. 6 (H)	Prop. 6	Prop. 6	Prop. 6 (H, G)	Prop. 6 (H, G)
	–	–	Prop. 9	Prop. 8	Prop. 8	Prop. 7	Prop. 7
	–	–	–	Prop. 10	Prop. 10	Prop. 12	Prop. 12
	–	–	–	–	Prop. 11	Prop. 13	Prop. 12
	–	–	–	–	–	Prop. 12	Prop. 13
	–	–	–	–	–	–	Prop. 14 (G?)
	–	–	–	–	–	–	–

piece spawns at the top of the grid, and play continues. Completely filling a row causes the row to clear, and all squares above that row move downward by one unit. For more detailed rules, see [16].

To study Tetris from a computational complexity perspective, we generally assume that the player is given a sequence of pieces and an initial board state of filled cells, making the game perfect information (as introduced in [4]). The two main objectives we consider here are “clearing” and “survival” (as introduced in [7]). In *Tetris clearing*, we want to determine whether we can clear the entire board after placing all the given pieces. In *Tetris survival*, we want to determine whether the player can avoid losing before placing all the given pieces. Previous work shows that these problems are NP-complete, even to approximate various metrics within $n^{1-\epsilon}$ [4], or with only 8 columns or 4 rows [2], or with additional constraints on drops [12], or with k -ominoes for $k \geq 3$ clearing or $k \geq 4$ survival [7].

1.1 Our Results

One of the open problems posed in the original paper proving Tetris NP-hard twenty years ago [4] is to determine which subsets of the seven Tetris piece types $\{\text{cyan 1x4}, \text{yellow 2x2}, \text{purple T}, \text{green 2x2}, \text{red 2x2}, \text{blue L}, \text{orange 2x2}\}$ suffice for NP-hardness, and which admit a polynomial-time algorithm. All existing Tetris NP-hardness proofs [4, 2, 12] use at least five of the seven piece types. In particular, [4, Section 6.2] mentions various sets of five piece types that suffice. What about fewer piece types?

Our main results are the first to make progress on this question: for any *size-2* subset $A \subseteq \{\text{cyan 1x4}, \text{yellow 2x2}, \text{purple T}, \text{green 2x2}, \text{red 2x2}, \text{blue L}, \text{orange 2x2}\}$, Tetris clearing is NP-complete with pieces restricted to A . Most pairs of piece types require different constructions for their reductions; refer to Table 1. Our results require us to specify more details of the piece rotation model, specifically what happens when the player rotates a piece in a way that collides with a filled square. We assume the *Super Rotation System (SRS)* [14], first introduced in the 2001 game *Tetris Worlds* and as part of the Tetris Company’s Tetris Guideline for how all modern (2001+) Tetris games should behave [15].

For every *size-2* subset A of piece types, we also establish #P-hardness for the corresponding problem of counting the number of ways to clear the board. Here we distinguish solutions by the final placement of each piece, not the sequence of moves to make those placements (as long as the placement is valid). This definition lets us ignore e.g. the null effect of moving a piece repeatedly left and right.

For certain size-3 subsets of piece types, we further establish ASP-completeness for Tetris clearing. Recall that an NP search problem is *ASP-complete* [17] if there is a parsimonious reduction from all NP search problems (including a polynomial-time bijection between solutions). In particular, ASP-completeness implies NP-hardness of finding another solution given k solutions, for any $k \geq 0$, as well as #P-completeness. These results hold for piece types $\{\text{I}, \text{O}, \text{S}\}$ and $\{\text{I}, \text{S}, \text{Z}\}$.

We also study Tetris under two more restrictive models on piece moves:

- **Hard drops only:** In this model, pieces do not move downward on their own, and if the player moves a piece downward, the piece moves maximally downward before locking into place (a *hard drop* maneuver). The player is still free to rotate or move the piece left or right before hard-dropping the piece. This model is motivated by most Tetris games awarding higher scores for hard drops, and was posed in [4].
- **20G:** In this model, instead of periodically moving down one unit, all pieces move maximally downward *instantly and on their own*, and the player is not allowed to control how fast a piece moves downward. The player is still free to rotate or move the piece left or right before the piece locks. This model is motivated by levels with the maximum possible gravity, as in Level 20+ of regular Tetris with 20 rows [13].

For certain size-2 subsets of piece types, we establish NP-hardness of both Tetris survival and clearing under either of these models. Table 1 labels which of our Tetris clearing results hold in which models.

Along the way, we prove new results about 3-Partition and Numerical 3-Dimensional Matching (3DM): both problems are strongly ASP-complete even when all integers are assumed distinct. These results are of independent interest for ASP-hardness reductions. Previously, these problems were known to be ASP-complete with multisets of integers [3], and strongly NP-complete with distinct integers [10].

1.2 Outline

The structure of the rest of the paper is as follows. Section 2 details the Super Rotation System (SRS), an important aspect of modern Tetris and used in our constructions. Section 3 proves ASP-completeness of 3-Partition with Distinct Integers and Numerical 3-Dimensional Matching with Distinct Integers, two problems we reduce from. Section 4 discusses our hardness results for Tetris clearing with SRS with only two piece types. Section 5 discusses some Tetris survival results under the “hard drops only” and “20G” models. Section 6 proves ASP-completeness of Tetris clearing with SRS.

2 Super Rotation System (SRS)

Most previous Tetris results are not sensitive to exactly how Tetris pieces rotate: most reasonable rotation models work [4, Section 6.4]. By contrast, many of the results in this paper focus specifically on (and require) the *Super Rotation System (SRS)* [14], defined as follows.

Each piece has a defined *rotation center*, as indicated by dots in Figure 1, except for I and O , whose rotation centers are the centers of the 4×4 squares in Figure 1. When unobstructed, all non- O tetrominoes will rotate purely about the rotation center (note that O pieces cannot rotate). The key feature about SRS is *kicking*: if a tetromino is obstructed when a rotation is attempted, the game will attempt to “kick” the tetromino into one of four alternate positions, each tested sequentially; if all four positions do not work,

24:4 Tetris with Few Piece Types

then the rotation will fail. See Figure 2 for an example of this kicking process. The full data for wall kicks can be found in Tables 2 and 3, and at [14]. Of note is that SRS wall kicks are vertically symmetric for all pieces or pairs of pieces (i.e., $\begin{smallmatrix} \blacksquare \\ \blacksquare \\ \blacksquare \end{smallmatrix} \leftrightarrow \begin{smallmatrix} \blacksquare \\ \blacksquare \\ \blacksquare \end{smallmatrix}$ and $\begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix} \leftrightarrow \begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$) except for the $\begin{smallmatrix} \blacksquare & \blacksquare & \blacksquare \end{smallmatrix}$ piece, so all rotations can be mirrored.

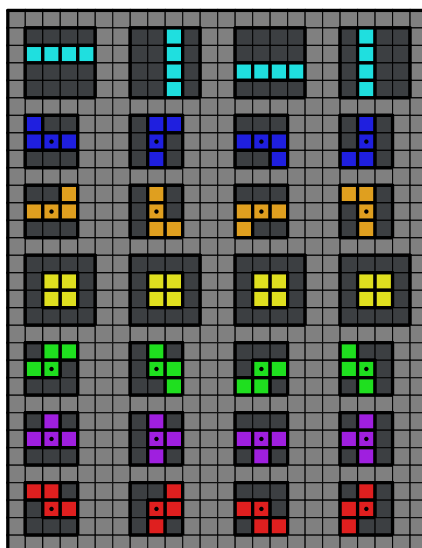


Figure 1 All tetromino pieces, in order from top to bottom: $\begin{smallmatrix} \blacksquare & \blacksquare & \blacksquare \end{smallmatrix}$, $\begin{smallmatrix} \blacksquare \\ \blacksquare \\ \blacksquare \end{smallmatrix}$, $\begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$, $\begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$, $\begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$, $\begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$, $\begin{smallmatrix} \blacksquare & \blacksquare \\ \blacksquare & \blacksquare \end{smallmatrix}$. The first column is the default orientation of a piece upon spawning in; each column to the right indicates a 90° rotation clockwise about the rotation center of the piece.

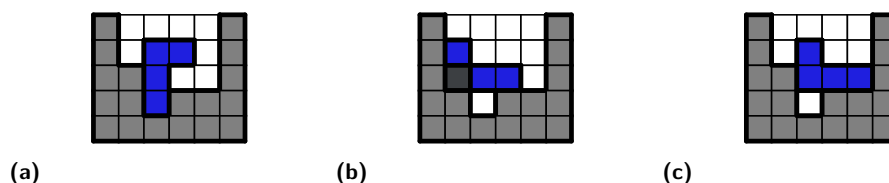







Figure 2 An example of the SRS kick system. Suppose the $\begin{smallmatrix} \blacksquare \\ \blacksquare \\ \blacksquare \end{smallmatrix}$ piece in (a) is being rotated 90° counter-clockwise. Test 1 (which is $(0,0)$) would fail, due to the dark gray square shown in (b). Test 2 (which is $(+1,0)$) would succeed, as shown in (c), and so the $\begin{smallmatrix} \blacksquare \\ \blacksquare \\ \blacksquare \end{smallmatrix}$ piece would rotate to the position in (c).

This system of kicking tetrominoes during rotations allows for moves which are often called *twists* or *spins*. All the spins that we utilize are detailed in the appendix of the full version of our paper.

3 3-Partition and Numerical 3DM with Distinct Integers

Our reductions to Tetris are all from one of the following two problems, which are strengthenings of two standard strongly NP-complete problems:

► **Definition 1 (3-Partition with Distinct Integers).** Given a set $A = \{a_1, a_2, \dots, a_n\}$ of n distinct positive integers such that $\frac{t}{4} < a_i < \frac{t}{2}$ for each i , where $t = \frac{3}{n} \sum_{i=1}^n a_i$, determine whether there is a partition of A into $\frac{n}{3}$ groups $D_1, \dots, D_{n/3}$ (each necessarily of size 3) having the same sum $\sum_{x \in D_j} x = t$.

■ **Table 2** Kick data for , , , , and  pieces. 0 indicates the default orientation, and R , 2 , and L indicate the orientation reached from a 90° , 180° , and 270° rotation clockwise (respectively) from the default orientation. An ordered pair (a, b) denotes a translation of the center by a units in the x direction and b units in the y direction. Positive x direction is rightwards, and positive y direction is upward.

	Test 1	Test 2	Test 3	Test 4	Test 5
$0 \rightarrow R$	(0, 0)	(-1, 0)	(-1, +1)	(0, -2)	(-1, -2)
$R \rightarrow 0$	(0, 0)	(+1, 0)	(+1, -1)	(0, +2)	(+1, +2)
$R \rightarrow 2$	(0, 0)	(+1, 0)	(+1, -1)	(0, +2)	(+1, +2)
$2 \rightarrow R$	(0, 0)	(-1, 0)	(-1, +1)	(0, -2)	(-1, -2)
$2 \rightarrow L$	(0, 0)	(+1, 0)	(+1, +1)	(0, -2)	(+1, -2)
$L \rightarrow 2$	(0, 0)	(-1, 0)	(-1, -1)	(0, +2)	(-1, +2)
$L \rightarrow 0$	(0, 0)	(-1, 0)	(-1, -1)	(0, +2)	(-1, +2)
$0 \rightarrow L$	(0, 0)	(+1, 0)	(+1, +1)	(0, -2)	(+1, -2)

■ **Table 3** Kick data for  pieces, with same notation as Table 2.

	Test 1	Test 2	Test 3	Test 4	Test 5
$0 \rightarrow R$	(0, 0)	(-2, 0)	(+1, 0)	(-2, -1)	(+1, +2)
$R \rightarrow 0$	(0, 0)	(+2, 0)	(-1, 0)	(+2, +1)	(-1, -2)
$R \rightarrow 2$	(0, 0)	(-1, 0)	(+2, 0)	(-1, +2)	(+2, -1)
$2 \rightarrow R$	(0, 0)	(+1, 0)	(-2, 0)	(+1, -2)	(-2, +1)
$2 \rightarrow L$	(0, 0)	(+2, 0)	(-1, 0)	(+2, +1)	(-1, -2)
$L \rightarrow 2$	(0, 0)	(-2, 0)	(+1, 0)	(-2, -1)	(+1, +2)
$L \rightarrow 0$	(0, 0)	(+1, 0)	(-2, 0)	(+1, -2)	(-2, +1)
$0 \rightarrow L$	(0, 0)	(-1, 0)	(+2, 0)	(-1, +2)	(+2, -1)

► **Problem 2 (Numerical 3-Dimensional Matching (3DM) with Distinct Integers).** *Given three sets*

$$A = \{a_1, a_2, \dots, a_n\}, B = \{b_1, b_2, \dots, b_n\}, \text{ and } C = \{c_1, c_2, \dots, c_n\}$$

*of n positive integers, where all $3n$ integers are **distinct**, and a target sum $t = \frac{1}{n} \sum_{i=1}^n (a_i + b_i + c_i)$, determine whether there is a partition of $A \cup B \cup C$ into n groups D_1, \dots, D_n , each with exactly one element from each of A , B , and C , and $\sum_{x \in D_j} x = t$ for all j .*

Without the “distinct” and “set” conditions, both problems are well-known to be **strongly NP-complete**, meaning that the problem is NP-hard even if the a_i integers are bounded by a polynomial in n . This property makes it feasible to represent each integer a_i (and t) in unary, which is the approach taken by all past Tetris NP-hardness proofs [4, 7, 2, 12], as then the total reduction size is still polynomial in n .

We want to ensure all integers are distinct in order to have more control over our reductions’ blowup in the number of solutions, as needed for #P- and ASP-hardness. Bosboom et al. [3] proved that numerical 3DM is strongly ASP-complete when A is restricted to be a set, but allowed for B and C to be multisets as usual, and did not forbid repeated integers between A, B, C . Hulett, Will, and Woeginger [10] proved that both 3-Partition and Numerical 3DM remain strongly NP-hard with distinct integers. We extend their proof to obtain ASP-completeness:

► **Theorem 3.** *3-Partition with Distinct Integers, and Numerical 3-Dimensional Matching with Distinct Integers, are strongly ASP-complete.*

To prove this result, we use the following intermediate problems (which are thus also ASP-complete):

► **Problem 4 (Positive 1-in-3SAT).** *Given a boolean formula in 3CNF (i.e., an AND of clauses consisting of 3 literals), where all literals are positive, does there exist an assignment of the variables to either true or false such that each clause has exactly one literal set to true?*

► **Problem 5 (Tripartite Edge-Disjoint Triangle Partition).** *Given an undirected tripartite graph $G = (V, E)$, can we partition E into disjoint triangles?*

Proof of Theorem 3. We give a chain of parsimonious reductions from 3SAT, which is known to be ASP-complete [17]:

1. **3SAT** → **Positive 1-in-3SAT:** Hunt, Marathe, Radhakrishnan, and Stearns [11, Theorem 3.8] gave such a parsimonious reduction.² See also [3, Lemma 2.1].
2. **Positive 1-in-3SAT** → **Tripartite Edge-Disjoint Triangle Partition:** We follow a simplification of a reduction from FCP 1-in-3SAT to FCP Tripartite Edge-Disjoint Triangle Partition [8, Theorem 12], which in turn is based on a reduction from 3SAT to Tripartite Edge-Disjoint Triangle Partition [9]. By reducing from Positive 1-in-3SAT, we simplify the reduction of [8] by avoiding negative literals.

We represent each variable by a sufficiently large triangular grid of vertices, with opposite sides of a parallelogram identified to form a flat torus, as shown in Figure 3a. This grid has exactly two solutions, corresponding to true (the triangles in Figure 3a) and false (the triangles in Figure 3b); note that the two solutions consist of exactly the same edges, and cover each exactly once. For each clause (x, y, z) , we pick one triangle of positive orientation, remove its edges, and unify the corresponding vertices of these triangles and of the three neighboring triangles of negative orientation, as shown in Figure 3c. Exactly one variable must choose the true state so as to cover the edges surrounding the unified hole exactly once. By choosing the variable gadgets large enough, we can ensure that the clause gadgets are disjoint from each other. Each gadget has a unique way to implement a given assignment, so this reduction is parsimonious.

3. **Tripartite Edge-Disjoint Triangle Partition** → **Numerical 3DM with Distinct Integers:** We combine a chain of reductions, from Triangle Edge-Disjoint Triangle Partition to Latin Square Completion [5], and from Latin Square Completion to Numerical 3DM with Distinct Integers [10].

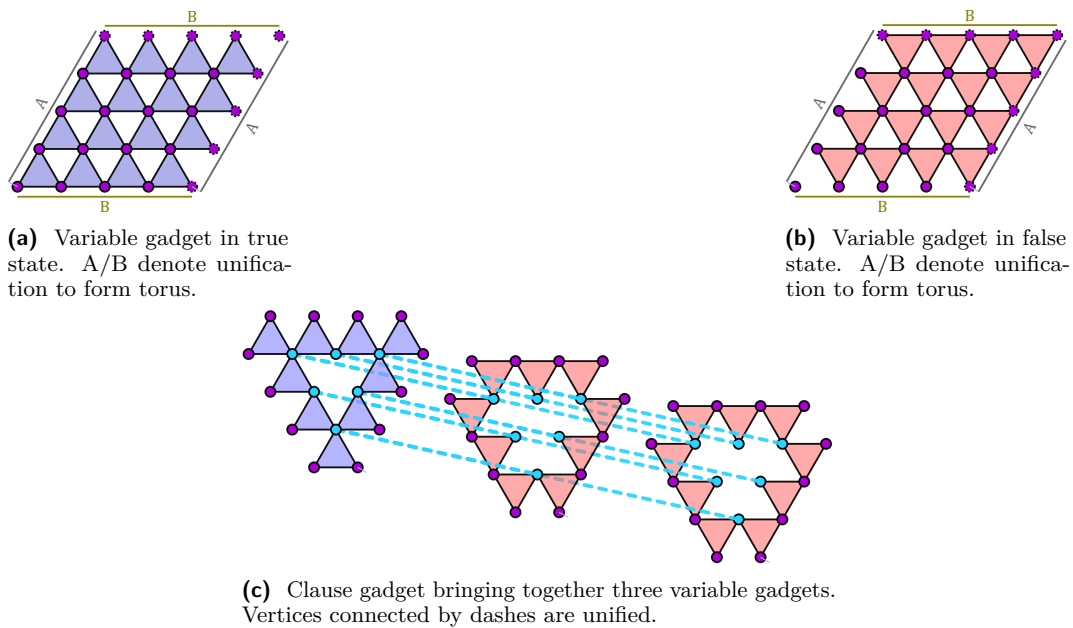
If $U = \{u_1, u_2, \dots\}$, $V = \{v_1, v_2, \dots\}$, $W = \{w_1, w_2, \dots\}$ is the vertex tripartition, then we do the following:

- Let $q = 2 \max\{|U|, |V|, |W|\}$, and let the target sum be $t = 19q^6$.
- Map each edge (u_i, w_k) to $2q^6 + iq - k \in A$.
- Map each edge (v_j, w_k) to $7q^6 + jq^2 + k \in B$.
- Map each edge (u_i, v_j) to $t - (9q^6 + jq^2 + iq) = 10q^6 - jq^2 - iq \in C$.

The lemmas in [10] show that all the integers in A , B , and C are distinct (i.e., we have a valid instance of Numerical 3DM with Distinct Integers); and that any triple summing to t consists of one element each from A , B , and C , with the elements corresponding to a triangle in the graph. Thus we obtain a bijection between triangle partitions and Numerical 3DM solutions, i.e., the reduction is parsimonious.

4. **Numerical 3DM with Distinct Integers** → **3-Partition with Distinct Integers:** We use standard techniques to relate these problems. Convert each integer a_i , b_i , and c_i in Numerical 3DM to integers $8a_i + 1$, $8b_i + 2$, and $8c_i - 3$, respectively, in 3-Partition;

² Their problem “1-EX3MONOSAT” is Positive 1-in-3SAT with the additional constraint that every clause has exactly three literals. Their reduction is also planarity preserving, so chaining with their parsimonious reduction from 3SAT to Planar 3SAT, we obtain that Planar 1-in-3SAT is also ASP-complete.



■ **Figure 3** Reduction from Positive 1-in-3SAT to Tripartite Edge-Disjoint Triangle Partition.

and convert t to $8t$. In particular, all integers are still distinct, because we scale up by a factor of 8 and then shift values by less than 4. Furthermore, working modulo 8, every triple of integers summing to t must take exactly one a_i , one b_j , and one c_k . Therefore we have a parsimonious reduction.





Composing these reductions, we obtain that 3-Partition with Distinct Integers, and Numerical 3DM with Distinct Integers, are ASP-hard. Both problems are NP search problems, so they are ASP-complete. ◀

4 Tetris with Two Piece Types

In this section, we will prove that for any size-2 subset $A \subseteq \{ \text{cyan 1x4}, \text{yellow 2x2}, \text{purple 2x3}, \text{green 3x2}, \text{red 3x3}, \text{blue 2x2}, \text{orange 2x2} \}$, Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the sequence of pieces given to the player only contains the piece types in A . We will also show that some of the reductions work under the “hard drop only” model and the “20G” model. Refer to Table 1 for a table of all of our results.

All of our reductions are from 3-Partition with Distinct Integers and are in the same flavor as the reduction for clearing 3-tris with rotation as given in the Total Tetris paper [7], which we will use some terminology from. In particular, the reductions will involve a starting board involving $\frac{n}{3}$ structures, which we will call “*bottles*”, of equal height of $\Theta(t \cdot \text{poly}(n))$, spaced sufficiently far apart so that bottles do not interact with each other except for line clears, and possibly along with an additional structure, which we will call a “*finisher*”, to the right of the rightmost bottle.

Each bottle consists of a neck portion with n constant-sized “*top segments*”, a body portion with $t \text{ poly}(n)$ -sized “*units*”, and possibly $O(n)$ extra lines either above the neck portion, between the “top segments”, between the neck portion and the body portion, and/or below the body portion that get cleaned up after the rest of the lines. To simplify our arguments, we make the size of each unit larger than the size of the neck portion.

The finisher will be a structure that specifically prevents the rows in the body portion from clearing before all of the top segments are cleared, and is located in same rows as the body portion of the bottles when required. We will use three types of finishers, a  finisher, an  finisher (which is a vertically symmetric version of the  finisher), and a  finisher, shown in Figure 4. Note that the finishers can be adapted to any number of rows larger than 4, and there is exactly one way to clear each type of finisher.

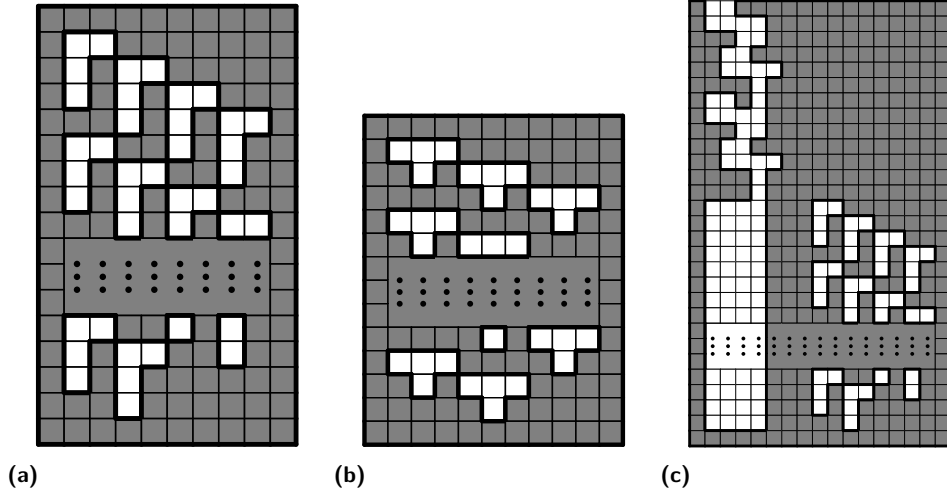









Figure 4 The  and  finishers (the  finisher can be obtained by reflecting the  finisher through a vertical line), and an example of the  finisher next to a bottle (in the ,  setup).

For each element $a_i \in A$, we create a sequence of pieces S_i , which can be decomposed into three subsequences:

- **Priming sequence:** A piece sequence that, if used correctly, properly blocks all bottles but one in the same “top segment”, and if used incorrectly, either directly “overflows” the bottle (i.e., puts blocks above the line under which all of our pieces must go) or “clogs” the bottle (i.e., improperly blocks the bottle and prevents the player from being able to clear the lines necessary to re-open the bottle). For all of the bottle structures except for the one for $\{\text{yellow square}, \text{purple T}\}$, the pieces in the priming sequence cannot rotate or translate below the topmost “top segment” under SRS, and any piece placed into a “top segment” of a bottle prevents any piece in the filling sequence from reaching the body portion of that bottle.
- **Filling sequence:** A piece sequence of length $\Theta(a_i)$ that “fills” a_i units in the body portion of the unblocked bottle. If there are not enough units left to fill, then the pieces corresponding to one of the units will cause an overflow due to there not being enough empty space in the neck portion for all of the pieces (using the fact that the size of each unit is larger than the size of the neck portion).
- **Closing sequence:** A piece sequence that properly clears the lines corresponding to the “top segment” blocked by the priming sequence and resets the states of the neck portion of the bottles (albeit with one less “top segment”).

We also have a *finale sequence* F , possibly the empty sequence, which helps clear any finishers on the board and the remaining lines on the board after the lines corresponding to the neck and body portions have been cleared.

In this section, when we write a sequence of pieces, we will use parentheses around sequences, commas between different piece types, and exponentiation to denote repeated pieces of the same piece type. For example, a sequence written as $(\text{I}^2, \text{J}^3, \text{K})$ consists of 2 I s, 3 J s, and an K , in that order. The sequence of pieces given to the player will be of the form $(S_1, S_2, \dots, S_n, F)$.

4.1 General Argument

We provide a very general argument for why these reductions work. If there exists a valid 3-partition $(D_1, \dots, D_{n/3})$ for $\{a_1, a_2, \dots, a_n\}$, then for each S_i , determine the corresponding j_i such that $a_i \in D_{j_i}$, then use the priming sequence to block all bottles properly except for the (j_i) th one, the filling sequence to fill a_i units in the body portion of the (j_i) th bottle, and the closing sequence to reset the states of the neck portion of the bottles. After all the S_i are used in this way, all lines corresponding to the “top segments” will be cleared as there are n such “top segments” with each S_i clearing exactly one of them, and each bottle will be filled to exactly t units. Thus, in the case where there are no pieces in the finale sequence, the lines corresponding to the body portions of the bottles will be cleared, meaning that no lines remain and we have cleared the board, and in the case where there are pieces in the finale sequence, the only lines that remain are those that can be cleared by the finale sequence. Thus, the sequence $(S_1, S_2, \dots, S_n, F)$ can clear the board.

Conversely, if the sequence $(S_1, S_2, \dots, S_n, F)$ can clear the board, then we claim that there is a corresponding 3-partition for $\{a_1, a_2, \dots, a_n\}$. In particular, for each S_i , the priming sequence must properly block all but one bottle, say the (j_i) th bottle, forcing all the pieces in the filling sequence into the (j_i) th bottle. The filling sequence must then fill exactly a_i units in the (j_i) th bottle before the closing sequence, and it must do so without overfilling the body portion of the bottle, as otherwise there will be an overflow in that bottle. In particular, this means that, for each $1 \leq j \leq \frac{n}{3}$, the sum of the a_i corresponding to the S_j that filled some units in the j th bottle must be at most t . However, since $\sum a_i$ is exactly $t(\frac{n}{3})$, the sum of the a_i corresponding to the S_j that filled some units in the j th bottle must actually be exactly t . In other words, there is a way to partition the a_i into $\frac{n}{3}$ subsets $D_1, \dots, D_{n/3}$ such that the sum of the elements in each subset is t . Thus, there is a corresponding 3-partition for $\{a_1, a_2, \dots, a_n\}$.

This general argument shows how YES instances of the two problems (3-Partition with Distinct Integers, Tetris clearing with SRS and restricted piece types) are equivalent, and hence that this reduction works. The rest of the subsections in this section show the bottle structures for each size-2 subset of piece types. Due to space constraints, we omit detailed construction-specific explanations; refer to the full version of our paper for more details.

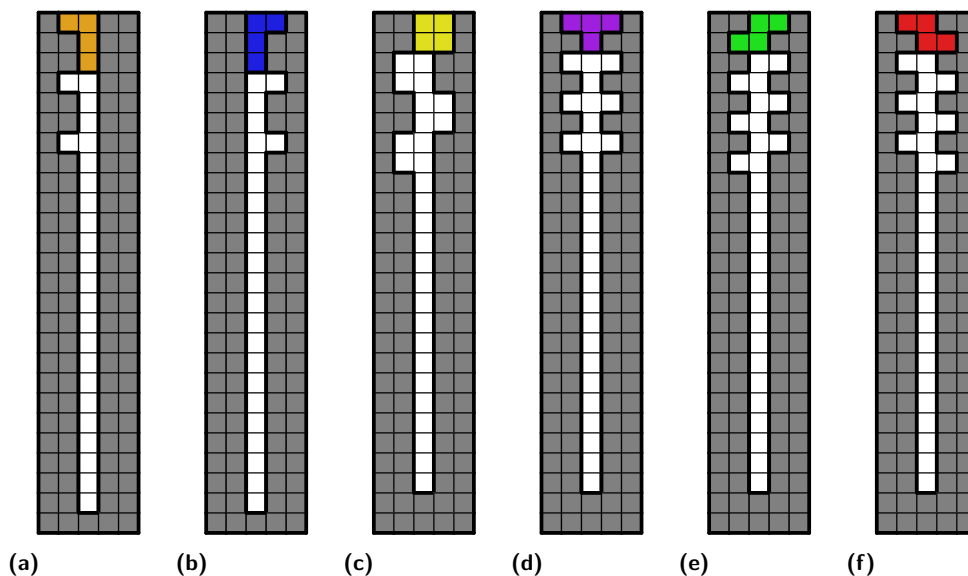
4.2 Subsets with IIII Pieces



First we show how the reduction in the Total Tetris paper [7] can be easily adapted to any subset of pieces with IIII pieces plus an additional piece type:

► **Proposition 6.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if:*


- The type of pieces in the sequence given to the player is restricted to any of $\{\text{IIII}, \text{L}\}$, $\{\text{IIII}, \text{O}\}$, $\{\text{IIII}, \text{J}\}$, $\{\text{IIII}, \text{K}\}$, $\{\text{IIII}, \text{I}\}$, or $\{\text{IIII}, \text{Z}\}$,
- The model being considered is “hard drops only” and type of pieces in the sequence given to the player is restricted to any of $\{\text{IIII}, \text{L}\}$, $\{\text{IIII}, \text{O}\}$, $\{\text{IIII}, \text{J}\}$, or $\{\text{IIII}, \text{I}\}$, or
- The model being considered is “20G” and the type of pieces in the sequence given to the player is restricted to either $\{\text{IIII}, \text{L}\}$ or $\{\text{IIII}, \text{O}\}$.

Refer to Figure 5 for the bottle structures.





■ **Figure 5** The bottle structures for the subsets containing , including how the non- piece must block a bottle during the priming and closing sequence.

4.3 Other Subsets with Pieces

We now move on to all the remaining subsets which contain  pieces.

► **Proposition 7.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to either $\{\text{yellow } 2 \times 2, \text{blue } 2 \times 2\}$ or $\{\text{yellow } 2 \times 2, \text{orange } 2 \times 2\}$.*

Refer to Figure 6a, which shows the bottle structure for $\{\text{yellow } 2 \times 2, \text{blue } 2 \times 2\}$. The bottle structure for $\{\text{yellow } 2 \times 2, \text{orange } 2 \times 2\}$ can be obtained by reflecting the bottle structure for $\{\text{yellow } 2 \times 2, \text{blue } 2 \times 2\}$ through a vertical line. We will also use a  (or ) finisher in our setup to prevent rows in the body portion from clearing early.

A demo of the $\{\text{yellow } 2 \times 2, \text{blue } 2 \times 2\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/80188>.

► **Proposition 8.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to either $\{\text{yellow } 2 \times 2, \text{green } 2 \times 2\}$ or $\{\text{yellow } 2 \times 2, \text{red } 2 \times 2\}$.*

Refer to Figure 6b, which shows the bottle structure for $\{\text{yellow } 2 \times 2, \text{green } 2 \times 2\}$. The bottle structure for $\{\text{yellow } 2 \times 2, \text{red } 2 \times 2\}$ can be obtained by reflecting the bottle structure for $\{\text{yellow } 2 \times 2, \text{green } 2 \times 2\}$ through a vertical line. We do not use a finisher in our setup.

A demo of the $\{\text{yellow } 2 \times 2, \text{green } 2 \times 2\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/81818>.

► **Proposition 9.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to $\{\text{yellow } 2 \times 2, \text{purple } 2 \times 2\}$.*

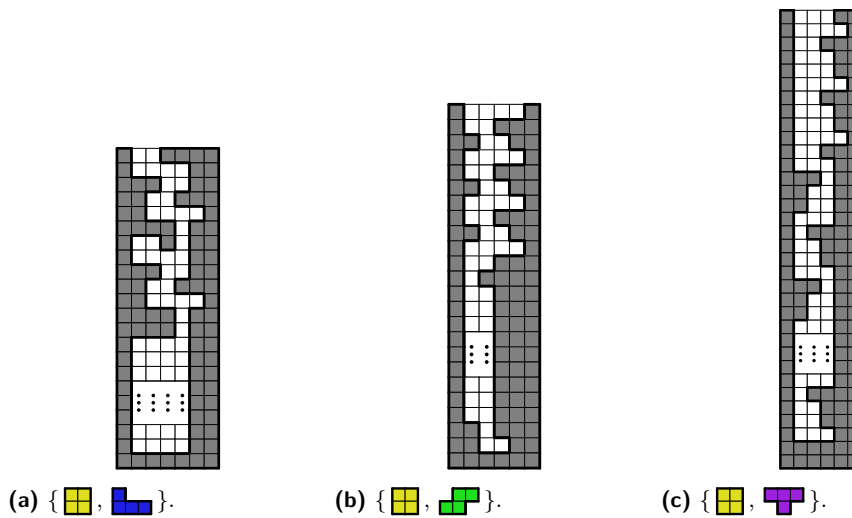




Figure 6 The bottle structures for the other subsets containing ; the bottle structures for $\{\langle \text{yellow square}, \text{orange horizontal piece} \rangle\}$ and $\{\langle \text{yellow square}, \text{red L-shaped piece} \rangle\}$ can be obtained by reflecting the bottle structures for $\{\langle \text{yellow square}, \text{blue L-shaped piece} \rangle\}$ and $\{\langle \text{yellow square}, \text{green Z-shaped piece} \rangle\}$ through a vertical line.

Refer to Figure 6c for the bottle structure. We will also use a  finisher in our setup to prevent rows in the body portion from clearing early.

A demo of the $\{\langle \text{yellow square}, \text{purple T-shaped piece} \rangle\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/80169>.

4.4 Two-Element Subsets of $\{\langle \text{green Z-shaped piece}, \text{purple T-shaped piece}, \text{red L-shaped piece} \rangle\}$

► **Proposition 10.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to either $\{\langle \text{green Z-shaped piece}, \text{purple T-shaped piece} \rangle\}$ or $\{\langle \text{purple T-shaped piece}, \text{red L-shaped piece} \rangle\}$.*

Refer to Figure 7a, which shows the bottle structure for $\{\langle \text{green Z-shaped piece}, \text{purple T-shaped piece} \rangle\}$. The bottle structure for $\{\langle \text{purple T-shaped piece}, \text{red L-shaped piece} \rangle\}$ can be obtained by reflecting the bottle structure for $\{\langle \text{green Z-shaped piece}, \text{purple T-shaped piece} \rangle\}$ through a vertical line. We do not use a finisher in our setup.

A demo of the $\{\langle \text{green Z-shaped piece}, \text{purple T-shaped piece} \rangle\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/80184>.

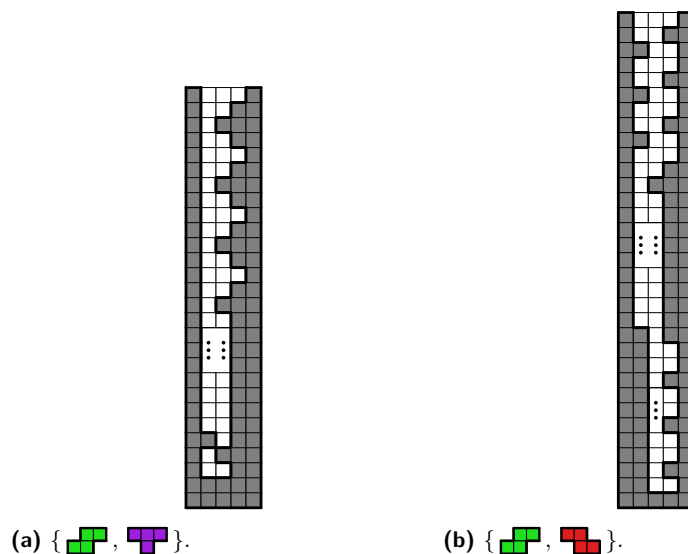
► **Proposition 11.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to $\{\langle \text{green Z-shaped piece}, \text{red L-shaped piece} \rangle\}$.*

Refer to Figure 7b for the bottle structure. We do not use a finisher in our setup.

A demo of the $\{\langle \text{green Z-shaped piece}, \text{red L-shaped piece} \rangle\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/80198>.

4.5 Remaining Subsets with More Complex Structures

► **Proposition 12.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to any of $\{\langle \text{blue L-shaped piece}, \text{purple T-shaped piece} \rangle\}$, $\{\langle \text{blue L-shaped piece}, \text{red L-shaped piece} \rangle\}$, $\{\langle \text{orange horizontal piece}, \text{green Z-shaped piece} \rangle\}$, or $\{\langle \text{orange horizontal piece}, \text{purple T-shaped piece} \rangle\}$.*



■ **Figure 7** The bottle structures for the two-element subsets of $\{\text{green}, \text{purple}, \text{red}\}$; the bottle structure for $\{\text{purple}, \text{red}\}$ can be obtained by reflecting the bottle structure for $\{\text{green}, \text{purple}\}$ through a vertical line.

Refer to Figure 8a, which shows the bottle structure for $\{\text{blue}, \text{red}\}$ and $\{\text{blue}, \text{purple}\}$. The bottle structure for $\{\text{orange}, \text{green}\}$ and $\{\text{orange}, \text{purple}\}$ can be obtained by reflecting the bottle structure for $\{\text{blue}, \text{red}\}$ and $\{\text{blue}, \text{purple}\}$ through a vertical line. We will also use a blue (or orange) finisher in our setup to prevent rows in the body portion from clearing early.

A demo of the $\{\text{blue}, \text{red}\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/80205>.

► **Proposition 13.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to either $\{\text{blue}, \text{green}\}$ or $\{\text{orange}, \text{red}\}$.*

Refer to Figure 8b, which shows the bottle structure for $\{\text{blue}, \text{green}\}$. The bottle structure for $\{\text{orange}, \text{red}\}$ can be obtained by reflecting the bottle structure for $\{\text{blue}, \text{green}\}$ through a vertical line. We will also use a blue (or orange) finisher in our setup to prevent rows in the body portion from clearing early.

A demo of the $\{\text{blue}, \text{green}\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/83069>.

► **Proposition 14.** *Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to $\{\text{blue}, \text{orange}\}$.*

Refer to Figure 8c(a) for the bottle structure for $\{\text{blue}, \text{orange}\}$. We will also use a blue finisher in our setup to prevent rows in the body portion from clearing early.

A demo of the $\{\text{blue}, \text{orange}\}$ bottle structure can be found at <https://jstris.jezevec10.com/map/80195>.

We also note the following:

► **Conjecture 15.** *The reduction in the proof of Proposition 14 works even if pieces experience $20G$ gravity.*

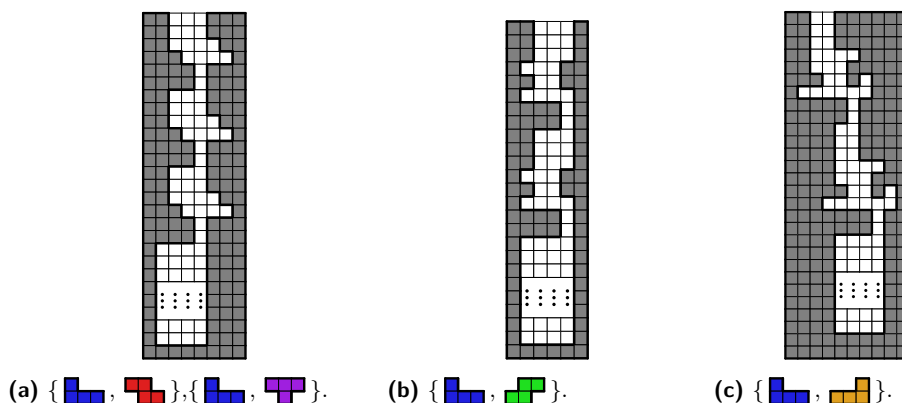


Figure 8 The bottle structures for the other subsets containing $\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}$; the bottle structure for $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ and $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ can be obtained by reflecting the bottle structure for $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ and $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ through a vertical line, and the bottle structure for $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ can be obtained by reflecting the bottle structure for $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ through a vertical line.

4.6 Putting It All Together

Combining all of these results, we get the following result:

► **Theorem 16.** *For any size-2 subset $A \subseteq \{\begin{smallmatrix} \square & \square & \square \\ \square & \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$, Tetris clearing with SRS is NP-hard, and the corresponding counting problem is #P-hard, even if the type of pieces in the sequence given to the player is restricted to the piece types in A .*

Proof. Propositions 6, 7, 8, 9, 10, 11, 12, 13, and 14 cover all size-2 subsets of piece types, as shown in Table 1; combining all of the reductions, we obtain the desired result. ◀




► **Remark 17.** All of our reductions involve a linear-factor blowup in the a_i for the filling sequences (i.e., we use $\Theta(na_i)$ pieces in the filling sequences); this makes it easier to argue about what happens when an overflow happens and makes the bottle analogy more fitting (since the neck portion is smaller while the body portion is much larger) but makes our reductions somewhat inefficient. Perhaps it is possible to reduce the blowup to a constant factor, though the argument may be a bit more complex.

5 Tetris Survival: Hard Drops Only and 20G

The previous section mentions NP-hardness of Tetris clearing under the “hard drops only” and “20G” Tetris models. Previous results about general Tetris [7, 2] have also proven NP-hardness of Tetris survival, so in this section, we prove that, in both of these variants, Tetris survival is NP-hard using $\{\begin{smallmatrix} \square & \square & \square \\ \square & \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$ pieces. This improves upon a result by Temprano [12] which proves hardness for “hard drops only” mode using the piece subset $\{\begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$.

► **Theorem 18.** *Tetris survival is NP-complete in the “hard drops only” and “20G” game modes, even if the type of pieces in the sequence given to the player is restricted to $\{\begin{smallmatrix} \square & \square & \square \\ \square & \square & \square \end{smallmatrix}, \begin{smallmatrix} \square & \square \\ \square & \square \end{smallmatrix}\}$.*

Proof. We reduce from 3-Partition with Distinct Integers using a similar bottle structure to other proofs in this paper. From a 3-Partition with Distinct Integers instance, we create a setup consisting of $\frac{n}{3}$ width 1 buckets, each of height $4t$, separated by width 1 columns. In

addition, we create a bucket of height $t - 1$ on the left which is blocked by a single square and has an open square on the upper left diagonal. We add one additional column on the left to obtain an even width board. We leave two rows at the top of the board empty. See Figure 9(a) for details. Each a_i is encoded by the sequence ( $n/3+1$,  a_i , ).

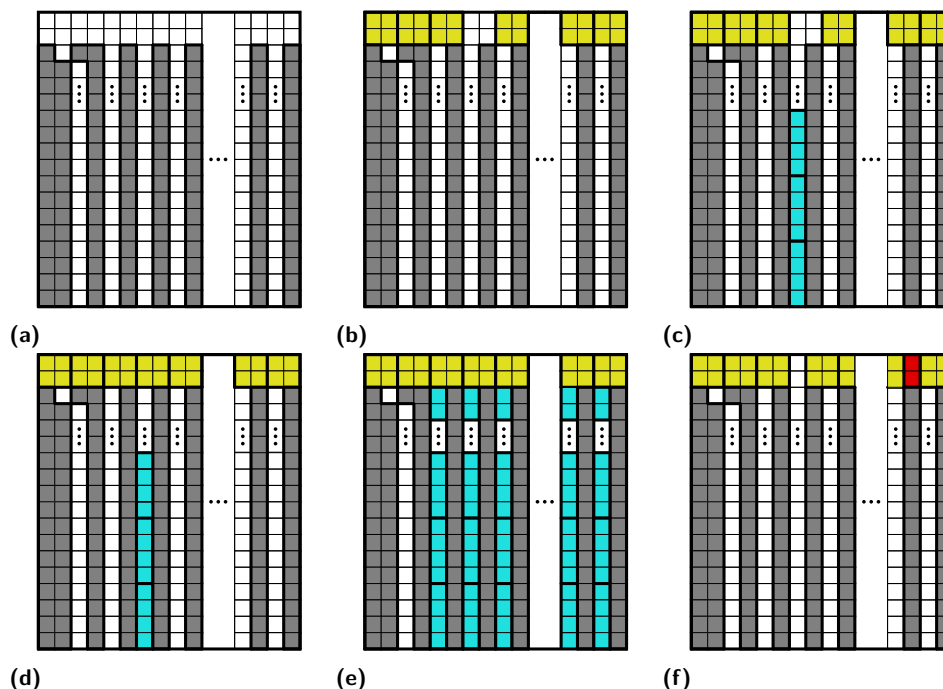













Figure 9 The bucket structure for “hard drops only” and “20G” Tetris modes. (b) shows how beginning sequence must block all but one bucket. (c) shows the result of a filling sequence. (d) shows how the closing sequence clears the top two rows. (e) shows a full board. (f) shows improper handling during the priming sequence.


The priming sequence is ( $n/3+1$). Due to parity constraints, the player is forced to block all but one of the buckets using these pieces. They can choose to leave either one two-block wide gap or two one-block wide gaps in the top two rows. However, if they choose to leave one-block gaps, they will create spaces that can never be filled without overflowing the screen and causing a game loss (see the squares highlighted in red in Figure 9(f)), so we can assume that they will leave a 2×2 gap, as shown in Figure 9(b).

The filling sequence is ( a_i); the  pieces must be placed in the pre-selected bucket, as shown in Figure 9(c). If pieces are not placed in accordance with a correct partition, then there will at some point be an extra  piece which does not fit in the open bucket. This will cause the player to lose as an  piece has length 4 and cannot fit in the 2×2 gap in the top 2 rows (note that no  piece will stick partially out of a bucket since each bucket has a height that is a multiple of 4).







The closing sequence is (). The  piece can only fit in the 2×2 square formed during the priming sequence (see Figure 9(e)), and it clears the top two rows, resetting the board to the initial state, except with a somewhat more filled bucket.

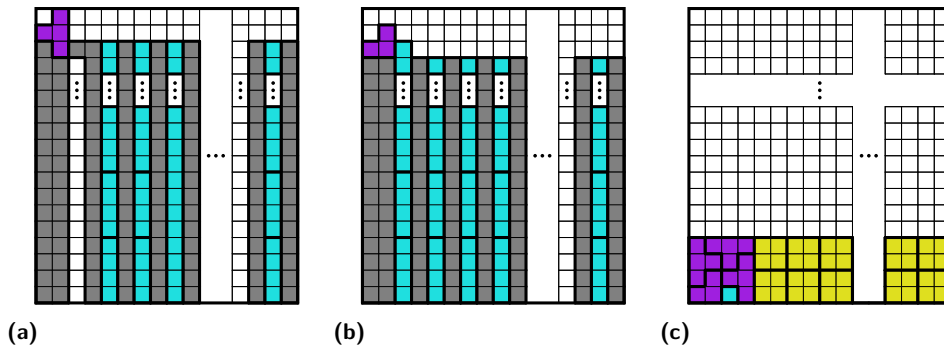
Once the player has received the entire sequence corresponding to each number in the 3-partition instance, including the final closing piece, the entire board will be full (see Figure 9(e)), with the exception of the extra inaccessible bucket. Because of this, the player must have filled each of the buckets to exactly the right height, solving the 3-partition instance, otherwise they would have lost.

For the entire duration of this sequence, every piece can be hard dropped into place as there are no overhangs in any of the buckets. Furthermore, each piece can be successfully maneuvered under 20G conditions. The  pieces cannot fall down any buckets, so they can be safely slid to any location in the top 2 rows, and the  pieces can move over the placed  pieces to the only open bucket. ◀

▶ **Corollary 19.** *The above reduction can be extended to NP-hardness for board clearing under “hard drops only” or “20G” conditions if we also allow for  pieces.*

▶ **Remark 20.** We have already proved that Tetris clearing under the “hard drops only” and “20G” models is hard with just two types of pieces. The primary reason for including hardness with a larger set of pieces is that this shows that, in both game modes, there is a board configuration and subset of pieces where the problems of surviving and clearing the board are both hard at the same time. In addition, our Tetris clearing results under the “20G” model does not include any proper subsets of $\{\text{cyan 1x4}, \text{yellow 2x2}, \text{purple 2x2}\}$, so this result is interesting in its own right.

Proof. We begin with the above proof that survivability with $\{\text{cyan 1x4}, \text{yellow 2x2}\}$ is hard. We extend the sequence with the finale sequence $(\text{purple 2x2}, \text{cyan 1x4}^t, \text{yellow 2x2}^{2n/3}, \text{purple 2x2}^3)$. Figure 10 shows the clearing process. We begin by using the first  piece to open the inaccessible bucket by clearing a row. We then use the  pieces to fill the previously inaccessible bucket, clearing all but the final two rows in the process. The final  piece protrudes one square from its bucket because of the row cleared by the first . We use the  pieces to fill the $\frac{2n}{3} \times 4$ space on the right of the board. Finally, we place the remaining three  pieces to fill the remaining space and completely clear the board. All of these pieces can be placed in both special game modes. ▶



■ **Figure 10** The clearing procedure for “hard drops only” and “20G” using $\{\text{cyan 1x4}, \text{yellow 2x2}, \text{purple 2x2}\}$.

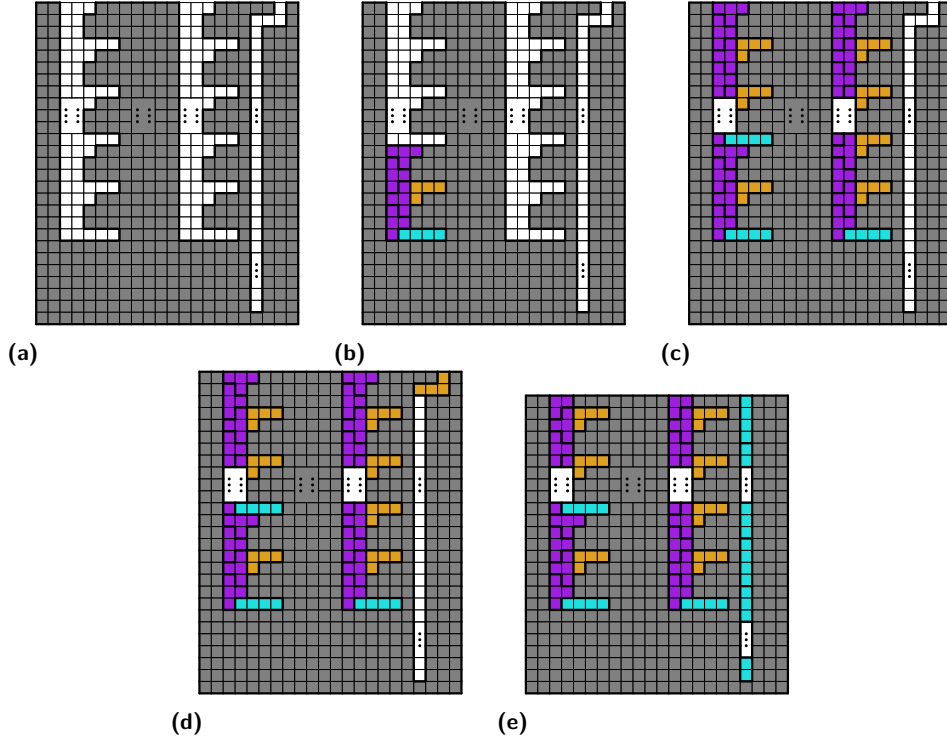
6 ASP-Completeness of Tetris

Even though the reductions in Section 4 are sufficient to prove #P-hardness, the reductions are not parsimonious, so they cannot be used to prove ASP-completeness. Indeed, the “blocking” bottles paradigm likely cannot be used to show ASP-completeness as there are many ways to permute the pieces that block all but one bottle. Thus, for ASP-completeness, we turn back to the “priming” buckets paradigm in [4]:

▶ **Theorem 21.** *Tetris clearing with SRS is ASP-complete even if the type of pieces in the sequence given to the player is restricted to either $\{\text{cyan 1x4}, \text{purple 2x2}, \text{orange 2x2}\}$ or $\{\text{cyan 1x4}, \text{purple 2x2}, \text{blue 2x2}\}$.*

24:16 Tetris with Few Piece Types

Proof. First we discuss the $\{\text{cyan}, \text{purple}, \text{orange}\}$ case. We give a parsimonious reduction from Numerical 3-Dimensional Matching with Distinct Integers; refer to Figure 11(a), which shows the bucket structure plus rightmost columns for $\{\text{cyan}, \text{purple}, \text{orange}\}$.



■ **Figure 11** The bucket structure plus rightmost columns for $\{\text{cyan}, \text{purple}, \text{orange}\}$. (b) shows how the cyan piece must prime a bucket (requires an cyan spin) and how the remainder of the pieces in a sequence must fit in the bucket ($m = 2$ here). (c) shows how the board looks like before the finale sequence. (d–e) show how the pieces in the finale sequence must be placed (requires an orange spin).


Here, a “unit” is the pattern that repeats every four rows. A positive integer m is encoded by the sequence of pieces $(\text{cyan}, (\text{purple}, \text{orange}, \text{purple})^{m-1}, \text{purple}, \text{purple})$. The finale sequence is $(\text{orange}, \text{cyan}^N)$, where $N = \text{poly}(nt)$ is much larger than the height of the buckets, and is used to clear the rightmost columns after the buckets have been filled.

In this case, the cyan piece serves as the “primer”, and must be placed as indicated in Figure 11(b) (the placement is possible due to cyan spins). The purple pieces cannot be placed in a non-primed bucket without blocking off certain holes, particularly the squares in which an cyan piece or an orange piece must be placed, and misplacing an orange piece (i.e., putting it in a different, non-primed bucket, putting it where the orange piece goes during the finale sequence, or putting it too high in the bucket) causes the next two purple pieces to block off squares in which an cyan piece or an orange piece must be placed. Thus, once the cyan piece is placed in a bucket, the placements of the rest of the pieces encoding the positive integer m are forced. Further discussion on and figures for improper piece placements can be found in the full version of our paper.



Lastly, to make the reduction parsimonious, from the instance

$$A = \{a_1, a_2, \dots, a_n\}, B = \{b_1, b_2, \dots, b_n\}, \text{ and } C = \{c_1, c_2, \dots, c_n\}$$

of Numerical 3-Dimensional Matching with Distinct Integers, we scale the a_i , b_i , and c_i and add/subtract constants such that each group must consist of exactly one a_i , one b_j , and



one c_k (i.e., no group with at least two elements from any of A , B , or C can sum to the target sum). In addition, we “pre-fill” all of the buckets so that the i th bucket from the left is already filled up to a_i units, and the sequence of pieces given to the player consist only of the pieces in the b_i sequences, the pieces in the c_i sequences, and the pieces in the finale sequence. This ensures that the subsets have a fixed ordering and must consist of exactly one a_i , one b_j , and one c_k . Combined with the fact that all the piece placements after each  piece in a sequence encoding a positive integer are forced, this means that each solution to the Numerical 3-Dimensional Matching instance corresponds to exactly one way to clear the Tetris board, and vice versa.

Therefore, we have a parsimonious reduction from Numerical 3-Dimensional Matching with Distinct Integers to Tetris clearing with $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{orange } 2 \times 2\}$, meaning that Tetris clearing with SRS is ASP-complete even if the type of pieces in the sequence given to the player is restricted to $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{orange } 2 \times 2\}$.

We get a similar argument for $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{blue } 2 \times 2\}$ by vertical symmetry; the  spin required still works when mirrored even though the kick tests for  are not vertically symmetric. ◀

Demos of the bucket structure can be found at <https://jstris.jezevec10.com/map/80170> for $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{orange } 2 \times 2\}$ and at <https://jstris.jezevec10.com/map/83325> for $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{blue } 2 \times 2\}$.

7 Open Problems

One big open problem that still remains is the computational complexity of Tetris clearing with SRS if the player is only given one piece type (for example, if the sequence consists of entirely  pieces). In this case, the “blocking” bottles paradigm no longer works, because the same piece type cannot be used both to block and to fill bottles without the reduction breaking, so a proof of hardness would involve an entirely different setup. It is also possible that Tetris clearing with SRS and with only one piece type is in P. For example, [4] conjectures polynomial time for the  piece type.

Similarly, it is open whether or not Tetris clearing with SRS is ASP-complete for subsets of pieces that are not supersets of $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{blue } 2 \times 2\}$ or $\{\text{cyan } 1 \times 4, \text{purple } 2 \times 2, \text{orange } 2 \times 2\}$. One could likely construct similar structures for other 3-piece subsets, but arguing whether Tetris clearing with SRS is ASP-complete for 1- or 2-piece subsets may require different ideas.

Some open questions arise regarding whether our results can be extended if we consider different objectives or add additional features. For example, can we establish results for 2-piece subsets, similar to those in Section 4, for Tetris survival? If we add a “holding” function, where the player can put one piece aside for later use, can get similar results?

Modern variants of Tetris also use different random generators to ensure that the player doesn’t receive the same piece arbitrarily many times in a row. One of the simplest random generators is called a *7-bag randomizer*. For this randomizer, the sequence of pieces is divided into groups (or “bags”) of 7, each group containing one of each tetromino in one of $7! = 5,040$ possible orderings. Can we show NP-hardness even if the sequence of pieces has to be able to be generated from this randomizer?



References

- 1 Tetris, 2023. Apple TV+ movie directed by Jon S. Baird and written by Noah Pink.
- 2 Sualeh Asif, Michael Coulombe, Erik D. Demaine, Martin L. Demaine, Adam Hesterberg, Jayson Lynch, and Mihir Singhal. Tetris is NP-hard even with $O(1)$ rows or columns. *Journal of Information Processing*, 28:942–958, 2020.
- 3 Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Adam Hesterberg, Roderick Kimball, and Justin Kopinsky. Path puzzles: Discrete tomography with a path constraint is hard. *Graphs and Combinatorics*, 36:251–267, 2020.
- 4 Ron Breukelaar, Erik D. Demaine, Susan Hohenberger, Hendrik Jan Hoogeboom, Walter A. Kosters, and David Liben-Nowell. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications*, 14(1–2):41–68, 2004. doi:10.1142/S0218195904001354.
- 5 Charles J. Colbourn. The complexity of completing partial Latin squares. *Discrete Applied Mathematics*, 8(1):25–30, 1984. doi:10.1016/0166-218X(84)90075-1.
- 6 Sopan Deb. Boy, 13, is believed to be the first to ‘beat’ tetris. *The New York Times*, 3 january 2024. URL: <https://www.nytimes.com/2024/01/03/arts/tetris-beat-blue-scuti.html>.
- 7 Erik D. Demaine, Martin L. Demaine, Sarah Eisenstat, Adam Hesterberg, Andrea Lincoln, Jayson Lynch, and Y. William Yu. Total Tetris: Tetris with monominoes, dominoes, trominoes, pentominoes, . . . *Journal of Information Processing*, 25:515–527, 2017. doi:10.2197/ipsjjip.25.515.
- 8 Erik D. Demaine, Fermi Ma, Erik Waingarten, Ariel Schvartzman, and Scott Aaronson. The fewest clues problem. *Theoretical Computer Science*, 748:28–39, November 2018. doi:10.1016/j.tcs.2018.01.020.
- 9 Ian Holyer. The NP-completeness of some edge-partition problems. *SIAM Journal on Computing*, 10(4):713–717, 1981. doi:10.1137/0210054.
- 10 Heather Hulett, Todd G. Will, and Gerhard J. Woeginger. Multigraph realizations of degree sequences: Maximization is easy, minimization is hard. *Operations Research Letters*, 36(5):594–596, 2008. doi:10.1016/j.orl.2008.05.004.
- 11 Harry B. Hunt III, Madhav V. Marathe, Venkatesh Radhakrishnan, and Richard E. Stearns. The complexity of planar counting problems. *SIAM Journal on Computing*, 27(4):1142–1167, 1998. doi:10.1137/S0097539793304601.
- 12 Oscar Temprano. Complexity of a Tetris variant. arXiv:1506.07204, 2015. URL: <https://arxiv.org/abs/1506.07204>, arXiv:1506.07204.
- 13 TetrisWiki. 20G. <https://tetris.wiki/20G>. Accessed: 2023-09-23.
- 14 TetrisWiki. Super Rotation System. https://tetris.wiki/Super_Rotation_System. Accessed: 2023-09-23.
- 15 TetrisWiki. Tetris guideline. https://tetris.wiki/Tetris_Guideline. Accessed: 2023-09-23.
- 16 Wikipedia. Tetris. <https://en.wikipedia.org/wiki/Tetris>. Accessed: 2023-09-23.
- 17 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*, E86-A(5):1052–1060, 2003. Also IPSJ SIG Notes 2002-AL-87-2, 2002. URL: <http://ci.nii.ac.jp/naid/110003221178/en/>.

Complexity of Planar Graph Orientation Consistency, Promise-Inference, and Uniqueness, with Applications to Minesweeper Variants

MIT Hardness Group¹

Massachusetts Institute of Technology, Cambridge, MA, USA

Della Hendrickson  

Massachusetts Institute of Technology, Cambridge, MA, USA

Andy Tockman  

Massachusetts Institute of Technology, Cambridge, MA, USA

Abstract

We study three problems related to the computational complexity of the popular game Minesweeper. The first is consistency: given a set of clues, is there any arrangement of mines that satisfies it? This problem has been known to be NP-complete since 2000 [4], but our framework proves it as a side effect. The second is inference: given a set of clues, is there any cell that the player can prove is safe? The coNP-completeness of this problem has been in the literature since 2011 [6], but we discovered a flaw that we believe is present in all published results, and we provide a fixed proof. Finally, the third is solvability: given the full state of a Minesweeper game, can the player win the game by safely clicking all non-mine cells? This problem has not yet been studied, and we prove that it is coNP-complete.

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

Keywords and phrases NP, coNP, hardness, minesweeper, solvability, gadgets, simulation

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.25

Related Version *Full Version:* <https://arxiv.org/abs/2404.14519>

Acknowledgements This paper was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.5440) taught by Erik Demaine in Fall 2023. We thank the other participants of that class for helpful discussions and providing an inspiring atmosphere.

1 Introduction

The puzzle-based video game Minesweeper was popularized by its inclusion in the default installation of Windows 3.1 in 1992, and to this day is one of the most widely recognizable computer games. Though the premise is very simple, it has spawned an endless stream of spinoffs, and a number of communities dedicated to challenges such as completing a randomly generated game as quickly as possible.

From a computational complexity perspective, Minesweeper is interesting in that there are multiple natural decision problems to study. The simplest is *consistency* (“given some clues, is there a possible arrangement of mines?”), which was proved NP-complete in 2000 [4].

But consistency doesn’t capture the essence of Minesweeper as a game, where the player has some partial information and tries to find a cell that they can click on safely, leading to more information. This suggests the *inference* problem (“given some clues, is there a

¹ Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as “MIT Hardness Group” (without “et al.”).



cell that is provably safe?”), which was shown coNP-complete in 2011 [6]. Unfortunately, this proof of coNP-hardness of Minesweeper inference is incorrect, and Thieme and Basten’s proof [7], which is designed to use very small gadgets, suffers from the same issue.

While inference asks about a single “turn” of Minesweeper, it is also natural to ask a related question about the entire game. We introduce the *solvability* decision problem, in which we are given both the current state of a Minesweeper game and also the secret arrangement of mines, and asked whether the player can make a sequence of safe clicks to solve the game.

In this paper, we develop a framework based on graph orientation to prove coNP-completeness of Minesweeper inference and solvability. As a side effect, the same gadgets also suffice to prove NP-hardness of Minesweeper consistency. Of particular note, we prove that Minesweeper solvability is coNP-complete even “after a single click”: from a nearly empty initial state with only one cell revealed.

Specifically, we define three graph orientation decision problems (consistency, promise-inference, and uniqueness) related to the Minesweeper problems, and show that each is hard with a particular set of simple abstract gadgets. It follows that finding well-behaved constructions in Minesweeper which behave like those gadgets is enough to prove hardness for all three Minesweeper problems.

In Section 2, we define each decision problem carefully, summarize the previously known results, and explain the flaw in the existing reduction for inference. In Section 3, we develop a framework of gadgets that makes it easy to prove hardness results for all three decision problems. In the full version of the paper, we apply the framework to Minesweeper and to many variants of Minesweeper from the video game *14 Minesweeper Variants*. For the most part, each application consists entirely of constructions of the relevant gadgets in the Minesweeper variant under consideration.

2 Prior work and definitions

2.1 Consistency

Research on the computational complexity of Minesweeper began when Sadie Kaye [4] posed the Minesweeper consistency problem. Informally, this problem asks whether a partially completed Minesweeper board has a legal arrangement of mines. We provide a formal definition here.

► **Definition 1.** A *partial board* is a two-dimensional rectangular array, where each entry is either a **covered cell** or an **uncovered cell**. A covered cell is an unknown cell which may or may not be a mine. An uncovered cell has an integer (and is known to not contain a mine), representing a Minesweeper clue. For a partial board B , we denote the set of covered cells $\mathcal{C}(B)$, and the set of uncovered cells $\mathcal{U}(B)$.

► **Definition 2 (Minesweeper consistency problem).** A partial board B is **consistent** if there exists a set $M \subseteq \mathcal{C}(B)$, representing all locations of mines, such that $M \cap \mathcal{U}(B) = \emptyset$ and the integer in each uncovered cell $c \in \mathcal{U}(B)$ counts the number of cells in M which are orthogonally or diagonally adjacent to c . Otherwise, B is **inconsistent**. See Figures 1 and 2. The input to the **Minesweeper consistency problem** is a partial board, and the problem asks whether it is consistent.

Kaye proves that the consistency problem is NP-complete [4]. We also prove NP-completeness as a side effect of our framework.

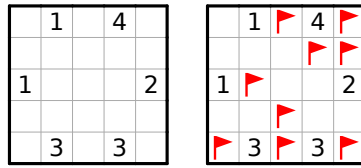


Figure 1 Left: an example of a consistent board. Right: one possible way to satisfy it.

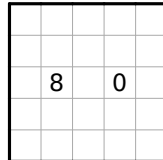


Figure 2 An example of an inconsistent board.

Proposition 3. The Minesweeper consistency problem is in NP.

Proof. Given a partial board B , a certificate of consistency is the M of Definition 2. We can iterate over $\mathcal{U}(B)$ and check that each clue is satisfied in polynomial time. ◀

We leave the proof of hardness to Section 3.

2.2 Inference

The Minesweeper inference problem was first posed by Allan Scott, Ulrike Stege, and Iris van Rooij [6]. Informally, this problem asks whether a partially completed Minesweeper board has a logical deduction available to the player that lets them click on a cell which is guaranteed to not have a mine. We provide a formal definition of a slight variation on the problem as originally posed.

Definition 4 (Minesweeper inference problem). Given a partial board B , an **inference** is a cell $c \in \mathcal{U}(B)$ such that for all consistent arrangements of mines $M \subseteq \mathcal{C}(B)$, we have $c \notin M$. The input to the **Minesweeper inference problem** is a partial board, and it asks whether there is an inference. See Figures 3 and 4.

Note that this definition has two key differences from the one originally given by Scott, Stege, and van Rooij [6]:

- Deducing the location of a mine does not count as an inference.
- No positions of known mines are given in the input.

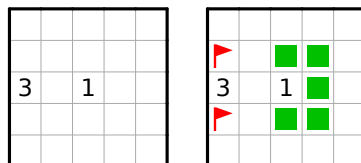


Figure 3 Left: an example of a board with an inference. Right: green squares mark cells that can be inferred.

1			2
3			4

■ **Figure 4** An example of a board that doesn't have an inference.

		1	u	1		1	2	3	2	1	
	1	2	3	2	1	1				1	
1	2		u		2	2	3	a	3	2	1
v	3	v	6	a	a	1	a	2	a	1	a
1	2		b		3	2	2	a	2	2	1
	2	3	4	2	2		3	3		1	
	2		b	3	3	4	a		3	1	
	2			x	y	z				2	
	1	3						3	1		
		1	2	3	3	3	2	1			

■ **Figure 5** The OR gate from [6] Figure 16. There is a minor typo: the lower cell with b should have \bar{b} .

We prefer Definition 4 because it eliminates the need to place conditions on the input such as “all given mines must be deducible from the clues,” which is otherwise necessary to avoid placing mines that could never be deduced in a real game. Furthermore, the flags representing known mines can be viewed as simply a player aid – one could in principle play a full game of Minesweeper without marking a single mine. Though we will proceed with Definition 4 only, we remark that all results in this paper work under either definition.

► **Proposition 5.** *The Minesweeper inference problem is in coNP.*

Proof. Given a partial board B , a certificate of noninference is, for each cell $c \in \mathcal{C}(B)$, a consistent arrangement of mines M with $c \in M$. We can iterate over the polynomially many arrangements given by the certificate and check consistency in polynomial time. ◀

Again, the proof of hardness will be in Section 3.

Error in existing proofs

Scott, Stege, and Rooij’s proof of coNP-hardness [6] has an issue where there is sometimes an unintended inference. Their OR gate is shown in Figure 5. A cell has a mine when the literal marking it is true. The inputs are u and v the output is a . The 6 enforces $a + b = u + v$, and the section at the bottom enforces $a \geq b$. Together this forces $a = u \vee v$ and $b = u \wedge v$.

The issue occurs when we know that u and v can’t both be true. This might happen if the OR gate is used inside an AND gate which merges two clauses which can’t simultaneously be false, such as if one contains x and the other contains $\neg x$.

3	2	1	
4	2	1	
3	2	1	0
2	1	0	0
1	1	0	0

■ **Figure 6** An example of a solvable board. Cyan cells indicate elements of K , and white cells are unknown to the player. Red flags indicate elements of M . From the initial state, the only provably safe cell is the fourth cell in the second row. After clicking it, the player can now deduce the third cell in the first row. Finally, this reveals enough information to deduce the second cell in the first row.

In this case, we can deduce that $b = u \wedge v$ is false, and thus the (higher) cell labeled b is safe to click. This is an inference. One strategy for resolving this issue would be to reveal that cell in the input, assuming one can find all inferences like this. But this doesn't work: That cell has either a 3 or a 4 depending on the value of u , so revealing it tells the player the value of u , allowing them to make more inferences and possibly learning further information.

Thieme and Basten's more compact proof [7] has a very similar issue.

Our approach to avoiding this issue is twofold. meaning clicking an inferred cell never reveals information. This allows us to eliminate inferences by revealing those cells in the input. This handles unintended "local" inferences, which we are able to detect.

Second, to prevent unintended larger-scale inferences, we design the network of gadgets carefully. Specifically, we ensure that for every gadget (OR gate, crossover, etc.), every locally valid combination of values can be achieved. The only exception is the "final" gate, which has a forced output when the input formula is unsatisfiable. We maintain this property across simulations by introducing a problem we call "promise-inference", which also partially relaxes the constraint that every locally valid solution is achievable.

2.3 Solvability

Informally, the Minesweeper solvability problem asks whether a player can make a sequence of inferences from a partially completed Minesweeper game and win by clicking on all non-mine cells. We provide a formal definition of this problem, which to our knowledge has not been studied.

► **Definition 6** (Minesweeper solvability problem). *Let B be a partial board with uncovered cells $K = \mathcal{U}(B)$, and let M be a set of mines consistent with B . Note that B is determined by M and K . Here M represents the secret set of mines, which is thought of as unknown to the player, and K represents the set of known cells.*

*Consider an ordering O of $G \setminus (M \cup K)$, meaning a list containing each element of $G \setminus (M \cup K)$ exactly once. For an element $o \in G \setminus (M \cup K)$, let $O = O_{init} ++ [o] ++ O_{tail}$, where $++$ denotes concatenation. We say M is **solvable** from K if there exists an O such that for all $o \in O$, o is a inference for the (unique) partial board consistent with M whose uncovered cells are $K \cup O_{init}$. Otherwise, M is **unsolvable** from K . See Figure 6.*

*The input to the **Minesweeper solvability problem** consists of the dimensions of a rectangular grid G and two disjoint subsets of cells $M \subseteq G$ and $K \subseteq G$. The problem asks whether M is solvable from K .*

Intuitively, solvability simulates a player who is given a Minesweeper puzzle on their computer to solve. If such a player wishes to guarantee that they do not click a mine (to win without any luck, or because they have antagonistic implementation which repositions

25:6 Complexity of PGO Consistency, Promise-Inference, and Uniqueness

mines to make the player lose if they click a cell that isn't provably safe, such as "expert mode" in *14 Minesweeper Variants*), they must click covered cells in an order O that satisfies Definition 6.

► **Proposition 7.** *The Minesweeper solvability problem is in coNP.*

Proof. Given a game (M, K) , a certificate of unsolvability is a set $K' \supseteq K$ together with a certificate of noninference for K' .

Suppose there is such a certificate. As long as the information available to the player is a subset of K' , they cannot infer that a cell outside K' is safe. Any order has a first cell outside K' , which by assumption is not an inference.

Conversely, suppose an instance is unsolvable. Starting from K , repeatedly make inferences and add them to the known information. By assumption, this must get stuck before all non-mine cells are uncovered, meaning we reach a point with no inferences. Then take K' to be the uncovered cells at that point. ◀

Once again, we prove hardness in Section 3.

3 The Minesweeper gadget framework

In this section, we describe an abstract framework which we will later apply to Minesweeper. The wires in Minesweeper hardness proofs generally have two states, which can be thought of as two orientations of an edge, so our framework is a general form of graph orientation.

► **Definition 8.** *A **gadget** is an abstract object which has*

- *a finite set of **ports**, in a specified cyclic order (we will generally list the ports in this order, and describe it more explicitly when relevant).*
- *a **constraint**, which is a set of subsets of the ports.*

Gadgets will interact via directed edges connecting ports. The constraint says which sets of edges pointing towards the gadget should be considered legal.

The gadgets we name are collected in Table 1, and will also be described as they come up.

► **Definition 9.** *A **network** of gadgets from a set S is an undirected graph where*

- *each vertex is labeled with a gadget from S*
- *if $G \in S$ has k ports, each vertex labeled G has degree k and its edge incidencies are labeled in a bijection with ports of G .*

*A **planar network** is such a graph equipped with a planar embedding, such that the cyclic order of edges around each vertex matches the order of the ports of the corresponding gadget.*

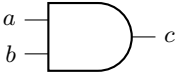

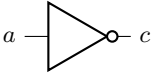
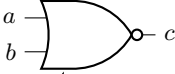
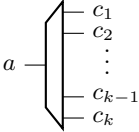
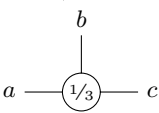
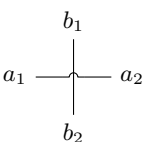
We often equivocate between vertices and gadgets, and between edge incidencies and ports – think of each vertex as a copy of its label, and think of edges as connecting ports to ports in a matching.

We draw planar networks using the icons in Table 1, which also indicate the correspondence between edges and ports (except for when it doesn't matter by symmetry).

► **Definition 10.** *An **assignment** to a (planar) network assigns a direction to each edge. A vertex is **satisfied** if the set of (labels of) edges pointing into it is in its (label's) constraint. An assignment is **satisfying** if every vertex is satisfied.*

Planar Graph Orientation (PGO) is the study of satisfying assignments of planar networks.

■ **Table 1** The gadgets we define in this paper, the icons we use to draw them (for those we show in networks), and their constraints.

Name	Icon	Constraint
fixed (true) terminal	$a \text{ --- } F$	$\{\{\}\}$
fixed (false) terminal	$a \text{ --- } T$	$\{\{a\}\}$
free terminal	$a \text{ ---}$	$\{\{\}, \{a\}\}$
AND gate		$\{\{c\}, \{a, c\}, \{b, c\}, \{a, b\}\}$
OR gate		$\{\{c\}, \{a\}, \{b\}, \{a, b\}\}$
NOT gate		$\{\{\}, \{a, c\}\}$
NOR gate		$\{\{\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$
(k -way) fanout gate		$\{\{a\}, \{c_1, \dots, c_k\}\}$
1-in-3 gadget		$\{\{a\}, \{b\}, \{c\}\}$
crossover		$\{\{a_1, b_1\}, \{a_2, b_1\}, \{a_1, b_2\}, \{a_2, b_2\}\}$

3.1 Gates as gadgets

One important kind of gadget is logic gates, which can be interpreted as gadgets: inputs are edges entering from the left and output are edges exiting on the right. Interpret pointing right as true. The gadget's constraint allows the inputs to be arbitrary, but forces the correct outputs for each combination of inputs. More precisely, for each subset S of input ports, the constraint contains $S \cup T$, where T is the set of outputs that are *false* when precisely the inputs in S are true.

It's important to keep in mind distinction between gate gadgets, which compute a value, and "normal" gadgets, which enforce a constraint.

For example, the *OR gate* has ports $\{a, b, c\}$ and constraint $\{\{c\}, \{a\}, \{b\}, \{a, b\}\}$. This constraint allows any subset of the input ports a and b to have edges pointing in, and enforces that the edge incident to the output port c to point out exactly when at least one input port points in. In other words, it computes the OR of a and b , and outputs it at c .

On the other hand, the *OR gadget* (which we don't need beyond this example) has only two ports $\{a, b\}$ and constraint $\{\{a\}, \{b\}, \{a, b\}\}$. It enforces that at least one edge points in, or that at least one input is true. This is equivalent to an OR gate with the output forced to be “true” (pointing away).

Any boolean circuit can be represented as a network of gate gadgets, by attaching input ports to output ports in the natural way. Allow us to leave inputs and outputs to the full circuit as “dangling” edges for now. It follows from the definition of gate gadgets – formally, one can induct on the depth of a gate – that this network has exactly one satisfying assignment for each combination of orientations of the dangling input edges, and in each the orientations of the output edges encode the output of the circuit (and internal edges encode the internal state of the circuit).

If an output connects to $k > 1$ inputs, we need to use a k -way *fanout gate*, which has ports $\{a, c_1, \dots, c_k\}$ and constraint $\{\{a\}, \{c_1, \dots, c_k\}\}$. This is the gadget representing the gate that duplicates its input k times.

A drawing of a circuit in the plane becomes a planar network of gate gadgets. A crossing pair of wires is translated to the *crossover gate*, which has two inputs and two outputs matching the inputs in the opposite order. As a gadget, the crossover gate has ports $\{a_1, b_1, a_2, b_2\}$, in that cyclic order, and constraint $\{\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}, \{a_2, b_2\}\}$. One can think of it as two wires $a_1 \rightarrow a_2$ and $b_1 \rightarrow b_2$ that cross in a circuit. However, the directions of these wires are arbitrary because the gadget is highly symmetric. As a gadget outside the perspective of networks of logic gates, the crossover gate is two edges that can independently be assigned orientations, and which cross each other. So we will also call it simply the *crossover*.

3.2 Decision problems

We consider three decision problems about planar graph orientation, corresponding to the Minesweeper decision problems we're interested in. Each of them is parameterized by a set of gadgets S – throughout we consider only finite sets of gadgets – and takes as input a planar network N of gadgets from S .

The problem corresponding to Minesweeper consistency is straightforward.

► **Problem 11.** PGO *consistency* with S asks whether N has a satisfying assignment.

For Minesweeper inference, we need to avoid a subtle issue, which is the error in prior claims of coNP-hardness [6, 7]. If there is a gadget for which we can deduce that some legal combination of edge orientations can't be extended to a full satisfying assignment, this information may allow us to infer the value of a Minesweeper cell internal to the gadget. This can happen even if we can't deduce the orientation of any particular edge, so the most obvious PGO inference problem fails to reduce to Minesweeper, and thus isn't useful.

Our strategy for resolving this issue will ultimately be to “click” all cells in the Minesweeper instance that could be deduced in this way. To make this work, we will need the values of those cells to not reveal any additional information (a property we will call “silence”), and we need the reduction to find all such cells in polynomial time. For our gadget framework to help here, we need to define the inference problem carefully, and in particular it needs to be aware of which combinations of edges are ruled out by “semilocal” deductions.

► **Problem 12.** PGO *promise-inference* with S is given a network N as well as, for each vertex in N , a *semilocal constraint* which is a subset of its constraint. We say an edge e is *locally forced* if the semilocal constraint of one of its vertices requires it has a particular orientation (because it's either in every element or in no element), and *locally free* otherwise.

We are promised that

- in every satisfying assignment, the set of edges pointing into a vertex is in its semilocal constraint; and
- either
 - there is a locally free edge to which every satisfying assignment assigns the same direction; or
 - for each vertex v and set of edges in its semilocal constraint, there's a satisfying assignment that makes exactly those edges point towards v .

We are asked to determine whether there is an edge whose orientation can be inferred but isn't immediate from semilocal constraints; that is, whether we are in the first option above.

Note that the two options can't simultaneously be true, since the existence of such an edge would imply that its incident vertices' semilocal constraints have elements that can't be extended to satisfying assignments.

The intent of semilocal constraints is to be between the levels of individual gadgets' "local" constraints and "global" deductions that require understanding the entire network. A semilocal constraint typically contains the sets of in-pointing edges that are attainable when looking at some constant-size neighborhood of a gadget.

The first part of the promise ensures that semilocal constraints are actually enforced by the structure of the network. In the case where there's no inferable edge, the second part says that it isn't possible to deduce more about the immediate neighborhood of a gadget than its semilocal constraint.

Finally, the PGO decision problem analogous to Minesweeper solvability is simpler.

► **Problem 13.** PGO *uniqueness* with S is given N as well as a satisfying assignment of N , and asks whether it is the unique satisfying assignment.

PGO uniqueness is related to Minesweeper solvability because it's possible to deduce the orientations of all edges if and only if there's a unique satisfying assignment.

► **Lemma 14.** For any set S of gadgets,

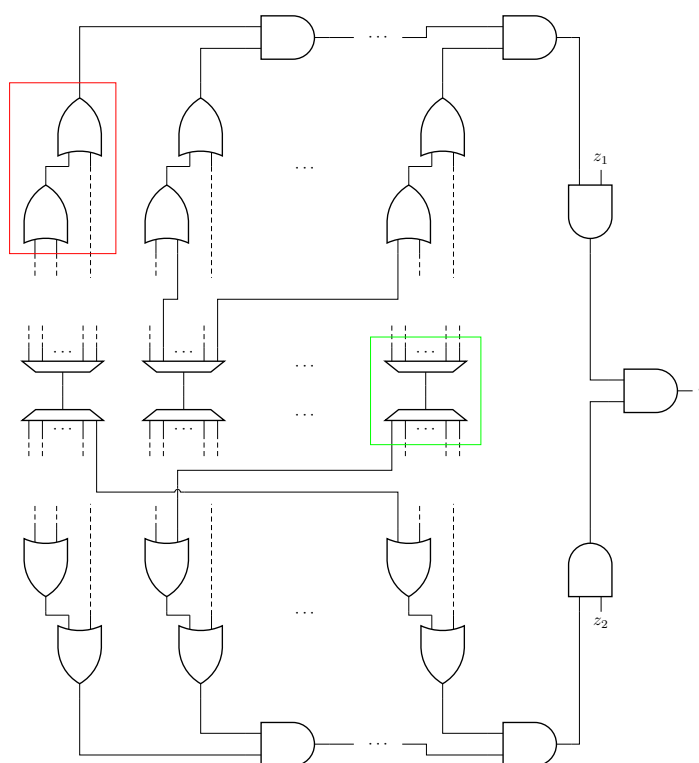
1. PGO consistency with S is in NP.
2. PGO promise-inference with S is in promise-coNP.
3. PGO uniqueness with S is in coNP.

Proof. Each problem has a straightforward certificate:

1. A satisfying assignment serves as a certificate of consistency.
2. A certificate that there is no inference consists of, for locally free edge and each orientation, a satisfying assignment that assigns the orientation to the edge.
3. A second satisfying assignment serves as a certificate of nonuniqueness. ◀

3.3 Hardness

We now prove hardness of each PGO decision problem for the appropriate class, with a specific set of gadgets. The gadget sets are chosen to make the hardness proofs simple; there are easy ways to reduce the number of different gadgets needed, and we will further simplify our gadget sets using simulation in Section 3.5. With Lemma 14, we have completeness in each case.



■ **Figure 7** Our reduction for coNP-hardness of PGO promise-inference. See Table 1 for gadget notation.

► **Theorem 15.** *PGO satisfiability with free terminals, fanout gates, and 1-in-3 gadgets is NP-hard.*

Some of these gadgets are new: the **1-in-3 gadget** has three ports, and its constraint says that exactly one of them must point in. The **free terminal** has one port, which is allowed to point in either direction.

Proof. We reduce from planar positive 1-in-3SAT, which is NP-hard [5]. Each variable with k occurrences becomes a free terminal that leads to a k -way fanout gate. Each clause becomes a 1-in-3 gadget. We connect the outputs of the fanout gates to 1-in-3 gadgets as in the 1-in-3SAT formula, which is planar. Satisfying assignments of this network correspond to satisfying assignments of the formula. ◀

► **Theorem 16.** *PGO promise-inference with free terminals, fanout gates, crossovers, OR gates, and AND gates is coNP-hard.*

Proof. We reduce from the complement of monotone 3SAT, which is NP-hard [1]. The layout of gadgets is depicted in Figure 7. The green outline is an example of a variable, and the red outline is an example of a clause. We think of the top half of the circuit as the positive clauses, and the bottom half of the circuit as the negative clauses. We remove duplicate clauses before constructing the circuit. Each gadget has as its semilocal constraint its full constraint set.

The output of the formula is given by r , which is connected to a free terminal. If the formula is unsatisfiable, there is an inference; namely, r is false (points towards the AND gate). So we just need to show that if the formula is satisfiable, there is no inference; that is,

for each gadget in the reduction, every set of edges in its constraint is possible to achieve (pointing in) in some consistent assignment. Because each gadget is a gate, this is equivalent to every combination of input values being attainable.

There is a unique consistent assignment for each choice of values for variables, z_1 , and z_2 . We will describe such a choice that achieves each combination of input values for each gadget.

Each fanout depends on a single variable, which can have either value. Each crossover is between two wires connected to different variables, so all combinations are possible. The inputs of each OR gate in a clause can be chosen by setting the input variables as desired.

Consider any of the AND gates combining the outputs of the clauses on one half of the circuit, and assume for simplicity that it's on the positive half. We can make both inputs true (pointing in) or both inputs false (pointing out) by setting all variables true or all variables false, respectively. To make exactly one input true, note that we can choose a single (positive) clause to be unsatisfied: make the three variables in that clause false and all other variables true. Then the clause is unsatisfied, but (since clauses have exactly three literals and there aren't duplicates), every other positive clause is satisfied. By choosing a clause that feeds into either side of the AND gate in question, we can make either of its inputs false and the other true. The same argument shows that the AND gates with z_1 and z_2 can also have any combination of inputs.

Finally, consider the rightmost AND gate, with output r . By hypothesis, the formula is satisfiable, so both inputs can be made true by satisfying the formula and setting z_1 and z_2 to both be true. We can then independently choose to make either input false by flipping z_1 or z_2 as appropriate. ◀

► **Theorem 17.** *PGO uniqueness with free terminals, fanout gates, crossovers, NOT gates, OR gates, AND gates, and fixed terminals is coNP-hard.*

The *fixed terminal* has one port, and its constraint forces the port's value. There are two kinds of fixed terminal; we will use the *fixed true terminal*, which forces the edge to point in.

Proof. We reduce from the complement of 3SAT, which is NP-hard [3]. Refer to Figure 8. Each variable becomes an edge connected to a fanout gate pointing down: that edge pointing down represents the variable being true. Each clause becomes a pair of OR gates connected in the natural way. We place edges connecting variables to clauses in the structure of the formula. If edges cross, we use a crossover gadget. For negated literals, we place a NOT gate along the edge.

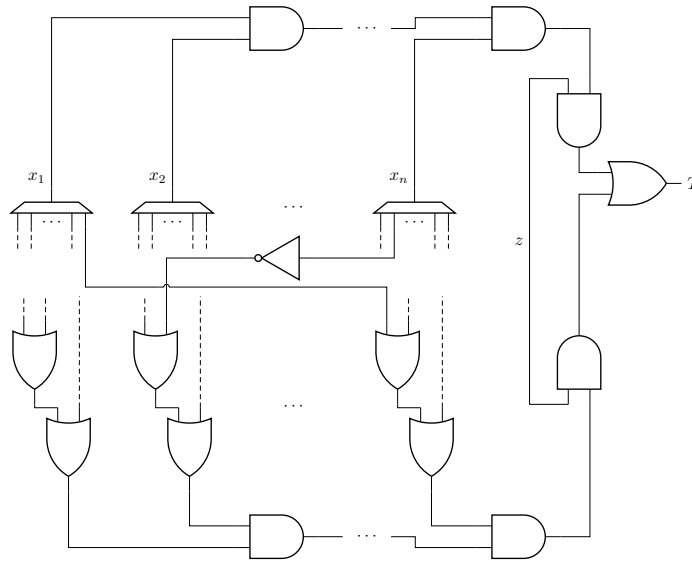
The outputs of the clauses are merged with AND gates, so the edge in the bottom right of Figure 8 points right (and up) exactly when the assignment (based on the orientations of edges representing variables) is satisfying.

In the top section of Figure 8, we merge the other ends of the variable edges with AND gates. The top right edge points right (and down) exactly when all variables are false.

On the right, there is an edge z which points towards one of two AND gates, with the results merged by an OR gate and then run into a fixed true terminal. For the output of that OR gate to be true (point right), either

- z points up, and all variables are false; or
- z points down, and the 3SAT formula is satisfied.

Note that the orientations of all edges are uniquely determined by those of z and variables, even ignoring the fixed terminal. In particular, the network has exactly one satisfying assignment (with z pointing down) for each satisfying assignment of the 3SAT formula, plus exactly one more, which has z pointing up and all variable edges pointing up.



■ **Figure 8** Our reduction for coNP-hardness of PGO uniqueness. See Table 1 for gadget notation.

The input to PGO uniqueness is the network described above and the satisfying assignment with z pointing up. This is the unique satisfying assignment exactly when the 3SAT formula is not satisfiable. ◀

3.4 Simulation

A key feature of our framework is that it allows us to abstractly construct gadgets out of other gadgets, greatly reducing the complexity of the gadgets we need to actually implement in Minesweeper. In particular, the results in Section 3.3 use many gadgets, some of which are hard to build directly, particularly with the properties our hardness proofs require.

► **Definition 18.** A *simulation* using gadgets from S is a network of gadgets from S , except it may have some “dangling” edges incident to only one vertex. Equivalently, the graph contains one special vertex called the “outside world” (which has the trivial constraint).

For *planar* simulations, we require that the dangling edges are in the external face, or equivalently that the graph including the outside world is planar.

See Section 3.5 for several examples of simulations.

► **Definition 19.** Given a simulation, the *simulated gadget* has dangling edges as ports, and its constraint contains each set of dangling edges for which there is a satisfying assignment making exactly those edges point into the simulation (away from the outside world).

In the planar case, the order of the ports is the order of the dangling edges around the simulation, or the reverse of their order around the outside world.

► **Definition 20.** We say S *simulates* G if there is a simulation using gadgets from S where the simulated gadget is G . For a set T , we say that S *simulates* T if S simulates each gadget in T .

For PGO uniqueness and Minesweeper solvability, we will need our simulations to be even better behaved.

► **Definition 21.** A simulation of G is *parsimonious* if for each legal configuration of the edges of G , there is exactly one satisfying assignment of the simulation that orients the dangling edges that way.

We say that S *parsimoniously* simulates G if the relevant simulation is parsimonious, and S *parsimoniously* simulates T if S parsimoniously simulates each element of T .

Much of the point of having a theory of simulations is that they can be composed. This reduces conceptual complexity by letting us break down complicated simulations into a sequence of simpler ones.

► **Lemma 22.** If S simulates T and T simulates G , then S simulates G . Moreover, this composition preserves parsimony.

Proof. In the simulation of G using gadgets from T , replace each gadget with its simulation using gadgets from S . In any satisfying assignment of the new simulation, each component simulation is consistent with the gadget it's simulating, so we can construct a satisfying assignment of the original simulation with the same orientations for dangling edges by looking at only the edges between simulated gadgets.

Conversely, any satisfying assignment of the original simulation can be extended to one of the new simulation by filling in each simulated gadget with an appropriate satisfying assignment. Thus we have a simulation of G using gadgets from S .

If each gadget is replaced with a parsimonious simulation, there is only one way to fill in simulated gadgets this way. So we have defined a bijection between satisfying assignments of the original and new simulations of G . If the original is parsimonious, so is the new simulation. ◀

The other half of the point of simulations is to simplify hardness proofs, so we need them to preserve hardness. This is straightforward for satisfiability.

► **Lemma 23.** If S simulates T , then there is a polynomial-time reduction from PGO satisfiability with T to PGO satisfiability with S .

Proof. Given a network N of gadgets from T , replace each gadget with a (constant-size) simulation using gadgets from S to construct a network N' of gadgets from S .

If there is a satisfying assignment of N' , looking only at the edges connecting simulated gadgets gives a satisfying assignment of N .

If there is a satisfying assignment of N , we can set the edges connecting simulated gadgets to match it, and then each simulated gadget has a local solution compatible with those edges. ◀

This is somewhat more complicated for promise-inference, but it's not so bad with the right definition for the decision problem.

► **Lemma 24.** If S simulates T , then there is a polynomial-time reduction from PGO promise-inference with T to PGO promise-inference with S .

Proof. We are given an instance of promise-inference with T , which is a network N of gadgets from T and semilocal constraints. Construct a network N' of gadgets from S in the same way as above.

To complete the instance of PGO promise-inference with S , we must define semilocal constraints for the vertices in N' . Consider a vertex v , which is inside a simulation of $G \in T$. The semilocal constraint of v shall contain the sets the of edges that point into v in satisfying

25:14 Complexity of PGO Consistency, Promise-Inference, and Uniqueness

assignments of the simulation that are compatible with the semilocal constraint of G (as a vertex of N). These semilocal constraints can be computed in polynomial time because each one requires considering only a simulation, and simulations have constant size.

It remains to check that this instance satisfies the promise, and falls into the same option as the input instance. For the first part of the promise, consider a vertex v in N' and a satisfying assignment. Then v is inside a simulation of some $G \in T$, the corresponding assignment of N (which has only edges between simulated gadgets) is compatible with the semilocal constraint of G . So we have a satisfying assignment of the simulation of G compatible with the semilocal constraint of G , and thus by construction it is compatible with the semilocal constraint of v .

Suppose N has a locally free edge whose orientation is forced. Since every satisfying assignment of N' contains a satisfying assignment of N in the inter-simulation edges, the corresponding edge of N' also has forced orientation. It must also be locally unforced: each orientation is compatible with the semilocal constraints of its vertices in N , and therefore each orientation is compatible with some appropriate satisfying assignment of the simulations of its vertices. Hence N' also has a locally free edge with forced orientation.

Now suppose N falls into the second option, namely there are satisfying assignments achieving every element of its semilocal constraints. Consider a vertex v in N' which is inside a simulation of G , and consider a set L in the semilocal constraint of v . By definition, there is a satisfying assignment of the simulation which makes exactly the edges in L point towards v , and which is compatible with the semilocal constraint of G as a vertex of N . By assumption, there is a satisfying assignment of N which orients the dangling edges of the simulation in the same way. Combining these, and filling in the assignment for other simulations, we obtain a satisfying assignment of N' which directs exactly the edges in L towards v , as desired. ◀

To prove an analogous result for PGO uniqueness, we need our simulations to preserve unique solutions. That is, they should be parsimonious.

► **Lemma 25.** *If S parsimoniously simulates T , then there is a polynomial-time reduction from PGO uniqueness with T to PGO uniqueness with S*

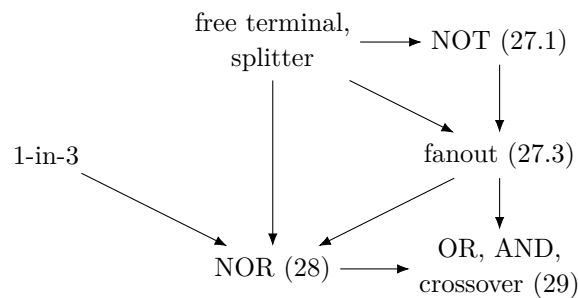
Proof. We are given a network N of gadgets from T and a satisfying assignment A . Construct a network N' of gadgets from S in the same way as the two previous proofs – replace each gadget with its simulation.

Our instance of PGO uniqueness with S also needs a satisfying assignment of N' . To construct one, direct inter-simulation edges to match A , and extend this to a full satisfying assignment A' by consistently orienting edges inside simulations. Since the simulations are parsimonious, there is a unique way to do this for each simulation, and since simulations are constant-size, A' can be computed in polynomial time.

As before, there is a correspondence between satisfying assignments of N and satisfying assignments of N' . This time, however, parsimony ensures that the correspondence is a bijection, since A' is well-defined. In particular, there is a satisfying assignment of N other than A if and only if there is a satisfying assignment of N' other than A' . ◀

3.5 Simpler gadget sets

Now we put the theory of simulations into practice: we will demonstrate several simulations and use them with the results of Section 3.4 to improve the results of Section 3.3 to use more convenient sets of gadgets. The simulations we use are summarized in Figure 9.



■ **Figure 9** The (parsimonious) simulations we use to simplify our gadget set. Each gadget is simulated by the collection of gadgets pointing towards it, except that we will need to build 1-in-3 gadgets, free terminals, and splitters directly.

When we build gadgets in variants of Minesweeper, we will construct gadgets that are like the fanout gate, but may have more than three ports and may have some ports reversed.

► **Definition 26.** A *generalized fanout* is a gadget with at least three ports whose constraint has exactly two sets, which are complements. That is, it has two legal configurations, which differ by flipping all edges.

For instance, the fanout gate is a generalized fanout, and the NOT gate is almost a generalized fanout, except it has only two ports.

► **Lemma 27.** Let F be a generalized fanout. Then F and free terminals parsimoniously simulate

1. the NOT gate
2. any generalized fanout F'
3. fanout gates (with any number of outputs).

Proof. We describe each simulation.

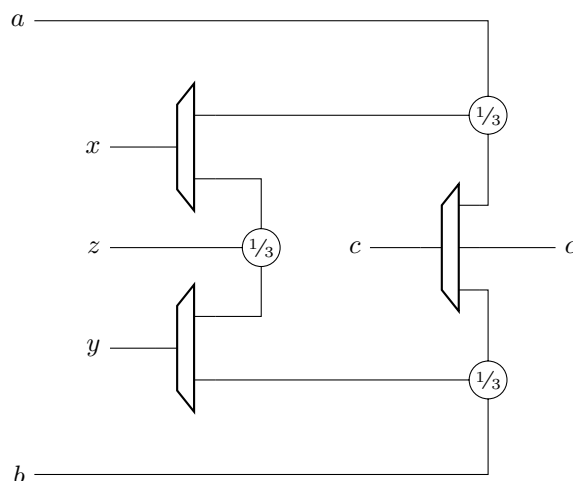
1. Let F have ports P and constraint $\{T, F\}$, where $T \sqcup F = P$. Since $|P| \geq 3$, at least one of T and F has size at least 2; assume without loss of generality that $|T| \geq 2$, and $a, b \in T$. Attach free terminals to all ports of F other than a and b . This simulation has two satisfying assignments, corresponding to T and F . The simulated gadget has ports a and b , and constraint $\{\{a, b\}, \emptyset\}$, so it is the NOT gate.
2. Connect copies of F in a tree until there are at least as many dangling edges as ports of F' . Note that there are exactly two satisfying assignments; all copies of F must flip state together. Now assign each port of F' to a dangling edge, respecting cyclic order. If there are extra dangling edges, put free terminals on them.

The result is a generalized fanout with the same ports as F' , but the partition into the two legal configurations may be wrong. For each port on the wrong side of the partition, attach a NOT gate (composing simulations using Lemma 22). This changes which of the two satisfying assignments has the edge at that port pointing in. Now the simulated gadget partition its ports into two legal configurations in the same way as F' , so it is in fact F' .

3. This is a special case of the above. ◀

► **Lemma 28.** 1-in-3 gates, free terminals, and fanouts parsimoniously simulate the NOR gate.

25:16 Complexity of PGO Consistency, Promise-Inference, and Uniqueness



■ **Figure 10** A simulation of a NOR gate using 1-in-3 gadgets, free terminals, and fanouts.

Proof. The simulation consists of three 1-in-3 gadgets, four free terminals, two 2-way fanouts, and a 3-way fanout, and is shown in Figure 10. The inputs are a and b and the output is c . True means “pointing right”, for inputs, outputs, and labeled free terminals.

The simulation enforces a 1-in-3 constraint on each of $\{a, x, c\}$, $\{b, y, c\}$, and $\{x, y, z\}$. The easiest way to verify that this is a parsimonious simulation of a NOR gate is to list all satisfying assignments:

a	b	x	y	z	c
T	T	F	F	T	F
T	F	F	T	F	F
F	T	T	F	F	F
F	F	F	F	T	T

We see that c is always $\neg(a \vee b)$, and there is a unique assignment for each combination of values for a and b . When comparing against the definition of the NOR gate in Table 1, recall that the correspondence between true/false and in/out is reversed for c relative to a and b . ◀

► **Lemma 29.** *NOR gates and fanout gates parsimoniously simulate OR gates, AND gates, and crossovers.*

Proof. In Section 3.1 we observed that boolean circuits can be converted to gadget networks; these networks can be thought of as parsimonious simulations. The NOR operation is logically complete and can build crossovers in planar circuits [2]. Hence for each gate we want to simulate, we can find a planar circuit of NOR gates that computes it, then convert this into a planar network of NOR-gate gadgets and fanout-gate gadgets. ◀

We now pull everything together by applying the results on simulation in Section 3.4 to the hardness results of Section 3.3, using the simulations in this section. We can either compose reductions or compose simulations using Lemma 22; either way, we obtain our main result about planar graph orientation decision problems.

► **Corollary 30.** *PGO consistency with any generalized fanout, fixed terminals, free terminals, and 1-in-3 gadgets is NP-hard. PGO promise-inference and PGO uniqueness with the same set of gadgets are coNP-hard.*

3.6 Applying the framework

We now conclude the development of our gadget framework by discussing what it takes to use it to prove hardness for a game like Minesweeper. We do not attempt to precisely define “like Minesweeper”, and this results of this section are not stated precisely. Ultimately, one needs a polynomial-time reduction from the appropriate PGO problem to the corresponding question for the game under consideration, but the games we will consider are similar enough that the reductions share most of their structure, and the discussion here applies to many or all of them.

It is conceptually helpful to distinguish between gadgets as abstract formal objects and the constructions we build in concrete games to embed gadgets in them. We call the latter “implementations”, to parallel the distinction between the specification and the implementation of a function.

► **Definition 31.** An *implementation* of a gadget is a construction in a game like Minesweeper which behaves like the gadget, in that there are covered cells (usually on the boundary of the construction) corresponding to ports, and the construction has a solution exactly when the configuration of edges represented by the values of those cells is legal for the gadget.

Unfortunately, because it matches how the term is typically used, we will frequently call implementations “gadgets” when the intended meaning is clear from context.

For implementations to interact in a way that simulates gadget networks, we need some kind of *wire*. Each of our wires will have two possible states, corresponding to the two orientations of the edge it represents. We must be able to route our wires in an arbitrary planar graph – the ability to extend and turn them is sufficient. We need to be able to plug wires into the ports of implementations, which sometimes requires the ability to adjust the alignment of a wire by a single cell.

Our wires and implementations also need to have the following properties.

- Every solution uses the same number of mines. This means the total number of mines in the puzzle doesn’t reveal any information that might break our reduction.
- Every uncovered cell has a clue, as in most implementations of Minesweeper. Some implementations and variants allow cells that are known to be safe but don’t have additional information (e.g. showing a question mark instead of a number), but we don’t want to rely on this feature.
- All given mines can be deduced from uncovered non-mine cells. This provides robustness against changes in the definition, e.g. whether the locations of mines can be provided as part of a puzzle in addition to uncovered cells without mines.

Implementing some gadgets in a game allows us to build a network of those gadgets in the game. The resulting instance of the game has a solution if and only if the network is satisfiable.

▷ **Claim 32.** If a game like Minesweeper has implementations of the gadgets in S , then there is a polynomial-time reduction from PGO consistency with S to the game’s consistency problem.

Note that our definition of Minesweeper consistency in Section 2 is specific to Minesweeper. For other games or variants, “partial board” and “consistent” need to be defined appropriately, but the other definitions (for inference and solvability) work as is.

For the other two decision problems, we need well-behaved implementations.

25:18 Complexity of PGO Consistency, Promise-Inference, and Uniqueness

► **Definition 33.** *An implementation is **silent** if clicking a known-safe internal cell can never reveal information that wasn't already known. In other words, for each covered internal cell, the information that cell provides when clicked (e.g. the number of adjacent mines) is the same in every solution in which it doesn't have a mine.*

▷ **Claim 34.** If a game like Minesweeper has silent implementations of the gadgets in S , then there is a polynomial-time reduction from PGO promise-inference with S to the game's inference problem.

Proof. Embed a network of gadgets from an instance of PGO promise-inference with S in the game. For each (constant size) gadget in the network, consider all solutions to its implementation which are consistent with its semilocal constraint. If there are **commonalities**, or cells which are safe in all such solution (or mines in all such solutions), “click on” them, revealing them in the instance of the game. Silence guarantees that the information revealed is the same in all such solutions, so the information to put on that cell can be determined from the semilocal constraint. If a commonality dictates the orientation of an edge, we click on all covered cells in the edge in the same way. It doesn't matter whether this includes the cell(s) representing the port on the other end of the edge – if that gadget's semilocal constraint doesn't force the edge in the same way, we must be in the first case of the promise so there's an inferable locally unforced edge anyway.

If the network has an inferable locally unforced edge, then cells inside the wire representing that edge (or cells representing the ports that edge connects to) can be inferred.

Otherwise, we are in the second case of the promise. We've already clicked all cells that can be deduced from each semilocal constraint – for any cell that is still covered, the gadget (or wire) it's in has solutions compatible with its semilocal constraint in which that cell has a mine and in which it doesn't. Thus the entire instance also has both of these kinds of solutions because we are promised that every element of a semilocal constraint is achieved by a satisfying assignment. That is, the cell in question can't be inferred. This is where it is crucial that we use promise-inference, and not a simpler PGO inference decision problem. ◁

For PGO uniqueness and Minesweeper solvability, we also need parsimony as we did for Lemma 25.

► **Definition 35.** *An implementation is **parsimonious** if it has exactly one solution corresponding to each legal configuration of the gadget.*

▷ **Claim 36.** If a game like Minesweeper has silent parsimonious implementations of the gadgets in S , then there is a polynomial-time reduction from PGO uniqueness with S to the game's solvability problem.

Proof. Embed a network N of gadgets from an instance of PGO uniqueness with S in the game. Thanks to parsimony, satisfying assignments of the network are in bijection with consistent solutions to this instance of the game. The secret solution is the solution corresponding to the given satisfying assignment of N , but cells are only uncovered if they are uncovered in the embedding of N , which doesn't depend on the satisfying assignment.

If N has another satisfying assignment, the game instance has multiple solutions. The player may be able to safely click some cells – perhaps the orientation of some edge is forced, or they can deduce that a cell inside a gadget is safe. However, since our implementations are silent, the player can't gain any information by doing this. In particular, all solutions consistent with the initial state of the game are also consistent after the player makes any sequence of safe clicks. In order to solve the instance, they would need to distinguish between these consistent solutions, which is impossible.

Conversely, suppose the only satisfying assignment to N is the one given to the reduction. Then the secret solution is the only solution consistent with the initial state of the game (we need parsimony to ensure there is only one). Thus the value of every cell can be deduced, and all cells without mines can be safely clicked in any order. \triangleleft

It follows that if we can silently and parsimoniously implement that gadgets needed by Corollary 30, we have hardness of consistency, inference, and solvability for the game in question. To simplify a little further, fixed terminals are trivial to construct: just let a wire end, and reveal some cells in it to force its orientation. Alternatively, modify a free terminal (possibly including a bit of wire) by revealing cells that determine its configuration.

► **Corollary 37.** *Suppose that for some game like Minesweeper, we have silent parsimonious implementations of any generalized fanout, free terminals, and 1-in-3 gadgets, and we are able to route, turn, and adjust the alignment of wires enough to embed any planar network of those gadgets. Then consistency is NP-hard, and inference and solvability are coNP-hard.*

We now go through one final layer of abstraction, which will handle routing of wires and filling empty space. We lay a network of the gadgets above on a square grid, where edges run horizontally and vertically and can turn. In particular, each tile contains either one of the gadgets above, a straight edge section, and turning edge section, or nothing.

This lets us reduce from problems about planar networks on grids, so we can construct straight and turning tiles instead of describing general wire routing.

► **Corollary 38.** *Suppose that for some game like Minesweeper, we have silent parsimonious implementations of gadgets which are all square tiles that fit in a grid and interact appropriately with adjacent tiles. Suppose in particular that we have empty tiles, straight wires, turning wires, any generalized fanout, free terminals, and 1-in-3 gadgets. Then consistency is NP-hard, and inference and solvability are coNP-hard.*

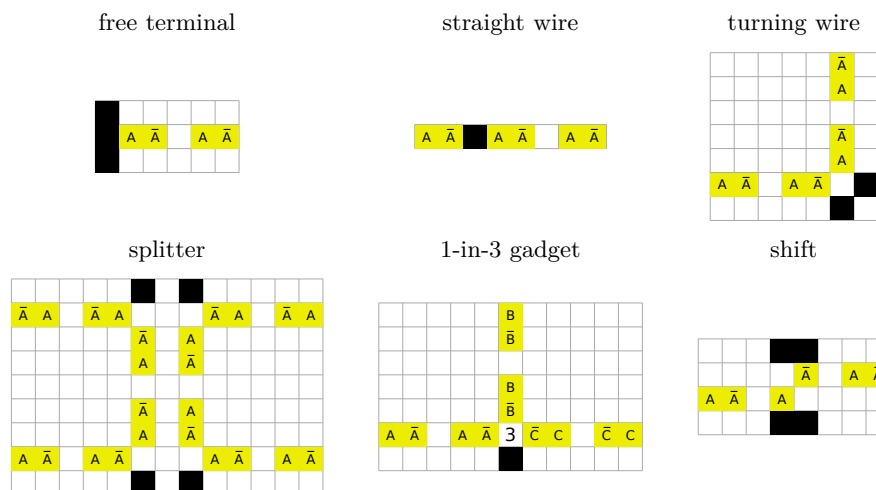
For a few of our applications, it is more convenient to build a 2-in-3 or a 1-in-4 instead of a 1-in-3 gadget. These are also sufficient, since they can simulate a 1-in-3 gadget, using NOT gates and a free terminal, respectively.

4 Hardness proofs

To demonstrate the utility of this PGO framework and the ease with which it facilitates writing hardness proofs, we provide a number of example applications that prove hardness of Minesweeper and Minesweeper-like games, found in the full version of this paper. We reproduce some gadgets for standard Minesweeper here, as a representative example.

In the original Minesweeper game, the player does not start with any cells revealed, and the player starts by clicking any cell on the entirely unrevealed board (which the game guarantees to be safe). Hence, we construct the following “transparent” gadgets, where all clues can be deduced starting from this initial state. This shows that Minesweeper solvability is coNP-complete even in the setting where the player is initially given no clues. (To guarantee the click is safe, we can double the width of the grid, and delay the decision of which half to place the circuit on until after the click is made.)

25:20 Complexity of PGO Consistency, Promise-Inference, and Uniqueness



References

- 1 E. Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.
- 2 Leslie M. Goldschlager. The monotone and planar circuit value problems are log space complete for p. *ACM sigact news*, 9(2):25–29, 1977.
- 3 Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.
- 4 Sadie Kaye. Minesweeper is np-complete. *The Mathematical Intelligencer*, 22:9–15, 2000. doi:10.1007/BF03025367.
- 5 Philippe Laroche. Planar 1-in-3 satisfiability is np-complete. *Comptes Rendus de L Academie des Sciences Serie I-Mathematique*, 316(4):389–392, 1993.
- 6 Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be np-complete but is hard nonetheless. *The Mathematical Intelligencer*, 33:5–17, 2011. doi:10.1007/s00283-011-9256-x.
- 7 Alex Thieme and Twan Basten. Minesweeper is difficult indeed! *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, 13560:472, 2022.

Coordinating “7 Billion Humans” Is Hard

Alessandro Panconesi  

Sapienza University of Rome, Italy

Pietro Maria Posta  

Sapienza University of Rome, Italy

Mirko Giacchini  

Sapienza University of Rome, Italy

Abstract

In the video game “7 Billion Humans”, the player is requested to direct a group of workers to various destinations by writing a program that is executed simultaneously on each worker. While the game is quite rich and, indeed, it is considered one of the best games for beginners to learn the basics of programming, we show that even extremely simple versions are already NP-Hard or PSPACE-Hard.

2012 ACM Subject Classification Theory of computation → Complexity classes

Keywords and phrases video games, computational complexity, NP, PSPACE

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.26

1 Introduction

In a world where robots have occupied every possible job position, humans are finally free to dedicate themselves to their favourite pastimes. However, in this utopian world, the work ethic of yesteryear reigns supreme and the only thing humans desire is good-paying jobs. To appease them, the robots construct one colossal building, so colossal to be visible from outer space, and hire all 7 billion humans living on Earth as well-paid white-collar workers. Now, robots are faced with the challenge of coordinating the human workforce to keep them all constantly entertained.

It is in such a world that “7 Billion Humans” takes place. Released in 2018 as the successor of “Human Resource Machine”, “7 Billion Humans” is a puzzle video game praised by tech reviewers as one of the best games to learn the basics of programming [16, 17, 18]. The player, who takes on the role of the robots in the story, must coordinate a group of workers by specifying their actions by means of an ad-hoc programming language. While the programming language in the game is quite rich, also containing “if” statements and “go-to” commands, in this paper, we will focus on the core mechanic of the game: moving the workers. In particular, the goal is to move the workers into an accepted configuration by writing a program that is executed simultaneously by each one of them. While moving, the workers will have to navigate through walls, desks, plants, and other objects that block their movement, as well as holes where workers can fall through. This extremely limited set of commands and objects constitutes the core of the game since they appear in essentially all the game levels. Even under these limitations, we show that the player (and hence the robots) will have a hard time coordinating the humans.

Our work falls within the rich area of video-game computational complexity. In recent years, several extremely popular video games have been proven to be NP-Hard or PSPACE-Hard, such as Super Mario Bros. and other Nintendo games [3, 6], Portal and several other 3D games [5], Trainyard [1, 2], Candy Crush [9], and many others [10, 14].

Let us now describe the game “7 Billion Humans” and our contributions.



© Alessandro Panconesi, Pietro Maria Posta, and Mirko Giacchini;
licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 26; pp. 26:1–26:16

Leibniz International Proceedings in Informatics

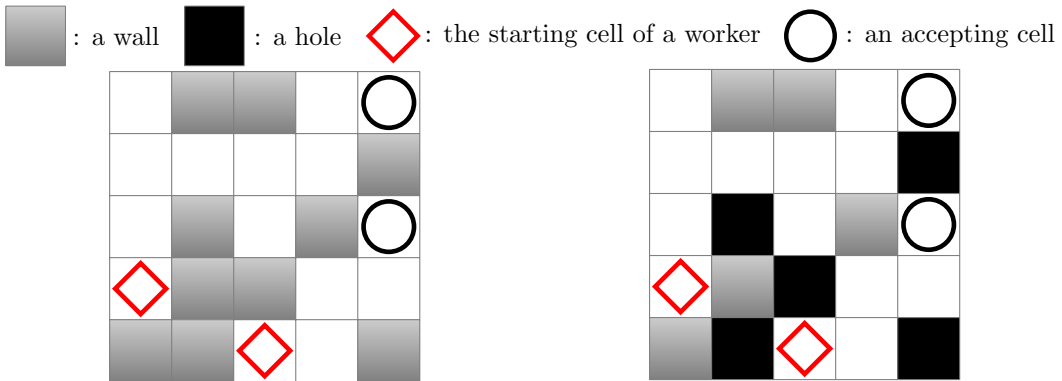


LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.1 Our Contributions

A level of “7 Billion Humans” consists of a grid of cells containing *workers* and *objects*. The player must write a program, which is executed by *every* worker simultaneously, in order to satisfy the requests of the level designer. We consider only the most basic kind of request: the workers must be moved from their starting configuration into an accepted configuration. More precisely, some cells of the grid are *accepting cells* and to solve a level, all the workers must be standing on an accepting cell after executing the program. There are many commands at the disposal of the user to write the program, but we make use only of the most basic one: `step {direction}`, which is used to move the workers (all at the same time) by one cell in a given direction, which can be one of `up`, `down`, `left`, `right`, `up-left`, `up-right`, `down-left`, or `down-right`. A cell can either be empty or contain an object.¹ The simplest type of objects are the *walls* and, as one might expect, stepping into a wall results in a non-movement.²

We show in figure 1a an example of a level. We abbreviate the command to step in one direction with `u`, `d`, `l`, `r`, `(ul)`, `(ur)`, `(dl)`, `(dr)`, and a sequence of steps in the same direction using exponentiation (e.g., `r4` instead of `rrrr`). For a generic finite alphabet Σ , we denote with Σ^* the set of all finite strings consisting of symbols of Σ . A program is therefore represented as a string over the alphabet $\{u, d, l, r, (ul), (ur), (dl), (dr)\}$. For simplicity, when a program $\pi \in \{u, d, l, r, (ul), (ur), (dl), (dr)\}^*$ solves a level, we also say that the level *accepts* the string π .



(a) This level can be solved, for example, by the program `u2r3uru` or by the program `ru2r3ur`. Instead, `ruru` does not solve the level because only one worker reaches an accepting cell. (b) This level can be solved by the program `ru2r3ur`. Instead, the program `u2r3uru` does not work anymore because one of the two workers would get stuck in a hole.

■ **Figure 1** Two examples of game levels. The level on the left contains only walls and empty cells, while the one on the right also contains holes. We assume that the grids are surrounded by walls.

The decision problem that we consider is: given a level of “7 Billion Humans”, say if the level is solvable or not. Since we included only the essential elements of the game, we call this problem **7BH-Essential**. We will show that even this extreme simplification is already NP-Hard.

¹ We can assume that workers start in empty cells and the accepting cells are all empty.
² In the game there are also other obstacles, such as desks and plants: since they all act the same, we will always talk about walls. Other workers also behave as obstacles if hit. However, this will never happen in our reductions.

► **Theorem 1.** *It is NP-Hard to check if a given level of “7 Billion Humans” is solvable, even using only walls, empty cells, and the `step` command. That is, `7BH-Essential` is NP-Hard.*

Holes are another common object in the game. When a worker steps on a cell containing a *hole*, it gets stuck for the rest of the computation. Since our goal is to have each worker on an accepting cell, even if a single worker steps on a hole, the level is lost. An example of a level with holes is shown in figure 1b. We call `7BH-Holes` the decision problem previously described where we add *holes* in addition to the already mentioned elements. Adding holes might make the problem more difficult, in fact, `7BH-Holes` is PSPACE-Hard.

► **Theorem 2.** *It is PSPACE-Complete to check if a given level of “7 Billion Humans” containing only walls, holes, empty cells, and using only the `step` command, is solvable. That is, `7BH-Holes` is PSPACE-Complete.*

The paper is organized as follows. In Section 2 we highlight some connections between ours and other problems. In Sections 3 and 4 we prove our two main theorems and finally, in Section 5, we discuss some final remarks.

2 Relations with other problems

In our reductions, the game level of “7 Billion Humans” will be divided into isolated sub-levels, each containing a single worker. Then, to solve the level, all the sub-levels must be solved simultaneously. Moreover, in our reductions, we will prevent diagonal movements: the only admissible directions are `right`, `left`, `up`, and `down`. This special structure of the level can be used to draw some relations with other problems.

2.1 Simultaneous Maze Solving

Each sub-level can be interpreted as a grid maze: the worker must find a path from its starting position to one of the accepting cells,³ avoiding the holes and navigating through the walls. Our results, then, entail that solving multiple mazes simultaneously is NP-Hard (Theorem 1), or PSPACE-Hard if the mazes can contain holes (Theorem 2). The only other work on the topic, to the best of our knowledge, is the one of Funke et al. [7], which, however, studies very special mazes that are always solvable simultaneously.

2.2 Intersection Non-Emptiness

Each sub-level can also be interpreted as a deterministic finite automaton (DFA for short). In particular, a sub-level $w \times h$ naturally translates into a DFA with at most $w \cdot h$ states (corresponding to the cells), and with the transition function on the alphabet $\{\text{u}, \text{d}, \text{l}, \text{r}\}$ that simulates the behavior of the cells. Solving all the sub-levels is equivalent to finding a string that is accepted by a set of DFAs: this is a fundamental problem in automata theory known as Intersection Non-Emptiness Problem [4, 12, 15] and first shown to be PSPACE-Complete by Kozen [11]. The structure of our DFAs is very special: the undirected transition graph, excluding self-loops, is a subgraph of the $w \times h$ grid graph. Therefore, our Theorem 2 entails that Intersection Non-Emptiness is PSPACE-Complete even with this class of automata.

³ Note that passing upon an accepting cell is not enough: each worker must be standing on an accepting cell at the end of the sequence of moves.

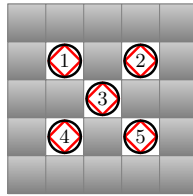
To the best of our knowledge, given the strong restrictions that we have to make on the DFAs, our results are not derivable from existing work. As an example, in the original PSPACE-Hardness proof of Kozen [11], the standard construction of the DFAs contains vertices having in-degree $(|Q| + |\Sigma| + 1)^3$, where Q and Σ are the set of states and input symbols of a Turing Machine, therefore, the in-degree is at least 27 and possibly much larger. Instead, our DFAs have an in-degree of at most 8, considering self-loops. Moreover, in such proof, the automata use several one-way transitions; instead, the transitions of our DFAs are reversible (except for states associated with holes).

3 NP-Hardness of 7BH-Essential

In this section we prove Theorem 1. In particular, we show a polynomial-time reduction from Positive 1-in-3-SAT, notoriously known to be NP-Hard (see, e.g., [8, page 259, problem LO4]), to 7BH-Essential.

► **Definition 3** (Positive 1-in-3-SAT). *The input of Positive 1-in-3-SAT consists of n boolean variables, x_1, x_2, \dots, x_n , and a set of m clauses, each containing exactly three distinct positive variables (i.e., there are no negated literals). The goal is to find a truth assignment to the variables such that each clause contains exactly one true variable.*

Fixed an instance of Positive 1-in-3-SAT, we make use of three types of gadgets: (i) the *diagonal* gadget, to prevent diagonal movements, (ii) the *assignment gadget* that, intuitively, allows assigning truth values to the variables, and (iii) one *clause gadget* for each clause, to ensure that the truth assignment satisfies the original 1-in-3 formula. Each gadget will be built with multiple independent sub-levels that must be solved simultaneously. The final game level is obtained by stacking the sub-levels together and isolating them via walls.



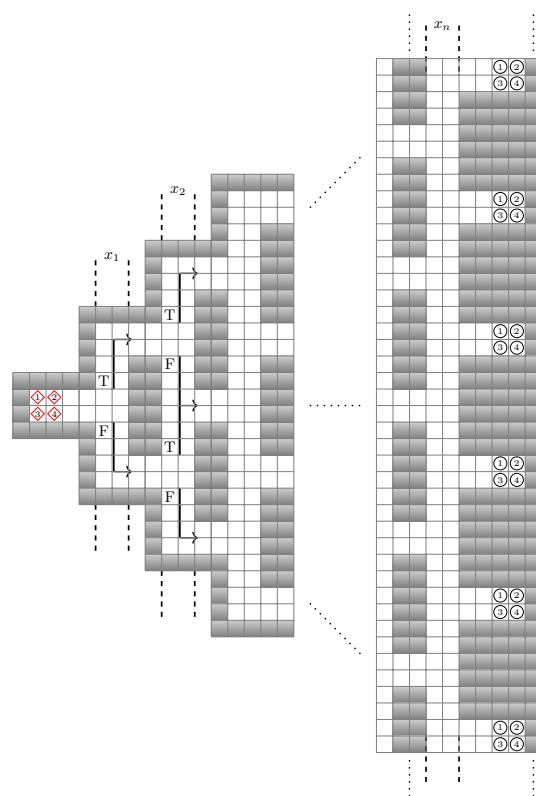
■ **Figure 2** The *diagonal* gadget, used to prevent diagonal movements, consists of five sub-levels: for $i \in \{1, 2, 3, 4, 5\}$, the i -th sub-level contains only the worker and the accepting cell labeled with i .

The diagonal gadget is reported in figure 2. It consists of five sub-levels that we draw together for brevity. In particular, for $i \in \{1, 2, 3, 4, 5\}$, the i -th sub-level contains only the one worker and the one accepting cell labeled with i . Suppose the program contains a diagonal movement (i.e., (u1), (ur), (d1), or (dr)), then, worker 3 would “overlap”⁴ with another worker and it would become impossible to solve all the five sub-levels of the gadget (indeed, once two workers are overlapped in different sub-levels having the same walls, it is impossible to separate them).

3.1 Assignment Gadget

This gadget consists of four sub-levels, reported together in figure 3. For $i \in \{1, 2, 3, 4\}$, the i -th sub-level contains only the i -th worker and the accepting cells labeled with i .

⁴ the workers are in different sub-levels, so they “overlap” if we imagine the sub-levels on top of each other



■ **Figure 3** Assignment Gadget. It consists of four sub-levels, drawn together for brevity. In particular, the i -th sub-level, for $i \in \{1, 2, 3, 4\}$, contains only the worker with label i on the left, and only the accepting cells with label i on the right. The workers select the truth value of the variables by moving **up** or **down**.

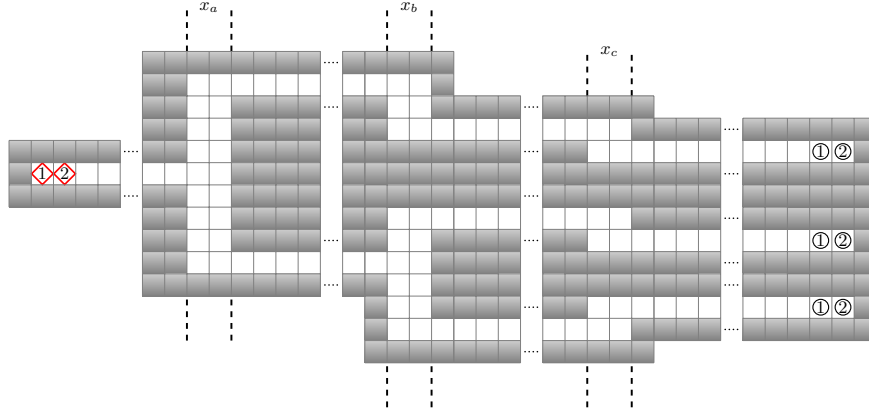
By using four sub-levels, the gadget ensures that the workers cannot hit the walls, indeed, if that happened, two workers would overlap and at least one sub-level would become unsolvable (e.g., using the program \mathbf{r}^5 , workers 1 and 2 overlap, making it impossible for them to simultaneously stand on an accepting cell). This property will be used in the clause gadgets.

Since the accepting cells are at the right end of the sub-levels, the workers must pass through all the variables, and select their truth values by moving **up** or **down**. Specifically, when the workers are in correspondence with the variable x_i (that is, when the workers are in columns $4i + 1$ and $4i + 2$, assuming that the columns are numbered from left to right starting with 0 on the left wall), moving **up** will set x_i to true in all clauses while moving **down** will set it to false. Intuitively, we can think that the workers will move via a program of the form $\mathbf{r}^4 \sigma_1^4 \mathbf{r}^4 \dots \sigma_n^4 \mathbf{r}^4$ with $\sigma_i \in \{\mathbf{u}, \mathbf{d}\}$, and σ_i determines the truth value of the i -th variable ($\mathbf{u} = \text{True}$, $\mathbf{d} = \text{False}$).

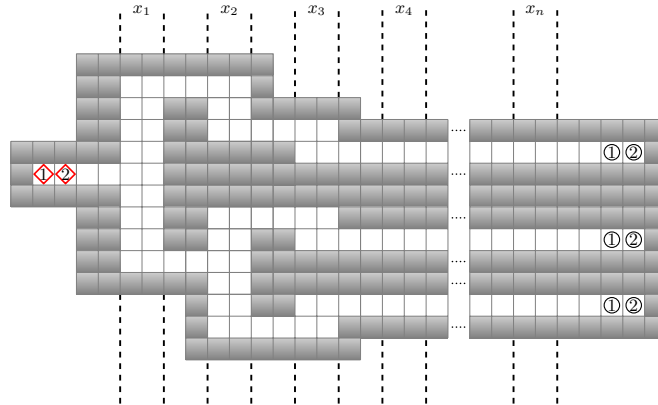
► **Remark.** Note that the workers have more freedom of movement than what we would like. For example, they can move **left**, and in correspondence with a variable, they can move **up** and **down** multiple times before going **right**. However, our clause gadgets are such that this extra freedom does not permit cheating. More precisely, if the level can be solved, then it can also be solved by a program of the form $\mathbf{r}^4 \sigma_1^4 \mathbf{r}^4 \dots \sigma_n^4 \mathbf{r}^4$ with $\sigma_i \in \{\mathbf{u}, \mathbf{d}\}$.

3.2 Clause Gadgets

For each clause $C = (x_a, x_b, x_c)$, with $a < b < c$, we create a clause gadget consisting of two sub-levels. In figure 4a, we report the sub-levels for the generic clause C . For the reader’s convenience, we also show, in figure 4b, the gadget for the clause (x_1, x_2, x_3) . For $i \in \{1, 2\}$, the i -th sub-level contains only the i -th worker and the accepting cells labeled with i .



(a) Gadget for the generic clause (x_a, x_b, x_c) , with $1 \leq a < b < c \leq n$.



(b) Gadget for the clause (x_1, x_2, x_3) . We can handle all the clauses by stretching opportune sections of the gadget (see figure 4a).

■ **Figure 4** Figure (a) shows the Clause Gadget for the generic clause (x_a, x_b, x_c) . It consists of two sub-levels drawn together for brevity. The i -th sub-level, $i \in \{1, 2\}$, contains only the worker with label i on the left and the accepting cells with label i on the right. The columns associated with the variables are aligned with those of the assignment gadget (figure 3). Solely for the sake of clarity, we also show in figure (b) the gadget for the particular clause (x_1, x_2, x_3) .

Let us first argue that the flexibility of the *assignment gadget* cannot be exploited to generate invalid truth assignments. Observe that the two workers of the clause gadget (of the two distinct sub-levels) must always remain side by side, indeed, if they were to split, the assignment gadget would become unsolvable since a worker would have hit a wall. Given that the two workers must remain side by side, the workers in the clause gadget cannot hit a wall horizontally (otherwise they would overlap, making the level unsolvable). This means that using the `left` command is useless: in fact, it can only be used to go back to a variable and possibly change its truth value in *all* the clauses. Moreover, moving `up` and `down` multiple times when selecting a truth value does not lead to benefits either. Indeed, assuming that

the user moves `left` or `right` only when none of the workers would hit a wall, then only the last `up` or `down` movement is used to decide the truth value of the variable in *all* the clauses. In other words, if the final game level is solvable, then it is also solvable by a program of the form $\mathbf{r}^4\sigma_1^4\mathbf{r}^4\dots\sigma_n^4\mathbf{r}^4$ with $\sigma_i \in \{\mathbf{u}, \mathbf{d}\}$.

Now, observe that the workers of the clause gadget for $C = (x_a, x_b, x_c)$ arrive at an accepting cell if and only if they move `up` exactly once in correspondence with x_a , x_b , or x_c . Note that the vertical movement in correspondence with the other variables is ignored. Since the `up` movement is associated with the “True” truth value, the clause gadget behaves exactly like a clause of the Positive 1-in-3-SAT formula.

3.3 Conclusion of the proof

Proof of Theorem 1. Suppose the Positive 1-in-3-SAT instance is solvable, and let $\alpha_i \in \{\text{True}, \text{False}\}$ for $i \in \{1, \dots, n\}$ be the truth assignment to the variables that solves the

instance. Then, letting $\sigma_i = \begin{cases} \mathbf{u}, & \text{if } \alpha_i = \text{True} \\ \mathbf{d}, & \text{if } \alpha_i = \text{False} \end{cases}$ the program $\mathbf{r}^4\sigma_1^4\mathbf{r}^4\dots\sigma_n^4\mathbf{r}^4$ solves the

corresponding 7BH-Essential instance. Conversely, if the 7BH-Essential instance is solvable, we can assume without loss of generality that it is solved by a program of the form

$\mathbf{r}^4\sigma_1^4\mathbf{r}^4\dots\sigma_n^4\mathbf{r}^4$ with $\sigma_i \in \{\mathbf{u}, \mathbf{d}\}$. Let $\alpha_i = \begin{cases} \text{True}, & \text{if } \sigma_i = \mathbf{u} \\ \text{False}, & \text{if } \sigma_i = \mathbf{d} \end{cases}$ then, as we argued, α satisfies

all the 1-in-3-SAT clauses.

Moreover, note that each sub-level of the assignment gadget contains $O(n^2)$ cells, each sub-level of a clause gadget contains $O(n)$ cells, and the five sub-levels of the diagonal gadget have a constant number of cells. Therefore, the complete game level obtained by stacking all the sub-levels contains $O(n^2 + nm)$ cells and can be constructed in polynomial time. Our reduction is complete. ◀

4 PSPACE-Completeness of 7BH-Holes

In this section, we prove Theorem 2. For a positive integer z , we use the notation $[z] = \{1, 2, \dots, z\}$. Let us start by showing that 7BH-Holes can be solved in polynomial space.

► **Observation 4.** *7BH-Holes* \in PSPACE

Proof. Consider a game level $n \times m$ containing k workers. Since the workers are the only non-static elements of the level, each cell can be in at most one of two states: either it contains a worker, or it does not. Then, there are at most $\binom{nm}{k} \leq 2^{nm}$ possible configurations for the level. Then, consider the non-deterministic Turing Machine M that, given in input a level of 7BH-Holes, maintains the current configuration of the level and a counter of the number of steps performed so far. If all the workers are standing on an accepting cell, then M accepts; if instead the counter exceeds 2^{nm} , then M rejects. In all other cases, M guesses non-deterministically the next step in $\{1, \mathbf{r}, \mathbf{u}, \mathbf{d}, (\mathbf{u}\mathbf{l}), (\mathbf{u}\mathbf{r}), (\mathbf{d}\mathbf{l}), (\mathbf{d}\mathbf{r})\}$, updating the configuration and increasing the counter accordingly. It is evident that M solves 7BH-Holes because if the level is solvable, there is a solution using at most 2^{nm} instructions. Moreover, the space required in any computation branch is $O(nm + \log(2^{nm})) = O(nm)$. Therefore, we showed that 7BH-Holes \in NPSPACE, then, by Savitch’s theorem [13], 7BH-Holes \in PSPACE. ◀

Now it remains to show that 7BH-Holes is PSPACE-Hard. We do so by exhibiting a polynomial-time reduction from the intersection non-emptiness problem for finite automata.

4.1 Intersection Non-Emptiness Problem

Recall that a deterministic finite automaton (DFA for short) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $q_0 \in Q$ is the starting state and $F \subseteq Q$ is the set of accepting states. The language accepted by the DFA A , called $L(A)$, is the set of strings $x \in \Sigma^*$ such that, starting from q_0 and applying δ repeatedly, A ends up in an accepting state.

The following is a classic decision problem in automata theory and it was proved to be PSPACE-Complete by Kozen [11]:

► **Definition 5** (Intersection Non-Emptiness Problem). *Given in input a set of k DFAs $\{A_1, A_2, \dots, A_k\}$, with $A_i = (Q_i, \Sigma, \delta_i, q_0^i, F_i)$ for $i \in [k]$, say if $\bigcap_{i=1}^k L(A_i) \neq \emptyset$*

To simplify the exposition, let us assume that all the DFAs have the same set of states Q and the same starting state q_0 . This is without loss of generality because we can rename the states and add fictitious extra states, without changing the languages of the automata.

4.2 Reduction Overview

Consider an instance $I = \{A_1, A_2, \dots, A_k\}$ of the Intersection Non-Emptiness Problem, with $A_i = (Q, \Sigma, \delta_i, q_0, F_i)$ for each $i \in [k]$. Without loss of generality, from now on we assume that $|Q| = n$, $Q = \{q_0, q_1, \dots, q_{n-1}\}$, and $|\Sigma| = m$, $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_m\}$. Note that our reduction must be polynomial in k, n , and m .

We represent the computation of the DFAs using a string. Specifically, let $\Gamma = \Sigma \cup Q \cup \{\#\}$, where $\#$ is a new symbol, and consider a string of the following form:

$$R_1 \# R_2 \# \dots R_r \# \in \Gamma^*$$

where:

$$\begin{aligned} R_j &= \sigma^{(j)} q^{(1,j)} q^{(2,j)} \dots q^{(k,j)} && \text{for } j \in [r], \\ \sigma^{(j)} &\in \Sigma \text{ and } q^{(i,j)} \in Q && \text{for } i \in [k], j \in [r] \end{aligned}$$

Call \mathcal{R} the set of all such strings. Intuitively, R_j contains the j -th input symbol and the state of the DFAs after processing the first $j - 1$ input symbols.

We say that a string $R_1 \# R_2 \# \dots R_r \# \in \mathcal{R}$ is *accepting* if it describes a valid accepting computation for all the automata, that is, if for all $i \in [k]$ the following holds:

- $q^{(i,1)} = q_0$
- $q^{(i,j+1)} = \delta_i(q^{(i,j)}, \sigma^{(j)})$, for all $j \in [r - 1]$
- $q^{(i,r)} \in F_i$

The main idea of our reduction is to define (i) an encoding of the alphabet Γ with strings in $\{1, r, u, d\}^*$, and (ii) a game level \mathcal{G} of 7BH-Holes, such that, a program solves \mathcal{G} if and only if it is an encoding of some accepting string in \mathcal{R} . This will be enough to conclude our reduction. Indeed, finding an accepting string in \mathcal{R} is clearly equivalent to finding a string accepted by all the DFAs. More formally:

► **Observation 6.** *Given an instance $\{A_1, A_2, \dots, A_k\}$ of the Intersection Non-Emptiness Problem, $\bigcap_{i=1}^k L(A_i) \neq \emptyset \iff \exists x \in \mathcal{R}$ accepting*

Proof. If $\sigma_1 \sigma_2 \dots \sigma_r \in \bigcap_{i=1}^k L(A_i)$, then the string $R_1 \# R_2 \# \dots R_{r+1} \#$ where (i) $\sigma^{(i)} = \sigma_i$ for $i \in [r]$ and $\sigma^{(r+1)}$ is any symbol in Σ , and (ii) the states are set according to the computation, is an accepting string. Conversely, if $R_1 \# R_2 \# \dots R_r \# \in \mathcal{R}$ is accepting, then $\sigma^{(1)} \sigma^{(2)} \dots \sigma^{(r-1)} \in \bigcap_{i=1}^k L(A_i)$ (if $r = 1$, the empty string is accepted by all DFAs). ◀

4.3 The Encoding

We first associate an integer value to each element in $Q \cup \Sigma = \{q_0, \dots, q_{n-1}, \sigma_1, \dots, \sigma_m\}$, specifically, we define $\text{num} : Q \cup \Sigma \rightarrow \mathbb{N}$ as:

$$\begin{aligned} \text{num}(q_i) &= 9 \cdot (k+2) \cdot (i+1) & \forall i \in \{0, 1, \dots, n-1\} \\ \text{num}(\sigma_i) &= 9 \cdot (k+2) \cdot (n+2) \cdot (i+1) & \forall i \in [m] \end{aligned}$$

Let us now define \mathcal{C} , the set of *clockwise* strings⁵, as:

$$\mathcal{C} = \{\mathbf{r}^{x_1} \mathbf{d}^{x_2} \mathbf{l}^{x_3} \mathbf{u}^{x_4} \mid x_1, x_2, x_3, x_4 \geq 6\}$$

Our encoding will associate to each element of Γ a subset of clockwise strings, specifically, let $\text{enc} : \Gamma \rightarrow \mathcal{P}(\mathcal{C})$, where $\mathcal{P}(\mathcal{C})$ is the powerset of set \mathcal{C} , be defined as:

$$\text{enc}(\gamma) = \{\mathbf{r}^{\text{num}(\gamma)} \mathbf{d}^{\text{num}(\gamma)} \mathbf{l}^{x_3} \mathbf{u}^{x_4} \in \mathcal{C} \mid x_3, x_4 \geq 6\} \quad \forall \gamma \in Q \cup \Sigma \quad (1)$$

$$\text{enc}(\#) = \{\mathbf{r}^{w\#} \mathbf{d}^{x\#} \mathbf{l}^{y\#} \mathbf{u}^{z\#}\} \quad \text{where:} \quad (2)$$

$$\begin{aligned} w\# &= 9 \cdot (k+2) \cdot (n+2) \cdot (m+2) \\ x\# &= 18 \cdot (k+2) \cdot (n+2) \cdot (m+2) \\ y\# &= w\# + 4k + 3 \\ z\# &= x\# + 3k + 3 \end{aligned}$$

Note that the sets are disjoint, therefore it is possible to decode a clockwise string. In particular, if $x \in \text{enc}(\gamma)$, with a slight abuse of notation, we say that $\text{enc}^{-1}(x) = \gamma$. Our goal now is to build the game level \mathcal{G} , such that, if $\gamma_1 \gamma_2 \dots \gamma_t \in \mathcal{R}$ is accepting, then, the program $x_1 x_2 \dots x_t$ must solve the level \mathcal{G} , where $x_i \in \text{enc}(\gamma_i)$. Conversely, if a program solves \mathcal{G} , then it must be a concatenation of clockwise strings $x_1 x_2 \dots x_t$ such that they can be decoded into $\text{enc}^{-1}(x_i) = \gamma_i$, and $\gamma_1 \gamma_2 \dots \gamma_t \in \mathcal{R}$ is accepting.

► **Remark.** At this stage, the numbers used in the encoding might appear arbitrary and obscure. We will point out where these numbers are used as we move forward in the proof.

4.4 The Game Level

We build several independent sub-levels, each containing a single worker. In particular, we build $2k+4$ sub-levels: $\mathbf{S} = \{CW_1, CW_2, CW_3, \text{enforce}\# \} \cup \{M_i^{\text{even}}, M_i^{\text{odd}}\}_{i \in [k]}$. The final game level \mathcal{G} is created by stacking the sub-levels together and isolating them via holes. Formally, $\mathcal{G} = \text{stack}(\mathbf{S})$.

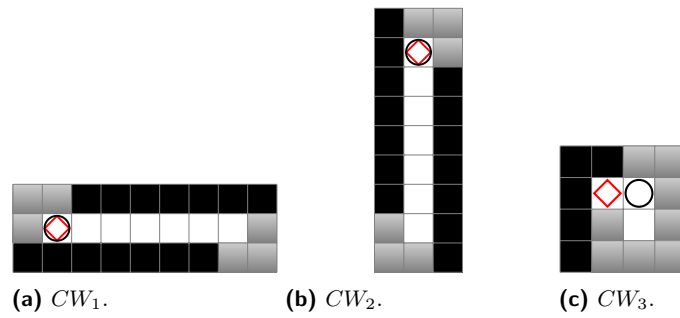
The three sub-levels CW_1, CW_2, CW_3 , reported in figure 5, are solved by all and only the programs that are concatenations of clockwise strings in \mathcal{C} (note that CW_1 and CW_2 also prevent diagonal movements).

4.4.1 The Enforce# Sub-Level

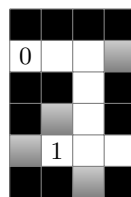
Let us first introduce the *counter gadget*, in figure 6. This gadget is such that, if the worker is standing on the cell labeled with 0, then any clockwise string in \mathcal{C} brings the worker to cell 1. Multiple counter gadgets can be concatenated together, and intuitively, these gadgets can be used to *skip* clockwise strings that we do not need to process.

⁵ named after the fact that **right, down, left, up** is a clockwise movement

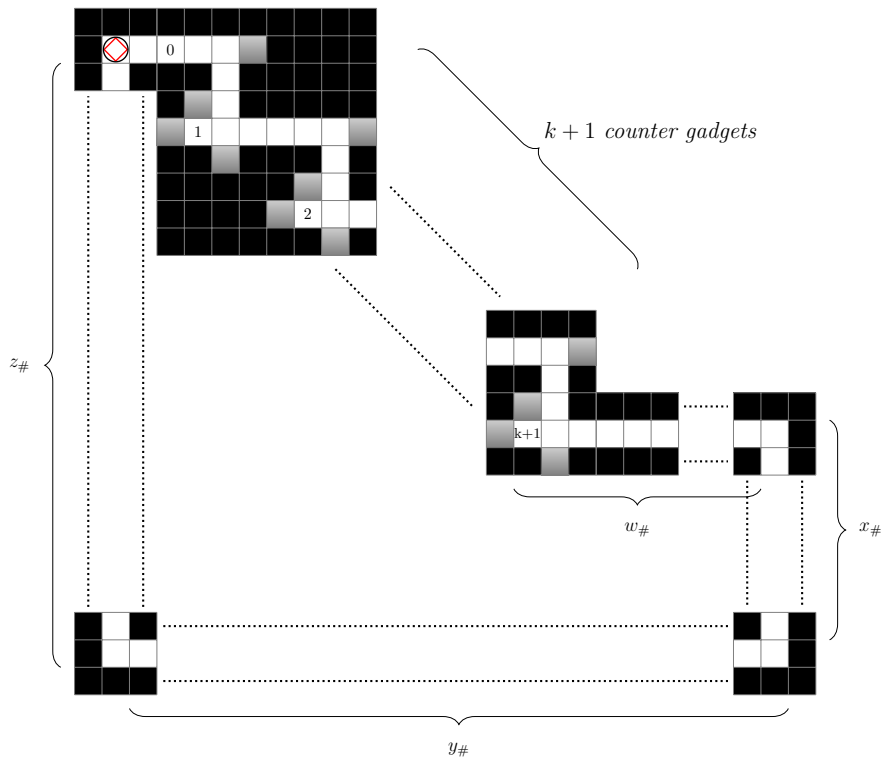
26:10 Coordinating “7 Billion Humans” Is Hard



■ **Figure 5** Sub-levels CW_1, CW_2, CW_3 , together they ensure that the program is a concatenation of clockwise strings in \mathcal{C} . Note that CW_3 is needed to ensure that the very first movement is right.



■ **Figure 6** Counter gadget. A worker starting from cell 0 that processes any clockwise string in \mathcal{C} ends up in cell 1.



■ **Figure 7** enforce $\#$ sub-level. The values $w_{\#}, x_{\#}, y_{\#}, z_{\#}$ are defined in Equation (2). This gadget ensures that accepting programs must be a concatenation of a multiple of $k + 2$ clockwise strings each ending with the encoding of $\#$.

The `enforce#` sub-level, in figure 7, skips the first $k + 1$ clockwise strings using $k + 1$ counter gadgets, then, it forces the next clockwise string to be $\mathbf{r}^{w\#}\mathbf{d}^{x\#}\mathbf{1}^{y\#}\mathbf{u}^{z\#}$, which is the only encoding of $\# \in \Gamma$, as defined in Equation (2). Indeed, if the $(k + 2)$ th clockwise string was not the encoding of $\#$, the worker would end up in a hole. After processing the whole encoding of $\#$, the worker will be in its starting position again (which is also the only accepting one), ready to possibly process more clockwise strings.

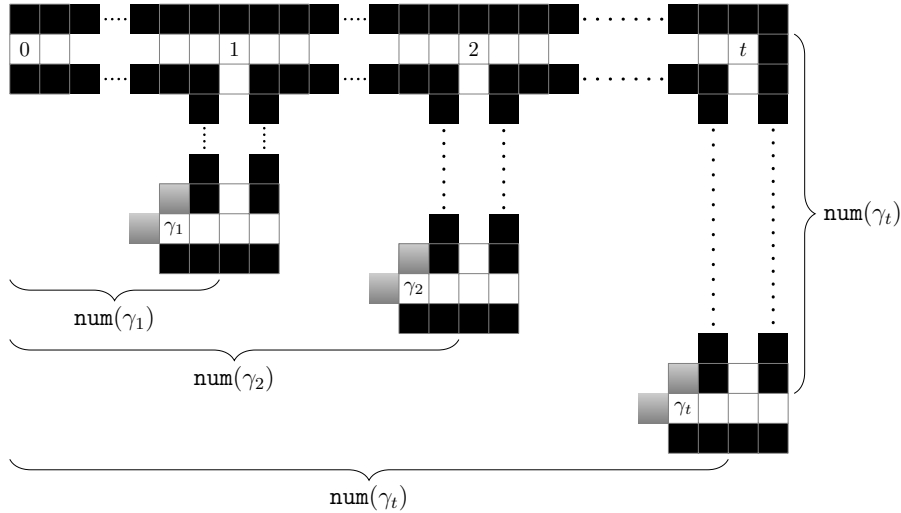
Therefore, to be more formal, the `enforce#` sub-level together with $\{CW_1, CW_2, CW_3\}$, ensures that if π is a solving program, then it must be of the form $\overline{R_1\#}\overline{R_2\#}\dots\overline{R_r\#}$, where $\overline{\#}$ is the only encoding of $\#$, and each $\overline{R_j}$ is a concatenation of $k + 1$ clockwise strings. Note also that all the programs of such form solve these four sub-levels.

► **Remark.** The `enforce#` sub-level can be built if it holds (i) $y_{\#} = 2 + 4(k + 1) + w_{\#} - 3 = w_{\#} + 4k + 3$, and (ii) $z_{\#} = 3(k + 1) + x_{\#}$. Note that both these relations are satisfied by our encoding, as reported in Equation (2).

4.4.2 The Automata Sub-Levels

For each DFA A_i , $i \in [k]$, we introduce two sub-levels: M_i^{odd} and M_i^{even} . Consider a program of the form $\overline{R_1\#}\dots\overline{R_r\#}$, where $\overline{\#} \in \mathbf{enc}(\#)$, and each $\overline{R_j}$ is a concatenation of $k + 1$ clockwise strings. Intuitively, M_i^{odd} will ensure, for each odd j , that $\overline{R_{j+1}}$ follows from $\overline{R_j}$ according to the computation of A_i . M_i^{even} will guarantee the same, but for even j 's. These sub-levels will also guarantee that the last state is an accepting one. Therefore, adding all the sub-levels $\{M_i^{odd}, M_i^{even}\}_{i \in [k]}$ will ensure that an accepting program must describe an accepting computation for all the DFAs.

Before showing the construction of M_i^{odd} and M_i^{even} , we introduce three new gadgets.

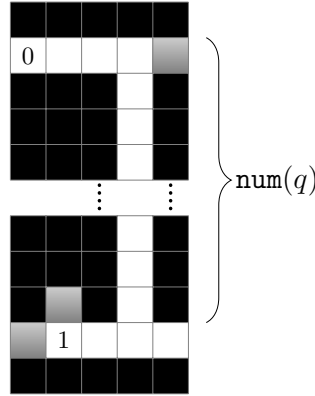


■ **Figure 8** The S -selector gadget for $S = \{\gamma_1, \gamma_2, \dots, \gamma_t\} \subseteq Q \cup \Sigma$. A clockwise string reaches the cell labeled with γ_ℓ if it moves **right** and **down** exactly $\mathbf{num}(\gamma_\ell)$ times.

The S -selector gadget, in figure 8, is parametrized by a set $S = \{\gamma_1, \gamma_2, \dots, \gamma_t\} \subseteq Q \cup \Sigma$ such that $\mathbf{num}(\gamma_j) < \mathbf{num}(\gamma_{j+1})$. If the worker is standing on the cell labeled with 0 of the selector gadget, then the next clockwise string must be any encoding of any element $\gamma_\ell \in S$, which will lead the worker to the cell labeled with γ_ℓ . Indeed, from our encoding in Equation (1), to verify that a clockwise string $\mathbf{r}^{x_1}\mathbf{d}^{x_2}\mathbf{1}^{x_3}\mathbf{u}^{x_4} \in \mathcal{C}$ is an encoding of a certain element

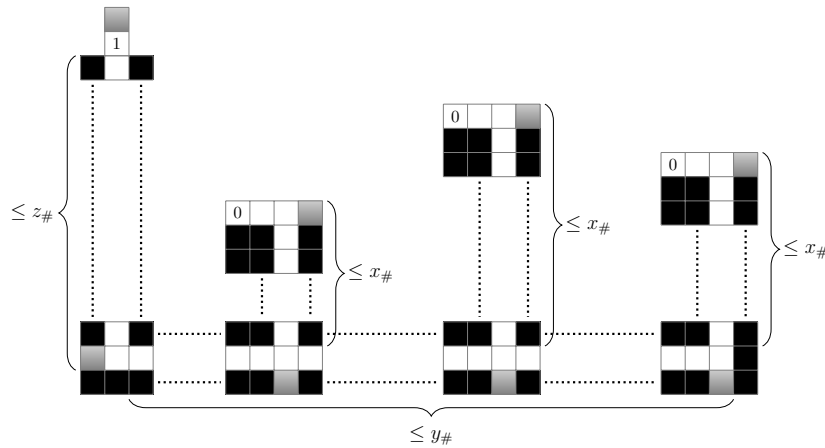
26:12 Coordinating “7 Billion Humans” Is Hard

$\gamma_\ell \in Q \cup \Sigma$, it suffices to check that $x_1 = x_2 = \text{num}(\gamma_\ell)$, which is what the gadget does. It is also easy to check that the worker will fall into a hole if the clockwise string is not an encoding of any element of S . This gadget will be useful for choosing the next input symbol and performing different checks depending on the current state of the automaton.



■ **Figure 9** The *q-forcer gadget* for $q \in Q$. Starting from cell 0, the worker reaches cell 1 with a clockwise string if and only if the number of **d** symbols is equal to $\text{num}(q)$.

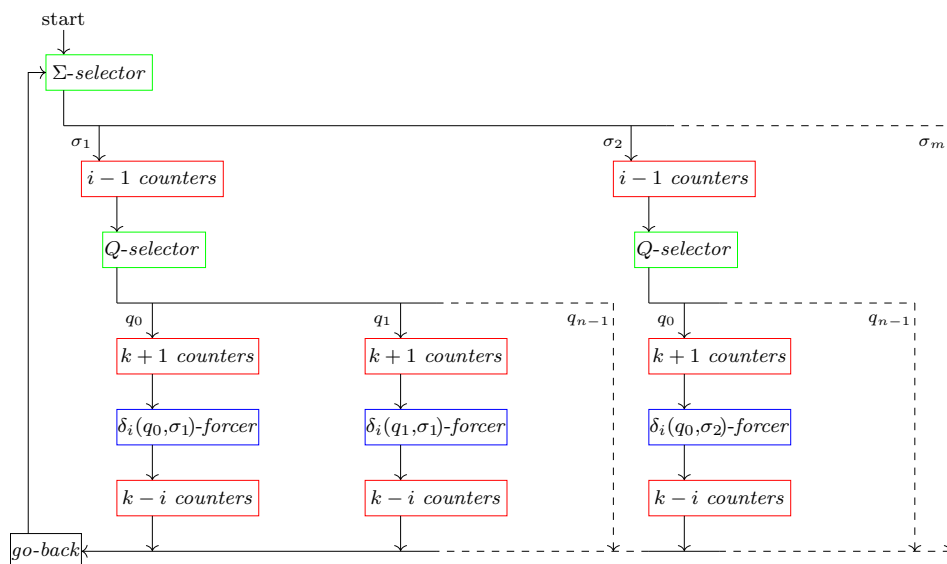
For a state $q \in Q$, the *q-forcer gadget*, in figure 9, forces the next clockwise string to have a number of **down** steps equal $\text{num}(q)$. Therefore, if we know that $c \in \mathcal{C}$ is an encoding of some state, then, by using the *q-forcer gadget* we impose the constraint that c must be an encoding of $q \in Q$. This gadget will be useful to force the string to respect the transition function.



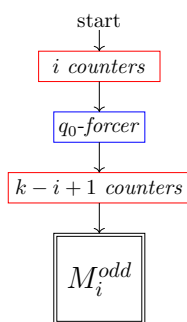
■ **Figure 10** The *go-back gadget*. Starting from any cell 0, the clockwise string $\mathbf{r}^{w_{\#}} \mathbf{d}^{x_{\#}} \mathbf{1}^{y_{\#}} \mathbf{u}^{z_{\#}}$, as described in Equation (2), brings the worker back to the cell 1. Note that more cells 0 can be added as needed, provided that the highlighted constraints are respected.

The last gadget we need is the *go-back gadget*, in figure 10. Suppose the worker is in one of the cells labeled with 0, then, by processing the only encoding of $\# \in \Gamma$, it will go back to cell 1. This gadget will be useful to “go back” to a selector gadget and analyze the next chunk of the program.

Note that these three gadgets and the counter gadget can be concatenated together. The only precaution to take is that when concatenating a selector gadget to a counter gadget, the cell 0 of the selector (figure 8) must coincide with the cell 1 of the counter (figure 6). Similarly, when concatenating a *go-back gadget* to a selector gadget, the cell 1 of the *go-back* must coincide with the cell 0 of the selector.



■ **Figure 11** M_i^{odd} sub-level associated to the DFA A_i , $i \in [k]$. The starting position of the single worker is at the beginning of the Σ -selector (i.e., the cell 0 of the Σ -selector, taking figure 8 as a reference). The first cell of the Σ -selector is also an accepting cell. Moreover, each branch of the Q -selectors corresponding to an accepting state $q_{acc} \in F_i$, contains an accepting cell after the first $k + 1 - i$ counters (i.e., the accepting cell is the cell labeled with 1, taking figure 6 as a reference, in the $(k + 1 - i)$ th of the $k + 1$ counters).



■ **Figure 12** M_i^{even} sub-level associated to the DFA A_i , $i \in [k]$. The starting position of the single worker is at the beginning of the first counter. The accepting cells are the same described for the M_i^{odd} sub-level.

Fixed $i \in [k]$, we report, in figure 11, the sub-level M_i^{odd} , and, in figure 12 the sub-level M_i^{even} . Both are represented in a schematic way. Both sub-levels use the *go-back gadget*, which requires processing exactly the encoding of $\#$: this is guaranteed by the *enforce $\#$* sub-level.

26:14 Coordinating “7 Billion Humans” Is Hard

We now argue that the sub-levels $\{CW_1, CW_2, CW_3, \text{enforce}\#, M_i^{odd}, M_i^{even}\}$ are solved by all and only the programs describing a valid accepting computation of DFA A_i .

We know, from $\{CW_1, CW_2, CW_3, \text{enforce}\#\}$, that a solving program π must of the form: $\overline{R_1\#\overline{R_2\#\dots\overline{R_r\#\overline{\#}}}}$, where $\overline{\#} \in \text{enc}(\#)$, and $\overline{R_j} = c_1^j c_2^j \dots c_{k+1}^j$ is a concatenation of $k+1$ clockwise strings. Let us first show that π can always be decoded:

- The Σ -selector of M_i^{odd} (resp. M_i^{even}) ensures that the first symbol of each $\overline{R_j}$, for odd j (resp. even j), is the encoding of a symbol in Σ . Therefore, for all $j \in [r]$, it must be $c_1^j \in \text{enc}(\sigma)$, for some $\sigma \in \Sigma$.
- Similarly, the Q -selectors of M_i^{odd} (resp. M_i^{even}) ensure that the $(i+1)$ th symbol of each $\overline{R_j}$, for odd j (resp. even j), is the encoding of a state. Therefore, for all $j \in [r]$, it must be $c_{i+1}^j \in \text{enc}(q)$, for some $q \in Q$.

Moreover, π describes a valid computation:

- the first forcer of M_i^{even} ensures that the computation starts from the starting state q_0 : $c_{i+1}^1 \in \text{enc}(q_0)$
- Suppose that the computation is valid up to $\overline{R_j}$, that is, the state $q = \text{enc}^{-1}(c_{i+1}^j)$ is correctly reached from q_0 . Let us assume that j is odd. Then, M_i^{odd} will follow the branch σ in the Σ -selector, for some $\sigma \in \Sigma$, and the branch q in the Q -selector. Then, the $\delta_i(q, \sigma)$ -forcer in M_i^{odd} forces c_{i+1}^{j+1} to be in $\text{enc}(\delta_i(q, \sigma))$, therefore, the state reached in $\overline{R_{j+1}}$ is correct too. If instead j is even, M_i^{even} ensures that the state reached in $\overline{R_{j+1}}$ is correct.

It is left to show that the computation described by π is accepting for A_i . Suppose r is odd. Then, at the end of the computation, the worker of M_i^{even} will be at the beginning of the Σ -selector (which is an accepting cell), and the worker of M_i^{odd} , which is “shifted forward” by $k+2$ clockwise strings, will be at the end of the $(k+1-i)$ th counter right after the Q -selector, but note that there is an accepting cell in such position only if the worker is in a branch corresponding to a state $q_{acc} \in F_i$, therefore $c_{i+1}^r \in \text{enc}(q_{acc})$ and the computation is accepting. If r is even, the situation is analogous: the worker of M_i^{odd} is at the beginning of the Σ -selector, but the one of M_i^{even} is at the end of the $(k+1-i)$ th counter after a Q -selector, and therefore it must be in an accepting branch.

Moreover, one can easily see that any encoding of an accepting computation solves all the sub-levels. Indeed, the Σ -selector allows the user to choose the input string, and then, since the computation is accepting, one of M_i^{even} and M_i^{odd} will end up in the $(k+1-i)$ th counter of an accepting branch and the other will stay at the beginning of the Σ -selector. Therefore, all the sub-levels would be solved.

► **Remark.** Our encoding allows the construction of M_i^{odd} and M_i^{even} . First, from our encoding, we have that for $j \in \{0, 1, \dots, n-2\}$, $\text{num}(q_{j+1}) - \text{num}(q_j) = 9(k+2)$, while the space actually needed between two branches of the Q -selector is: $4(k+1)+5+4(k-i)+4+6 < 8(k+1)+11 \leq 9(k+2)$ where the last 6 takes into account the overhead of the selector. Similarly, for $j \in [m-1]$, $\text{num}(\sigma_{j+1}) - \text{num}(\sigma_j) = 9(k+2)(n+2)$, while the space required between two branches of the Σ -selector is at most $4(i-1) + 9(k+2)(n+1) + 6 < 9(k+2)(n+2)$. To conclude, we need only to check that the constraints for the go-back gadget are satisfied. The width of the whole M_i^{odd} sub-level is at most $9(k+2)(n+2)(m+2) = w_{\#} < y_{\#}$. The height of the sub-level is instead at most $\text{num}(\sigma_m) + 4(i-1) + \text{num}(q_{n-1}) + 4(2k+1-i) + \text{num}(q_{n-1}) + 3 \leq 18(k+2)(n+2)(m+1) + 8k < 18(k+2)(n+2)(m+2) = x_{\#} < z_{\#}$. Therefore all the gadgets can be concatenated together.

4.5 Conclusion of the proof

Putting all the pieces together, we can prove the theorem.

Proof of Theorem 2. The problem is in PSPACE by Observation 4. Consider the following reduction from the Intersection Non-Emptiness Problem: an instance $I = \{A_1, A_2, \dots, A_k\}$ is associated with the level $\mathcal{G} = \text{stack}(\{CW_1, CW_2, CW_3, \text{enforce}\#\} \cup \{M_i^{\text{odd}}, M_i^{\text{even}}\}_{i \in [k]}).$

Suppose I can be solved, then, by Observation 6, there exists an accepting string $x_1 x_2 \dots x_t \in \mathcal{R}$. Consider any encoding $\pi = y_1 y_2 \dots y_t$ such that $y_j \in \text{enc}(x_j)$. Such program can be rewritten in the form $\overline{R_1\#} \dots \overline{R_r\#}$. Therefore, as we argued, π solves $\{CW_1, CW_2, CW_3, \text{enforce}\#\}$, and, given that I is solved, for each $i \in [k]$, π describes an accepting computation for A_i , then, it also solves M_i^{odd} and M_i^{even} . Therefore, \mathcal{G} is solved: all its workers will be standing on an accepting cell at the end of the program.

Suppose now that there exists a program $\pi \in \{1, r, u, d\}^*$ solving \mathcal{G} . As we argued, such a program can be decoded into a string $R_1\# \dots R_r\# \in \mathcal{R}$. Given that, for each $i \in [k]$, M_i^{odd} and M_i^{even} are solved, it means that the string represents an accepting computation for A_i . That is, letting $R_j = \sigma^{(j)} q^{(1,j)} \dots q^{(k,j)}$, it holds: (i) $q^{(i,1)} = q_0$, (ii) $q^{(i,j+1)} = \delta_i(q^{(i,j)}, \sigma^{(j)})$ for $j \in [r-1]$, and (iii) $q^{(i,r)} \in F_i$. Therefore, the string in \mathcal{R} is accepting. Using Observation 6, it follows that I is solvable.

Finally, observe that the number of cells in each sub-level is at most $O(y_{\#} \cdot z_{\#}) = O(k^2 n^2 m^2)$, and since there are $2k + 4$ sub-levels, \mathcal{G} has at most $O(k^3 n^2 m^2)$ cells, therefore the reduction can be carried out in polynomial time. ◀

5 Conclusions

We analyzed the computational complexity of the video game “7 Billion Humans”. The game involves controlling multiple workers simultaneously to direct them to some destination cells. When each cell is either empty or contains a wall, the problem of deciding if a level is solvable is NP-Hard, while adding holes makes the problem PSPACE-Complete. We also observed that the simple structure of our reductions entails hardness results for the problem of simultaneous maze solving and for the intersection non-emptiness problem.

While **7BH-Essential**, the problem where levels only contain walls and empty cells, is NP-Hard and clearly in PSPACE, it is not known whether it lies in NP. We leave this as an interesting open problem.

References

- 1 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is np-hard. *Theoretical Computer Science*, 2018. FUN with Algorithms.
- 2 Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Tracks from hell — when finding a proof may be easier than checking it. *Theoretical Computer Science*, 2020. FUN with Algorithms.
- 3 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 2015. FUN with Algorithms.
- 4 Emmanuel Arrighi, Henning Fernau, Stefan Hoffmann, Markus Holzer, Ismaël Jecker, Mateus de Oliveira Oliveira, and Petra Wolf. On the Complexity of Intersection Non-emptiness for Star-Free Language Classes. In *41st IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2021)*, 2021.
- 5 Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The computational complexity of portal and other 3d video games. *FUN with Algorithms*, 2018.


26:16 Coordinating “7 Billion Humans” Is Hard

- 6 Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super mario bros. is harder/easier than we thought. *FUN with Algorithms*, 2016.
- 7 Stefan Funke, André Nusser, and Sabine Storandt. The simultaneous maze solving problem. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
- 8 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.
- 9 Luciano Gualà, Stefano Leucci, and Emanuele Natale. Bejeweled, candy crush and other match-three games are (np-)hard. *2014 IEEE Conference on Computational Intelligence and Games*, 2014.
- 10 Graham Kendall, Andrew Parkes, and Kristian Spoerer. A survey of np-complete puzzles. *ICGA Journal*, 2008.
- 11 Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, 1977.
- 12 Klaus-Jörn Lange and Peter Rossmanith. The emptiness problem for intersections of regular languages. In *Mathematical Foundations of Computer Science*, 1992.
- 13 Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2), 1970.
- 14 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! In *Fun with Algorithms*, 2012.
- 15 Michael Wehar. Hardness results for intersection non-emptiness. In *Automata, Languages, and Programming (ICALP)*, 2014.
- 16 Commonsense review. <https://www.commonsense.org/education/reviews/7-billion-humans>. Accessed: 2024-02-26.
- 17 Hackr review. <https://hackr.io/blog/coding-games>. Accessed: 2024-02-26.
- 18 Hubspot review. <https://blog.hubspot.com/website/best-coding-games>. Accessed: 2024-02-26.

Arimaa Is PSPACE-Hard

Benjamin G. Rin¹  

Utrecht University, The Netherlands

Atze Schipper 

Utrecht University, The Netherlands

Abstract

Arimaa is a strategy board game developed in 2003 by Omar Syed, designed to be hard for AI to win because of its large branching factor. In this paper, its theoretical complexity is considered. We prove that Arimaa (suitably generalized to an $n \times n$ board) is PSPACE-hard. This result is found by reducing a known PSPACE-hard variant of Generalized Geography to a variant of Arimaa that we call Arimaa', which in turn is then reduced to $(n \times n)$ Arimaa. Since the game is easily seen to be solvable in exponential time, it follows that its complexity lies somewhere between being PSPACE-complete and EXPTIME-complete.

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

Keywords and phrases Arimaa, complexity theory, PSPACE-hardness, board games, Generalized Geography

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.27

Acknowledgements We would like to thank Rosalie Iemhoff for her useful feedback and suggestions, as well as Colin Caret, Dominik Klein, Johannes Korbmacher, Jaap van Oosten, and all other attendees at The Utrecht Logic in Progress Series for their comments on a presentation of an early version of this paper. We also would like to thank the anonymous referees for their feedback.

1 Introduction

Arimaa is a two-player strategy game played on an 8×8 board. Like chess, checkers, and Go, it is a perfect-information, deterministic, turn-based game with at most one winner. Arimaa has been a focus of much attention in applied AI research, as it was originally designed purposely to be difficult for computers to play [18]. The present paper considers the more theoretical question, “Given a position in Arimaa and specified player, does that player have a winning strategy?” We call this the *Arimaa decision problem*. As with chess and checkers, this question is trivially solvable in constant time if the game is played on a standard 8×8 board (albeit for an impractically large constant). Accordingly, we focus instead on the question for the generalized case of an $n \times n$ board (cf. [8], [11], [12], [13]). For the aforementioned strategy games, the corresponding question is known to be PSPACE-hard.² The present paper shows that the same is true for Arimaa. We prove this result by reduction from the known PSPACE-hard problem Generalized Geography, adapting a technique from [16] originally used to show the PSPACE-hardness of $n \times n$ chess.³ We do not prove here that

¹ Corresponding author. Authors listed alphabetically.

² Results showing PSPACE-hardness of strategy and puzzle games continue to be produced to this day, with some recent examples including [1], [3], and [4]. Depending on the details of how the game is generalized – e.g., concerning rules such as the 50-move rule in chess – the question may in fact be shown to be EXPTIME-hard (see, e.g., [8], [12], and [13]).

³ Our proof has its origins in the bachelor thesis of the second author, written under supervision of the first [15].



© Benjamin G. Rin and Atze Schipper;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 27; pp. 27:1–27:24

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$n \times n$ Arimaa is in PSPACE. However, it is not hard to see that it falls in EXPTIME.⁴ In fact, we suspect it is EXPTIME-complete (see Conjecture 1), which we leave for future research.

1.1 History

In the last twenty-five years, AI has been increasingly defeating humans in strategy games such as chess and Go. This emerging trend inspired computer engineer Omar Syed to design Arimaa as a game intentionally hard for computers while still relatively easy to understand and fun to play for humans. It is played on a chess board with chess pieces, but the mean branching factor is around 17,000 moves per turn [9] (compared to about 35 for chess).

In 2003, the game was ready and published in The International Computer Games Association (ICGA) Journal [18], together with a prize of \$10,000 USD for the first person or organization to create a computer program that could beat the best human players before the year 2020.⁵ The challenge sparked much research into the development of Arimaa bots and the practical complexity of Arimaa from people like Brian Haskin [9], Christ-Jan Cox [5] and experienced Go programmer David Fotland [6]. Fotland had won the Arimaa computer tournament five times, but could never manage to win the contest against the best human players. Many approaches were tried by different people, resulting in numerous academic papers, theses and technical reports before the challenge was finally completed (e.g., [7], [17], [19]). In 2015, David Wu with his bot *SHARP* [22] did the unthinkable and defeated the top human players. The March 2015 ICGA Journal issue [20] was themed around the Arimaa challenge being won.

All this research is on the practical complexity of making a machine play Arimaa well. But its theoretical complexity is relatively unexplored. As the editorial board of ICGA wrote in the March 2015 issue:

The last scientific challenge, establishing the game-theoretical value of Arimaa, is still waiting for its solution. Arimaa is now in the class of games in which computers outperform the best human players, just like chess. Yet, the ultimate question is: can we solve the game? As readers of this Journal know we distinguish between weakly solving and strongly solving a game (introduced by Allis, 1994). So, we would like to encourage all researchers to continue the development of advanced techniques to find the ultimate truth of Arimaa. [20]

1.2 Main theorem

The present paper proves the following result.

► **Theorem 1.** *The Arimaa decision problem is PSPACE-hard.*

⁴ In [8] it is stated that $n \times n$ chess is decidable in exponential time by brute force. A similar case can be made for $n \times n$ Arimaa. There is an upper bound 13^{n^2} on the number of possible board states, since each of the n^2 squares must be empty or occupied by one of the six pieces of either player. Since each position is caused by one of two players, and since a position can occur no more than three total times in a game (by the threefold repetition rule), the game tree has not more than $6 \cdot 13^{n^2}$ nodes, which is exponential in the number n^2 of squares. Optimal moves can then be found by, for example, alpha-beta pruning.

⁵ The prize was later raised to \$12,000 USD [2].

We will reduce a known PSPACE-hard variant of Generalized Geography, which we call $\text{AGG}_{1,1}$, to a variant of Arimaa we call Arimaa' , which is then reduced to $n \times n$ Arimaa⁶. These reductions are both computable in polynomial time, so it will follow that Arimaa is PSPACE-hard.

The proof in this paper follows in the footsteps of Storer in [16] for $n \times n$ chess. Unlike chess, however, Arimaa has no pieces that can move long distances in a single turn. Accordingly, one of the main challenges of our proof, and arguably one of its main contributions, is in showing how to apply Storer’s proof method for games with only short-range pieces like Arimaa.

Since it is straightforward to see that the Arimaa decision problem is in EXPTIME for reasons similar to those for chess (see fn. 4), it follows that the complexity of Arimaa lies somewhere between PSPACE- and EXPTIME-completeness. In the future, we hope to settle the following conjecture.

► **Conjecture 1.** *The Arimaa decision problem is EXPTIME-complete.*

2 Background

2.1 Game rules

Arimaa is played on a chess board using chess pieces, but otherwise has little to do with chess. The pieces are renamed as follows to make it easier to remember their hierarchy (listed in order of decreasing strength): *elephant* for king, *camel* for queen, *horse* for rook, *dog* for bishop, *cat* for knight, and *rabbit* for pawn. The 8×8 board remains the same except for the squares c3, c6, f3 and f6. These squares are *trap* squares, where pieces can be captured.

2.1.1 Setup

► **Definition 2.** *The home ranks of a player are the first two ranks on the board from that player’s perspective.*

► **Definition 3.** *The goal rank of a player is the first home rank of his opponent.*

The board starts empty, with Gold being the first player to choose the setup in which he⁷ would like to start the game. After he has positioned his pieces (this is visible to Silver), Silver may choose her starting setup. Both players may place their 16 pieces in any order they want on their home ranks. After the setup, Gold will be the first player to make a move.

2.1.2 Playing

► **Definition 4.** *A position describes all piece locations and which player is next to move. Switching pieces of the same team and strength does not result in a different position.*

Figure 1 shows a sample Arimaa position. The game is shown from Gold’s point of view. Traps are represented by \times symbols. For readability, we use numbers to denote the different pieces in figures throughout this paper. The numbers correspond to the pieces’ places in the hierarchy – e.g., an elephant is denoted by the number 6, a camel by 5, etc. Hereafter, let us call the 1s, 2s, and 3s the *weak* pieces and the 4s, 5s, and 6s the *strong* pieces.

⁶ Hereafter, we may refer to $n \times n$ Arimaa simply as Arimaa, if confusion is unlikely.

⁷ For referential convenience, in this paper we (alphabetically) use “he” for Gold and “she” for Silver.

8								
7	1	1				3	1	
6	3	3	×	4	1	×		
5				6				
4			1					
3		3	×	6	1	×	1	
2			4	1		4		
1			2	2	1	1	1	1
	a	b	c	d	e	f	g	h

■ **Figure 1** An example Arimaa position from [21].

All pieces move the same way, one step at a time and only in cardinal directions. The only exception to this rule is the rabbit: like the pawn in chess, the rabbit cannot move itself backward. The only way a rabbit can move back is by being pushed or pulled by an enemy piece (see below). The biggest difference from other board games such as chess, checkers, and Go is that players are allowed to distribute up to four moves per turn over their pieces. For example, one may let four pieces move one step, one piece move four steps, or anything in between, as long as the turn ends with a net change to the position. Pieces can only step onto an adjacent square if it is not occupied by another piece, unless the piece making the move is pushing the other piece away.

Pieces may *push* or *pull* a single enemy piece that is strictly lower in the strength hierarchy. This costs two steps, as two pieces are moving one square, and the piece performing the push or the pull must be directly adjacent to the enemy piece. Any square in one of the cardinal directions of the piece being pushed can be chosen to push that piece onto as long as it is able to move, and the pushing piece will move on to the square previously occupied by the pushed piece. Observe the position depicted in Figure 1. The gold dog on a6 could, for example, push the silver rabbit on a7 to a8, moving to a7 itself. Pulling is similar: the piece performing the pull can move to any square it is allowed to move to, and the pulled piece will move to the square previously occupied by the pulling piece. For example, the gold elephant on d3 could pull the silver rabbit on d2 to d3, and could then pull it to d4, ending the turn on e4 itself.

Another game mechanic is *freezing*. Any piece that is next to an enemy piece that is higher in hierarchy is *frozen* and therefore not allowed to move (thus also not allowed to push or pull). It follows automatically that the elephant, the highest ranked piece, cannot be frozen. A piece that is currently not frozen is allowed to move to a square next to a bigger enemy piece, resulting in freezing itself. As an exception, a piece can protect itself from being frozen by being next to a friendly piece. In that case, neither piece can be frozen. Likewise, a piece can unfreeze a currently frozen friendly piece by moving to a square next to it. In the example position in Figure 1, the pieces on a7, b7 and d6 are currently frozen, while the piece on c1 is not.

Another way to immobilize a piece is to *blockade* it. A piece is blockaded when it is surrounded by pieces it cannot push. Examples include the c3 rabbit in Figure 1 and the elephants in Figure 5. So, while an elephant can never be frozen, it can be blockaded if surrounded by two pieces in each direction.

The trap squares are the only way for pieces to be removed from the board (captured). Trap squares can be *controlled* by having at least one friendly piece next to it, meaning your pieces are safe to stand in the trap square without being captured. However, as soon as all

four squares next to the trap are either unoccupied or are occupied by enemy pieces, any friendly piece standing in the trap will be removed from the game. Shared trap control is thus also possible, when both friendly and enemy pieces are standing around a trap, resulting in the trap square being safe to occupy for both players.

All consequences of moving pieces are effective immediately, not at the end of the turn. For example: if a piece is frozen after making a certain move, the player is not allowed to make any more moves with that piece, even if not all four moves in the current turn are used yet. However, pushing and pulling are moves considered to happen simultaneously. This means that a player can, for instance, move a friendly piece onto an uncontrolled trap square while pulling a weaker enemy piece. Although this results in the friendly piece getting captured, the pull still gets to be completed. See, for example, the gold dog on b3: the moment it moves south next to the silver horse on c2 to pull the c3 silver rabbit west, it is frozen, but the pull gets completed anyway.

The final movement rule is the *threefold repetition rule*: a player may not end a turn having created the same position as one that he or she has previously created twice.

2.1.3 Winning conditions

There are three ways for a player to win the game:

1. Ending a turn with at least one of the player's rabbits on the goal rank.
2. Making the other player lose all his or her rabbits. If both players lose their last rabbit in the same turn, the player who took the turn wins.
3. Giving the other player no legal moves to play (say, by freezing and/or blockading all his or her pieces).

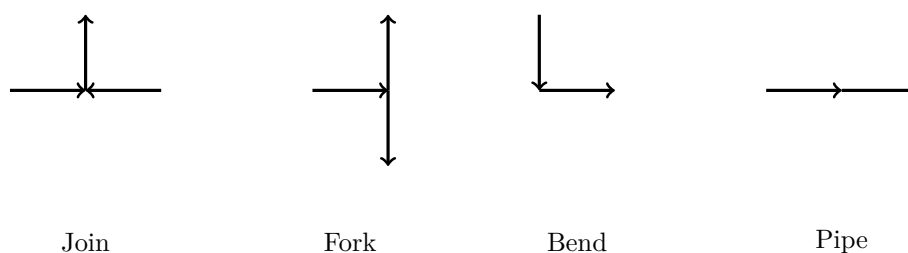
2.2 Generalization to $n \times n$ Arimaa

As already discussed, 8×8 is trivially computable in constant time, so we generalize to an $n \times n$ board. However, stretching the board to variable dimensions raises an important decision on how to place the trap squares. Standard Arimaa has four trap squares on an 8×8 board, placed on squares c3, c6, f3 and f6. We have opted to generalize this by concatenating copies of the standard board (with standardly placed trap squares) in all directions, resulting in an arbitrarily large board with many groups of four trap squares (see Figure 5 for an example). More on trap placement can be found in Section 4, where we discuss other ways of distributing trap squares over a large board.

2.3 Generalized Geography variations

Our main theorem is proved by reduction from a known PSPACE-complete variant of Generalized Geography. We present the definition of this variant in the present section.

Geography is a game in which two players take turns in naming cities. The first letter of the next city must be the same as the last letter of the previous city, and the next city must not have been named before. The first player who cannot think of a new city loses. Generalized Geography (GG) is a generalized version of the game. GG is played on a directed graph in which the two players (usually called White and Black, but we may say Gold and Silver) each take turns placing markers on nodes. The game starts with Gold placing a marker on the start node, after which it is Silver's turn. The players then take turns placing markers on nodes with an incoming edge from the last node chosen. This continues until a player places a marker on a node already marked, after which the game ends and that player loses.



■ **Figure 2** The four node shapes.

The problem of determining which player has a winning strategy in GG is proven to be PSPACE-complete by Schaefer [14]. In [10], some restrictions to the graph are added while preserving the complexity of the game. Storer uses this restricted definition of GG in his proof on the complexity of chess [16], where he calls it Restricted GG (RGG). Based on RGG, Storer defines another variant called Array GG (AGG), wherein further restrictions are added and the winning condition is reversed: a player who marks a previously marked node now *wins* the game. In detail:

► **Definition 5.** *The AGG problem is defined as: “Given a directed graph G and start node s , with the features described below, does Gold have a winning strategy (see rules above)?”*

The nodes in G have total degree 3 or have indegree 1 and outdegree 1. The start node s is the only exception, with indegree 0 and outdegree 1.

The nodes of G form a finite subset of $\mathbb{Z} \times \mathbb{Z}$ in the Cartesian plane. Edges must be either vertical or horizontal, and may only connect pairs of nodes that have no other nodes between them (see Figure 3).

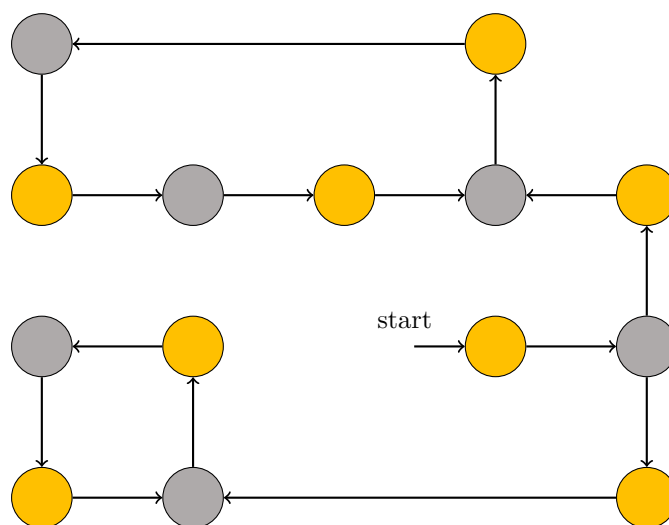
Other restrictions on the graph are:

1. *The graph is planar (no crossing edges when the graph is embedded on a plane).*
2. *The graph is bipartite (no odd-length cycles).*
3. *The graph is connected.*
4. *There are no self-loops.*

Storer proves that AGG is PSPACE-complete in [16], by showing how to transform an RGG problem (given an instance (G, s) of RGG) into an AGG problem. This is done by scaling up the graph when needed by adding new nodes (preserving the original parity of the game by adding them in pairs) and changing the length of edges. The transformation is done in such a way that we only end up with nodes of the forms depicted in Figure 2, and straight edges between them.

In Corollary 1.1 from Storer’s paper it is shown that AGG remains PSPACE-complete even if we add the further restriction that every non-start node must be one of the types depicted in Figure 2, and the nodes of each type other than the bend are also in the orientations depicted. Let us call this restricted problem $\text{AGG}_{1,1}$. This means that we need only eleven different types of nodes, other than the starting node, to build any $\text{AGG}_{1,1}$ graph: the join, fork and pipe in one orientation each, and bends in all eight orientations (only north-to-east is depicted here).⁸ As in AGG, edges may be in any cardinal direction. Note that there can be a *long* edge connecting two nodes that are distant in the Cartesian plane, if no node lies between them.

⁸ Storer shows how a grid graph with nodes of other types can be simulated by one with only these types.



■ **Figure 3** An $AGG_{1,1}$ example graph. Node color indicates which player can mark that node.

► **Remark 6.** Because the graph is bipartite, and because players alternate turns choosing nodes adjacent to previously chosen nodes, the nodes can be partitioned into two disjoint subsets – one for each player. Since it is predefined which player will mark the start node, each node is already determined at the beginning of the game to be potentially markable by precisely one particularly player. And given any node, its incoming *edge(s)* are also deterministically assigned as potentially usable for exactly one player before the game starts. See Figure 3 for an example of an $AGG_{1,1}$ graph.

2.4 Arimaa'

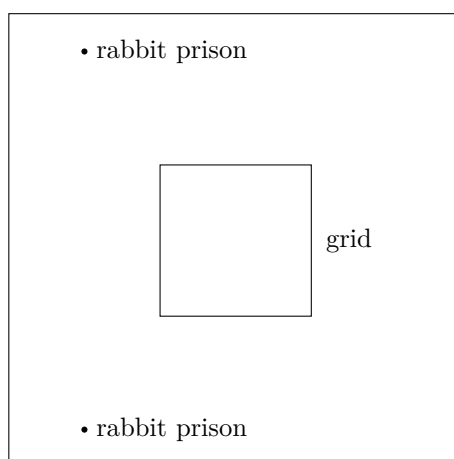
To simplify the proof of the main theorem, we show that a variant of Arimaa called Arimaa' reduces to $n \times n$ Arimaa, and we then show that this variant is PSPACE-hard. We define Arimaa' in the present section.

► **Definition 7.** A board state is an arrangement of pieces on the board, without consideration of which player is moving next.

► **Definition 8.** Arimaa' is the problem of determining whether Gold has a winning strategy given an $n \times n$ Arimaa board state P' under the following conditions:

1. There is a new threefold repetition rule: players may now legally cause a position to occur for a third time, but a player who does this immediately loses.⁹
2. The rule making players lose when starting a turn with no movable pieces is removed.
3. The rule preventing players from passing their turn without making a net change to the position is removed. Passing a turn in any position still counts as causing that position, with respect to the threefold repetition rule.

⁹ This change is strictly to make it easier to discuss repetition, so that we may sometimes say a player repeats the position and loses, rather than say the player cannot repeat the position.



■ **Figure 4** Overview of an Arimaa' position. In the area labeled “grid” is a simulated $AGG_{1.1}$ graph.

4. In the board's center is a simulated $AGG_{1.1}$ graph with its start node already marked, the details of which will be specified in the proof of Theorem 1. (We will see that Gold will have a winning strategy in P' iff the Gold player in the simulated $AGG_{1.1}$ game also has a winning strategy.) Outside this area, the board is empty, save for two gadgets called rabbit prisons defined later. See Figure 4.
5. The most recent turn was Gold's and was a pass. Before that, Silver moved a piece.

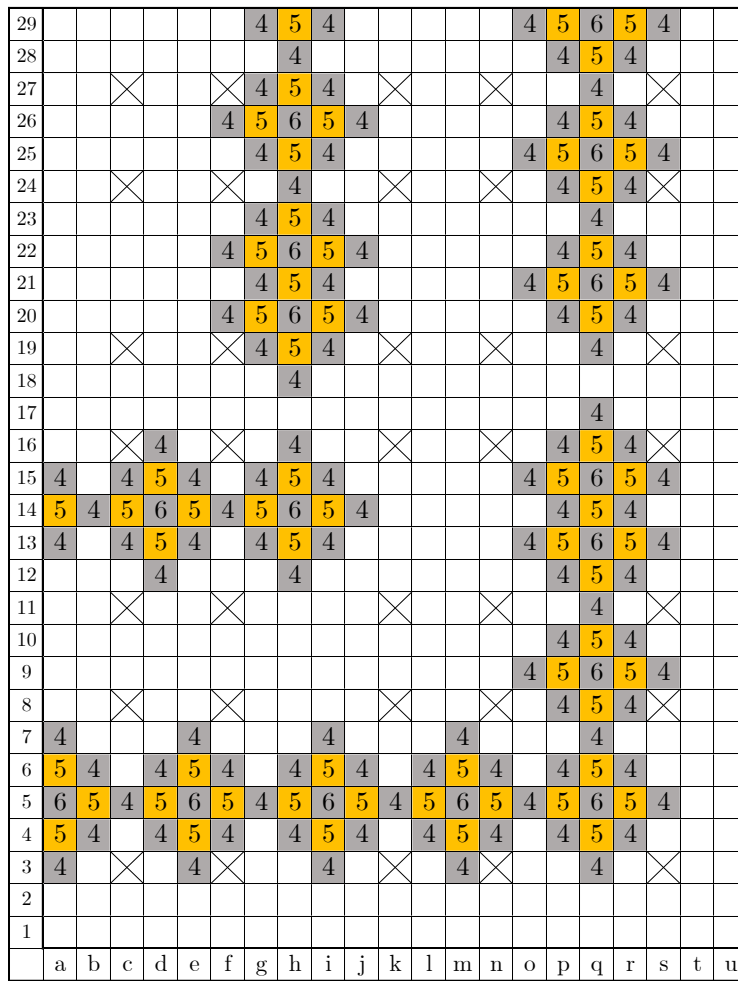
► **Remark 9.** By condition 8.3, if a player (say, Gold) moves a piece and causes a position X , after which the opponent (Silver) passes the turn, then Gold cannot profitably pass back. If he tries, then Silver can pass again, after which Gold cannot pass back again without instantly losing from having caused position X to occur for the third time. Therefore, a player's decision to pass the turn cannot be trivially undone by the opponent. Indeed, once a player moves a piece, the opponent can repeatedly pass and force the player to make another piece move, for as many turns as desired.

2.4.1 Big picture

► **Definition 10.** The simulation grid, or simply grid, is the simulated $AGG_{1.1}$ graph located in the middle of the Arimaa' board. The simulation is the ongoing process of players making moves that simulate placing markers on nodes of the $AGG_{1.1}$ graph.

In Figure 4 we see the grid in the center and the rabbit prisons (see Section 2.4.3) near their respective goal ranks. The Gold player in Arimaa' will simulate the actions of Gold in the $AGG_{1.1}$ game. As detailed later, the Arimaa' players simulate the marking of a node by moving a cat into the corresponding gadget on the board, after which it becomes captured or otherwise made irrelevant and simultaneously frees a previously frozen enemy cat. An $AGG_{1.1}$ node marked by Gold (Silver) will therefore correspond to an Arimaa' gadget with a gold (silver) cat entering it and a silver (gold) cat leaving. The latter cat will then travel to an adjacent node gadget. In case of the start node, a silver cat is leaving, so we can think of it as being already marked by Gold.

Surrounding the grid are *walls*, which we describe in the next section. Outside these walls are two rabbit prisons, whose construction and purpose we describe in the section after.



■ **Figure 5** A part of a silver corridor containing a 90° turn.

2.4.2 Walls

► **Definition 11.** A free piece is a piece that is allowed to move, because it is not frozen or blocked by other pieces.

► **Definition 12.** A stable group of pieces is one containing no free piece.

A wall, as the name suggests, is a special kind of continuous line of stable pieces that is designed to contain and constrain the motion of nearby free pieces. Specifically, walls constrain only *weak* pieces (recall that this means pieces of strength 3 or less). A wall is designed for just one color; its main function is to stop weak pieces of the appropriate color that are on one side of the wall from crossing over to the other side. As we explain below, no appropriately colored weak piece can get through them or break them down. Although a wall is designed for only one color, pieces of both colors are needed to construct it. The color of weak pieces that the wall constrains is always the opposite of the wall’s outermost pieces (the 4s – see Figure 5). We say a wall is *gold* (*silver*) if its outer 4s are gold (silver). For example, the walls in Figure 5 are silver walls.

The main building block of a wall is a 6 with four enemy 5s and eight friendly 4s around it. The 6 is unable to push or pull because of the double layer of pieces around it, and the 5s and 4s are frozen, so the group of pieces as a whole is stable. These building blocks can be laid out in any desired direction to build walls of various shapes.

The parallel walls in Figure 5 jointly form a *corridor*, inside which a free piece may move. The inside of the corridor shown spans from row 8 to 11 and from column k to n . The 90° turn is included for demonstrative purposes; corridors can extend in a single direction for an arbitrary length. Straight corridors will correspond to edges in our reduction from $AGG_{1,1}$.

Observe how a silver (gold) wall constrains the movement of weak gold (silver) pieces, since the weak piece gets frozen the moment it touches any outer 4. In the case of Figure 5, the walls depicted have gaps on squares h17 and q18, but since each gap is only one square wide and has 4s next to it, a weak gold piece cannot pass through it without getting frozen. However, any strong gold piece can move through these gaps. If its strength is 5 or greater, that piece can even destabilize the wall by pushing/pulling 4s. Furthermore, *any* free *silver* piece can also move through the gaps and can destabilize the wall by unfreezing a 4.

It will normally hold that only weak gold (silver) pieces get free near silver (gold) walls.¹⁰ Since we always know which player may use a given edge in $AGG_{1,1}$ (see Remark 6), we can carefully construct corridors of correct colors accordingly. All corridors will have gaps similar to the two in Figure 5, but they are impenetrable by the weak free pieces.

In addition to color, walls can also be categorized by their *parity*. We define the parity of a wall based on the color of pieces that lie on the diagonal of the upper-left/bottom-right traps (in the groups of four traps). If the color of the pieces on that diagonal is silver, we say the wall has *standard* parity (regardless of the wall's color). Otherwise, the parity is *nonstandard*. In Figure 5, the wall depicted has standard parity.

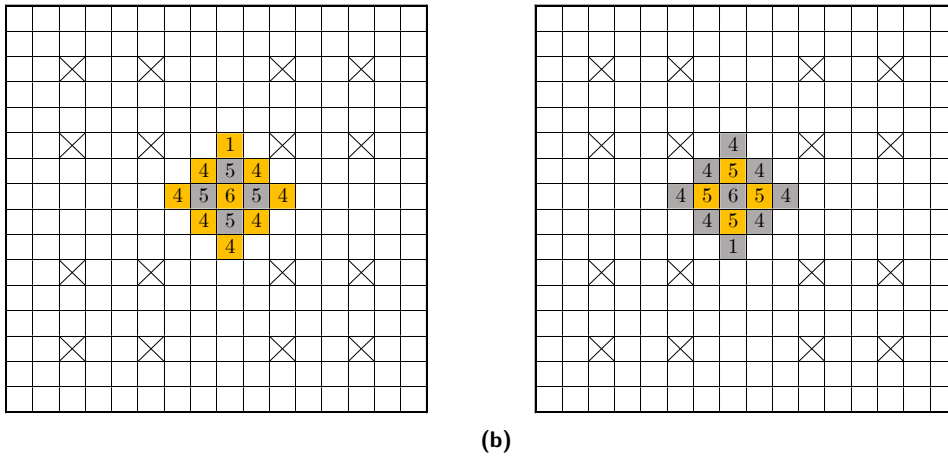
Gold walls are made in almost the same way as silver walls, but aside from reversing the piece colors, we also shift the wall one square to maintain standard parity. This is because reversing the piece coloring changes the wall's parity, so the shift is done to undo that change. We desire all walls to be built with the same parity so they can connect properly. More on connecting corridors can be found in Section 3.2.

2.4.3 Rabbit prisons

The rabbit prisons are similar to wall building blocks, but with a 1 in place of a 4. See Figure 6. Each player's rabbit prison is located near that player's goal rank. The prisons' sole purpose is to ensure that players comply with the $AGG_{1,1}$ simulation in the Arimaa' game, by giving the opponent of a non-complying player a way to punish them.

Specifically, a player who makes a move in the Arimaa' game that breaks the simulation will let the other player get a loose 4 or 6. (Details are explained in Section 3.3.2.) Such pieces are strong enough to pass through gaps in the walls and walk outside the grid. When this happens, the loose 4 or 6 can reach the appropriate rabbit prison and unfreeze the rabbit there, letting it step onto the goal rank and win the game.

¹⁰The only exception occurs when a player fails to uphold the simulation of the $AGG_{1,1}$ game within the Arimaa' game. In Section 3.3.2, we see how any player who does this allows the opponent to have a free piece near a same-colored wall, leading to a forced win for that opponent. This dynamic serves to ensure the simulation is upheld until the end.



■ **Figure 6** A gold and a silver rabbit prison.

2.4.4 Viable winning conditions

As detailed in the game rules (Section 2.1.3), there are multiple ways to win at Arimaa. However, some winning conditions have been altered in Arimaa'. In the setup presented here, in which Arimaa' is used to simulate an AGG_{1.1} game, the normal method of victory has a player move a piece into a node gadget for the second time (always a join, the only node type with in-degree 2 – see Figure 2), which leads, as we will later see, to an eventual threefold repetition for the opponent. As discussed above, there is also an alternative winning method possible against an opponent who does not comply with the AGG_{1.1} simulation.

The second Arimaa winning condition – capturing all opposing rabbits – can be disregarded in Arimaa'. This is because, apart from all the rabbits in the simulation, the one in the rabbit prison must also be captured. In order to do this, a player must get a free piece outside the grid, and as soon as a player achieves that, it is equally possible to win by freeing the player's own imprisoned rabbit and walking it to the goal rank.

The third Arimaa winning condition has been simply removed from Arimaa', as noted in Definition 8.2. But note that in Arimaa', if a player moves a piece and then *ends* (rather than begins) their turn with no movable pieces, then their opponent can force a win by passing twice and causing them to lose the game by the new repetition rule.

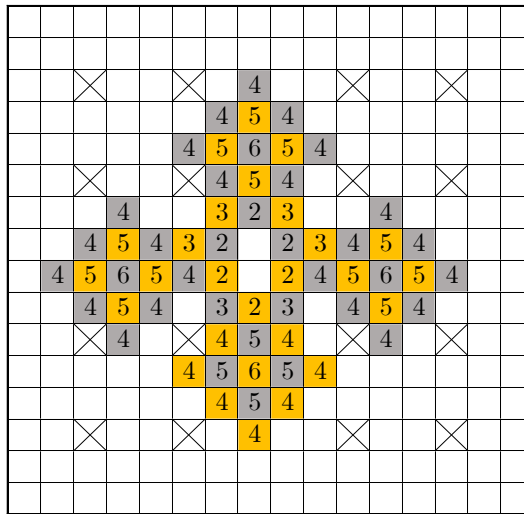
2.4.5 Reducing Arimaa' to Arimaa

We now show that reducing AGG_{1.1} to Arimaa' suffices to prove our main theorem.

► **Lemma 2.** *$n \times n$ Arimaa is PSPACE-hard if Arimaa' is PSPACE-hard.*

Proof. For any Arimaa' position with board state P' , we show how to convert it to an $n \times n$ Arimaa board state P such that Gold wins in P if and only if Gold wins in P' (with the same player to move). This works in polynomial time. We begin by introducing the concept of a *cat cage*, seen in Figure 7. The cat cage depicted is empty, but in the actual construction a (silver or gold) cat is placed on one of the two empty squares in the middle. A cat cage is *gold* (respectively, *silver*) if a gold (respectively, silver) cat is put into one of the two central empty squares.

In P , the board state is identical to that of P' except that a gold and a silver cat cage are placed outside the grid. By the cages' design, a cat in one of the two middle squares of a cage is only able to move back and forth between those squares, and all other pieces in the



■ **Figure 7** An empty cat cage.

cage remain immobile no matter the position of the cat. Further, note that there will never be interaction between the pieces of the cat cages and those of the grid, because any player who would somehow get a free piece outside the grid would just move it to the rabbit prison to free the rabbit and win. These considerations lead us to the following definition.

► **Definition 13.** A grid state is an arrangement of pieces on the grid, without consideration of which player is moving next or the caged cats' positions. A cat cage move is a turn in which the player simply moves the caged cat to the other accessible square and ends the turn. Thus, a cat cage move leaves the grid state intact and changes only a caged cat's location.

We now argue that the rule differences between $n \times n$ Arimaa and Arimaa' described in conditions 8.2-8.3 do not change the outcome of the game. By assumption, the Arimaa players can play the same grid moves as the Arimaa' players. Passing moves are not legal, but these can be simulated by cat cage moves. The cat cages also ensure that players will not lose the game by beginning a turn with no free pieces. Still, we must show that our remark about Arimaa' players losing if they end a turn with no free pieces applies to the Arimaa game.

During the course of play, suppose the Arimaa board is in an arbitrary board state and a player (without loss of generality, Gold) has just ended a turn with no free non-cage pieces. We will show that Gold loses in this situation.

Assume without loss of generality that each cat is on the south square of its cage. Given a grid state C , let $C \Downarrow\Downarrow$ denote the board state consisting of grid state C with both caged cats on the south square. The current position can then be written $gC \Downarrow\Downarrow$, denoting that Gold has just caused board state $C \Downarrow\Downarrow$. From position $gC \Downarrow\Downarrow$, Silver can make a cat cage move (moving its cat to the north square), resulting in position $sC \Downarrow\Uparrow$. As Gold has no more legal moves left in the grid, he is forced to make a cat cage move, resulting in position $gC \Uparrow\Uparrow$. Because Gold has no other options, Silver can force the following sequence: $gC \Downarrow\Downarrow$ (initial position), $sC \Downarrow\Uparrow$, $gC \Uparrow\Uparrow$, $sC \Uparrow\Downarrow$, $gC \Downarrow\Downarrow$ (second occurrence), $sC \Downarrow\Uparrow$ (second occurrence), $gC \Uparrow\Uparrow$ (second occurrence), $sC \Uparrow\Downarrow$ (second occurrence). From here, Gold has to create position $gC \Downarrow\Downarrow$, but this is illegal in Arimaa as it repeats the position a third time. Accordingly, Gold has no legal moves and loses by Arimaa rules.

We now see that the presence of cat cages and the use of the repetition rule compensates for the difference between Arimaa and Arimaa' caused by the 8.2-8.3 rule changes. On the other hand, while passing can be simulated by with cat cage moves, Arimaa player can actually make cat cage moves back and forth for longer than players can pass back and forth in Arimaa' (compare the move sequence above in this proof with Remark 9). However, just as in Arimaa', players cannot trivially undo an opponent's decision to simulate passing the turn with cat cage moves; we have seen that if a player's opponent insists on only cat-cage movement, then the player must eventually progress the simulation with a move in the grid. ◀

▶ **Remark 14.** In the course of play, there may arise positions in which the grid contains arrangements of pieces that function similarly to a cat cage. That is, there may arise groups of pieces internally within the grid such that a cat imprisoned inside is mobile, but can only move through a fixed region of squares within. We call such a region a *synthetic cat cage*. This may occur after a player uses a turn switcher gadget (see Section 3.3.2).

We observe that synthetic cages have no effect on the outcome of the game, for either the Arimaa' or Arimaa players discussed above. If an Arimaa player makes a synthetic cage move, the result is equivalent to making a cat cage move.¹¹ If an Arimaa' player makes a synthetic cage move, the opponent can respond with a pass, forcing the first player to do something productive.

2.4.6 Reachability from the starting position

Before we proceed to the proof of the main theorem, we note that any simulated AGG_{1,1} position on an Arimaa and Arimaa' board can be reached from the start of the game. This is easy to see, as pieces can move any direction (except rabbits, which cannot move backward). The board must be large enough that the home ranks can contain all pieces needed for the simulation. Then the players offer each other no resistance when moving pieces to form the simulated AGG_{1,1} position. Excess pieces can be captured in traps.

3 Proof

In this section we present the formal proof of Theorem 1:

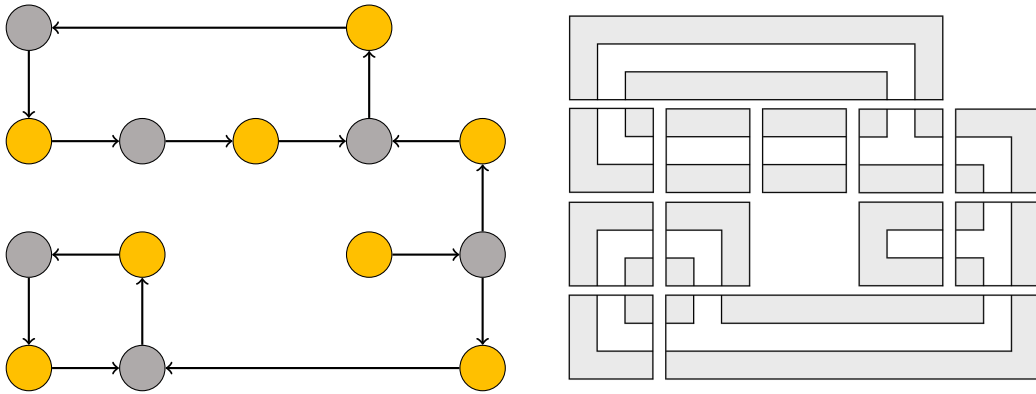
▶ **Theorem 1.** *The Arimaa decision problem is PSPACE-hard.*

It will be clear that the reduction from AGG_{1,1} to Arimaa', like the reduction from Arimaa' to Arimaa, is polynomial-time computable.

3.1 Movement through the grid

The simulation of the AGG_{1,1} game in Arimaa' works by always having only one player, called the *active player*, incentivized to move pieces, while the other player, called the *inactive player*, simply passes the turn repeatedly, until the active player completes a series of moves that simulate marking a node. This process involves a partial gadget called a *turn switcher* (see Section 3.3.2). After this, the roles switch as the second player becomes incentivized to move pieces while the first player repeatedly passes the turn, and so on. This continues until the simulated AGG_{1,1} game is complete.

¹¹The reader can verify that if both players are making (synthetic or non-synthetic) cage moves in a sequence similar to the eight-turn sequence in the proof above, then switching mid-sequence to moving a cat from a different cage does not help.



■ **Figure 8** An $AGG_{1.1}$ graph (left) and sketched representation by Arimaa' gadgets (right).

► **Definition 15.** *The key piece in the grid is the loose cat, a free cat that is the only piece on the board able to travel. This cat is capable of moving from one gadget to the next. For as long as the simulation is running, the active player will have a loose cat.*

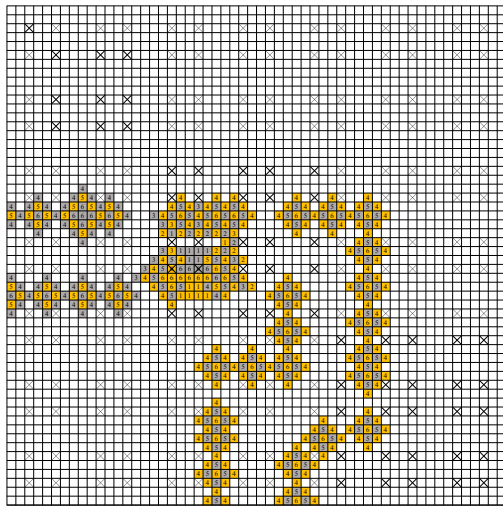
The simulation begins with Silver active (see Section 3.3.1). Players keep their roles until forced to switch in a turn switcher (see Section 3.3.2). We consider two cases. If the active player has a winning Geography strategy, the active player will have incentive to continue making moves that progress the $AGG_{1.1}$ simulation. If the active player does not have a winning strategy, the inactive player will keep passing turns and thereby force the active player to keep making moves in the simulation. In this case, while the active player may initially make irrelevant moves, the threat of threefold repetition will eventually force the loose cat to make progress in the simulation. This principle ensures that the losing player keeps moving through the corridors and turn switchers. So, in either case the simulation will eventually reach its natural conclusion.

3.2 Connecting gadgets

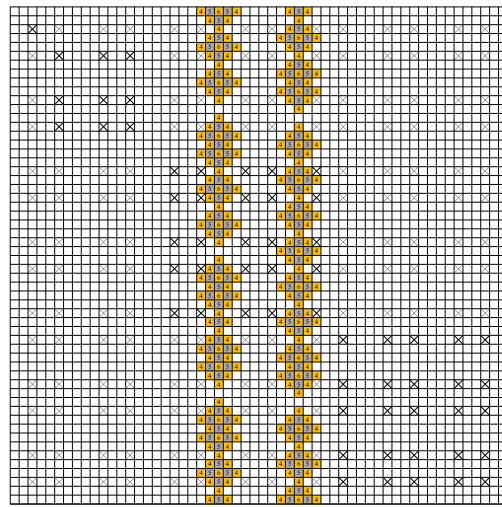
Before we present the details of the gadgets representing the different nodes of the $AGG_{1.1}$ game, we look at how they are stitched together to form the grid. In Figure 8 we see how the example $AGG_{1.1}$ graph from Figure 3 looks once transformed into square blocks. The example is made out of a starting node, one fork, two joins, two pipes, and eight bends. These blocks will become the gadgets in the reduction. For some examples, see Figure 9. The gadgets each use 49 concatenated standard Arimaa boards, arranged in a 7×7 square. The connecting corridors are all centered and have standard parity, ensuring that the gadgets connect correctly to each other. The detailed figures in the rest of this section cut off some of the corridor and empty space, leaving only the most critical parts of the gadgets or partial gadgets visible.

3.3 Gadgets

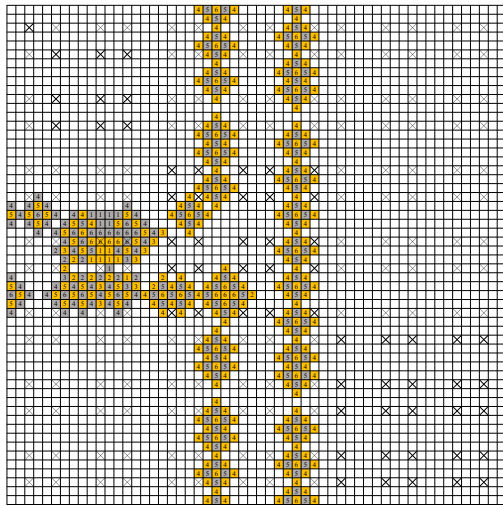
All the stable structures forming the simulation are built in standard parity, except for the turn switcher. Most gadgets in this section are depicted in just one color, but each can be transformed by reversing the piece colors, while preserving the parity by shifting the wall position one square as discussed in Section 2.4.2. The turn switcher poses a more difficult challenge, as we will see in Section 3.3.2.



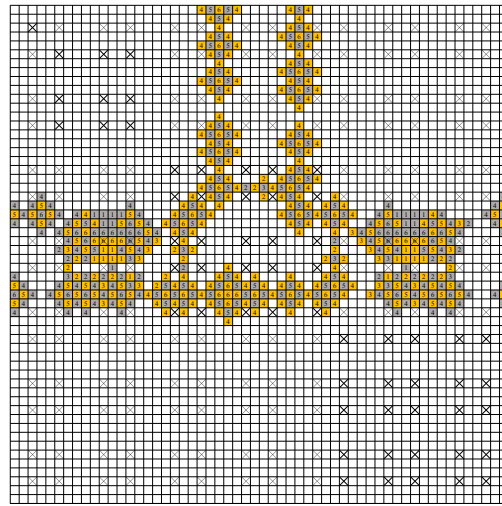
(a) A silver south-to-west bend gadget.



(b) A long vertical silver edge.



(c) A gold fork gadget.



(d) A gold join gadget.

■ **Figure 9** Complete gadget examples.

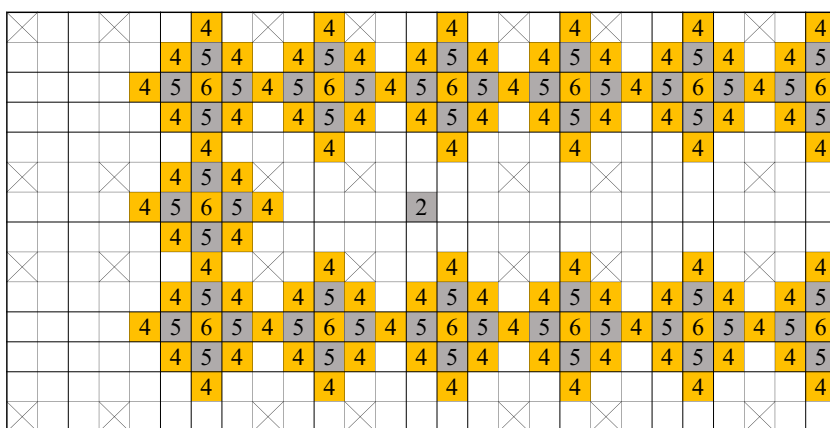
3.3.1 Starting gadget

Recall that an $AGG_{1,1}$ game starts with Gold marking the start node and Silver then marking the next node. The player exiting the gadget simulating the start node should therefore be Silver. This gadget representing the start node is nothing more than a corridor that is closed on one side, with the other side connecting to the gadget representing the second node (see Figure 10). In it is a loose silver cat to start the simulation. This is the only free piece at the outset; the other pieces on the board are either part of the grid or rabbit prisons, hence stable.

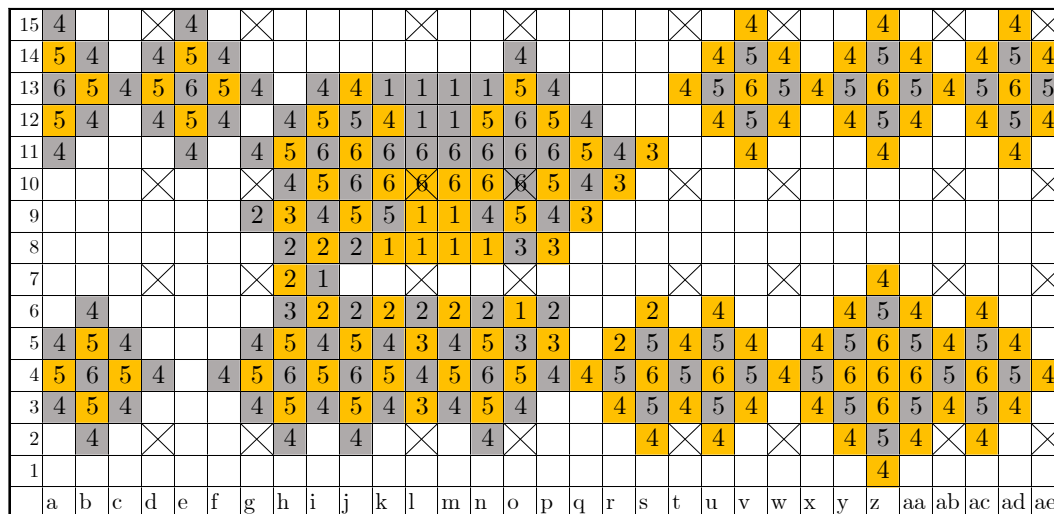
3.3.2 Turn switcher

This is the most important and complex partial gadget, responsible for switching the active and inactive player. In broad terms, it activates when a loose cat approaches and frees a friendly cat that then pushes an opposing rabbit down a narrow passage. At some point,

27:16 Arimaa Is PSPACE-Hard



■ Figure 10 The critical part of a starting gadget.



■ Figure 11 A turn switcher changing Gold as active player to Silver.

the inactive player will have to respond, as otherwise the second cat will be able to push all the way through the passage and end up on the other side of the turn switcher, where it will be able to free an elephant that can be used to win the game. Now follows a detailed explanation.

Figure 11 shows a turn switcher, which we see is completely stable before it gets activated. Avoiding repetition, a loose cat from Gold will eventually enter the switcher via the left corridor, where it will have no choice but to step on the g7 trap square and free the cat on h7. The h7 cat then has no choice but to push the silver rabbit on i7 east, causing the g7 cat to be captured. The remaining free cat is then forced to stay east of h7, because on h7 it would get frozen again, allowing Silver to win by repeatedly passing the turn. At some point, Gold will have to push the silver rabbit further right to j7 (case 1), k7 (case 2), l7 (case 3), m7 (case 4), n7 (case 5), o7 (case 6), and/or p7 (case 7). As Gold can push the rabbit up to twice per turn before Silver gets to move, Gold must choose which squares to allow the rabbit to be on when Silver starts her turn. For example, if Gold wants to avoid leaving the rabbit on m7, then Silver will necessarily have turns in which her rabbit begins on l7 and n7.

We consider all cases below. In each case, we will assume Gold keeps the loose cat adjacent to the silver rabbit between j7 and p7. If Gold instead decides to abandon the silver rabbit and bring the cat back west, nothing changes in how Silver should act.

Case 1 (j7)

The silver cat on j8 is now free and can push the k8 rabbit, freeing the elephant on j10 and allowing it to break down the wall and complicate the position. However, this series of events would never happen, because this requires both players to cooperate; Gold would have to choose to push the rabbit to j7 and end the turn, and Silver would then have to move the j8 cat. Whichever player has a winning strategy in the $AGG_{1,1}$ simulation could decline to let this sequence of moves occur.

Case 2, 4 (k7, m7)

The silver rabbit is frozen, so Silver simply passes and soon Gold must push the rabbit again.

Case 3 (l7)

Similar to cases 2 and 4. The rabbit can only move one square east before it gets frozen again, but it will have no incentive to do so as Gold can reply by immediately starting to pass all turns, causing Silver to lose the game. Meanwhile, the silver cat on l6 is technically unfrozen now, but remains still unable to move because it has become blockaded.

Case 5 (n7)

Similar to case 1, with the n6 cat replacing the j8 cat.

The interesting cases are 6 and 7, as Silver will be forced to stop passing and make a move in the simulation. At the end of case 6, the roles of active and inactive player will have been switched. Case 7, we will see, will never occur.

Case 6 (o7)

In this position, we claim that Gold is now making a game-winning threat, forcing Silver to stop passing and instead move her now free rabbit. The threat is that if Silver passes now, Gold can double-push the rabbit east to q7 in his next turn, after which Silver is unable to stop Gold from going to the elephant prison and winning the game. We now explain how Gold would accomplish this, after which we describe Silver's answer to the threat.

The elephant prison is centered around the square z4. The purpose of the elephant prison is to give the active player a way to free an elephant without freeing the opponent's pieces.¹² In the event that Silver has not stopped the loose cat of Gold from walking all the way to p7 and pushing the silver rabbit all the way to q7, Silver now has no free pieces in this turn switcher. This allows Gold's cat to keep the rabbit frozen and push it into a trap at his leisure, then move toward the gold 4 on z7. Unfreezing the 4 activates the elephant prison. The 4 can move aside to give room to the elephant on z5 to push out the 5 on z6 (keeping it always frozen). This 5 may be captured with any of the available traps, leaving the elephant completely free. The silver 5s on y5 and aa5 stay frozen by the gold elephants below them.

¹²As it turns out, this construction is more elaborate than strictly needed. It would suffice to replace the elephant prison with normal wall, letting one of the wall's outer 4s play the elephant's role, as 4s are also large enough to escape the grid and win. Still, we aesthetically favor using the strongest piece.

After the elephant is freed, it can move through the gaps in the walls and escape from the grid to the large empty space outside. Once there, it will go to its friendly rabbit prison to free the rabbit there and win the game.

Returning to our initial case 6 position, Silver can and must prevent all this by moving her rabbit from o7 to p7, which unfreezes the cat on p6 so that it can push the gold rabbit on o6 to the o7 trap.¹³ The purpose of Silver pushing the gold rabbit from o6 onto o7 is to restrict the mobility of Gold's n7 cat. If Silver doesn't do this, Gold's n7 cat would have the freedom to travel east and escape the turn switcher, which could benefit only Gold. So, having now used three out of four movement steps this turn, Silver can either end the turn now or move her cat back to p6 and then end the turn. Assume for simplicity that she chooses to move now to p6, since we will see that the cat must move to p6 soon anyway.

After this turn by Silver, it is Gold to move. In this position, we claim Gold now has no choice but to pass his turns. Observe that the rabbit now on the o7 trap square is immobile, as it is blocked laterally and to the north, and gold rabbits can never move south. Further observe that if Gold moves his n7 cat, this rabbit will be captured by the trap, and Gold would only end up making irrelevant moves with his cat between i7 and n7 (moving all the way to h7 would just get it frozen). Meanwhile, Silver would simply be able to pass the turn repeatedly and make Gold lose the game by repetition.¹⁴ So, with Gold forced to keep his cat and rabbit still, we see that he must pass, at which time Silver becomes the active player and the silver cat now located on p6 becomes the loose cat. It is now free to go into the rest of the gadget this turn switcher is a part of and continue the simulation. The gold cat, meanwhile, stays permanently imprisoned between i7 and n7 and will never again be relevant in the game (see Remark 14).

As an aside, note that Silver cannot leave the switcher with both her newly loose cat and her rabbit, as each cannot free the other without freezing itself. (The useless rabbit is anyway confined to only six squares – see fn. 13.) So there is always only one loose piece.

Case 7 (p7)

Gold will not allow this to occur. If he does, Silver wins as follows. First, Silver captures the cat on the o7 trap by pulling the adjacent o6 rabbit with her p6 cat. Then she freezes that rabbit by pushing it back to o6. Gold has no choice now but to pass. The silver cat now pushes the rabbit to the o7 trap, capturing it, and travels backwards through the turn switcher to the west side. Careful to avoid the g7 trap, it frees the frozen 3, which frees a 4, which then escapes the grid and wins the game, similarly to gold's threat from case 6.

► **Remark 16.** In the construction of turn switchers, there are some subtleties in changing the switcher's color and/or direction, since almost every piece in it has its own function and since it is precisely built around and onto the trap squares. It also has an internal parity

¹³ It would be a losing strategy for Silver to move her rabbit further east than p7, as that would allow Gold's cat to catch up and freeze the rabbit, move it into a trap and win via the elephant prison. In any case, silver rabbits cannot move north, so the s6 cat will keep it forever confined to squares q7, r7, s7, q6, r6, and q5.

¹⁴ Note that even if this were an Arimaa game rather than Arimaa', Gold would still lose the game. It might seem at first glance that Gold's initial westward movement of his cat is itself a sort of synthetic cat cage move, making Gold the initiator (and therefore winner) of an eight-move repetition sequence (as he shuffles his cat between m7 and n7), but this is not the case. Gold's first movement to m7 causes the o7 rabbit to be captured, creating a necessarily new Arimaa position and allowing *Silver* to be the first to mark time in a cat cage.

change¹⁵ where there is a double column of adjacent rabbits and elephants of the same color. The upper left and the entire bottom side of the turn switcher in Figure 11 are of standard parity, so they need no adjustment to connect to walls of standard parity. The upper right corner, however, is not, so an extra gap of space is left between that part of the turn switcher that is not of standard parity and the wall that is. The gap is not wide enough for enemy cats to go through, as they would still be frozen by the pieces making up the wall.

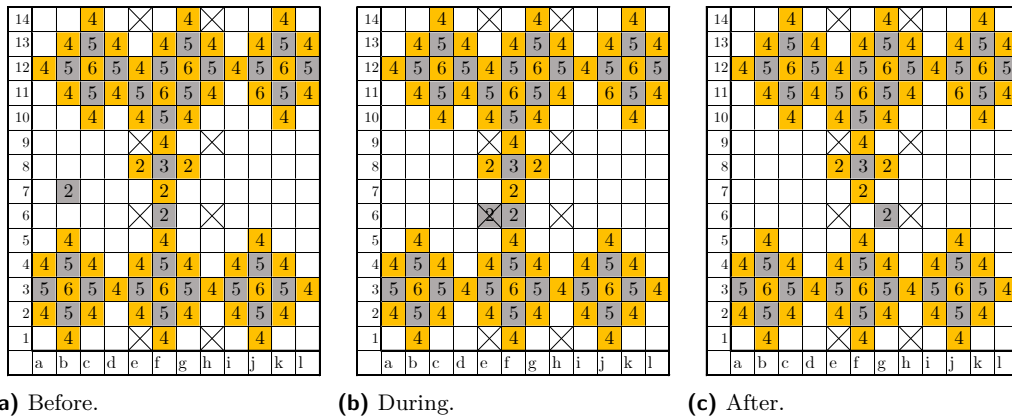
Our reduction requires four types of turn switchers: rightward gold-to-silver (as in Figure 11), leftward gold-to-silver (as in Figure 13, right turn switcher), rightward silver-to-gold (not depicted), and leftward silver-to-gold (as in Figure 9a). All are constructed similarly, but with the following adjustments. Changing the turn switcher’s direction requires mirroring it over its vertical axis, resulting in a parity change. Changing its color requires switching its piece colors and mirroring it over the horizontal axis (since gold and silver rabbits are mirrored in their vertical movements). In all cases, a wall gap similar to the one in s12-t12 in Figure 11 is included where needed.

3.3.3 Pipes and Bends

A pipe is easily constructed by a straight corridor with a turn switcher. A bend is similar, but with a 90° corner. Bends can be made in any direction, with the turn switcher in whichever of the two corridors connected by the corner is horizontal. See Figure 9a for an example.

3.3.4 Joins and Sleds

A join can be constructed from a two facing turn-switchers that feed into a common central area that leads into a vertical corridor. Within the central area are two occurrences of a small partial gadget that we call a *sled*. We now describe the workings of sleds, after which we discuss the join itself and how it uses sleds.



■ **Figure 12** A sled between two walls in a corridor in different stages of use.

Sleds are built between two corridor walls. They are designed to let just a single loose cat pass a single time. See Figure 12 for a sled in action. In 12a we can see a loose cat (b7) traveling east inside a corridor, encountering a sled. The sled functions like a dead end for

¹⁵The reason for this internal change of parity is that the turn switcher needs to be in standard parity both above and below the entrance – i.e., both h6 and h8 have silver pieces in Figure 11 – in order to keep the h7 cat initially frozen. By contrast, o6 and o8 have pieces of opposite color, as do p6 and p8, in order for the turn switcher mechanism to work.

this cat, and eventually it will have no choice but to move onto the trap square e6, as seen in Figure 12b. Now the cat on f6 is unfrozen and has no choice but to move east, becoming the new loose cat as the e6 cat gets captured (see Figure 12c). This new loose cat is free to go north (g7) and east (h7), then continue movement through this corridor. Observe that the sled can only be used in its intended direction and only one time.¹⁶

Recall that the main feature of an $AGG_{1.1}$ join is that it is the only type of node that may be marked twice, and subsequently the player who marked it wins the game. The Arimaa' join gadget is designed to work similarly. In Figure 13 we see a join ready to be marked by Gold.¹⁷ So Gold enters this join and Silver leaves it. Without loss of generality, assume Gold enters from the west. Silver emerges from the turn switcher and has no choice but to use the sled on the west side. With the loose silver cat now in the middle of the join, the only place it can go is the vertical corridor exiting the gadget toward the turn switcher of the next node gadget (north of this join gadget), since the east sled cannot be used from this side. The next time this join gadget is used, Gold will activate the turn switcher on the east to enter. Silver again emerges and has no choice now but to use the eastern sled and again go to the vertical corridor leading to the next gadget. But since the next gadget's turn switcher cannot be used twice (the entrance works like a sled), Silver is now unable to get past this point. Stuck in a dead end, she will eventually run out of new moves and lose the game by repetition.

3.3.5 Forks

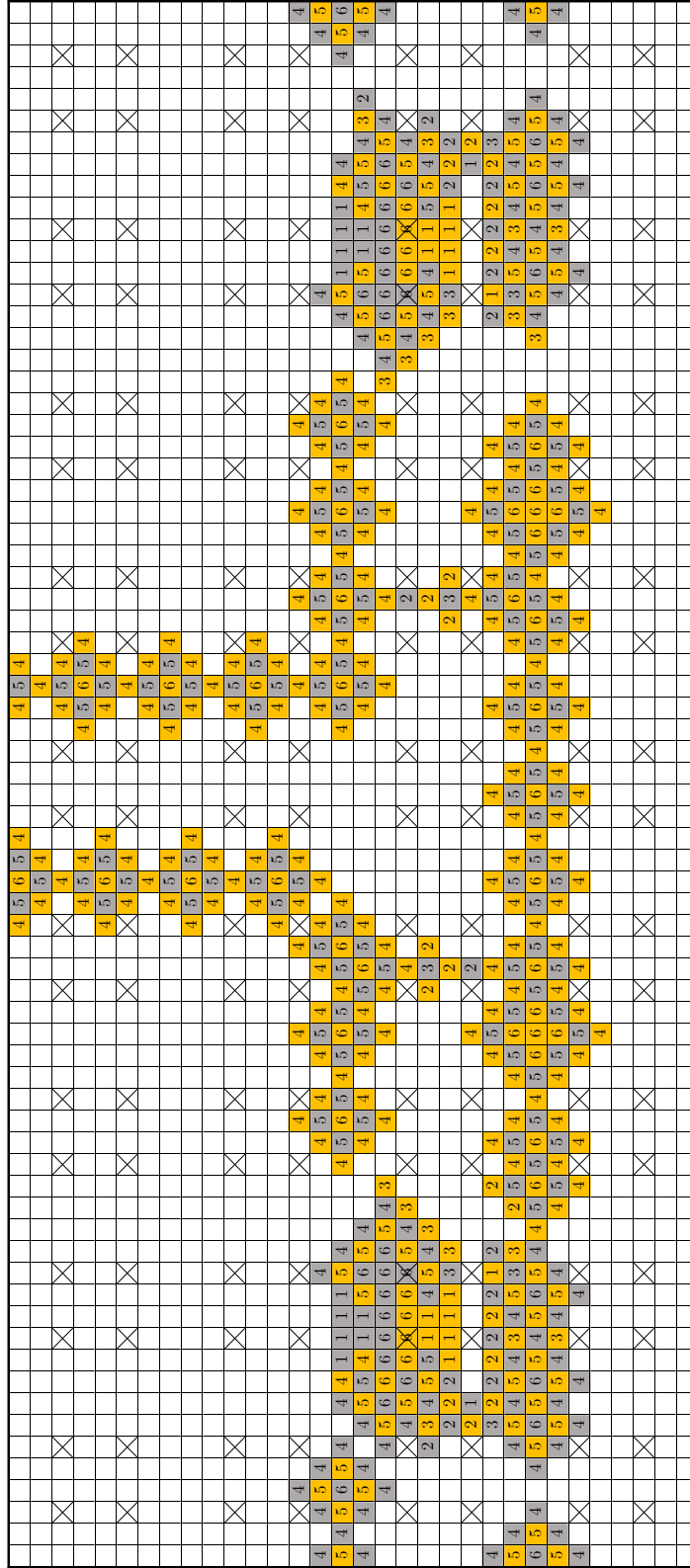
Forks gadgets represent nodes with indegree 1 and outdegree 2. The fork shown in Figure 14 is a gold fork. The silver loose cat that will leave the turn switcher may choose to take either the northern or southern outgoing corridor. Backtracking at a later stage in the simulation to also use the other direction of the fork is impossible, as each node the fork feeds into has a turn switcher, which a cat cannot go through in reverse.

3.4 Proof conclusion

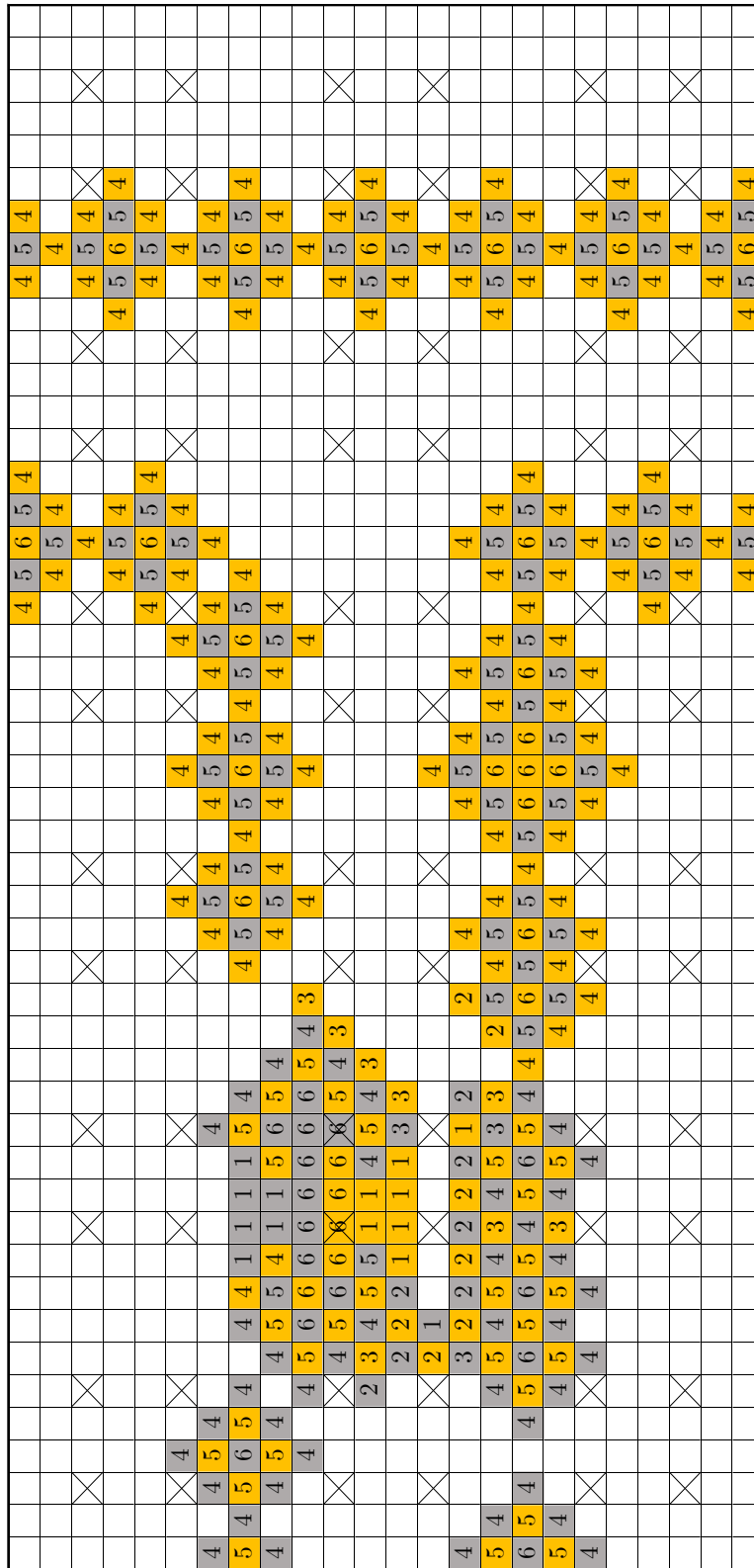
We see that every gadget works as required. This completes the proof of Theorem 1. ◀

¹⁶The metaphor of a sled is imperfect, since the cat entering the contraption is not the same physical cat as the one coming out. But it is our attempt to convey a form of uni-directional travel that (for a cat, at least) can be used only once.

¹⁷This depiction is modified from the one in Figure 9c, in order to show its functioning more clearly. The official join from 9c is more compressed and has only one elephant prison that is shared by both turn switchers.



■ Figure 13 The critical part of a join gadget, expanded.



■ **Figure 14** The critical part of a fork gadget.

4 Discussion and Conclusion

4.1 Overview

The inception of the game Arimaa has inspired many researchers to explore the practical complexity of the game when developing their bots. The result of this paper on the theoretical complexity of the game proves that the problem of solving Arimaa is intractable (under the assumption $P \subsetneq PSPACE$). It also gives us a way to compare the game to other games in terms of difficulty, and of course adds one more problem to the library of known PSPACE-hard problems.

As remarked earlier, one notable feature of this proof is that it shows PSPACE-hardness of a game whose pieces only move short distances in a single turn. While our reduction strategy is largely based on that of [16] for chess, chess does not present this same challenge because some pieces can move arbitrarily many squares per turn. In Arimaa, the strictly local movement of pieces calls for the idea of cat cages and Arimaa' in our reduction.

4.2 Future research

This paper shows that Arimaa is PSPACE-hard because $AGG_{1,1}$ reduces to Arimaa', which in turn reduces to Arimaa. The reduction presented in the proof works with traps placed on the corners of a 4×4 square, with four empty spaces between adjacent groups of traps and two empty spaces separating traps within a group. However, another way to generalize the standard 8×8 Arimaa board could be to place traps with two empty spaces between them in all directions. For this alternative version of $n \times n$ Arimaa, the reduction in this paper would no longer work, as it is impossible to create any kind of stable building block like the one we used here for the walls. This does not rule out all other possible reductions, but at the moment we do not know how they would look. With minor adjustments, our reduction can work with a constant number of three or more empty squares between traps, although that might be less in line with the spirit of the original game. In future research it would be interesting to see a reduction for a version containing traps spaced with two squares between them in every direction (rather than alternating gaps of two and four squares), if possible.

Further, as discussed in this paper's introduction, we expect that it is possible to prove Arimaa is EXPTIME-hard, perhaps using a reduction similar to that of [8]. Raising the lower bound of Arimaa's complexity to match the upper bound, this would make Arimaa EXPTIME-complete, just like chess (without a generalized 50-move draw rule) [8], checkers [13], and Go (with Japanese ko rules) [12]. However, we also consider it possible, though less likely, that Arimaa is in PSPACE and is therefore PSPACE-complete.

References

- 1 Joshua Ani, Jeffrey Bosboom, Erik D. Demaine, Yevhenii Diomidov, Dylan H. Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara, editors, *10th International Conference on Fun with Algorithms, FUN 2021, May 30 to June 1, 2021, Favignana Island, Sicily, Italy*, volume 157 of *LIPICs*, pages 3:1–3:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.FUN.2021.3.
- 2 Arimaa.com. The 2015 Arimaa Challenge. <http://arimaa.com/arimaa/challenge/2015/>, 2015. [Online; accessed 20-02-2024].
- 3 Davide Bilò, Luciano Gualà, Stefano Leucci, Guido Proietti, and Mirko Rossi. On the PSPACE-completeness of Peg Duotaire and other Peg-Jumping Games. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms (FUN 2018)*, volume 100 of *Leibniz International Proceedings in Informatics*

- (*LIPICs*), pages 8:1–8:15, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.FUN.2018.8.
- 4 Edouard Bonnet, Florian Jamain, and Abdallah Saffidine. Havannah and TwixT are PSPACE-complete. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaatt, editors, *Computers and Games - 8th International Conference, CG 2013, Yokohama, Japan, August 13-15, 2013, Revised Selected Papers*, volume 8427 of *Lecture Notes in Computer Science*, pages 175–186. Springer, 2013. doi:10.1007/978-3-319-09165-5_15.
 - 5 Christ-Jan Cox. Analysis and implementation of the game Arimaa. *MSc diss., Universiteit Maastricht, The Netherlands*, 2006.
 - 6 David Fotland. Building a world-champion Arimaa program. In *International Conference on Computers and Games*, pages 175–186. Springer, 2004.
 - 7 David Fotland. Computer Arimaa: The beginning. *International Computer Games Association Journal*, 38(1):12–18, 2015. doi:10.3233/icg-2015-38103.
 - 8 Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n . *J. Comb. Theory, Ser. A*, 31(2):199–214, 1981. doi:10.1016/0097-3165(81)90016-9.
 - 9 Brian Haskin. A look at the Arimaa branching factor. http://arimaa.janzert.com/bf_study, 2006. [Online; accessed 20-02-2024].
 - 10 David Lichtenstein and Michael Sipser. Go is polynomial-space hard. *Journal of the ACM*, 27(2):393–401, apr 1980. doi:10.1145/322186.322201.
 - 11 Stefan Reisch. Hex ist PSPACE-vollständig. *Acta Informatica*, 15:167–191, 1981. doi:10.1007/BF00288964.
 - 12 J. M. Robson. The complexity of Go. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 413–417. North-Holland/IFIP, 1983.
 - 13 J. M. Robson. N by N checkers is Exptime complete. *SIAM Journal on Computing*, 13(2):252–267, 1984. doi:10.1137/0213018.
 - 14 Thomas J. Schaefer. Complexity of decision problems based on finite two-person perfect-information games. In Ashok K. Chandra, Detlef Wotschke, Emily P. Friedman, and Michael A. Harrison, editors, *Proceedings of the 8th Annual ACM Symposium on Theory of Computing, May 3-5, 1976, Hershey, Pennsylvania, USA*, pages 41–49. ACM, 1976. doi:10.1145/800113.803629.
 - 15 Atze Schipper. $n \times n$ Arimaa is PSPACE-hard. Bachelor’s thesis, Utrecht University, 2021.
 - 16 James A. Storer. On the complexity of chess. *Journal of Computer and System Sciences*, 27(1):77–100, 1983. doi:10.1016/0022-0000(83)90030-2.
 - 17 Omar Syed. The Arimaa Challenge: From inception to completion. *International Computer Games Association Journal*, 38(1):3–11, 2015. doi:10.3233/ICG-2015-38102.
 - 18 Omar Syed and Aamir Syed. Arimaa - A new game designed to be difficult for computers. *International Computer Games Association Journal*, 26(2):138–139, 2003. doi:10.3233/ICG-2003-26213.
 - 19 Gerhard Trippen. Plans, patterns, and move categories guiding a highly selective search. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers*, volume 6048 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2009. doi:10.1007/978-3-642-12993-3_11.
 - 20 Jaap van den Herik. Table of contents / editorial. *International Computer Games Association Journal*, 38(1):1–2, March 2015. doi:10.3233/icg-2015-38101.
 - 21 Wikibooks. Arimaa/Playing the game. https://en.wikibooks.org/wiki/Arimaa/Playing_The_Game, 2019. [Online; accessed 20-02-2024].
 - 22 David J. Wu. Move ranking and evaluation in the game of Arimaa. Bachelor’s thesis, Harvard College, 2011. URL: <https://api.semanticscholar.org/CorpusID:504661>.

No Tiling of the 70×70 Square with Consecutive Squares

Jiří Sgall  

Computer Science Institute of Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic

János Balogh  

University of Szeged, Hungary

József Békési  

University of Szeged, Hungary

György Dósa  

University of Pannonia, Veszprém, Hungary

Lars Magnus Hvattum  

Molde University College, Norway

Zsolt Tuza  

University of Pannonia, Veszprém, Hungary

HUN-REN Alfréd Rényi Institute of Mathematics, Budapest, Hungary

Abstract

The total area of the 24 squares of sizes $1, 2, \dots, 24$ is equal to the area of the 70×70 square. Can this equation be demonstrated by a tiling of the 70×70 square with the 24 squares of sizes $1, 2, \dots, 24$? The answer is “NO”, no such tiling exists. This has been demonstrated by computer search. However, until now, no proof without use of computer was given. We fill this gap and give a complete combinatorial proof.

2012 ACM Subject Classification Mathematics of computing \rightarrow Combinatorics

Keywords and phrases square packing, Gardner’s problem, combinatorial proof

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.28

Funding *Jiří Sgall*: Partially supported by grant 24-10306S of GA ČR.

János Balogh: Supported by the grant TKP2021-NVA-09 of the Ministry for Innovation and Technology, Hungary.

1 Introduction

The total area of the 24 squares of sizes $1, 2, \dots, 24$ is equal to the area of the 70×70 square. In fact, this is the only nontrivial solution of the Diophantine equation $1^2 + 2^2 + \dots + n^2 = m^2$, see [7]. Can this equation be demonstrated by a tiling of the 70×70 square with the 24 squares of sizes $1, 2, \dots, 24$?

This natural question was popularized by Martin Gardner [3], who attributes the problem to R. B. Britton (unpublished). The answer is “NO”, no such tiling exists. This has been demonstrated by computer search, as claimed first in [2] without giving details and later in [4, 5]. However, until now, no proof without use of computer was given. We fill this gap and give a complete combinatorial proof.

Some initial steps towards a combinatorial proof were given in [6]. Using an extensive case analysis, it is shown there that squares of size up to 5 cannot be placed on the edges of the 70×70 board; in addition some further combinations of squares on edges, typically involving 6, 7, and/or 8, are also excluded.



© Jiří Sgall, János Balogh, József Békési, György Dósa, Lars Magnus Hvattum, and Zsolt Tuza; licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 28; pp. 28:1–28:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

This problem belongs to a wide class of Mrs. Perkins's Quilt problems, that in general ask for "squaring the square", i.e., tiling of a given square with smaller squares. See [1] for an overview of this rich area. Our problem is one special case. In addition to exact tiling, there are numerous results on packings that do not cover the whole square or coverings where some small squares overlap. One can then state various optimization problems, like minimizing the bounding box that allows packing a given set of small squares, or minimizing the overlap. One can also allow to omit some small squares and find a packing of a subset of squares in the given square that minimizes the empty space. In our case, the 70×70 square can accommodate all smaller squares except for the 7×7 square and the computer search in [5] also shows that 49 is indeed the smallest possible empty space.

1.1 New techniques and proof overview

The key to a shorter proof avoiding a long case analysis is a new insight into the global structure of a tiling. We assume that we have some tiling of the 70×70 board with the 24 squares and seek an eventual contradiction.

First observe that no three squares can together intersect all the rows as $22 + 23 + 24 = 69 < 70$. This allows us to select four **important rows** such that each square intersects at most one of them. Namely, select as two important rows the bottom row and the row just above the largest square intersecting the bottom row. Similarly, select the top row and the row just below the largest square intersecting the top row. Actually, there is some flexibility in this selection, and we may choose the important rows so that the distance among any two of them is at least 22 by moving the middle two important rows towards the center, if necessary. This simplifies some arguments later. Analogously we define four **important columns**. See Section 2 for precise definitions.

Now we classify all squares in the solution. A square is called **Major** if it intersects both an important row and an important column. **Minor** if it intersects an important row or an important column but not both. **Orphan** if it intersects neither an important row nor an important column.

See Fig. 1.1 for an illustration of the square types.

We continue by a counting argument that shows that there are at most two orphans which in addition have total size at most 4, see Theorem 2.4.

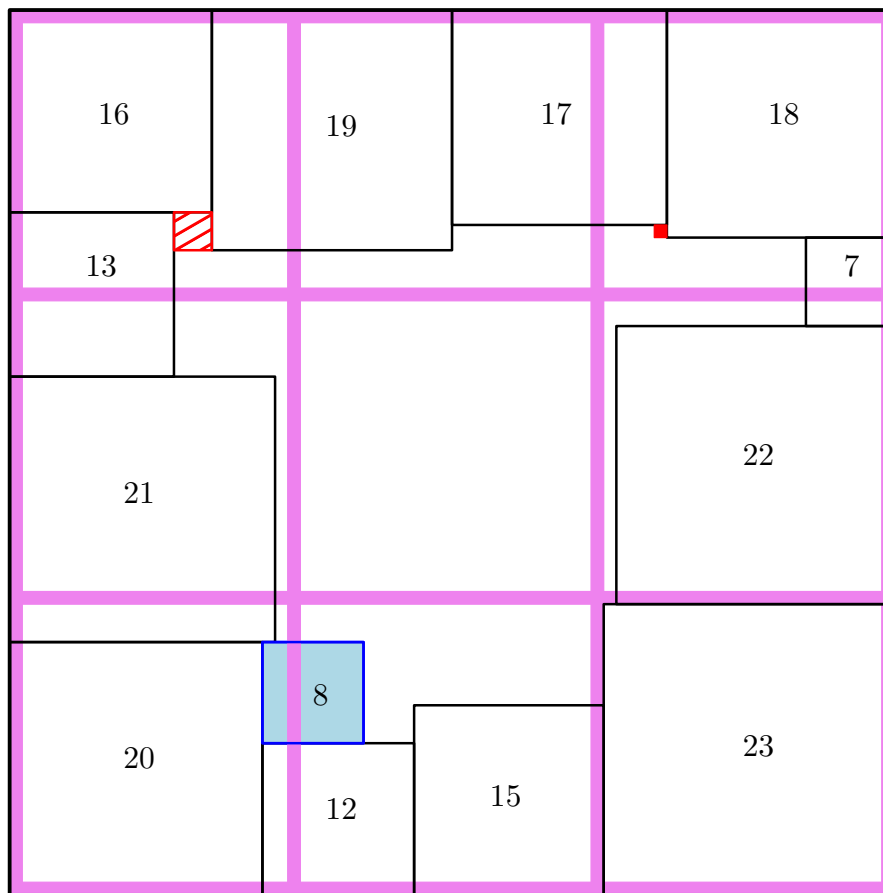
The limit of two orphans, additionally of small size, is a serious restriction. E.g., when trying to sketch a possible tiling, one realizes that a corner square often has an orphan adjacent to it, similarly to the top left corner in Figure 1.1, unless the corner square is very large. One can actually prove that a smallest square on an edge must have an orphan adjacent to it whenever it has size at most 11.

With this classification and restrictions in mind, we examine the tiling and exclude possible cases step by step.

We start by excluding small squares on the edges of the 70×70 board, somewhat similar to [6]. The bounds on orphans allow us to exclude squares of size up to 9 fairly easily, see Sections 3 and 4.

If all the squares on an edge have size at least 10, we show that there are only four squares on each edge, total of twelve squares along all edges, including the four corner squares touching two edges, see Lemma 3.6. With this restriction, it is not hard to exclude also a square of size 10 on an edge, see the end of Section 4.

More importantly, this restriction to only four squares on each edge allows us to use another counting technique introduced at the beginning of Section 5. We illustrate it on the case where all the squares on the edges have size at least 13. This means that the twelve



■ **Figure 1.1** Important rows and columns; square types.
 As a convention, in all our figures we use colors and shading as follows:

- the important rows and columns are drawn as violet thick lines;
- major squares are drawn in black with no fill;
- minor squares are drawn in blue with a light fill;
- orphans are drawn in red, filled if size is 1, hatched if size is larger.

squares on edges are the twelve largest squares and have total size equal to $13 + \dots + 24 = 222$. Furthermore, every corner square is adjacent to a smaller square on one of the edges and these squares are unique. It follows that the sizes of corner squares are at least 14, 16, 18, 20 and thus their total size is at least 68. However, this would imply that the circumference of the board is at least $222 + 68 = 290$, contradicting the obvious fact that the circumference is equal to $4 \cdot 70 = 280$.

The remaining case where a square of size 11 or 12 is on an edge is the most delicate part of the proof and needs a careful combination of all the techniques above, this is covered in Section 5. First we exclude the option that both 11 and 12 touch an edge. This gives a good lower bound on the total size of the twelve squares on the edges. Then, using some case analysis, we give a lower bound on the size of the corner squares. This leads to the final contradiction.

2 Important lines and classification of squares

As already used above, the board refers to the 70×70 square which should be tiled. The board consists of 70 rows and 70 columns of 1×1 squares that we call cells.

From now on, by a square we always mean one of the square tiles as positioned in the solution. Edges always mean edges of the board, while sides are the sides of a square. A square that is adjacent to an edge of the board is called an edge square; a square adjacent to two edges of the board is called a corner square.

We assume that we have some solution, i.e., a tiling of the board with the 24 squares. It is obvious that every square in the solution is aligned with the cells, as the solution covers the board perfectly. From now on, we consider only such positions of squares, i.e., only positions that have an integral offset from a corner of the board.

► **Definition 2.1.** *We define four important rows of cells:*

Bottom row: *The row along the bottom edge.*

Low row: *Let \mathbb{A} be the largest square touching the bottom edge, let $x = \max(22, \mathbb{A})$. The low row is the $(x + 1)$ -st row from bottom.*

High row: *Let \mathbb{B} be the largest square touching the top edge, let $y = \max(22, \mathbb{B})$. The high row is the $(y + 1)$ -st row from top.*

Top row: *The row along the top edge.*

Similarly, we define the first, left, right, and last columns and together call them important columns.

See Fig. 1.1 for an illustration of important rows and columns on a partial tiling. Note that the second important row and column do not touch any square on the top or left edge, as all these squares are smaller than 22 and thus the important row and column are moved towards the center, to maintain the desired spacing.

► **Observation 2.2.**

- (i) *No square intersects two of the important rows.*
- (ii) *Any 22 consecutive rows contain at most one important row.*
- (iii) *No square intersects two of the important columns.*
- (iv) *Any 22 consecutive columns contain at most one important column.*

Proof. We prove the first two items for rows. The claims for columns follow by symmetry.

By definition, no square intersects both bottom and low rows, and also no square intersects both top and high rows.

Let x be the number of rows below the low row and y the number of rows above the high row (as in the definition). For contradiction, assume that a square of size z intersects both low and high rows. Note that this together covers all rows, implying $x + y + z \geq 70$.

At most one of x, y, z can be equal to 24, as these options imply that the square 24 is either at the top edge, at the bottom edge, or none of the two, respectively. Similarly, at most one of x, y, z can be equal to 23. Thus $x + y + z \leq 24 + 23 + 22 = 69$, a contradiction.

For the second claim, we need to observe that there are at least 21 rows between two important rows. This follows by definition for bottom and low rows, and also for top and high rows. As $x + y \leq 23 + 24 = 47$, there are at least $70 - 47 = 23$ rows between low and high rows including these two important rows, and the claim follows as well. ◀

For completeness, we repeat the classification of the squares in the solution.

► **Definition 2.3.** A square is called

Major if it intersects both an important row and an important column.

Minor if it intersects an important row or an important column but not both.

Orphan if it intersects neither an important row nor an important column.

There are exactly 16 major squares, as there are exactly 16 cells in the intersections of important rows and important columns, each of these cells is covered by a square and no square contains two of these cells due to Observation 2.2.

In the next theorem we show that the total size of orphans is at most 4. This implies that there are at most two orphans and if there are two orphans, the smaller one has size equal to 1. In addition, we give all combinations of sizes of minor and major squares for all orphan sizes.

► **Theorem 2.4.** The total size of orphans is at most 4. There are at most two orphans and at most one orphan larger than 1. All possible combinations of sizes of orphans, minor and major squares are given in Table 2.1.

■ **Table 2.1** Possible combinations of minor and major squares. The dots in the first two rows in the column of minor squares denote all the smaller squares that are not among the orphans.

orphans	minor squares	major squares
1,3 4	... ,5,6,7,8	9,10,11,12,13,14,... ,24
1,2 3	... ,4,5,6,7,9	8,10,11,12,13,14,... ,24
2	1,3,4,5,6,8,9 1,3,4,5,6,7,10	7,10,11,12,13,14,... ,24 8,9,11,12,13,14,... ,24
1	2,3,4,5,7,8,9 2,3,4,5,6,8,10 2,3,4,5,6,7,11	6,10,11,12,13,14,... ,24 7,9,11,12,13,14,... ,24 8,9,10,12,13,14,... ,24
none	1,2,3,4,6,7,8,9 1,2,3,4,5,7,8,10 1,2,3,4,5,6,9,10 1,2,3,4,5,6,8,11 1,2,3,4,5,6,7,12	5,10,11,12,13,14,... ,24 6,9,11,12,13,14,... ,24 7,8,11,12,13,14,... ,24 7,9,10,12,13,14,... ,24 8,9,10,11,13,14,... ,24

Proof. Let MAJOR be the sum of sizes of all major squares, MINOR the sum of sizes of all minor squares, and ORPHAN the sum of sizes of orphans.

The expression $2 \cdot \text{MAJOR} + \text{MINOR}$ is equal to the sum of sizes of squares on all important rows and columns. Thus we have $2 \cdot \text{MAJOR} + \text{MINOR} = 8 \cdot 70 = 560$. The sum of all 24 square sizes is $1 + \dots + 24 = 12 \cdot 25 = 300$, so $\text{MAJOR} + \text{MINOR} = 300 - \text{ORPHAN}$. Together this implies $\text{MAJOR} = 260 + \text{ORPHAN}$.

Furthermore, MAJOR is at most the sum of the 16 largest squares, which gives $\text{MAJOR} \leq 9 + 10 + \dots + 24 = 8 \cdot 33 = 264$. Thus $\text{ORPHAN} \leq 4$. As the three smallest square sizes sum to 6, there are at most two orphans. Also, there cannot be two orphans larger than 1, as their total size would be at least 5.

It remains to examine all possible combinations of squares that sum up to ORPHAN and the corresponding value of MAJOR. The results are given in Table 2.1. ◀

3 Preliminary observations and small MIN cases

As in previous literature on the problem, it is convenient to consider various cases where we have some partial tiling of the 70×70 board and we try to extend it. Our goal is to achieve a contradiction, eventually covering all possibilities. To make our presentation complete, we cover all cases, including those already excluded in the previous literature.

We denote the squares by blackboard-bold letters \mathbb{A} , \mathbb{B} , etc. We overload this notation and use it for the size of squares, too. Similarly, square 1 denotes the unique square of size 1, square 2 denotes the unique square of size 2, etc.

The smallest edge square is denoted MIN . (Recall that an edge square is a square touching an edge of the board.) We will gradually restrict the possible sizes of MIN . Typically we will assume that MIN touches the bottom edge. Some of our claims speak about an edge square that is a local minimum, which means that on both sides it has either a larger edge square or an edge (i.e., it is a corner square). Obviously, MIN is always a local minimum.

The following observation shows that MIN is never a corner square, but is useful in other situations, too.

► **Observation 3.1.** *A corner square is always adjacent to a smaller edge square.*

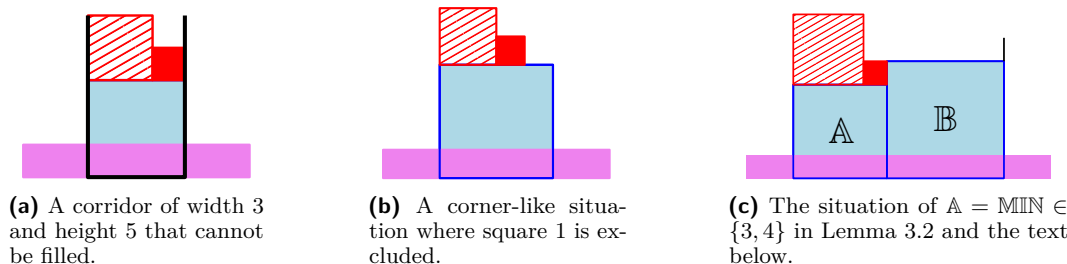
Proof. If a corner square is adjacent to two larger edge squares, then these two squares intersect, which is impossible. As there are no two equal squares, it follows that one of the adjacent edge squares is smaller than the corner square. ◀

Some configurations lead to an easy contradiction based only on local considerations. One of them is so-called corridor.

Given a partial tiling, a corridor is a rectangular space bounded by already placed squares from three sides and open on the last side. Its width is the length of the middle bounding segment and its height is the minimum of the two remaining bounding segments. Narrow corridors are quite restrictive. The only way to fill a corridor of width 1 or 2 is to use square 1 or 2, respectively, so if such a corridor has height of 3 or more, it cannot be filled at all. Similarly, a corridor of width 3 and height 5 cannot be filled, as the only way to cover the width is to use square 3 or squares 1 and 2 together. See Fig. 3.1(a).

Any square \mathbb{A} on the bottom edge that is a local minimum creates a corridor of width \mathbb{A} on top of it. This corridor must be filled with at least two squares, as size \mathbb{A} is already used. This immediately excludes the case $\text{MIN} \leq 2$, as the corridor on top of MIN cannot be filled.

Observation 3.1 excludes the situation where square 1 is placed in a corner, as it cannot be adjacent to two larger edge squares. This applies also to a corner-like configuration similar to Figure 3.1(b) where instead of a corner of the board we have two squares.



■ **Figure 3.1** Some configurations that are easy to exclude.

The following observation will be useful at multiple places. See Figures 3.1(c) and 3.3 for an illustration.

► **Lemma 3.2.** *Suppose a square \mathbb{A} on the bottom edge is a local minimum, i.e., any adjacent edge square on the bottom edge is larger, and it has square 1 on top. Then square 1 is the leftmost or rightmost square on top of \mathbb{A} and the adjacent edge square exists and has size $\mathbb{A} + 1$.*

Proof. If square 1 is in the middle of the top side of \mathbb{A} , it creates a corridor of width 1 that cannot be filled, as 1 is already used. Similarly, if it is next to a vertical edge or a square larger than $\mathbb{A} + 1$, it creates a corridor of width 1 that cannot be filled. ◀

Now we are ready to exclude the case of $\text{MIN} \leq 4$, which is known but the proof is significantly easier with our classification of squares. Suppose $\text{MIN} \in \{3, 4\}$ is at the bottom edge. By Theorem 2.4, MIN is a minor square, i.e., it cannot intersect an important column. The only possible way to fill the corridor above it is to use squares 1 and $\text{MIN} - 1$, and both of these squares are orphans. See Fig. 3.1(c) for reference. Lemma 3.2 now implies that the neighboring edge square \mathbb{B} has size $\text{MIN} + 1$. This is necessarily also a minor square, as we now have orphans. Also, the neighbor of \mathbb{B} on the other side than MIN is larger than \mathbb{B} or \mathbb{B} is a corner square. In both cases, there is a corridor above 1 and \mathbb{B} , thus any square on top of \mathbb{B} is another orphan, using also the fact that such squares cannot reach the low row. This is a contradiction as we have three orphans.

Similar ideas lead to following observations that limit the possibilities on top of local minimum squares and solve some more cases.

► **Lemma 3.3.** *Suppose that \mathbb{A} is a local minimum, w.l.o.g. on the bottom edge, and $5 \leq \mathbb{A} \leq 11$. Then \mathbb{A} is a major square, one square on top of it is minor and the other square(s) on top of it are orphans.*

Proof. Since $\mathbb{A} \leq 11$, all squares on top of it have size at most 10. Thus none of these squares intersects the low row. It follows that these squares are orphans or minor. They cannot be all orphans, as $\mathbb{A} \geq 5$. Thus one, and only one, of them is intersected by an important column. Consequently, \mathbb{A} is a major square. ◀

We immediately see that $\text{MIN} = 5$ is excluded, as Theorem 2.4 implies that a major square of size 5 cannot exist if there is an orphan.

At this point, it is good to summarize possible configurations on top of a local minimum $\mathbb{A} \leq 11$, see Fig. 3.2. In addition to the minor square, there are one or two orphans, i.e., at most three squares. If there are two orphans, one of them must have size 1 and it is not the middle square.

The following lemma shows that for $\mathbb{A} = \text{MIN} \leq 10$, only the configuration in Fig. 3.2(a) is possible.

► **Lemma 3.4.** *Suppose that $6 \leq \text{MIN} \leq 10$. Then MIN cannot have square 1 on top.*

Proof. Suppose for a contradiction that square 1 is on top of MIN . Lemma 3.3 implies that it is an orphan as otherwise there are orphans of total size $\text{MIN} - 1 \geq 5$. Then by Lemma 3.2, it is next to an adjacent edge square \mathbb{B} of size $\text{MIN} + 1$. See Fig. 3.3. The squares \mathbb{B} and 1 together create a corridor of width $\text{MIN} + 2$. This corridor is not intersected by an important column, as together with MIN they intersect at most $\text{MIN} + \text{MIN} + 1 \leq 21$ consecutive columns.

28:8 No Tiling of the 70×70 Square with Consecutive Squares

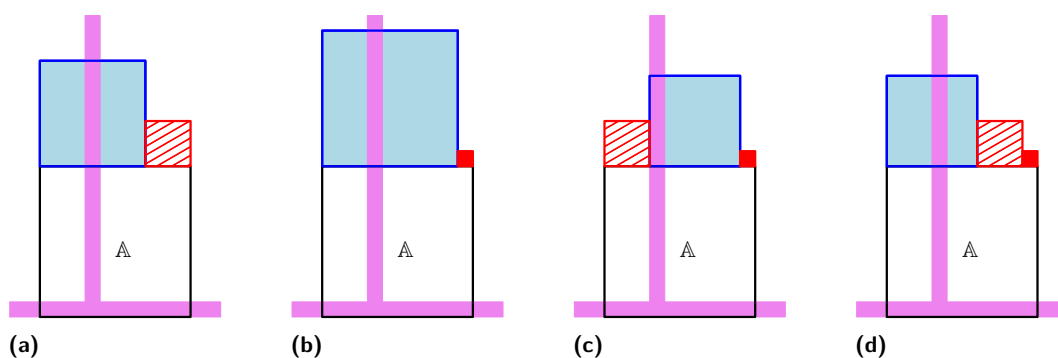


Figure 3.2 Possible configurations in the corridor above a local minimum A .

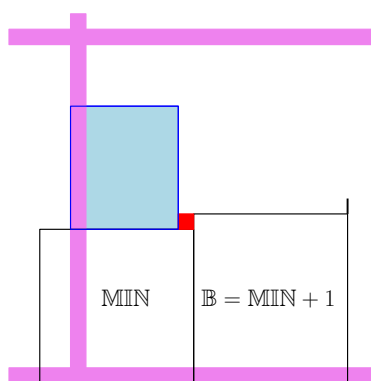


Figure 3.3 An illustration of the proof of Lemma 3.4. On the right, a big square or the edge of the board creates a corridor.

Thus any square filling this corridor must be an orphan, unless it intersects the low row. The only configuration where the low row may be intersected is when $\text{MIN} = 10$ and the whole corridor of width 12 is filled by a single square of size 12; as it sits on square $\text{MIN} + 1 = 11$, it reaches row 23. However, then the square 12 is minor, as it is not intersected by an important column. By Theorem 2.4 this is impossible, as there is an orphan.

In all the other cases the corridor is filled by orphans of total size $\text{MIN} + 2 \geq 8$, which is also a contradiction. ◀

The last lemma excludes the case $\text{MIN} = 6$, as a major square of size 6 is not compatible with orphans larger than 1.

The following proposition summarizes the cases excluded in this section.

► **Proposition 3.5.** *We have $\text{MIN} \geq 7$. Furthermore, if $\text{MIN} \leq 10$, then MIN is major and there is a single orphan on top of MIN and it has size at least 2.*

We note that already our Proposition 3.5 is as strong as the results of approximately 40 pages of [6].

The next lemma limits the number of edge squares for large MIN . We use it in the later parts of the proof.

► **Lemma 3.6.** *If $\text{MIN} \geq 10$, then each edge touches exactly four squares. There are exactly twelve edge squares, all of them are major squares.*

Proof. For a contradiction, assume that some edge touches more than four squares. Then there must be a minor edge square.

If $\text{MIN} = 10$ then Lemmata 3.3 and 3.4 imply that MIN is a major square and there is an orphan of size at least 2. This in turn, using Theorem 2.4, excludes the existence of a minor square of size 11 or more, thus we have a contradiction.

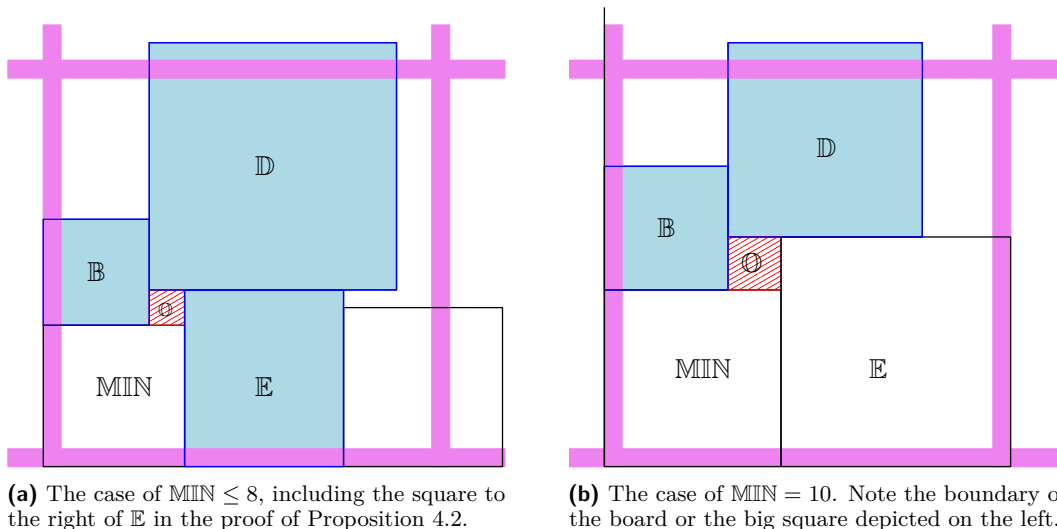
If $\text{MIN} = 11$ then by Lemma 3.3 it is a major square and there is an orphan on top of it. This excludes a minor square of size 12 or more, thus we have a contradiction.

Otherwise the minor edge square must have size 12 and $\text{MIN} = 12$. Then at least one of the squares on top of it does not intersect the low row, and thus it is an orphan. We have a contradiction again.

We have shown that each edge touches four squares. Thus we have four corner squares plus eight remaining edge squares, two per edge, a total of twelve edge squares. ◀

4 Medium MIN cases

In this section we exclude the remaining cases with $7 \leq \text{MIN} \leq 10$. Proposition 3.5 and Theorem 2.4 imply that MIN is a major square and there are two squares on top of it, a minor square \mathbb{B} and an orphan $\mathbb{O} \geq 2$; w.l.o.g. we assume that \mathbb{B} is on the left of \mathbb{O} . Let \mathbb{D} be the square touching both \mathbb{B} and \mathbb{O} . See Fig. 4.1.



■ **Figure 4.1** An illustration of the medium MIN case.

► **Lemma 4.1.** *Suppose that $7 \leq \text{MIN} \leq 10$ and \mathbb{B} , \mathbb{O} , and \mathbb{D} are as above. Then $\mathbb{D} \geq 9$ and if $\text{MIN} \leq 8$ then $\mathbb{D} \geq 17$. Furthermore, the edge square touching MIN on the right exists and has size exactly $\text{MIN} + \mathbb{O}$.*

Proof. We examine the possible types and sizes of \mathbb{D} .

If \mathbb{D} is an orphan, then $\mathbb{D} = 1$ and $\mathbb{O} \leq 3$. This in turn implies that $\mathbb{B} = \text{MIN} - \mathbb{O} \geq 4$. However, the combination of a major square $\text{MIN} = 7$ and an orphan $\mathbb{O} = 3$ is excluded by Theorem 2.4, so $\mathbb{B} \geq 5 \geq \mathbb{O} + 2$. It follows that \mathbb{O} is in a corner-like position similar to Fig. 3.1(a), which leads to a contradiction.

28:10 No Tiling of the 70×70 Square with Consecutive Squares

If \mathbb{D} is not an orphan, it has to intersect an important row or column or both. If \mathbb{D} intersects the low row, then $\text{MIN} + \mathbb{O} + \mathbb{D} \geq 23$, thus $\mathbb{D} \geq 23 - \mathbb{O} - \text{MIN} \geq 19 - \text{MIN} \geq 9$. If \mathbb{D} intersects an important column, then it is not the same important column as the one intersecting MIN and \mathbb{B} , thus \mathbb{B} and \mathbb{D} span at least 23 columns; we obtain $23 \leq \mathbb{B} + \mathbb{D} = \mathbb{D} + \text{MIN} - \mathbb{O}$ and thus $\mathbb{D} \geq 23 + \mathbb{O} - \text{MIN} \geq 23 + 2 - 10 = 15$. The bound $\mathbb{D} \geq 9$ follows.

Furthermore, if $\text{MIN} \leq 8$ then we get a stronger bound $\mathbb{D} \geq 19 - \text{MIN} \geq 11$. However, as we have an orphan $\mathbb{O} \geq 2$, Theorem 2.4 implies that $\mathbb{D} \geq 11$ has to be a major square. Thus it intersects both an important row and column and the same calculation as in the previous paragraph yields $\mathbb{D} \geq 23 + \mathbb{O} - \text{MIN} \geq 23 + 2 - 8 = 17$.

As $\mathbb{D} \geq 9$, it extends at least $\mathbb{D} - \mathbb{O} \geq 5$ squares to the right of MIN and \mathbb{O} . We claim that this implies that the edge square touching MIN on the right exists and has size at most $\text{MIN} + \mathbb{O}$. Suppose for a contradiction that it has a strictly smaller size. The size needs to be strictly larger than MIN . Then above this edge square and below \mathbb{D} we have a corridor of width 1, 2, or 3 and length at least 5, which cannot be filled, see Fig. 3.1(b), a contradiction. Thus the size of the edge square is equal to $\text{MIN} + \mathbb{O}$ and the last claim of the lemma follows. \blacktriangleleft

► **Proposition 4.2.** *We have $\text{MIN} \geq 11$.*

Proof. Let \mathbb{B} , \mathbb{O} , and \mathbb{D} be as above and let \mathbb{E} be the edge square touching MIN on the right.

If $\text{MIN} \leq 8$ then $\mathbb{O} \leq 3$ and $\mathbb{E} = \text{MIN} + \mathbb{O} \leq 11$. See Fig. 4.1(a) for an illustration. Lemma 4.1 implies that \mathbb{D} extends at least 3 cells to the right of \mathbb{E} . Thus there exists an edge square to the right of \mathbb{E} and it has size strictly between MIN and $\mathbb{E} \leq \text{MIN} + 3$. Thus above this edge square and below \mathbb{D} we have a corridor of width 1 or 2 and length at least 3, which cannot be filled, a contradiction.

If $\text{MIN} = 9$ then $11 \leq \mathbb{E} \leq 13$. Thus $\text{MIN} + \mathbb{E} \leq 22$ and by Observation 2.2 only one important column intersects MIN and \mathbb{E} ; we know that it intersects the major square MIN . Thus \mathbb{E} is a minor square. However, Theorem 2.4 excludes a combination of an orphan $\mathbb{O} \geq 2$ with a minor square $\mathbb{E} \geq 11$, a contradiction.

If $\text{MIN} = 10$ then Lemma 3.6 implies that there are only four edge squares on the edge of MIN . See Fig. 4.1(b) for an illustration. As $\mathbb{E} = \text{MIN} + \mathbb{O} \leq 14$, it follows that the remaining two edge squares are either 24 and 23 (and $\mathbb{E} = 13$) or 24 and 22 (and $\mathbb{E} = 14$). Thus $\mathbb{O} \in \{3, 4\}$ and $\mathbb{B} = 10 - \mathbb{O} \in \{6, 7\}$; also note that MIN and \mathbb{B} span at most 17 rows. Observe that on the left \mathbb{B} touches the edge or a square of size at least 22 and on the right it touches $\mathbb{D} \geq 9$ which extends above $\mathbb{B} \leq 7$ by at least $\mathbb{D} + \mathbb{O} - \mathbb{B} \geq 5$ cells. Thus there is a corridor on top of \mathbb{B} of length at least 5. It follows that there are at least two squares on top of \mathbb{B} . They cannot have size 1, as the cell above 1 cannot be covered, so they have size at least 2 and at most 5. None of them intersect the low row as they reach at most $17 + 5 = 22$ rows from bottom, and only one of them intersects an important column. Thus we have another orphan of size at least 2, a contradiction. \blacktriangleleft

5 Large MIN cases

At this point we know that $\text{MIN} \geq 11$. To exclude the cases with $\text{MIN} \geq 11$, we use a technique based on counting the sizes of edge squares.

Recall that by Lemma 3.6 there are exactly four squares along each edge, a total of twelve squares, all of them major. For the rest of the section, let E denote the sum of sizes of the edge squares and C the sum of sizes of the four corner squares.

5.1 Edge-square counting technique and $\text{MIN} = 13$

We introduce and demonstrate the new technique on the case $\text{MIN} \geq 13$, which is relatively easy.

► **Observation 5.1.** *We have $E + C = 280$.*

Proof. The sum of all the squares touching a given edge is 70. Summing over all edges we get $4 \cdot 70 = 280$. Considering that all corners touch two edges and the remaining edge squares only one edge, the sum is equal to $E + C$. ◀

► **Proposition 5.2.** *We have $\text{MIN} \leq 12$.*

Proof. Suppose for a contradiction that $\text{MIN} \geq 13$.

Then the sizes of the twelve edge squares are $13, \dots, 24$ (note that $\text{MIN} > 13$ is impossible). Thus $E = 13 + \dots + 24 = 6 \cdot 37 = 222$.

Every corner square is adjacent to a smaller edge square and these are unique. It follows that the sizes of corner squares are at least 14, 16, 18, 20 and thus $C \geq 14 + 16 + 18 + 20 = 68$.

We have $E + C \geq 222 + 68 = 290$, a contradiction. ◀

5.2 Small corners and edge squares

To use the edge counting technique for $\text{MIN} = 11, 12$, we use the following ingredients, to get the necessary bounds on E and C .

- (1) We prove that only one of 11 and 12 is used as an edge square. This gives us a bound $E \geq 209$ for $\text{MIN} = 11$ and $E \geq 210$ for $\text{MIN} = 12$.
- (2) We prove that MIN has an orphan of size at least 2 adjacent to it.
- (3) We prove that any corner square has size at least 14.
- (4) We carefully analyze certain corner squares of small size, to show that each of them enforces an orphan. This typically allows us to conclude that there is only one corner of size at most 17, giving us a bound $C \geq 71$.
- (5) In case $\text{MIN} = 11$, we need to work a bit harder to obtain the bound $C \geq 72$.

The first three items are covered in this subsection.

► **Observation 5.3.** *Suppose that a corner square \mathbb{C} neighbors with two edge squares such that \mathbb{A} has size $\mathbb{C} - \delta$ for $\delta \in \{1, 2\}$ and \mathbb{B} is larger than \mathbb{C} . Then \mathbb{B} has size at most $\mathbb{C} + \delta$.*

Proof. Suppose w.l.o.g. that \mathbb{C} is the bottom right corner and \mathbb{A} is on the bottom edge. Then we have a corridor of width δ on top of \mathbb{A} below \mathbb{B} . This corridor cannot be longer than δ , as it can be filled only by square δ . Thus \mathbb{C} cannot extend more than δ to the left of \mathbb{C} and the claim follows. ◀

► **Lemma 5.4.** *There is no corner square of size 12.*

Proof. Suppose for a contradiction that 12 is a corner square.

The corner 12 must have a smaller neighbor, which is necessarily $\text{MIN} = 11$. Observation 5.3 implies that the other neighbor of 12 has size 13. Assume w.l.o.g. that 12 is the bottom right corner and MIN neighbors it on the bottom edge.

It follows that the other two squares along the bottom edge have size 23 and 24. The remaining two squares along the right edge then have sizes at most 21 and 22. However, then the total size along the right edge is at most $12 + 13 + 21 + 22 = 68 < 70$, a contradiction. ◀

► **Lemma 5.5.** *We cannot have two edge squares of sizes 11 and 12.*

28:12 No Tiling of the 70×70 Square with Consecutive Squares

Proof. For a contradiction, assume that we have edge squares $\text{MIN} = 11$ and $\mathbb{B} = 12$. We distinguish two cases.

Case 1. $\text{MIN} = 11$ and $\mathbb{B} = 12$ are on the same edge, w.l.o.g. the bottom one. As none of them can be a corner by Lemma 5.4, they are the two middle squares. The two bottom corners have sizes 23 and 24.

Now consider the top row. All its squares have size at most 22, as 23 and 24 are the two bottom corners. Thus the two top corners have size at least $70 - 22 - 21 = 37$ and overall we get $C \geq 37 + 23 + 24 = 84$. Using again the fact that 23 and 24 are on the bottom edge, we can bound $E \geq (11 + 12 + \dots + 20) + 23 + 24 = 5 \cdot 31 + 47 = 202$. Thus we have $E + C \geq 202 + 84 = 284 > 280$, a contradiction.

Case 2. $\text{MIN} = 11$ and $\mathbb{B} = 12$ are on different edges. W.l.o.g. assume that \mathbb{B} is on the bottom edge and note that it is a local minimum.

Consider the squares on top of \mathbb{B} . As 11 is used elsewhere, they have size at most 10 and do not intersect the low row. It follows that one of them is an orphan of size at least 2.

As 12 is not adjacent to $\text{MIN} = 11$, it follows by Lemma 3.2 and 3.3, that there is an orphan of size at least 2 adjacent to MIN . This is a contradiction as there cannot be two orphans of size at least 2. \blacktriangleleft

► **Corollary 5.6.** *If $\text{MIN} = 11$ then $E \geq 209$. If $\text{MIN} = 12$ then $E \geq 210$.*

Proof. We have $E \geq \text{MIN} + 13 + 14 + \dots + 23 = \text{MIN} + 198$. The claim follows. \blacktriangleleft

► **Corollary 5.7.** *If $\text{MIN} = 11$, then MIN is adjacent to an orphan of size at least 2.*

Proof. By Lemma 3.3 there is an orphan on top of $\text{MIN} = 11$. If the orphan has size one, Lemma 3.2 implies that MIN is adjacent to 12, which is excluded by Lemma 5.5 \blacktriangleleft

► **Lemma 5.8.** *There is no corner square of size 13.*

Proof. Suppose for a contradiction that 13 is a corner square.

There is a smaller edge square adjacent to the corner 13, and Lemma 5.5 implies that this is $\text{MIN} \in \{11, 12\}$, as no other edge square smaller than 13 is available. Assume w.l.o.g. that 13 is the bottom right corner and MIN neighbors it on the bottom edge. Observation 5.3 implies that the other neighbor of 13 has size 14 or 15.

Now we examine the total size of the seven squares along the left and top edges. The total, counting the top left corner twice, is equal to 140. The seven squares can contain one of 14 and 15 (the one not adjacent to the corner 13) and then squares of size at least 16. Thus the total size of the seven squares is at least $14 + 16 + 17 + \dots + 21 = 14 + 3 \cdot 37 = 125$. Furthermore, the size of the top left corner is at least 16, as it has a smaller adjacent edge square. Thus the total along the left and top edges is at least $125 + 16 = 141 > 140$, a contradiction. \blacktriangleleft

We conclude this subsection by showing that MIN is adjacent to an orphan of size at least 2. We already know this for $\text{MIN} = 11$. The proof for $\text{MIN} = 12$ follows the line of proof of Lemma 5.5.

► **Lemma 5.9.** *If $\text{MIN} = 12$, then we cannot have two adjacent edge squares of sizes 12 and 13.*

Proof. For a contradiction, assume that we have adjacent edge squares $\text{MIN} = 12$ and 13 ; w.l.o.g. assume they are at the bottom edge. By Lemmata 5.4 and 5.8, these are not corners. It follows that the two bottom corners have total size $70 - 12 - 13 = 45$. The top corners have sizes at least 15 and 17 , as they are adjacent to smaller edge squares. Thus $C \geq 45 + 15 + 17 = 76$. Together with $E \geq 210$ we have $E + C \geq 210 + 76 = 286 > 280$, a contradiction. \blacktriangleleft

► **Corollary 5.10.** *If $\text{MIN} = 12$, then MIN is adjacent to an orphan of size at least 2.*

Proof. If there is 1 on top of $\text{MIN} = 12$ then Lemma 3.2 implies that MIN is adjacent to 13 , which is excluded by Lemma 5.9.

Thus all squares on top of MIN have size at least 2 . As there are at least two squares on top of MIN , they are all of size at most 10 . Thus they do not intersect the low row. This in turn implies that one of them is an orphan, as at most one of them is intersected by an important column, and we have obtained an orphan of size at least 2 . \blacktriangleleft

5.3 Medium corners and the final analysis

The core of the remaining argument is to examine the corners of medium size more carefully. Essentially, we need to extend Observation 3.1 to see that on one side of the corner square, the adjacent squares do not extend beyond the corner square. The neighboring edge square on that side is then called the key neighbor.

► **Definition 5.11.** *Suppose \mathbb{C} is a corner square and \mathbb{A} is an adjacent edge square. W.l.o.g. assume that \mathbb{C} is the bottom right corner and \mathbb{A} is on the bottom edge.*

We say that \mathbb{A} is a key neighbor of \mathbb{C} if $\mathbb{A} < \mathbb{C}$ and no square adjacent to the left side of \mathbb{C} extends beyond the top of \mathbb{C} .

► **Observation 5.12.** *Any corner square has at least one key neighbor.*

Proof. Suppose that \mathbb{C} is the bottom right corner, $\mathbb{A} < \mathbb{C}$ is an edge square adjacent to \mathbb{C} on the bottom edge and \mathbb{B} is an edge square adjacent to \mathbb{C} on the right edge.

If $\mathbb{B} > \mathbb{C}$ then no square adjacent to the left side of \mathbb{C} may extend above the top of \mathbb{C} , as it would intersect \mathbb{B} . So \mathbb{A} is a key neighbor of \mathbb{C} .

Suppose now that $\mathbb{B} < \mathbb{C}$ and it is not a key neighbor of \mathbb{C} . Then there exists a square \mathbb{D} adjacent to the top side of \mathbb{C} and extending to the left of \mathbb{C} . Now again no square adjacent to the left side of \mathbb{C} may extend above the top of \mathbb{C} , as it would intersect \mathbb{D} . So \mathbb{A} is a key neighbor of \mathbb{C} . \blacktriangleleft

Now we are ready to formulate the needed extension of Observation 5.3 for small corners.

► **Lemma 5.13.** *Suppose \mathbb{C} is a corner square and \mathbb{A} is its key neighbor and \mathbb{B} the other edge square adjacent to \mathbb{C} ; furthermore let $\delta = \mathbb{C} - \mathbb{A}$. W.l.o.g. assume that \mathbb{C} is the bottom right corner and \mathbb{A} is on the bottom edge, thus \mathbb{B} is on the right edge.*

Then the following holds:

- (1) *If $\mathbb{A} \neq \text{MIN}$ and $\mathbb{C} + \delta \leq 22$ then $\delta = 1$ and $\mathbb{B} \leq \mathbb{C} + \delta$.*
- (2) *If $\mathbb{A} = \text{MIN}$ and $\mathbb{C} \leq 17$ then $\delta \in \{2, 3, 4\}$ and either $\mathbb{B} = \mathbb{C} + \delta$ or $\mathbb{B} = 13$, $\mathbb{C} = 16$, $\text{MIN} = 12$, and $\mathbb{O} = 4$.*

Furthermore, in both cases there is an orphan of size δ adjacent to \mathbb{C} .

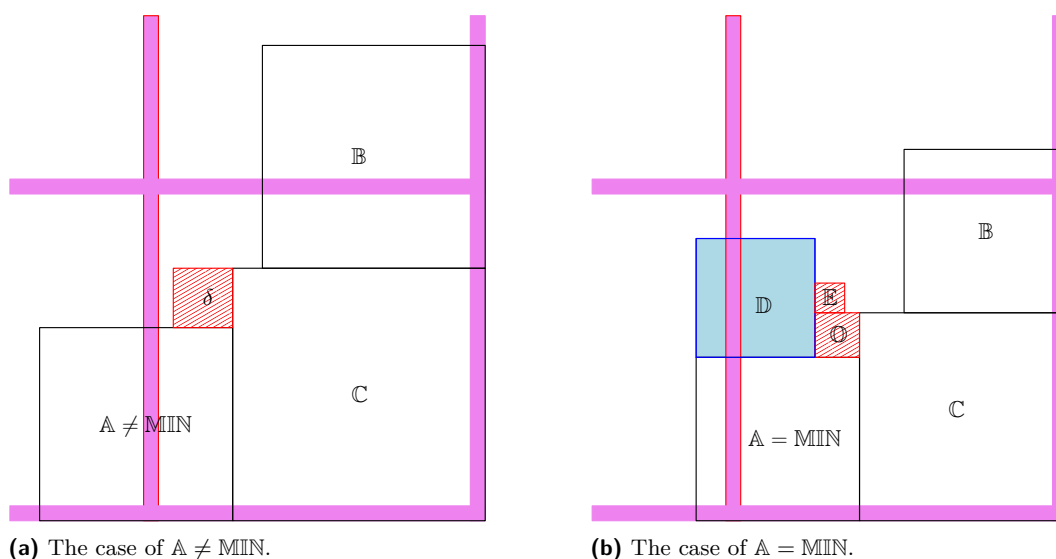


Figure 5.1 An illustration of the proof of Lemma 5.14.

Proof. Suppose first that $A \neq \text{MIN}$. See Fig. 5.1(a) for an illustration. Consider the δ cells adjacent to the left side of C and above A . These cells are covered by squares that do not extend above C . Thus all these squares have size at most δ and do not intersect any important column due to the condition $C + \delta \leq 22$; they also do not intersect the low row, as the low row is above C . It follows that all these squares are orphans. However, by Corollaries 5.7 and 5.10 we already have a different orphan of size at least 2 adjacent to MIN (it is different from the squares above A as $\text{MIN} \neq A$); this implies that all these squares are in fact only a single orphan of size 1. Thus $\delta = 1$, we have an orphan of size 1 adjacent to C and the lemma follows.

Suppose now that $A = \text{MIN}$. See Fig. 5.1(b) for an illustration. We know that there are exactly two squares on top of MIN , namely one orphan $\mathbb{O} \in \{2, 3, 4\}$ and one minor square \mathbb{D} of size $\text{MIN} - \mathbb{O} \geq 11 - 4 = 7$. Since $C \leq 17$, we have $\delta \leq 17 - \text{MIN} \leq 6$. It follows that \mathbb{D} cannot be adjacent to C . Thus \mathbb{O} is adjacent to C and \mathbb{D} is to the left of \mathbb{O} . We claim first that $\delta = \mathbb{O}$ and thus $C = \text{MIN} + \mathbb{O}$. Indeed, if $\delta > \mathbb{O}$ then the cells just above \mathbb{O} must be covered by at least two orphans, which leads to a contradiction.

Finally, we claim that $B = C + \mathbb{O}$, except for one special case described in the lemma. Indeed, if $B < C + \mathbb{O}$, there is a corridor above \mathbb{O} between \mathbb{D} and $B \geq 13$ of nonzero width $C + \mathbb{O} - B = \text{MIN} + 2 \cdot \mathbb{O} - B \leq 12 + 2 \cdot 4 - 13 = 7$ (the corridor may be wider than \mathbb{O} if $B < C$). If the width of the corridor is 7, the previous calculation has to be tight, which leads to $\text{MIN} = 12$, $\mathbb{O} = 4$, $C = \text{MIN} + \mathbb{O} = 16$, and $B = 13$, i.e., exactly the special case in the lemma. In the remaining case the corridor has width at most 6; this leads to a contradiction as follows: Consider the square \mathbb{E} adjacent to \mathbb{O} and \mathbb{D} . It reaches at most the row $A + \mathbb{O} + 6 \leq 22$, so it does not reach the low row. It also does not intersect any important column, so \mathbb{E} is an orphan. It cannot have size 1, as it is in a corner-like position which we excluded, see Fig. 3.1(2). It also cannot have size at least 2, as we would have two orphans of size at least 2. So we have a contradiction. ◀

► **Lemma 5.14.** *If $\text{MIN} = 12$ then $C \geq 71$. If $\text{MIN} = 11$ then $C \geq 72$.*

Proof. We start with several observations.

First, suppose we prove that there is a single corner of size at most 17. Then this corner has size at least 14 and altogether we have $C \geq 14 + 18 + 19 + 20 = 71$. Thus we are almost done, except that we need to improve the bound by 1 if $\text{MIN} = 11$.

Second, if we have a corner $\mathbb{C} \leq 17$ with a key neighbor $\mathbb{A} \neq \text{MIN}$, then in Lemma 5.13 we have $\delta \leq 17 - 13 = 4$ and $\mathbb{C} + \delta \leq 21$; thus the case (1) of Lemma 5.13 applies.

Third, we can have only one corner where the case (1) of Lemma 5.13 applies, as there is only one orphan of size 1.

With these observations in mind, we proceed to examine some cases.

Suppose first that we have a corner \mathbb{C} of size at most 17 with a key neighbor MIN . We examine all possible cases which we list in the form of triples listing MIN , the size of the corner, and the size of the other neighbor. We have $\mathbb{C} \geq 14$, furthermore Lemma 5.13 implies that $\delta = \mathbb{C} - \text{MIN} \leq 4$ and the size of the other neighbor equals $\mathbb{C} + \delta$, except for one special case. Thus for the key neighbor $\text{MIN} = 11$, the only possibilities are (11, 14, 17) and (11, 15, 19), while for $\text{MIN} = 12$, the possibilities are (12, 14, 16), and (12, 15, 18), (12, 16, 20), and (12, 16, 13) (the special case in Lemma 5.13).

(11, 15, 19), (12, 16, 20), or (12, 16, 13): Then we have an orphan of size $\delta = 4$, so no other corner of size at most 17 exists. Thus $C \geq 15 + 18 + 19 + 20 = 72$ and the lemma holds.

(12, 14, 16): Then there is no other corner of size at most 17, as there is no consecutive pair of sizes for the corner and its key neighbor. Thus $C \geq 14 + 18 + 19 + 20 = 71$ and the lemma holds. (Note that $\text{MIN} = 12$ in this case.)

(12, 15, 18): If there is no other corner of size at most 17, we are done. The only remaining possibility is a corner 17 with a key neighbor 16. Then $C \geq 15 + 17 + 19 + 20 = 71$ and the lemma holds as well. (Note that $\text{MIN} = 12$ in this case, too.)

(11, 14, 17): We have two somewhat subtle subcases. If there is another corner of size at most 17, the only possibility is a corner 16 with a key neighbor 15 and the other neighbor 13. This exhausts all edge squares up to size 17. Now Lemma 5.13 implies that the other corners have size at least 21. Indeed, for a corner of size at most 20 with a key neighbor of size at least 18 we have $\delta \leq 2$ and thus the case (1) of Lemma 5.13 would apply; this would imply another orphan of size 1, which is impossible. It follows that $C \geq 14 + 16 + 21 + 22 = 73$ and the lemma holds.

In the second subcase we have no other corner of size at most 17. We already know that $C \geq 14 + 18 + 19 + 20 = 71$ and we need to improve the bound by 1. To do this, consider the edge of MIN . It has squares 11 and 14, thus its other corner has size at least $70 - 11 - 14 - 24 = 21$. Thus $C \geq 14 + 18 + 19 + 21 = 72$ and the lemma holds.

In the remaining case, we have no corner of size at most 17 with a key neighbor MIN . From the observations at the beginning of the proof, it follows that there is at most a single corner $\mathbb{C} \leq 17$. Thus we already know that $C \geq 14 + 18 + 19 + 20 = 71$ and we need to improve the bound by 1 if $\text{MIN} = 11$. This means that it is sufficient to exclude the case where the corner sizes are exactly 14, 18, 19, and 20.

For a contradiction, assume that $\text{MIN} = 11$ and the corners are $\mathbb{C} = 14, 18, 19$, and 20. Then $\mathbb{C} = 14$ has a key neighbor 13 and another neighbor of size at most 15; this other neighbor is either 15 or $\text{MIN} = 11$, as no other small edge square is available. Now consider the key neighbors of the two corners 18 and 19. One of them may have size at most 15, namely it may be the one of 15 and $\text{MIN} = 11$ not adjacent to \mathbb{C} . The other key neighbor of

28:16 No Tiling of the 70×70 Square with Consecutive Squares

18 or 19 has size at least 16. Thus for this corner $\delta \leq 19 - 16 = 3$ and case (1) of Lemma 5.13 applies. This implies the existence of a second orphan of size 1, in addition to the one adjacent to C . This is the final contradiction. ◀

► **Theorem 5.15.** *There exists no tiling of the 70×70 board with squares of sizes $1, 2, \dots, 24$.*

Proof. Suppose we have a perfect tiling.

By Propositions 4.2 and 5.2 we have $\text{MIN} \in \{11, 12\}$.

If $\text{MIN} = 11$ then Corollary 5.6 and Lemma 5.14 imply that $E + C \geq 209 + 72 = 281$.



If $\text{MIN} = 12$ then Corollary 5.6 and Lemma 5.14 imply that $E + C \geq 210 + 71 = 281$.

In both cases we get a contradiction with $E + C = 280$ from Observation 5.1. ◀

References

- 1 Stuart Anderson. Mrs Perkins's quilt, 2020. Accessed: April 10, 2024. URL: <http://www.squaring.net/quilts/mrs-perkins-quilts.html>.
- 2 James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Commun. ACM*, 18(11):651–656, 1975. doi:10.1145/361219.361224.
- 3 Martin Gardner. Mathematical games: The problem of Mrs. Perkins' quilt. *Scientific American*, 215(3):264–272, 1966.
- 4 Richard E. Korf. Optimal rectangle packing: New results. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 142–149. AAAI, 2004. URL: <http://www.aaai.org/Library/ICAPS/2004/icaps04-019.php>.
- 5 Richard E. Korf, Michael D. Moffitt, and Martha E. Pollack. Optimal rectangle packing. *Ann. Oper. Res.*, 179(1):261–295, 2010. doi:10.1007/S10479-008-0463-6.
- 6 Brian Lavery and Thomas Murphy. Optimal rectangle packing for the 70 square. *Recreational Mathematics Magazine*, 5(9):5–47, 2018.
- 7 George Neville Watson. The problem of the square pyramid. *Messenger of Mathematics*, 48:1–22, 1918.

Achieving the Highest Possible Elo Rating

Rikhav Shah   

University of California, Berkeley, CA, USA

Abstract

Elo rating systems measure the approximate skill of each competitor in a game or sport. A competitor's rating increases when they win and decreases when they lose. Increasing one's rating can be difficult work; one must hone their skills and consistently beat the competition. Alternatively, with enough money you can rig the outcome of games to boost your rating. This paper poses a natural question for Elo rating systems: say you manage to get together n people (including yourself) and acquire enough money to rig k games. How high can you get your rating, asymptotically in k ? In this setting, the people you gathered aren't very interested in the game, and will only play if you pay them to. This paper resolves the question for $n = 2$ up to constant additive error, and provides close upper and lower bounds for all other n , including for n growing arbitrarily with k . There is a phase transition at $n = k^{1/3}$: there is a huge increase in the highest possible Elo rating from $n = 2$ to $n = k^{1/3}$, but (depending on the particular Elo system used) little-to-no increase for any higher n . Past the transition point $n > k^{1/3}$, the highest possible Elo is at least $\Theta(k^{1/3})$. The corresponding upper bound depends on the particular system used, but for the standard Elo system, is $\Theta(k^{1/3} \log(k)^{1/3})$.

2012 ACM Subject Classification Mathematics of computing \rightarrow Combinatorics; Mathematics of computing \rightarrow Discrete mathematics

Keywords and phrases Elo, rating system, monotonic invariant, Euler's method, mass movement

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.29

Related Version *Full Version*: <https://arxiv.org/abs/2203.14567>

1 Introduction

The Elo Rating system was proposed by Arpad Elo in the mid 20th century to estimate the relative skill of chess players [2]. It was quickly adopted by the international chess community, and in the decades since has seen adoption in many competitive contexts. This paper considers a simple combinatorial question about the Elo system. To the author's knowledge, this is the first time this question has been posed in print: **given n players starting with equal rating, what is the highest a player could be rated after a total of k games are played?**

We begin with a definition of the system then provide its motivation. Each player is given some "rating" value (measured in "points" or simply "Elo"), which updates as they play games. These rating points are somewhat analogous to poker chips: when player A and player B play a game, they each place some of their rating points into a pot. In the case of a draw, the players split the pot evenly. If one player wins, they take the entire pot. The heart of the Elo system is dictating how many points each player must ante up. To do so, each implementation of the system specifies a "pot function" σ satisfying

1. σ is non-negative and monotonically increasing, and
 2. $\sigma(z) + \sigma(-z) = 1$ for all $z \in \mathbb{R}$.
- (1)

Let r_A and r_B be the ratings of players A and B respectively. When players A and B play, the number of points they ante up are $K \cdot \sigma(r_A - r_B)$ and $K \cdot \sigma(r_B - r_A)$ respectively, for a total pot size of K . Players are allowed to go into debt if they don't have the required



© Rikhav Shah;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 29; pp. 29:1–29:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

29:2 Achieving the Highest Possible Elo Rating

points, i.e. negative ratings are perfectly fine. The value of K itself is a parameter of the system. The resulting rating updates for different outcomes of a game between A and B are as follows:

	A wins	A and B draw	B wins
$r'_A =$	$r_A + K \cdot \sigma(r_B - r_A)$	$r_A + \frac{K}{2} \cdot (\sigma(r_B - r_A) - \sigma(r_A - r_B))$	$r_A - K \cdot \sigma(r_A - r_B)$
$r'_B =$	$r_B - K \cdot \sigma(r_B - r_A)$	$r_B + \frac{K}{2} \cdot (\sigma(r_A - r_B) - \sigma(r_B - r_A))$	$r_B + K \cdot \sigma(r_A - r_B)$

(2)

The motivation for this system comes from thinking of the outcome of a game as a random variable. For some symmetric random variable η , the event $r_A - r_B + \eta > 0$ is recorded as a victory for A , $r_A - r_B + \eta < 0$ as a victory for B , and $r_A - r_B + \eta = 0$ as a draw. Set

$$\sigma(z) = \Pr(\eta < z) + \frac{1}{2} \Pr(\eta = z) \quad (3)$$

so that σ is a kind of “symmetrized” cumulative distribution function of η (which coincides with the usual cumulative distribution function when η has no atoms). Given η , setting σ to satisfy (3) guarantees it satisfies (1). Conversely, given any σ satisfying (1), one can define a symmetric random variable η satisfying (3), if some of the mass of η is allowed to be at infinity (i.e. $\lim_{z \rightarrow \infty} \Pr(\eta \leq z)$ need not be 1 and $\lim_{z \rightarrow -\infty} \Pr(\eta < z)$ need not be 0). Under this probabilistic model, observe that $\mathbb{E}(r'_A) = r_A$ and $\mathbb{E}(r'_B) = r_B$. So one gains points for performing “better than expected” and loses points for performing “worse than expected”. One additional natural assumption is that η has a finite expectation, though it is not required.

Given some real-world game, one should attempt to pick η so that these estimated probabilities match the empirical win-loss rates observed. The proposal by Elo in the 1960s was to take η to be Gaussian, citing the ubiquity of the normal distribution in nature [1]. However, the community soon decided a logistic random variable was more suitable, leading to the pot function of $\sigma(z) = \frac{1}{1+e^{-cz}}$ for some constant c [2]. The International Chess Federation (FIDE) has long used $c = \log(10)/400 \approx 5.76 \times 10^{-3}$ [4, Chapter B02], though recent analysis suggests $c = \frac{5}{6} \cdot \log(10)/400 \approx 4.80 \times 10^{-3}$ reflects real-world chess data much better [6]. Depending on various factors, FIDE uses K in the range of 10 to 40. The range of ratings exhibited by human players is roughly 0 to 3000 [4, 3]. This paper considers a generic pot function σ , and applies the results to several specific families of pot functions listed in Table 1.

There are additional complications in real-world implementations of Elo. For legibility and practicality, fractional and negative rating points are avoided by scaling and shifting points up and rounding to the nearest integer, and by imposing an artificial floor on possible ratings (by gifting a player points if they would otherwise dip below the floor). The total size of the pot K may also vary depending on various factors, such as how many games each player has played before. For example, K may be large for a new player to facilitate faster convergence of their rating to their true skill level, and may decrease over time to reduce arbitrary fluctuations for experienced players. Sometimes rating updates are batched. That is, one accumulates their pot winnings and losses over several games, and updates their rating once at the end. Often times all the games played at a single tournament are batched together in that manner. All these details and even more complications are outlined thoroughly by different organizations implementing Elo; one may read about them in, for example, the FIDE Handbook [4].

1.1 Setting and results

This paper considers n players starting with equal ratings. Players' ratings update on a game-by-game basis according to (2) with $K = 1$ kept fixed. In particular, both fractional and negative points are possible and rating updates are not batched. Since the dynamics depend only on the difference between ratings, we can take everyone's initial rating to be 0 without loss of generality. Asymptotically in k , we seek the highest one of the n players may be rated after a total of k games are played amongst all of them. This question is interesting both for fixed n , and for n allowed to grow with k . We find a phase transition at $n = k^{1/3}$: there is a huge increase in the highest possible Elo rating from $n = 2$ to $n = k^{1/3}$, but (depending on σ) little-to-no increase for any higher n . This paper is organized into four main sections.

Section 2: the highest Elo problem for $n = 2$ is resolved up to constant additive error.

Section 3: a lower bound is provided for each n by finding a family of strategies which achieve a highest rating of $\Theta(\min(n, k^{1/3}))$ for any pot function. Note the bound does not improve past $n = \Omega(k^{1/3})$.

Section 4: an upper bound on the highest possible rating for each n is provided with a mild natural assumption on σ .

Section 5: open questions and a surprising connection to the maximum overhang problem are discussed.

The gap between the upper and lower bounds depends on which σ is used. In particular, the key quantity is the left-tail behavior of σ . A quickly decaying tail corresponds both to a small asymptotic rate in the case of $n = 2$, and to nearly matching upper and lower bounds for general n . Our main result is stated in terms of the following function, which quantifies the rate of decay of the left-tail of σ :

$$f(x) = \int_0^x \frac{1}{\sigma(-\tau)} d\tau \quad (4)$$

Note that faster the left tail of σ decays to 0, the faster f diverges to infinity. Note that non-negativity and monotonicity of σ means that f is increasing and convex. This main theorem summarizes the results from each section.

► **Theorem 1.** *Let $R(n, k)$ be the highest possible Elo rating achievable with k games and n players starting with 0 Elo points. Fix any pot function σ and let f be as in (4). Then*

$$\frac{1}{2}f^{-1}(2k) \leq R(2, k) \leq \frac{1}{2}f^{-1}(2k - 2) + \frac{5}{2}.$$

Now assume that $\sup_z z \sigma(-z) < \infty$. Fix $n = h(k)$. If $h(k) = \Omega(k^{1/3})$, then there exists constants C_1, C_2 such that for sufficiently large k , one has

$$C_1 k^{1/3} \leq R(h(k), k) \leq C_2 k^{1/3} f^{-1}(k)^{1/3}.$$

Furthermore, if $h(k) = o(k^{1/3})$ then

$$C_1 h(k) \leq R(h(k), k) \leq C_2 h(k) f^{-1}(k/h(k)).$$

Proof. $R(2, k)$ handled in Section 2. The lower bounds for $R(h(k), k)$ are in Section 3 and the upper bounds are in Section 4. ◀

► **Remark 2.** The results hold even if the players all start with distinct ratings, provided that all the initial ratings fall into some bounded interval independent of n and k , and that there's some point around which the initial ratings are symmetric. That is, there exists some value r_0 such that for every player rated r there's one rated $2r_0 - r$.

29:4 Achieving the Highest Possible Elo Rating

Table 1 lists the value of f^{-1} for several natural families of σ and names the corresponding η where appropriate. For the logistic pot function, f^{-1} is logarithmic and Theorem 1 implies

$$R(2, k) = \frac{1}{2c} \log(2k) + O(1) \quad \text{and} \quad R(n, k) = \tilde{\Theta}\left(\min\left(n, k^{1/3}\right)\right)$$

where $\tilde{\Theta}$ suppresses log factors. For any pot function that eventually hits 0 (for example, the “uniform” pot function $\max(0, \min(1, cz + 1/2))$), let x be such that $\sigma(-x) = 0$ and note f has a vertical asymptote at x . This means f^{-1} is bounded, so Theorem 1 implies

$$R(n, k) = \Theta\left(\min\left(n, k^{1/3}\right)\right)$$

where Θ only suppresses constant factors. At the other extreme, σ need not converge to 0 at all. In this case, $f^{-1}(k) = \Theta(k)$. The upper bounds in Theorem 1 no longer hold, but just by noticing the total pot size for each game is bounded, one sees $R(n, k) = \Theta(f^{-1}(k)) = \Theta(k)$ anyway.

■ **Table 1** The value of f^{-1} for some selected families of pot functions. Most of the families are parameterized by some constant c , which correlates with the slope of each σ at 0. The logistic pot function in the top row is the usual one used by real-world implementations of Elo.

η	$\sigma(z)$	$f^{-1}(k)$
Logistic	$\frac{1}{1+e^{-cz}}$	$\frac{1}{c} \log(ck) \cdot (1 + O(1/k))$
–	$\frac{1}{2} \frac{cz}{(1+ cz ^p)^{1/p}} + \frac{1}{2}$	$\left(\frac{1}{2c^p} \cdot \frac{p+1}{p}\right)^{\frac{1}{p+1}} \cdot k^{\frac{1}{p+1}} \cdot (1 + o(1))$
Gaussian	$\frac{1}{2} \operatorname{erf}(cz/\sqrt{2}) + \frac{1}{2}$	$\frac{1}{c} \sqrt{\log k} \cdot (1 + o(1))$
Uniform	$\min\left(1, \max\left(0, cz + \frac{1}{2}\right)\right)$	$\frac{1}{2c} - \frac{e^{-ck}}{2c}$
Cauchy	$\frac{1}{\pi} \arctan(cz) + \frac{1}{2}$	$\sqrt{2/c\pi} \cdot k^{1/2} + O(1)$
–	$\frac{c}{2} \operatorname{sign}(x) + \frac{1}{2}$	$\frac{2}{1-c} \cdot k$

2 Case of $n = 2$

When there are only two players, H and L , as their ratings r_H and r_L grow further apart, the fewer rating points the higher player H can earn from the lower L each time. If $\sigma(-x) = 0$ for some x , the difference in the ratings of H and L cannot exceed $x + 2$. To see this, observe that a game can only increase the value of $r_H - r_L$ if $|r_H - r_L| < x$, and furthermore, r_H and r_L can each change by at most one point in each game. Since $r_H + r_L = 0$ is invariant, this immediately implies

$$r_H \leq \frac{x}{2} + 1$$

independently of how many games are played. On the other hand, if $\sigma(-z) > 0$ for all z , then $r_H - r_L$ will diverge. To see this first note monotonicity of σ means σ is bounded away from 0 on each compact interval. Then, note H beating L will change the value of $r_H - r_L$ by applying the map $g : z \mapsto z + 2\sigma(-z)$. The orbit can only converge if $\sigma(-g^j(z))$ converges to 0, which cannot happen for bounded z by assumption. In the former case, we may still ask how quickly the bound $r_H \leq \frac{z}{2} + 1$ is achieved, and in the latter case we ask how quickly the orbit diverges. Intuitively, the faster $\sigma(-z)$ decays to 0 as $z \rightarrow \infty$, the slower the rate.

Indeed, as the next theorem makes explicit, there is a simple expression for the rate in terms of the f defined in (4). Note f is only well defined for $\{x : \sigma(-x) > 0\}$. Nevertheless, it's continuous and strictly increasing, and its range is all of \mathbb{R} . This implies for any σ that it has a well defined inverse f^{-1} defined on all of \mathbb{R} . The following theorem takes advantage of that fact to unify the analysis for every σ .

► **Theorem 3.** *Fix any pot function σ . Let $r(t)$ be the highest possible rating after t games with two players. Then*

$$\frac{f^{-1}(2t)}{2} \leq r(t) \leq \frac{f^{-1}(2t-2)}{2} + \frac{5}{2}.$$

Proof. Since the ratings of the two players sum to zero, it suffices to keep track of just the higher rated player. Call the players H and L for “higher” and “lower”. Let $r(t)$ be the rating of H after t games. Then $-r(t)$ is the rating of L after t games. Under the assumption that H wins every game, we have the simple recurrence

$$\begin{aligned} r(0) &= 0 & r(t+1) &= r(t) + \sigma(-r(t) - r(t)) \\ & & &= r(t) + \sigma(-2r(t)). \end{aligned}$$

This can be viewed as running Euler's method on the differential equation

$$y(0) = 0 \quad y' = \sigma(-2y)$$

with a step size of 1. Since y' is positive and monotonically decreasing in y , the Euler approximation upper bounds the exact solution. That is,

$$r(t) \geq y(t).$$

Using separate and integrate, one sees the exact solution to the differential equation is

$$y(t) = \frac{1}{2}f^{-1}(2t).$$

This establishes the lower bound. For the upper bound, we cannot assume that the optimal strategy is for H to win every game. In particular, the function $z \mapsto z + \sigma(-2z)$ need not be monotone, so we cannot exclude the possibility one can achieve a higher rating by first losing a game and “slingshotting” to a higher rating exploiting the fact that $\sigma(-2y)$ is larger for smaller y . Construct a sequence x_j based on the recurrence

$$x_0 = 1 \quad x_{j+1} = x_j + \sigma(-2x_j + 2). \tag{5}$$

This sequence defines a partition of the positive real line, $(0, x_0], (x_0, x_1], (x_1, x_2], (x_2, x_3] \dots$. Again we take $r(t)$ to be the rating of H after t games, but instead of a strict recurrence we only have an inequality $r(t+1) \leq r(t) + \sigma(-2r(t))$, which is tight only when H wins game $t+1$. We now relate $r(t)$ to x_j . If $r(t) \leq 0$, then $r(t+1) \leq r(t) + 1 \leq 1 = x_0$. If $r(t) \in (0, x_0]$, then $r(t+1) \leq r(t) + \sigma(-2r(t)) \leq x_0 + \sigma(-2x_0 + 2) = x_1$. If $r(t) \in (x_{j-1}, x_j]$ for some j , then

$$\begin{aligned} r(t+1) &\leq r(t) + \sigma(-2r(t)) \\ &\leq x_j + \sigma(-2x_{j-1}) \\ &\leq x_j + \sigma(-2x_j + 2) \\ &= x_{j+1}. \end{aligned}$$

29:6 Achieving the Highest Possible Elo Rating

In each case, we see that

$$r(t) \leq x_j \implies r(t+1) \leq x_{j+1}.$$

Since $r(0) = 0 < 1 = x_0$, we have $r(t) \leq x_t$ regardless of the sequence of wins and losses. Now similar to before, x_t can be viewed as the result of running Euler's method with a step size of 1 on the differential equation

$$y' = \sigma(-2y + 2). \tag{6}$$

Using separate and integrate, see that the solution to the differential equation is

$$y(t) = \frac{1}{2}f^{-1}(2t) + 1.$$

Again the Euler approximation is an upper bound to the exact solution. That is,

$$x_t \geq y(t). \tag{7}$$

Suprisingly, this lower bound leads us to an upper bound. We first plug the recurrence (5) back into itself to express x_t as a sum, then use (7) with monotonicity of σ .

$$x_t = x_0 + \sum_{j=0}^{t-1} \sigma(-2x_j + 2) \leq x_0 + \sum_{j=0}^{t-1} \sigma(-2y(j) + 2). \tag{8}$$

The summand is the same as the right hand side of the differential equation (6). Then the sum is Reimann sum of step size 1. But since the summand is monotone, we can bound the Reimann sum by the integral.

$$\sum_{j=0}^{t-1} \sigma(-2y(j) + 2) = \sum_{j=0}^{t-1} y'(j) \leq y'(0) + \int_0^{t-1} y'(s) ds = y'(0) + y(t-1). \tag{9}$$

Note $y'(0) = 1/2$ and $x_0 = 1$, so combining (8) and (9) gives the final upper bound of

$$x_t \leq 1 + \frac{1}{2} + y(t-1) = 1 + \frac{1}{2} + 1 + \frac{1}{2}f^{-1}(2t-2). \quad \blacktriangleleft$$

3 Lower bound for general n

This section describes two strategies for any number of players and games. The first strategy does not depend at all on the pot function used, but requires that everyone's initial ratings be exactly equal. The second strategy has a small dependence on the pot function used, but works for any symmetric list of initial ratings. Both strategies produce higher ratings for larger n , up to $n = \Theta(k^{1/3})$, at which point the asymptotic highest rating in k is $\Theta(k^{1/3})$. The first strategy is very simple: **pick any pair of players with equal rating and have one beat the other. Repeat until all players have a distinct rating or k games have been played.** This strategy is guaranteed to produce a player of either very high or very low rating. If it produces a player of very low rating, simply re-do the strategy picking the same sequence of pairs of players but have the opposite player win. Since game outcomes are symmetric, this will produce a player of high rating instead.

► **Theorem 4.** *The aforementioned strategy achieves a highest rating of*

$$\min\left(\frac{k^{1/3}}{2}, \frac{n}{4}\right)$$

when all players start with 0 rating points. In particular, the rating achieved is $\Omega(k^{1/3})$ for $n = \Omega(k^{1/3})$.

Proof. Let $\mathbf{r}(j)$ be the multiset of player ratings after j games are played and $|\mathbf{r}(j)|$ its element absolute value. We claim that every entry of $\mathbf{r}(j)$ is a half-integer. To see this, note $\sigma(0) = 1/2$ for every pot function so player ratings change in increments of $1/2$ and all start at 0. We next claim

$$\sum_{r \in \mathbf{r}(j)} r^2 = j/2. \quad (10)$$

We argue inductively. The claim is clearly true initially when all ratings are 0. When two players rated a points play, the increase in the functional is

$$\Delta = (a + 1/2)^2 + (a - 1/2)^2 - a^2 - a^2 = 1/2.$$

Suppose the strategy terminates when k games have been played. Then

$$\frac{k}{2} = \sum_{r \in \mathbf{r}(k)} r^2 \leq n \max |\mathbf{r}(k)|^2 \implies \max |\mathbf{r}(k)| \geq \sqrt{k/2n}.$$

Now suppose the strategy terminates when all players have distinct ratings. Then since all ratings are half integers, that means $\max |\mathbf{r}(j)| \geq n/4$. Without a guarantee on which way the strategy terminates, we get the worst of both bounds. So the maximum rating is at least $\min(\sqrt{k/2n}, n/4)$. If $n \leq 2k^{1/3}$, the minimum is $n/4$. If $n > 2k^{1/3}$, we can rerun the strategy ignoring all but the first $2k^{1/3}$ players, giving a bound of exactly $k^{1/3}/2$. ◀

► **Remark 5.** Theorem 4 is unfortunately very brittle. If the initial ratings are perturbed slightly, we can no longer assume all ratings are half-integers, nor can we often expect players to have equal rating. For

$$\sigma(x) = \frac{1}{2} \text{sign}(x) + \frac{1}{2} = \begin{cases} 1 & x > 0 \\ 1/2 & x = 0 \\ 0 & x < 0 \end{cases},$$

this is indeed the best we can do. If all players have slightly different ratings, no Elo can be transferred between them. However, there's a robust version of the strategy that works for every other σ : first fix some $\delta > 0$ for which $\sigma(-\delta) > 0$. **Pick any pair of players whose ratings are within δ of each other and have the higher rated player beat the lower rated player. Repeat until no two players are within δ rating points or until k games have been played.** Note for $\delta = 0$, one recovers the original strategy. As before, we may have to flip everyone's rating to ensure we end with a very high rating, as opposed to a very low rating. We need not assume that the initial ratings are all 0 anymore, but we do need to allow ourselves this possible reflection.

► **Theorem 6.** *Fix any $\vec{r} = (r_1, \dots, r_n)$. The aforementioned strategy achieves a highest rating of*

$$\min\left(\delta^{1/3} \sigma(-\delta)^{1/3} \cdot k^{1/3}, \frac{\delta}{2} \cdot n\right)$$

for at least one choice of the initial ratings $\mathbf{r}(0) = \vec{r}$ or $\mathbf{r}(0) = -\vec{r}$.

29:8 Achieving the Highest Possible Elo Rating

Proof. The proof is very similar to the proof of Theorem 4. Again we consider the functional $\sum_{r \in \mathbf{r}(j)} r^2$. In a game between players rated $a < b$, the change in the functional is

$$\begin{aligned} \Delta &= (b + \sigma(a - b))^2 + (a - \sigma(a - b))^2 - b^2 - a^2 \\ &= 2b\sigma(a - b) - 2a\sigma(a - b) + 2\sigma(a - b)^2 \\ &= 2(b - a)\sigma(a - b) + 2\sigma(a - b)^2 \\ &\geq 2\sigma(-\delta)^2. \end{aligned}$$

Suppose the strategy terminates when k games have been played. Then

$$2\sigma(-\delta) \cdot k \leq \sum_{r \in \mathbf{r}(k)} r^2 \leq n \max |\mathbf{r}(k)|^2 \implies \max |\mathbf{r}(k)| \geq \sqrt{2\sigma(-\delta)k/n}.$$

Now suppose the strategy terminates when no two players are within δ rating points. Since there are a total of n players, that means $\max |\mathbf{r}(j)| \geq \delta n/2$. As before, without a guarantee on which way the strategy terminates, we get the worst of both bounds. When

$$n \leq \frac{2\sigma(-\delta)^{1/3}}{\delta^{2/3}} \cdot k^{1/3},$$

the smaller bound is $\delta n/2$. When n is larger, one simply ignores the excess players, as before. \blacktriangleleft

► **Remark 7.** The theorem is best applied with the vector of initial ratings is symmetric, i.e. \vec{r} is a permutation of $-\vec{r}$. In fact, by shifting all the points up or down by a constant amount, they can be symmetric around any (constant valued) point, i.e. \vec{r} is a permutation of $(r_0, \dots, r_0) - \vec{r}$.

► **Remark 8.** This bound is stronger the heavier the tail of σ . Consider, for instance $\sigma(z) = \frac{1}{1+e^{-cz}}$ as $c \rightarrow \infty$, or any other family of σ approaching the pathological $\frac{1}{2}\text{sign}(x) + \frac{1}{2}$. The bound of Theorem 6 becomes weaker and weaker. This contrasts with the bound of Theorem 4, which is completely independent of the pot function but requires initial ratings be exactly equal.

4 Upper bound for general n

In this section, we wish to show the algorithm presented in Section 3 is nearly optimal. Our strategy is to show that achieving a rating of r requires many games to be played. We start with a relaxation of the setup: instead of considering a discrete sequence of games resulting in a discrete sequence of player ratings, we consider a continuous path in the space of possible player ratings. Call a path $\mathbf{r} : [0, c] \rightarrow \mathbb{R}^n$ **valid** if there exists a finite sequence $0 = t_0 < \dots < t_k = c$ such that for each $j \in [k]$ there exists $w, \ell \in [n]$ with

$$\mathbf{r}'(t) = \mathbf{e}_w - \mathbf{e}_\ell \quad \forall t \in (t_{j-1}, t_j)$$

where $\mathbf{e}_w, \mathbf{e}_\ell$ are elementary basis vectors. For convenience, we write $\mathbf{r}'(t_{j-1}) = \mathbf{e}_w - \mathbf{e}_\ell$ to make \mathbf{r}' right-continuous. In other words, \mathbf{r} has constant speed and consists of line segments (called **edges**) along which only two coordinates change. The points $\mathbf{r}(t_j)$ are referred to as **vertices**. We will use the notation

$$A \rightarrow B \rightarrow C$$

to denote the path with vertices A, B, C . A sequence of games corresponds naturally to a valid path: let $\mathbf{r}(t_j)$ be the list of the players' ratings after game j . The indices “ w ” and “ ℓ ” conveniently correspond to the “winner” and “loser” for each game. In the event of a draw, the initially lower rated player is considered the winner as their rating increases. Note

$$t_{j+1} - t_j = \text{amount of Elo transferred for game } j + 1. \quad (11)$$

Define a weight function

$$W(\mathbf{x}, \mathbf{y}) = \begin{cases} \frac{1}{\sigma(2 - \langle \mathbf{x}, \mathbf{y} \rangle)} & \langle \mathbf{x}, \mathbf{y} \rangle > 0 \\ 0 & \langle \mathbf{x}, \mathbf{y} \rangle \leq 0 \end{cases} \quad (12)$$

and define the length of the path to be

$$\text{len}(\mathbf{r}) = \int_0^c W(\mathbf{r}'(t), \mathbf{r}(t)) dt.$$

Whereas the Euclidean length of a path is simply $c\sqrt{2}$, the value of $\text{len}(\mathbf{r})$ depends highly on the motion of the path. When a valid path comes from a sequence of games, Lemma 9 shows $\text{len}(\mathbf{r})$ cannot be larger than k . The intuition is that $\langle \mathbf{r}'(t), \mathbf{r}(t) \rangle = \mathbf{r}_w(t) - \mathbf{r}_\ell(t)$ records the difference in two players' ratings, so the Euclidean length of the $(j + 1)$ th edge is $\sqrt{2}\sigma(-\langle \mathbf{r}'(t_j), \mathbf{r}(t_j) \rangle)$, which approximately cancels with $W(\mathbf{r}'(t), \mathbf{r}(t))$ resulting in constant contribution.

► **Lemma 9.** *Let \mathbf{r} be a valid path arising from a sequence of k games. Then*

$$\text{len}(\mathbf{r}) \leq k.$$

Proof. It suffices to show for each edge $\mathbf{r}(t_j) \rightarrow \mathbf{r}(t_{j+1})$ that

$$\text{len}(\mathbf{r}(t_j) \rightarrow \mathbf{r}(t_{j+1})) \leq 1$$

since summing over all edges would produce the final result. Fix any j . Let w, ℓ be the indices such that

$$\mathbf{r}'(t) = \mathbf{e}_w - \mathbf{e}_\ell \quad \forall t \in [t_j, t_{j+1}).$$

Let $z = \langle \mathbf{r}'(t_j), \mathbf{r}(t_j) \rangle$ so that by (11) we have

$$\sigma(-z) = t_{j+1} - t_j.$$

Also note $\langle \mathbf{r}'(t), \mathbf{r}(t) \rangle = z + 2(t - t_j) \leq z + 2\sigma(-z) \leq z + 2$. Then

$$\begin{aligned} \text{len}(\mathbf{r}(t_j) \rightarrow \mathbf{r}(t_{j+1})) &= \int_{t_j}^{t_{j+1}} W(\mathbf{r}'(t), \mathbf{r}(t)) dt \\ &\leq \int_{t_j}^{t_{j+1}} \frac{1}{\sigma(2 - \langle \mathbf{r}'(t), \mathbf{r}(t) \rangle)} dt \\ &\leq \int_{t_j}^{t_{j+1}} \frac{1}{\sigma(-z)} dt \\ &= 1. \end{aligned} \quad \blacktriangleleft$$

29:10 Achieving the Highest Possible Elo Rating

Lemma 9 means that instead of bounding k , we can bound $\text{len}(\mathbf{r})$. Directly doing so for any valid path is difficult, so we introduce two additional notions that allow us to make some additional restrictions on \mathbf{r} . Call a valid path **ordered** if

$$\mathbf{r}_1(t) \geq \cdots \geq \mathbf{r}_n(t) \quad \forall t \in [0, c].$$

Each valid path corresponds to an ordered valid path of the same length by doing the following: each time \mathbf{r} intersects a hyperplane of the form $x_w = x_\ell$, reflect the remaining part of the path across it. This makes it possible to refer unambiguously to the p th highest rated player. It also means $\langle \mathbf{r}', \mathbf{r} \rangle$ does not change sign on each edge. The last notion we introduce is that of an **upset**. This is when a lower rated player beats or draws with a higher rated player. Stated in terms of valid paths, an upset is an edge for which

$$\langle \mathbf{r}', \mathbf{r} \rangle = \mathbf{r}_w - \mathbf{r}_\ell < 0$$

at the starting vertex. We essentially show an optimal strategy, when converted to an ordered valid path, does not make use of upsets. The precise statement is given in Lemma 10. The intuition here is that upsets bring the players' ratings closer together, whereas in order to make one rating large you need the ratings to be spread out. The proof strategy is to take any valid path with upsets and convert it to one achieving a higher maximum rating without upsets. This lemma is the longest and most technical as it involves some casework in describing this conversion procedure.

► **Lemma 10.** *For each valid ordered path $\mathbf{r} : [0, c] \rightarrow \mathbb{R}^n$ there exists a valid ordered path $\tilde{\mathbf{r}} : [0, \tilde{c}] \rightarrow \mathbb{R}^n$ with*

$$\text{len}(\tilde{\mathbf{r}}) \leq \text{len}(\mathbf{r}) \quad \text{and} \quad \max \tilde{\mathbf{r}}(\tilde{c}) \geq \max \mathbf{r}(c)$$

such that \mathbf{r}' has no upsets.

Proof. Given \mathbf{r} , we start by constructing $\tilde{\mathbf{r}}$ with a slightly different property than the requirement of the theorem. We require $\mathbf{r}(c) = \tilde{\mathbf{r}}(\tilde{c})$ and that all the upsets in $\tilde{\mathbf{r}}$ occur at the end. That is, once $\langle \mathbf{r}', \tilde{\mathbf{r}} \rangle$ becomes negative, it stays negative. Once we have done that, observe that upsets only decrease the value of $\max \tilde{\mathbf{r}}$ so we obtain a larger maximum value by truncating the path just before the upsets.

Fix vector $\mathbf{u} = (1, 2, \dots, n)$. Then each valid path \mathbf{r} has an associated sequence $S(\mathbf{r})$ of integers $S(\mathbf{r})_j = \langle \mathbf{u}, \mathbf{r}'(t_j) \rangle$ with one integer for each edge. Since that path is ordered, each integer is positive if and only if the corresponding edge is an upset. Equip the set of possible values of $S(\mathbf{r})$ with the lexicographic ordering. Our method for constructing $\tilde{\mathbf{r}}$ is iterative, where each iteration strictly decreases $S(\mathbf{r})$ and does not increase $\text{len}(\mathbf{r})$. The strict decreasing of $S(\mathbf{r})$ guarantees that this procedure terminates after at most $n^{\# \text{ edges in } \mathbf{r}}$ steps.

We now describe one iteration. Given a path \mathbf{r} , locate three consecutive vertices $\mathbf{r}(t_j), \mathbf{r}(t_{j+1}), \mathbf{r}(t_{j+2})$ with

$$S(\mathbf{r})_j > 0 \quad \text{and} \quad S(\mathbf{r})_{j+1} < 0.$$

That is, the two connecting edges are an upset followed by a non-upset. We modify the path to obtain $\tilde{\mathbf{r}}$ by either deleting the middle vertex $\mathbf{r}(t_{j+1})$ from the path and adding an edge directly from $\mathbf{r}(t_j)$ to $\mathbf{r}(t_{j+2})$, or replacing the middle vertex with new vertex \mathbf{s} . If $\mathbf{r}(t_{j+1})$ is deleted, $S(\mathbf{r})$ is shortened and therefore decreased. If $\mathbf{r}(t_{j+1})$ is replaced by \mathbf{s} , note that only the j th and $(j+1)$ th entries in $S(\mathbf{r})$ are affected. So requiring that

$$S(\tilde{\mathbf{r}})_j < S(\mathbf{r})_j$$

suffices for $S(\mathbf{r})$ to decrease overall. Note the right hand side is positive, so it suffices for $S(\tilde{\mathbf{r}})_j$ to be negative. We additionally require

$$\text{len}(\tilde{\mathbf{r}}) \leq \text{len}(\mathbf{r}).$$

The modification differs based on how many coordinates change on those two edges. Edge edge modifies two coordinates, so this number can be 2, 3, or 4.

Case of 4 coordinates: this corresponds to the players of the two games being disjoint pairs. Intuitively, the order of the games is irrelevant to the outcome. Set $\mathbf{s} = \mathbf{r}(t_j) - \mathbf{r}(t_{j+1}) + \mathbf{r}(t_{j+2})$. Then $S(\tilde{\mathbf{r}})_j = S(\mathbf{r})_{j+1} < 0$ and $\text{len}(\mathbf{r}) = \text{len}(\tilde{\mathbf{r}})$. **Case of 2 coordinates:** this corresponds to the same two players playing in consecutive games. Let those players be w, ℓ with $w < \ell$. Let Π be the projection onto the w, ℓ coordinates. Then for some $a, b > 0$,

$$\Pi \mathbf{r}(t_j) = \begin{bmatrix} x \\ y \end{bmatrix} \quad \Pi \mathbf{r}(t_{j+1}) = \begin{bmatrix} x - a \\ y + a \end{bmatrix} \quad \Pi \mathbf{r}(t_{j+2}) = \begin{bmatrix} x - a + b \\ y + a - b \end{bmatrix}.$$

Our modification simply deletes $\mathbf{r}(t_{j+1})$, so S certainly decreases. The original length is

$$\begin{aligned} \text{len} \left(\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x - a \\ y + a \end{bmatrix} \rightarrow \begin{bmatrix} x - a + b \\ y + a - b \end{bmatrix} \right) &= \int_0^b \frac{1}{\sigma(2 + (y + a - t) - (x - a + t))} dt \\ &= \int_0^b \frac{1}{\sigma(2 + y - x + 2a - 2t)} dt. \\ &= \int_{-a}^{b-a} \frac{1}{\sigma(2 + y - x - 2t)} dt. \end{aligned}$$

If $a \geq b$, the new length is 0. If $a < b$, then

$$\begin{aligned} \text{len} \left(\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} x - a + b \\ y + a - b \end{bmatrix} \right) &= \int_0^{b-a} \frac{1}{\sigma(2 + (y - t) - (x + t))} dt \\ &= \int_0^{b-a} \frac{1}{\sigma(2 + y - x - 2t)} dt \end{aligned}$$

which is strictly less than the original length since the integrand is the same with a smaller range. **Case of 3 coordinates:** this corresponds to one player playing two different opponents. Let Π be the projection onto those three coordinates in ranked order, i.e.

$$\Pi \mathbf{r}(t_j) = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

for $x \geq y \geq z$. There are six possible sign patterns of $\Pi(\mathbf{r}(t_{j+2}) - \mathbf{r}(t_j))$, each with a different selection of \mathbf{s} .

Subcase (+, -, +): For some $a, c > 0$ we have

$$\Pi \mathbf{r}(t_{j+2}) = \begin{bmatrix} x + a \\ y - a - c \\ z + c \end{bmatrix}.$$

There are three possible values of $\Pi \mathbf{r}(t_{j+1})$ corresponding to the three possible upsets that can occur. No matter which one we observe, the selection of their replacement \mathbf{s} is the same.

$$\Pi \mathbf{r}(t_{j+1}) = \begin{bmatrix} x \\ y - c \\ z + c \end{bmatrix}, \begin{bmatrix} x \\ y - a - c \\ z + a + c \end{bmatrix}, \begin{bmatrix} x - c \\ y \\ z + c \end{bmatrix}, \quad \mathbf{s} = \begin{bmatrix} x + a \\ y - a \\ z \end{bmatrix}$$

29:12 Achieving the Highest Possible Elo Rating

The possible original lengths are as follows.

$$\begin{aligned}
 L_1 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y-c \\ z+c \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y-a-c \\ z+c \end{bmatrix} \right) = \int_0^a \frac{1}{\sigma(2+(y-c-t)-(x+t))} dt \\
 &= \int_0^a \frac{1}{\sigma(2+y-x-c-2t)} dt, \\
 L_2 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y-a-c \\ z+a+c \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y-a-c \\ z+c \end{bmatrix} \right) = \int_0^a \frac{1}{\sigma(2+(z+a+c-t)-(x+t))} dt \\
 &= \int_0^a \frac{1}{\sigma(2+z-x+a+c-2t)} dt, \\
 L_3 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-c \\ y \\ z+c \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y-a-c \\ z+c \end{bmatrix} \right) = \int_0^{a+c} \frac{1}{\sigma(2+(y-t)-(x-c+t))} dt \\
 &= \int_0^{a+c} \frac{1}{\sigma(2+y-x+c-2t)} dt \\
 &= \int_{-c}^a \frac{1}{\sigma(2+y-x-c-2t)} dt
 \end{aligned}$$

The new length is

$$\begin{aligned}
 L_4 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y-a \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y-a-c \\ z+c \end{bmatrix} \right) = \int_0^a \frac{1}{\sigma(2+(y-t)-(x+t))} dt \\
 &= \int_0^a \frac{1}{\sigma(2+y-x-2t)} dt.
 \end{aligned}$$

By monotonicity of σ , we automatically have $L_3 \geq L_1 \geq L_4$. We assume \mathbf{r} is an ordered path, so in the L_2 case we have $z+a+c \leq y-a-c$ giving $L_2 \geq L_4$ showing our modification did not increase the length. Also see that $\mathbf{r}(t_j) \rightarrow \mathbf{s}$ is not an upset so $S(\tilde{\mathbf{r}})_j < 0$.

Subcase $(-, +, -)$: For some $a, c > 0$ we have

$$\text{Pr}(t_{j+2}) = \begin{bmatrix} x-a \\ y+a+c \\ z-c \end{bmatrix}.$$

Again there are three possible values of $\text{Pr}(t_{j+1})$ corresponding to the three possible upsets that can occur, and our selection of their replacement \mathbf{s} is the same regardless.

$$\text{Pr}(t_{j+1}) = \begin{bmatrix} x-a \\ y+a \\ z \end{bmatrix}, \begin{bmatrix} x-a-c \\ y+a+c \\ z \end{bmatrix}, \begin{bmatrix} x-a \\ y \\ z+a \end{bmatrix}, \quad \mathbf{s} = \begin{bmatrix} x \\ y+c \\ z-c \end{bmatrix}.$$

The original lengths are as follows.

$$\begin{aligned}
L_1 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-a \\ y+a \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-a \\ y+a+c \\ z-c \end{bmatrix} \right) = \int_0^c \frac{1}{\sigma(2+(z-t)-(y+a+t))} dt \\
&= \int_0^c \frac{1}{\sigma(2+z-y-a-2t)} dt, \\
L_2 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-a-c \\ y+a+c \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-a \\ y+a+c \\ z-c \end{bmatrix} \right) = \int_0^c \frac{1}{\sigma(2+(z-t)-(x-a-c+t))} dt \\
&= \int_0^c \frac{1}{\sigma(2+z-x+a+c-2t)} dt, \\
L_3 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-a \\ y \\ z+a \end{bmatrix} \rightarrow \begin{bmatrix} x-a \\ y+a+c \\ z-c \end{bmatrix} \right) = \int_0^{a+c} \frac{1}{\sigma(2+(z-t)-(y+t))} dt \\
&= \int_0^{a+c} \frac{1}{\sigma(2+z-y-2t)} dt, \\
&= \int_{-a}^c \frac{1}{\sigma(2+z-y-2a-2t)} dt.
\end{aligned}$$

The new length is

$$\begin{aligned}
L_4 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y+c \\ z-c \end{bmatrix} \rightarrow \begin{bmatrix} x-a \\ y+a+c \\ z-c \end{bmatrix} \right) = \int_0^c \frac{1}{\sigma(2+(z-t)-(y+t))} dt \\
&= \int_0^c \frac{1}{\sigma(2+z-y-2t)} dt.
\end{aligned}$$

Again by monotonicity of σ , we automatically have $L_3 \geq L_1 \geq L_4$. We assume \mathbf{r} is an ordered path, so in the L_2 case we have $y+a+c \leq x-a-c$ giving $L_2 \geq L_4$ as required. Also see that $\mathbf{r}(t_j) \rightarrow \mathbf{s}$ is not an upset so $S(\tilde{\mathbf{r}})_j < 0$.

Subcase (+, +, -): For some $a, b > 0$ we have

$$\Pi\mathbf{r}(t_{j+2}) = \begin{bmatrix} x+a \\ y+b \\ z-a-b \end{bmatrix}.$$

There is only one possible value of $\Pi\mathbf{r}(t_2)$ corresponding to an upset.

$$\Pi\mathbf{r}(t_{j+1}) = \begin{bmatrix} x-b \\ y+b \\ z \end{bmatrix}, \quad \mathbf{s} = \begin{bmatrix} x \\ y+b \\ z-b \end{bmatrix}.$$

29:14 Achieving the Highest Possible Elo Rating

The original length is

$$\begin{aligned}
 L_1 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x-b \\ y+b \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y+b \\ z-a-b \end{bmatrix} \right) \\
 &= \int_0^{a+b} \frac{1}{\sigma(2+(z-t)-(x-b+t))} dt \\
 &= \int_0^{a+b} \frac{1}{\sigma(2+z-x+b-2t)} dt \\
 &= \int_0^b \frac{1}{\sigma(2+z-x+b-2t)} dt + \int_0^a \frac{1}{\sigma(2+z-x-b-2t)} dt
 \end{aligned}$$

and the new length is

$$\begin{aligned}
 L_2 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y+b \\ z-b \end{bmatrix} \rightarrow \begin{bmatrix} x+a \\ y+b \\ z-a-b \end{bmatrix} \right) \\
 &= \int_0^b \frac{1}{\sigma(2+(z-t)-(y+t))} dt + \int_0^a \frac{1}{\sigma(2+(z-b-t)-(x+t))} dt \\
 &= \int_0^b \frac{1}{\sigma(2+z-y-2t)} dt + \int_0^a \frac{1}{\sigma(2+z-x-b-2t)} dt.
 \end{aligned}$$

Again since we assume the path is ordered, we have $x \geq y+b$ so $L_2 \leq L_1$ by monotonicity of σ . Also see that $\mathbf{r}(t_j) \rightarrow \mathbf{s}$ is not an upset so $S(\tilde{\mathbf{r}})_j < 0$.

Subcase (+, -, -): For some $b, c > 0$, we have

$$\Pi\mathbf{r}(t_{j+2}) = \begin{bmatrix} x+b+c \\ y-b \\ z-c \end{bmatrix}.$$

There is only one possible value of $\Pi\mathbf{r}(t_2)$ corresponding to an upset.

$$\Pi\mathbf{r}(t_{j+1}) = \begin{bmatrix} x \\ y-b \\ z+b \end{bmatrix}, \quad \mathbf{s} = \begin{bmatrix} x+b \\ y-b \\ z \end{bmatrix}.$$

The original length is

$$\begin{aligned}
 L_1 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x \\ y-b \\ z+b \end{bmatrix} \rightarrow \begin{bmatrix} x+b+c \\ y-b \\ z-c \end{bmatrix} \right) \\
 &= \int_0^{b+c} \frac{1}{\sigma(2+(z+b-t)-(x+t))} dt \\
 &= \int_0^{b+c} \frac{1}{\sigma(2+z-x+b-2t)} dt \\
 &= \int_0^b \frac{1}{\sigma(2+z-x+b-2t)} dt + \int_0^c \frac{1}{\sigma(2+z-x-b-2t)} dt
 \end{aligned}$$

and the new length is

$$\begin{aligned} L_2 &= \text{len} \left(\begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x+b \\ y-b \\ z \end{bmatrix} \rightarrow \begin{bmatrix} x+b+c \\ y-b \\ z-c \end{bmatrix} \right) \\ &= \int_0^b \frac{1}{\sigma(2+(y-t)-(x+t))} dt + \int_0^c \frac{1}{\sigma(2+(z-t)-(x+b+t))} dt \\ &= \int_0^b \frac{1}{\sigma(2+y-x-2t)} dt + \int_0^c \frac{1}{\sigma(2+z-x-b-2t)} dt. \end{aligned}$$

Since we assume the path is ordered, we have $y-b \geq z$ so $L_2 \leq L_1$ once again. Also see that $\mathbf{r}(t_j) \rightarrow \mathbf{s}$ is not an upset so $S(\tilde{\mathbf{r}})_j < 0$.

Subcase $(-, +, +), (-, -, +)$: For some a, c we have

$$\text{Pr}(t_{j+2}) = \begin{bmatrix} x-a \\ y+a-c \\ z+c \end{bmatrix}.$$

Depending on the sign of $a-c$, there are two possible values of $\text{Pr}(t_{j+1})$. In either case, the selection of \mathbf{s} is the same.

$$\text{Pr}(t_{j+1}) = \begin{bmatrix} x-a \\ y \\ z+a \end{bmatrix}, \begin{bmatrix} x-c \\ y \\ z+c \end{bmatrix} \quad \mathbf{s} = \begin{bmatrix} x-a \\ y+a \\ z \end{bmatrix}.$$

In this case, both new edges are upsets so the new length is 0, but we cannot conclude $S(\tilde{\mathbf{r}})_j < 0$ as before. Instead, let $w_1 < w_2 < w_3$ be the indices of the three relevant coordinates and note

$$S(\mathbf{r})_j = w_1 - w_3 > S(\tilde{\mathbf{r}})_j = w_1 - w_2.$$

Finally note the sign patterns of $(+, +, +)$ and $(-, -, -)$ are not possible since the sum of all ratings is invariant. Since in all possible cases we have a decrease of S without an increase of length, this process terminates in a path that isn't longer and doesn't have any upsets followed by a non-upset. We end by truncating off any upsets at the end of the path. ◀

Lemma 10 implies we can restrict our attention to valid ordered upset-free paths. The length of these paths can be bounded in terms of the following potential function.

► **Definition 11.** Define $\Phi : \mathbb{R}^n \rightarrow \mathbb{R}$ by

$$\Phi(\mathbf{s}) = \|\mathbf{s}\|^2 + \sum_{p=1}^{n-1} f(-2 + \mathbf{s}_p - \mathbf{s}_{p+1})$$

where f is as in (4).

We seek to show Φ grows slowly as games are played, so that Φ of the end point of the path is upper bounded by (a multiple of) the length of the path. Note that the first term $\|\mathbf{s}\|^2$ in this potential function is exactly the $\sum r^2$ expression appearing in the proofs of Theorems 4 and 6. The intuitive idea behind the second term is that when player p beats player $p+1$, the ratings \mathbf{s}_p and \mathbf{s}_{p+1} move $2\sigma(\mathbf{s}_{p+1} - \mathbf{s}_p)$ apart, which by the definition of f means the corresponding f term increases by just a constant amount. Lemma 12 makes this intuition precise.

29:16 Achieving the Highest Possible Elo Rating

► **Lemma 12.** For a valid ordered upset-free path $\mathbf{r} : [0, c] \rightarrow \mathbb{R}^n$ one has

$$\Phi(\mathbf{r}(c)) - \Phi(\mathbf{r}(0)) \leq \left(2 + 2 \sup_z \sigma(2 - z)\right) \cdot \text{len}(\mathbf{r})$$

Proof. By the fundamental theorem of calculus, it suffices to show that

$$\frac{d}{dt} \Phi(\mathbf{r}(t)) \leq \left(2 + 2 \sup_z \sigma(2 - z)\right) W(\mathbf{r}'(t), \mathbf{r}(t)).$$

Since \mathbf{r} is upset-free we have, $\langle \mathbf{r}'(t), \mathbf{r} \rangle \geq 0$ so by the definition of W (12),

$$W(\mathbf{r}'(t), \mathbf{r}(t)) = \frac{1}{\sigma(2 - \langle \mathbf{r}'(t), \mathbf{r}(t) \rangle)}.$$

The chain rule gives

$$\frac{d}{dt} \Phi(\mathbf{r}(t)) = \langle \mathbf{r}'(t), \nabla \Phi(\mathbf{r}(t)) \rangle. \quad (13)$$

Since \mathbf{r} is ordered in addition to being upset-free, $\mathbf{r}'(t)$ is always of the form $\mathbf{r}'(t) = \mathbf{e}_j - \mathbf{e}_{j+a}$ for positive a . That is, the winning player has a lower index than the losing player. The j th entry of the gradient can be computed directly as

$$\begin{aligned} \frac{\partial}{\partial \mathbf{s}_j} \Phi(\mathbf{s}) &= 2\mathbf{s}_j + f'(-2 + \mathbf{s}_j - \mathbf{s}_{j+1}) \cdot \mathbf{1}_{j < n} - f'(-2 + \mathbf{s}_{j-1} - \mathbf{s}_j) \cdot \mathbf{1}_{j > 1} \\ &= 2\mathbf{r}_j + \frac{\mathbf{1}_{j < n}}{\sigma(2 + \mathbf{s}_{j+1} - \mathbf{s}_j)} - \frac{\mathbf{1}_{j > 1}}{\sigma(2 + \mathbf{s}_j - \mathbf{s}_{j-1})}, \end{aligned}$$

and in particular

$$\begin{aligned} \langle \mathbf{e}_j - \mathbf{e}_{j+a}, \nabla \Phi(\mathbf{s}) \rangle &= \left(\frac{\partial}{\partial \mathbf{s}_j} \Phi(\mathbf{s}) - \frac{\partial}{\partial \mathbf{s}_{j+a}} \Phi(\mathbf{s}) \right) \\ &\leq 2(\mathbf{s}_j - \mathbf{s}_{j+a}) + \frac{\mathbf{1}_{j < n}}{\sigma(2 + \mathbf{s}_{j+1} - \mathbf{s}_j)} + \frac{\mathbf{1}_{j+a > 1}}{\sigma(2 + \mathbf{s}_{j+a} - \mathbf{s}_{j+a-1})} \\ &\leq 2(\mathbf{r}_j - \mathbf{r}_{j+a}) + \frac{\mathbf{1}_{j < n}}{\sigma(2 + \mathbf{s}_{j+a} - \mathbf{s}_j)} + \frac{\mathbf{1}_{j+a > 1}}{\sigma(2 + \mathbf{s}_{j+a} - \mathbf{s}_j)} \\ &\leq \frac{2 \sup_z \sigma(2 - z)}{\sigma(2 + \mathbf{s}_{j+a} - \mathbf{s}_j)} + \frac{2}{\sigma(2 + \mathbf{s}_{j+a} - \mathbf{s}_j)} \\ &= \left(2 + 2 \sup_z \sigma(2 - z)\right) \cdot \frac{1}{\sigma(2 - \langle \mathbf{e}_j - \mathbf{e}_{j+a}, \mathbf{s} \rangle)}. \\ &= \left(2 + 2 \sup_z \sigma(2 - z)\right) \cdot W(\mathbf{e}_j - \mathbf{e}_{j+a}, \mathbf{s}). \end{aligned}$$

For $\mathbf{s} = \mathbf{r}(t)$ and $\mathbf{e}_j - \mathbf{e}_{j+a} = \mathbf{r}'(t)$, this is exactly what we needed to show. ◀

The last piece of the puzzle is showing $\Phi(\mathbf{s})$ has to be large whenever one entry of \mathbf{s} is large. Rapid growth of f means if any two consecutive players have a large rating difference, Φ will be large. On the other hand, if consecutive players are close in rating, many entries of \mathbf{s} are close to its largest entry forcing $\|\mathbf{s}\|^2$ to be large.

► **Lemma 13.** Let \mathbf{s} be any vector such that $\max \mathbf{s} = \mathbf{s}_1$ and at least one entry is negative. Then

$$\Phi(\mathbf{s}) \geq \mathbf{s}_1^2 \quad \text{and} \quad \Phi(\mathbf{s}) \geq \frac{\mathbf{s}_1^3/8}{f^{-1}(\mathbf{s}_1^2/4) + 2}.$$

Furthermore,

$$\Phi(\mathbf{s}) \geq nf\left(-2 + \frac{\mathbf{s}_1}{2n}\right).$$

Proof. The first result is immediate from $\Phi(\mathbf{s}) \geq \|\mathbf{s}\|^2 \geq \mathbf{s}_1^2$. By the assumption on \mathbf{s} , there exists m such that

$$\mathbf{s}_m \geq \mathbf{s}_1/2 \geq \mathbf{s}_{m+1}.$$

Then

$$\|\mathbf{s}\|^2 \geq \frac{m}{4}\mathbf{s}_1^2. \quad (14)$$

By convexity of f ,

$$\sum_{j=1}^m f(-2 + \mathbf{s}_j - \mathbf{s}_{j+1}) \geq \sum_{j=1}^m f\left(-2 + \frac{\mathbf{s}_1 - \mathbf{s}_{m+1}}{m}\right) \geq mf\left(-2 + \frac{\mathbf{s}_1}{2m}\right). \quad (15)$$

We claim (15) is monotonically decreasing in m . To see this, note f' is itself positive and monotonically increasing by monotonicity of σ . Then for $x > -2$,

$$f(x) = \int_0^x f'(t) dt \leq xf'(x) \leq (x+2)f'(x) \implies 0 \leq \frac{(x+2)f'(x) - f(x)}{(x+2)^2} = \frac{d}{dx} \frac{f(x)}{x+2}.$$

Setting $x = -2 + \mathbf{s}_1/2m$ and noting x is itself monotonically decreasing in m establishes the claim. m is the index of a player, so we must have $m \leq n$. This immediately implies by monotonicity of (15) that

$$\Phi(\mathbf{s}) \geq nf\left(-2 + \frac{\mathbf{s}_1}{2n}\right).$$

Combining (14) and (15) yields

$$\begin{aligned} \Phi(\mathbf{s}) &\geq \inf_m \left(\frac{m}{4}\mathbf{s}_1^2 + mf\left(-2 + \frac{\mathbf{s}_1}{2m}\right) \right) \\ &\geq \inf_m \max\left(\frac{m}{4}\mathbf{s}_1^2, mf\left(-2 + \frac{\mathbf{s}_1}{2m}\right) \right). \end{aligned}$$

The minimum of the maximum of two functions occurs when they intersect, which in this case is guaranteed to happen exactly once since $f(-2 + \mathbf{s}_1/2m)$ is monotone in m and its range contains $[0, \infty) \ni \mathbf{s}_1^2/4$. Therefore the minimizing m is

$$m = \frac{\mathbf{s}_1/2}{f^{-1}(\mathbf{s}_1^2/4) + 2}.$$

The last result of the lemma follows immediately by plugging that m into (14). ◀

Assembling the above lemmas together results in Theorem 14.

► **Theorem 14.** *Let r be the highest rating achieved by a group of any number of players who play a total of k games. Suppose the pot function σ satisfies*

$$C_1 = \sup_z z \sigma(2 - z) < \infty.$$

Then

$$r \leq 2n \cdot (f^{-1}(C_2 \cdot k/n) + 2).$$

29:18 Achieving the Highest Possible Elo Rating

and

$$r \leq 2 \cdot C_2^{1/3} \cdot k^{1/3} \cdot (f^{-1}(C_2/4 \cdot k) + 2)^{1/3}$$

for $C_2 = 2 + 2C_1 + f(-2)$ where f is defined in (4).

Proof. We may take $n \leq k - 1$ players without loss of generality by simply ignoring players who aren't connected to the player achieving the highest rating via some sequence of games. Let $\mathbf{r} : [0, c] \rightarrow \mathbb{R}^n$ be the valid path corresponding to the sequence of games achieving rating $r = \max \mathbf{r}(c)$. Perform all necessary reflections to make it ordered. Lemma 9 gives

$$\text{len}(\mathbf{r}) \leq k. \tag{16}$$

Lemma 10 constructs $\tilde{\mathbf{r}}$ without any upsets satisfying

$$r = \max \mathbf{r}(c) \leq \max \tilde{\mathbf{r}}(\tilde{c}) \tag{17}$$

and

$$\text{len}(\tilde{\mathbf{r}}) \leq \text{len}(\mathbf{r}). \tag{18}$$

Since $\tilde{\mathbf{r}}$ is ordered and upset-free, Lemma 12 implies

$$\Phi(\tilde{\mathbf{r}}(\tilde{c})) - \Phi(\tilde{\mathbf{r}}(0)) \leq (2 + 2C_1) \text{len}(\tilde{\mathbf{r}}).$$

Note $\Phi(\tilde{\mathbf{r}}(0)) = \Phi(0) = (n - 1)f(-2)$ so

$$\Phi(\tilde{\mathbf{r}}(\tilde{c})) \leq (2 + 2C_1) \text{len}(\tilde{\mathbf{r}}) + (n - 1)f(-2). \tag{19}$$

Set $C_2 = 2 + 2C_1 + f(-2)$ and assemble the chain of inequalities:

$$\begin{aligned} \Phi(\tilde{\mathbf{r}}(\tilde{c})) &\stackrel{(19)}{\leq} (2 + 2C_1) \text{len}(\tilde{\mathbf{r}}) + (n - 1)f(-2) \\ &\stackrel{(18)}{\leq} (2 + 2C_1) \text{len}(\mathbf{r}) + (n - 1)f(-2) \\ &\stackrel{(16)}{\leq} (2 + 2C_1)k + (n - 1)f(-2) \\ &\leq (2 + 2C_1 + f(-2)) \cdot k \\ &= C_2 \cdot k \end{aligned} \tag{20}$$

The three lower bounds on Φ provided by Lemma 13 are each used in different ways. First, the n -dependent bound implies

$$nf\left(-2 + \frac{r}{2n}\right) \stackrel{(17)}{\leq} nf\left(-2 + \frac{\max \tilde{\mathbf{r}}(\tilde{c})}{2n}\right) \stackrel{(13)}{\leq} \Phi(\tilde{\mathbf{r}}(\tilde{c})) \stackrel{(20)}{\leq} C_2 \cdot k.$$

Since r appears only once in the equation, we can simply rearrange to solve for r . In particular,

$$r \leq 2n \left(f^{-1} \left(C_2 \cdot \frac{k}{n} \right) + 2 \right).$$

giving the first result of the theorem. The second n -independent bound gives

$$\frac{r^3/8}{f^{-1}(r^2/4) + 2} \stackrel{(17)}{\leq} \frac{(\max \tilde{\mathbf{r}}(\tilde{c}))^3/8}{f^{-1}((\max \tilde{\mathbf{r}}(\tilde{c}))^2/4) + 2} \stackrel{(13)}{\leq} \Phi(\tilde{\mathbf{r}}(\tilde{c})) \stackrel{(20)}{\leq} C_2 \cdot k. \tag{21}$$

This bounds k in terms of a function of r . The last application of Lemma 13 converts this into a bound on r in terms of k .

$$r^2 \stackrel{(17)}{\leq} (\max \tilde{r}(\tilde{c}))^2 \stackrel{13}{\leq} \Phi(\tilde{r}(\tilde{c})) \stackrel{(20)}{\leq} C_2 \cdot k.$$

This can be plugged back into the left-hand side of (21),

$$C_2 \cdot k \geq \frac{r^3/8}{f^{-1}(r^2/4) + 2} \geq \frac{r^3/8}{f^{-1}(C_2/4 \cdot k) + 2}.$$

Rearranging gives

$$r^3 \leq 8 \cdot C_2 \cdot k \cdot (f^{-1}(C_2/4 \cdot k) + 2).$$

Taking cube roots establishes the final result. ◀

► **Remark 15.** The probabilistic interpretation (3) of the Elo system described in the introduction lends itself naturally to the requirement that $z\sigma(2-z)$ be bounded. In particular, take $\sigma(z) = \Pr(\eta < x) + \frac{1}{2} \Pr(\eta = x)$ to be the symmetrized cumulative distribution function of a symmetric random variable η **with finite expectation**. Note

$$\sup_z z\sigma(2-z) = \sup_z (2+z)\sigma(-z) \leq 2 + \sup_z z\sigma(-z).$$

It's clear that the supremum on the right will be achieved for $z \geq 0$. Since η has finite expectation, we may apply Markov's inequality to obtain

$$z\sigma(-z) \leq z \Pr(\eta \leq -z) = \frac{1}{2} \cdot z \cdot \Pr(|\eta| \geq z) \leq \frac{1}{2} \mathbb{E}|\eta| < \infty.$$

This shows $\mathbb{E}|\eta| < \infty$ is sufficient. However it isn't strictly necessary; for instance, $z\sigma(2-z)$ just barely does not diverge for a Cauchy random variable. However, one *does* need finite $(1-\varepsilon)$ th moment: suppose $\sigma(-z) \leq c/z$. Then

$$\mathbb{E}|\eta|^{1-\varepsilon} = \int \Pr(|\eta|^{1-\varepsilon} \geq t) dt = 2 \int \sigma(-t^{1/(1-\varepsilon)}) dt \leq 2 \int t^{-1/(1-\varepsilon)} dt < \infty$$

for each $\varepsilon \in (0, 1)$.

5 Discussion

5.1 Remaining questions

Can one close the gap between the upper and lower bounds in the $n = \omega(1)$ regime? This could occur by finding a better strategy than the ones in Section 3, for instance by slowly increasing δ as games are played, or by tightening the analysis in Theorem 14. It's possible that for heavy-tailed σ the lower bound is too loose, but for light-tail σ the upper bound is. Why is there a jump in the upper bound from $f(k)^{1/3}$ to $f(k/n)$ at $n = k^{1/3}$? Can one find a bound that smoothly crosses the phase transition? In Theorem 6, one cannot totally specify the initial set of ratings, but has to allow the possibility that the initial ratings are all flipped. This appears very strongly to be an artifact of the analysis; can one prove a version that allows you to assign any initial ratings subject only to the constraint that the average rating is non-negative?

The highest Elo problem also bears a passing resemblance to the Toda lattice: players are particles whose positions on the real line is given by their ratings; they exhibit a repulsive force when a higher rated player beats a lower rated player and an attractive force if vice versa. Is it possible to use tools from solid state physics to analyze this problem?

Another interesting variant is to constrain the number of *rounds* of games. In a tournament, many games will be happening parallel. Each round consists of any number of games, subject only to the constraint that each player only participates in at most one game each round. **Given n players and k rounds, what's the highest someone may be rated after all games have finished?**

5.2 Connection to maximum overhang

One may notice the jump from $\log k$ to $k^{1/3}$ as one increases n for $\sigma(-z) = \frac{1}{1+e^{-z}}$. This may be reminiscent of the maximum overhang problem. In that problem, one places k unit-length bricks on top of each other at the edge of a table, attempting to achieve the largest overhang possible. The classic solution if one is restricted to placing only a single brick at each height (the “single-wide” setting) achieves an overhang of $\Theta(\log k)$ units. However, if one is allowed to place as many bricks at each height as one likes (the “multi-wide” setting), the optimal solution achieves an overhang of $\Theta(k^{1/3})$ [5]. This connection does not appear to be a coincidence! The proof of optimality in the maximum overhang problem uses a reduction to a “mass movement problem.” But in fact, the highest Elo problem for the particular $\sigma(z) = \max(0, \min(1, z/2 + 1/2))$ can be reduced to nearly the same problem!

The mass movement problem described by [5] is as follows: consider some finite number of piles of mass placed on the real line. A valid “move” takes some unit interval and rearranges the mass within that interval so that the center of mass is unchanged. Negative mass is allowed. Formally, a “signed distribution” μ is a finite linear combination of dirac delta functions δ . A valid “move” replaces μ with $\mu + \nu$ where ν is itself a signed distribution whose support is contained in some unit interval and $\int \nu = 0$. A sequence of moves $\nu_1 \cdots, \nu_k$ corresponds naturally to a sequence of signed distributions μ_0, \cdots, μ_k . They place an additional “weight-constraint” on allowed sequences. In particular, the number of j for which ν_j is allowed to have support to the right of any threshold T is at most $\max_{0 \leq i \leq k} \int_T^\infty \mu_i$. The goal is to use at most k moves to move the distribution $k \cdot \delta$ to a distribution with a unit of mass as far right as possible.

To reduce highest Elo to mass movement, let $r_p(j)$ be the rating of player p after j games and set

$$\mu_j(x) = \sum_p \delta(x - r_p(j)).$$

Then consider a game where p beats q . Set $z = r_p - r_q$. By the selection of $\sigma(z) = \max(0, \min(z/2 + 1/2))$, this only produces a change in ratings if $z \leq 1$. Then μ changes by performing the following move:

$$\nu(x) = \delta(x - r_p - \sigma(-z)) + \delta(x - r_q + \sigma(-z)) - \delta(x - r_p) - \delta(x - r_q). \tag{22}$$

Suppose $z \geq -1$. Then the support of ν is contained in $[r_q - \sigma(-z), r_p + \sigma(-z)]$, which is an interval of length $z + 2\sigma(-z) = 1$ as required. Furthermore, ν only has support above T when there's a player above T Elo guaranteeing the weight-constraint. The objective is starting with $n \cdot \delta$, produce via k moves a distribution with a unit of mass (i.e. at least one player) as far right as possible (i.e. with the highest rating possible). Recall that for the purposes of an upper bound, we make take $n = k$, establishing the correct objective.


There is, however, one big discrepancy: the case of $z < -1$. In particular, in the highest Elo problem it is possible for a player rated -10 to beat a player rated 10 , garnering a full point of Elo. This does not correspond to a valid move, since the support of ν would be $\{-10, -9, 9, 10\}$. Intuitively, such moves should not ultimately help push mass far away, and indeed in Lemma 10 we show for a different relaxation of highest Elo that they don't help.

A second discrepancy is that mass movement places no restriction on the amount of mass moved in each move, whereas for highest Elo we always have $\|\nu\|_1 = 2$. This would seem to imply an upper bound of the “ k rounds” variant mentioned at the end of Section 5.1. Can that argument be made to work? Conversely, is there a variant or generalization of maximum overhang that can be analyzed using the proof of Theorem 14?

References

- 1 Arpad E. Elo. The proposed USCF rating system: Its development, theory, and applications. *Chess Life*, 23(8), August 1967.
- 2 Arpad E. Elo. *The Rating of Chess Players, Past and Present*. Arco Publishing, second edition, 1978.
- 3 FIDE. FIDE Ratings and Statistics. <https://ratings.fide.com/>.
- 4 FIDE. FIDE Handbook. <https://handbook.fide.com/>, 2017.
- 5 Mike Paterson, Yuval Peres, Mikkel Thorup, Peter Winkler, and Uri Zwick. Maximum overhang. *The American Mathematical Monthly*, 116(9):763–787, 2009. doi:10.4169/000298909X474855.
- 6 Jeff Sonas. The Elo rating system – correcting the expectancy tables. *ChessBase*, March 2011.

How to Covertly and Uniformly Scramble the 15 Puzzle and Rubik’s Cube

Kazumasa Shinagawa ✉ 

Ibaraki University, Japan

National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Kazuki Kanai ✉ 

National Institute of Technology, Kure College, Hiroshima, Japan

Kengo Miyamoto ✉ 

Ibaraki University, Japan

Koji Nuida ✉ 

Institute of Mathematics for Industry (IMI), Kyushu University, Fukuoka, Japan

National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

Abstract

A combination puzzle is a puzzle consisting of a set of pieces that can be rearranged into various combinations, such as the 15 Puzzle and Rubik’s Cube. Suppose a speedsolving competition for a combination puzzle is to be held. To make the competition fair, we need to generate an instance (i.e., a state having a solution) that is chosen uniformly at random and unknown to anyone. We call this problem a secure random instance generation of the puzzle. In this paper, we construct secure random instance generation protocols for the 15 Puzzle and for Rubik’s Cube. Our method is based on uniform cyclic group factorizations for finite groups, which is recently introduced by the same authors, applied to permutation groups for the puzzle instances. Specifically, our protocols require 19 shuffles for the 15 Puzzle and 43 shuffles for Rubik’s Cube.

2012 ACM Subject Classification Security and privacy → Information-theoretic techniques

Keywords and phrases Card-based cryptography, Uniform cyclic group factorization, Secure random instance generation, The 15 Puzzle, Rubik’s Cube

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.30

Funding *Kazumasa Shinagawa*: This work was supported in part by JSPS KAKENHI Grant Numbers 21K17702 and 23H00479, and JST CREST Grant Number MJCR22M1.

Kengo Miyamoto: This work was supported in part by JSPS KAKENHI Grant Numbers 20K14302 and 23H00479.

Koji Nuida: This work was supported in part by JSPS KAKENHI Grant Number JP19H01109.

1 Introduction

1.1 Secure Random Instance Generation Problem

A *combination puzzle* is a puzzle that consists of a set of pieces that can be rearranged into various combinations, such as the 15 Puzzle and Rubik’s Cube. Suppose we want to hold a speedsolving competition for a combination puzzle. For the competition, we need to generate an *instance* (i.e., a state having a solution) of the puzzle. It must be chosen uniformly at random from all instances, since otherwise some players may predict the instance. In an actual speedsolving competition, an instance is chosen randomly by a computer program, and a person called the *scrambler* applies the corresponding scrambling procedure to the puzzle. An obvious drawback of this method is that the scrambler, who should know the chosen instance, cannot fairly participate in the competition. One solution to this problem could be to use a robot that generates a uniformly random instance of the puzzle, but verifying the correctness of the robot’s behavior is not easy in general.



© Kazumasa Shinagawa, Kazuki Kanai, Kengo Miyamoto, and Koji Nuida;
licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 30; pp. 30:1–30:15

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

From this background, we need to generate an instance of the puzzle that satisfies the following conditions:

- It is chosen uniformly at random from all instances.
- It is hidden from all players, *including the scrambler*, until the competition starts.
- It is arranged by human hands without electronic devices (such as a robot).

We call this problem a *secure random instance generation problem* of the puzzle.

For the 15 Puzzle, the problem might seem to be trivial at first glance, because the structure of an ordinary 15 Puzzle board allows taking out the 15 blocks, turning them face-down, and scramble them randomly. However, there is actually an issue; a completely random arrangement of blocks may have no solution. Precisely¹, it is well-known that when the blocks are arranged according to a permutation σ in the symmetric group S_{15} , a solution exists if and only if σ is in the alternating group A_{15} , which happens only with probability $1/2$ for a uniformly random σ . Therefore, we need to scramble the blocks in a way that only permutations in A_{15} can appear, which is a non-trivial task.

For Rubik's Cube, similar to the 15 Puzzle, we need to covertly and uniformly generate a permutation of the *Rubik's Cube group* R , which is a subgroup of the 48th symmetric group S_{48} , but the situation is more complicated than for the 15 Puzzle (not just because the group R is more complicated than A_{15}). First, unlike the 15 Puzzle, each piece of the cube cannot be "face-down." Second, since all pieces are mechanically connected, applying a permutation to the cube is a non-trivial task.

In this paper, we consider the secure random instance generation problem for the first time and solve the problem for the 15 Puzzle and Rubik's Cube. For both puzzles, we need to covertly and uniformly generate a permutation from a finite group. A similar problem has been studied in the research area of *card-based cryptography*.

1.2 Card-Based Cryptography

Card-based cryptography [3, 5, 14] is a research area within cryptography, which is based on physical cards; cryptographic protocols such as secure computation protocols and zero-knowledge proof protocols are implemented by a deck of cards without electronic devices.

A *shuffle* is an operation to rearrange a card sequence covertly and randomly. Formally, a shuffle for a sequence of n cards is defined by a pair of a set of permutations $\Pi \subseteq S_n$, where S_n denotes the n -th symmetric group, and a probability distribution \mathcal{F} on Π , which is denoted by $(\text{shuffle}, \Pi, \mathcal{F})$. For a sequence $\vec{c} := (c_1, c_2, \dots, c_n)$ of face-down cards, it chooses a permutation $\pi \in \Pi$ according to \mathcal{F} covertly, and rearranges it into the sequence $\pi(\vec{c}) = (c_{\pi^{-1}(1)}, c_{\pi^{-1}(2)}, \dots, c_{\pi^{-1}(n)})$. Here, it is assumed that no player (including the player who actually operates the shuffle operation) can guess which permutation π was chosen beyond the fact that it was chosen according to \mathcal{F} . A shuffle $(\text{shuffle}, \Pi, \mathcal{F})$ is said to be *uniform* if \mathcal{F} is the uniform distribution on Π , *closed* if Π is a subgroup of S_n , and *uniform closed* if both conditions hold. Hereinafter, for a uniform shuffle $(\text{shuffle}, \Pi, \mathcal{F})$, we write it as $(\text{shuffle}, \Pi)$ and call it the Π -*shuffle*.

Although the definition of shuffles allows for an arbitrary permutation set Π and an arbitrary distribution \mathcal{F} , how to physically implement a shuffle $(\text{shuffle}, \Pi, \mathcal{F})$ given Π and \mathcal{F} is quite non-trivial. In the literature on card-based cryptography, five shuffles – a *random cut*, a *random bisection cut*, a *pile-shifting shuffle*, a *complete shuffle*, and a *pile-scramble shuffle*

¹ We assume that an instance of the 15 Puzzle always has the empty square placed at the bottom right (as well as the default instance of the puzzle), therefore an arrangement of the puzzle corresponds to a permutation in S_{15} rather than S_{16} .

– are considered easy to implement. This is because the first three shuffles are performed by random cyclic shifting and the last two shuffles are performed by completely random scrambling. (See also Ueda et al. [23] for how to implement a random cut, a random bisection cut, and a pile-shifting shuffle.) In this paper, we call these five shuffles *practical shuffles*.

From this context, there is a line of research to implement a class of shuffles from practical shuffles with the use of “helping cards”. Saito et al. [20] showed that any shuffle (shuffle, Π , \mathcal{F}) can be implemented by three practical shuffles (one random cut, one pile-shifting shuffle, and one pile-scramble shuffle) with helping cards if every probability of \mathcal{F} is a rational number. Although it is very general and important work, it requires at least $n \cdot |\Pi|$ helping cards; it is inefficient in terms of the number of cards when $|\Pi|$ is large. Another important example of previous work is the result by Koch [11]. He showed that any uniform closed shuffle (shuffle, Π) can be implemented by pile-shifting shuffles. However, it requires $O(|\Pi|)$ shuffles; it is inefficient in terms of the number of shuffles when $|\Pi|$ is large. In summary, up until now, there is no efficient implementation of a shuffle when $|\Pi|$ is large.

1.3 Our Contribution

In this paper, we propose physical protocols for secure random instance generation of the 15 Puzzle and Rubik’s Cube. Let G be the permutation group corresponding to the set of all instances of the puzzle. We divide the problem into the following problems:

- How to implement a G -shuffle from practical shuffles?
- How to apply a G -shuffle to the puzzle physically?

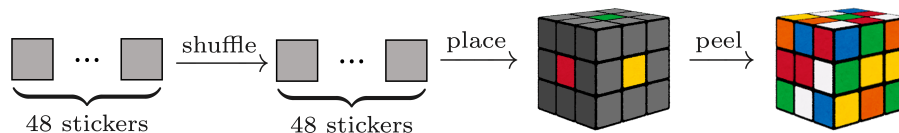
For the first problem, we utilize a decomposition of a finite group into cyclic subgroups called a *uniform cyclic group factorization* (UCF), which is recently proposed by the authors [10] (see Section 3.1). We show that if a group G has a UCF of length k , a G -shuffle can be implemented by a sequence of k *cyclic group shuffles*, which are C -shuffles for some cyclic group C , with no additional cards. This is an efficient method to implement a G -shuffle from practical shuffles since k is considerably smaller than $|G|$. Up until now, it is known that all solvable groups have a UCF [10] but whether any finite group has a UCF or not is still open. Fortunately, the alternating group and the Rubik’s Cube group R have a UCF. Based on the UCF, the A_{15} -shuffle and the R -shuffle can be implemented by a sequence of 19 and 43 cyclic group shuffles, respectively. In addition, these cyclic group shuffles are practical shuffles: random cuts, random bisection cuts, and pile-shifting shuffles. Note that since $|A_{15}| = 653837184000$ and $|R| = 43252003274489856000$, the existing methods [11, 20] require a large number of cards or a large number of shuffles at least these numbers.

For the 15 Puzzle, the second problem is almost trivial: Turn the blocks face-down and apply the A_{15} -shuffle to the sequence of face-down blocks just like a sequence of cards. Thus, the secure random instance generation problem of the 15 Puzzle is solved.

For Rubik’s Cube, the second problem is more complicated than that of the 15 Puzzle because there are two obstacles: (1) since each piece of the cube cannot be “face-down”, how to hide an instance of the cube is non-trivial; (2) since all pieces are mechanically connected, how to apply a permutation to the cube is non-trivial. To solve (1) and (2), we propose two methods for the construction of a secure random instance generation of Rubik’s Cube.

The first solution requires *color stickers*, which are concealed by peelable *films*: The faces of the cube are hidden by stickers and films, as a solution to (1), and a permutation is applied to a sequence of color stickers instead of the cube, as a solution to (2). The protocol proceeds as follows. First, an R -shuffle is applied to a sequence of color stickers. Then, each

sticker is placed on each face of the cube. Finally, just before the competition starts, all films are peeled off. We call it a protocol in the *free permutation model* since it applies a permutation freely. The protocol flow is summarized as Figure 1.



■ **Figure 1** The flow of our protocol for Rubik's Cube in the free permutation model.

The second solution requires a piece of cloth: The cube is covered by a piece of cloth, as a solution to (1), and standard operations of Rubik's Cube (i.e., F, B, U, D, L, and R) are applied from under the cloth, as a solution to (2). The protocol proceeds as follows. First, cover the solved cube by a large piece of cloth. Then, apply a sequence of standard operations repeatedly until the player who scrambles the puzzle cannot remember how many times it has been repeated (just like a *Hindu cut* of the playing card) from under the cloth. Finally, just before the competition starts, the cloth is removed. We call it a protocol in the *restricted permutation model* since the permutations applied to the puzzle is restricted to the standard operations.

In summary, we propose three protocols for secure random instance generation of the 15 Puzzle and Rubik's Cube. For the 15 Puzzle, we propose a protocol with 19 practical shuffles (7 random cuts, 7 random bisection cuts, and 5 pile-shifting shuffles). For Rubik's Cube, our protocol in the free permutation model applies 43 practical shuffles (22 random bisection cuts and 21 pile-shifting shuffles) to the sequence of color stickers, and one in the restricted permutation model applies a sequence of standard operations to the cube directly.

1.4 Related Work

Mathematics of the 15 Puzzle. It is NP-hard to determine whether an instance of the generalized 15 Puzzle can be solved at most k moves for a given integer k [17, 18]. It is shown that the length of shortest solutions ranges from 0 to 80 single-tile moves [1] or 43 multi-tile moves [16]. For the generalized 15 Puzzle with an $n \times n$ puzzle board, it is shown that the asymptotic mixing time is $O(n^4 \log n)$ when random moves are made [2].

Mathematics of Rubik's Cube. It is NP-complete to determine whether an instance of the generalized $n \times n \times n$ Rubik's Cube can be solved at most k moves for a given integer k [4]. It is shown that the minimum number of steps required to solve any instance of Rubik's Cube called *God's Number* is 20 [19].

Card-Based Cryptography. There is a line of research to implement somewhat complicated shuffles from practical shuffles. There are several studies on generating a *derangement*, which is a permutation without fixed points, covertly and uniformly at random [3, 6, 8, 9]. It can be seen as a uniform shuffle whose permutation set is the set of all derangements. Hashimoto et al. [7] proposed a *secure grouping protocol* that generates a random permutation with some conditions, which has an application to the Werewolf game. Also, it can be seen as a kind of uniform shuffle. Miyamoto and Shinagawa [12, 21] constructed a protocol for implementing *graph shuffles*, which is a class of uniform closed shuffles whose permutation set is the automorphism group of a graph, from pile-scramble shuffles.

Secure Computation Using the 15 Puzzle. Mizuki, Kugimoto, and Sone [13] showed that secure computation can be done by the use of the 15 Puzzle. In particular, they showed that any 4-variable Boolean function and any 14-variable Boolean symmetric function are securely computed by using the 15 Puzzle.

2 Preliminaries

2.1 Notations

Throughout this paper, any groups are assumed to be finite. We denote the n -th symmetric group by S_n and the n -th alternating group by A_n . We assume to multiply permutations from left to right, e.g., $(1, 2)(1, 3) = (1, 2, 3)$. Note that it is the convention used by the GAP and the community of Rubik's Cube.

2.2 Shuffle

A *shuffle* is an operation that rearranges a sequence of cards covertly and randomly (see also Section 1.2). In Sections 4 and 5, we use random cuts, random bisection cuts, and pile-shifting shuffles. These shuffles are special cases of cyclic group shuffles.

Cyclic Group Shuffle. A *cyclic group shuffle* is a G -shuffle for some cyclic group $G = \langle g \rangle$. Let k be the order of G . Then it applies a permutation g^r to a sequence of cards, where $r \in \{0, 1, \dots, k-1\}$ is chosen uniformly at random. For example, applying a cyclic group shuffle ($\text{shuffle}, \langle g \rangle$), where $g = (1, 2)(3, 4, 5, 6) \in S_6$, to a sequence of six cards yields the following result:

$$\vec{c} = \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \mapsto \pi(\vec{c}) = \begin{cases} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^0 \text{ (with prob. } 1/4); \\ \begin{array}{cccccc} 2 & 1 & 6 & 3 & 4 & 5 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^1 \text{ (with prob. } 1/4); \\ \begin{array}{cccccc} 1 & 2 & 5 & 6 & 3 & 4 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^2 \text{ (with prob. } 1/4); \\ \begin{array}{cccccc} 2 & 1 & 4 & 5 & 6 & 3 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^3 \text{ (with prob. } 1/4). \end{cases}$$

Random Cut. A *random cut* [5] (of length k) is a cyclic group shuffle whose generator g is a conjugate of $(1, 2, \dots, k) \in S_n$, i.e., there exists a permutation $\tau \in S_n$ such that $g = \tau^{-1}(1, 2, \dots, k)\tau$. For example, applying a random cut ($\text{shuffle}, \langle g \rangle$), where $g = (2, 1, 4)$, to a sequence of four cards yields the following result:

$$\vec{c} = \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \mapsto \pi(\vec{c}) = \begin{cases} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^0 \text{ (with prob. } 1/3); \\ \begin{array}{cccc} 2 & 4 & 3 & 1 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^1 \text{ (with prob. } 1/3); \\ \begin{array}{cccc} 4 & 1 & 3 & 2 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^2 \text{ (with prob. } 1/3). \end{cases}$$

Random Bisection Cut. A *random bisection cut* [15] (of size ℓ) is a cyclic group shuffle whose generator is a conjugate of $(1, \ell+1)(2, \ell+2) \cdots (\ell, 2\ell) \in S_n$. For example, applying a random bisection cut ($\text{shuffle}, g$), where $g = (1, 4)(2, 5)(3, 6)$, to a sequence of six cards yields the following result:

$$\vec{c} = \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \mapsto \pi(\vec{c}) = \begin{cases} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^0 \text{ (with prob. } 1/2); \\ \begin{array}{cccccc} 4 & 5 & 6 & 1 & 2 & 3 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^1 \text{ (with prob. } 1/2). \end{cases}$$

Pile-Shifting Shuffle. A *pile-shifting shuffle* [22] (of size ℓ with k piles) is a cyclic group shuffle whose generator g is a conjugate of the following permutation:

$$\vec{c} = (1, \ell+1, 2\ell+1, \dots, (k-1)\ell+1)(2, \ell+2, 2\ell+2, \dots, (k-1)\ell+2) \cdots (\ell, 2\ell, 3\ell, \dots, k\ell) \in S_n.$$

For example, applying a pile-shifting shuffle (shuffle, $\langle g \rangle$), where $g = (1, 3, 5)(2, 4, 6)$, to a sequence of six cards yields the following result:

$$\begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} \mapsto \pi(\vec{c}) = \begin{cases} \begin{array}{cc|cc|cc} 1 & 2 & 3 & 4 & 5 & 6 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^0 \text{ (with prob. } 1/3); \\ \begin{array}{cc|cc|cc} 5 & 6 & 1 & 2 & 3 & 4 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^1 \text{ (with prob. } 1/3); \\ \begin{array}{cc|cc|cc} 3 & 4 & 5 & 6 & 1 & 2 \\ \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} & \boxed{?} \end{array} & \text{if } \pi = g^2 \text{ (with prob. } 1/3). \end{cases}$$

3 Uniform Cyclic Group Factorization and Its Application to Shuffle

In this section, we recall the notion of uniform cyclic group factorizations of a finite group [10] and show that if a group G has a uniform cyclic group factorization, the G -shuffle can be implemented by a sequence of cyclic group shuffles with no additional cards.

3.1 Uniform Cyclic Group Factorization

Let G be a group. Let $\mathcal{H} = (H_1, H_2, \dots, H_k)$ be an ordered tuple of subsets of G . Define the multiplication map $\text{mult}_{\mathcal{H}} : H_1 \times H_2 \times \dots \times H_k \rightarrow G$ by $\text{mult}_{\mathcal{H}}(h_1, h_2, \dots, h_k) := h_1 h_2 \cdots h_k$. \mathcal{H} is called a *factorization* of G if $\text{mult}_{\mathcal{H}}$ is surjective. The integer k is called the *length* of \mathcal{H} . If H_1, H_2, \dots, H_k are proper subsets of G , then \mathcal{H} is called a *proper factorization* of G .

► **Definition 1** (Definition 2.1 in [10]). *Let G be a group and $\mathcal{H} = (H_1, H_2, \dots, H_k)$ a factorization of G .*

- (1) *The factorization \mathcal{H} is a uniform factorization of G if $|\text{mult}_{\mathcal{H}}^{-1}(g)|$ does not depend on $g \in G$. The integer $t := |\text{mult}_{\mathcal{H}}^{-1}(g)|$ is called the multiplicity of \mathcal{H} .*
- (2) *The factorization \mathcal{H} is a uniform group factorization (UGF) of G if \mathcal{H} is a uniform factorization of G and all H_1, H_2, \dots, H_k are subgroups of G .*
- (3) *The factorization \mathcal{H} is a uniform cyclic group factorization (UCF) of G if \mathcal{H} is a uniform group factorization of G and all H_1, H_2, \dots, H_k are cyclic subgroups of G .*

When (H_1, H_2, \dots, H_k) is a UGF (or a UCF) of G , we write $G = H_1 H_2 \dots H_k$.

We give some examples of UCF. For the n -th symmetric group S_n , by letting $H_i := \langle (1, \dots, i+1) \rangle$, the tuple $\mathcal{H}_1 = (H_1, H_2, \dots, H_{n-1})$ is a UCF of S_n . The length of \mathcal{H}_1 is $n-1$ and the multiplicity of \mathcal{H}_1 is 1. For the n -th dihedral group $D_n = \langle \sigma, \tau \mid \sigma^n = 1, \tau^2 = 1, \tau^{-1} \sigma \tau = \sigma^{-1} \rangle$, the tuple $\mathcal{H}_2 = (\langle \sigma \rangle, \langle \tau \rangle)$ is a UCF of D_n . The length of \mathcal{H}_2 is 2 and the multiplicity of \mathcal{H}_2 is 1.

► **Lemma 2.** *Let G_1, G_2 be groups. If G_1 and G_2 have UGF (resp. UCF), then the direct product $G_1 \times G_2$ and the semi-direct product $G_1 \rtimes G_2$ also have UGF (resp. UCF).*

Proof. Suppose that $G_1 = H_1 H_2 \dots H_k$ and $G_2 = M_1 M_2 \dots M_\ell$ where $H_i = \langle h_i \rangle$ and $M_j = \langle m_j \rangle$. Then the direct product $G_1 \times G_2 = \{(g_1, g_2) \mid g_i \in G_i\}$ has a UCF $G_1 \times G_2 = \widehat{H}_1 \widehat{H}_2 \dots \widehat{H}_k \widehat{M}_1 \widehat{M}_2 \dots \widehat{M}_\ell$ where $\widehat{H}_i := \langle (h_i, \text{id}_{G_2}) \rangle$ and $\widehat{M}_j := \langle (\text{id}_{G_1}, m_j) \rangle$. Since every element $x \in G_1 \times G_2$ is uniquely represented by $x = g_1 g_2$ for $g_i \in G_i$, the semi-direct product $G_1 \rtimes G_2$ has a UCF $G_1 \rtimes G_2 = H_1 H_2 \dots H_k M_1 M_2 \dots M_\ell$. ◀

Any solvable group has a UCF (Theorem 3.3 in [10]). It is an open problem whether any group has a UCF or not. The authors showed that any group has a UCF whenever any group has a proper UGF (Theorem 3.4 in [10]), i.e., the open problem can be affirmatively solved if any group has a UGF.

3.2 Shuffles Based on Uniform Cyclic Group Factorization

We show that if a group G has a uniform cyclic group factorization, the G -shuffle can be implemented by a sequence of cyclic group shuffles with no additional cards.

First, we define the equivalence between a shuffle and a sequence of shuffles. Intuitively, they are said to be equivalent if the resultant probability distributions are the same.

► **Definition 3.** Let G, H_1, \dots, H_s be subgroups of S_n . A shuffle $(\text{shuffle}, G, \mathcal{F})$ is said to be equivalent to a sequence of shuffles $(\text{shuffle}, H_1, \mathcal{F}_1), \dots, (\text{shuffle}, H_s, \mathcal{F}_s)$ if for all $g \in G$,

$$\mathcal{F}(g) = \sum_{\substack{(h_1, h_2, \dots, h_s) \in H_1 \times H_2 \times \dots \times H_s \\ h_1 h_2 \dots h_s = g}} \prod_{i=1}^s \mathcal{F}_i(h_i), \quad (1)$$

where the summation is taken over all $(h_1, \dots, h_s) \in H_1 \times \dots \times H_s$ such that $h_1 \dots h_s = g$.

The following lemma is easy but important for implementing shuffles.

► **Lemma 4.** Let G be a group and $\mathcal{H} = (H_1, H_2, \dots, H_k)$ a factorization of G . Let $\mathcal{F}_{\mathcal{H}}$ be a probability distribution on G defined as follows:

$$\mathcal{F}_{\mathcal{H}}(g) = \frac{|\text{mult}_{\mathcal{H}}^{-1}(g)|}{\prod_{i=1}^k |H_i|}.$$

Then a shuffle $(\text{shuffle}, G, \mathcal{F}_{\mathcal{H}})$ is equivalent to a sequence of k uniform shuffles as follows:

$$(\text{shuffle}, H_1), (\text{shuffle}, H_2), \dots, (\text{shuffle}, H_k).$$

Moreover, if \mathcal{H} is a uniform factorization, $\mathcal{F}_{\mathcal{H}}$ is a uniform distribution on G .

Proof. Since the probability that $h_i \in H_i$ is chosen by $(\text{shuffle}, H_i)$ is $\frac{1}{|H_i|}$ and the number of $(h_1, \dots, h_k) \in H_1 \times \dots \times H_k$ such that $h_1 \dots h_k = g$ is $|\text{mult}_{\mathcal{H}}^{-1}(g)|$, the right hand side of Eq. (1) is $\frac{|\text{mult}_{\mathcal{H}}^{-1}(g)|}{\prod_{i=1}^k |H_i|}$ which is equal to $\mathcal{F}_{\mathcal{H}}(g)$. Therefore, $(\text{shuffle}, G, \mathcal{F}_{\mathcal{H}})$ is equivalent to the sequence of k uniform shuffles. If \mathcal{H} is a uniform factorization, $\mathcal{F}_{\mathcal{H}}$ is a uniform distribution on G since $|\text{mult}_{\mathcal{H}}^{-1}(g)|$ does not depend on g . ◀

► **Corollary 5.** Let G be a group. If G has a uniform cyclic group factorization of length k , a uniform closed shuffle $(\text{shuffle}, G)$ is equivalent to a sequence of k cyclic group shuffles.

Proof. Follows from Lemma 4. ◀

4 Secure Random Instance Generation of the 15 Puzzle

In this section, we propose a secure random instance generation protocol of the 15 Puzzle. In Section 4.1, we give a UCF of the alternating group A_n based on the construction given by the authors [10]. In Section 4.2, we construct a protocol based on the UCF.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

■ **Figure 2** Cells of the 15 Puzzle.

4.1 Uniform Cyclic Group Factorization of the Alternating Group

The 15 Puzzle is a sliding block puzzle consisting of a *board* and 15 *blocks* from $\boxed{1}$ to $\boxed{15}$. The backs of all blocks are identical and denoted by $\boxed{?}$. The board has 4×4 *cells*, which are numbered from 1 to 16 as in Figure 2. Initially, all blocks are placed on cells from 1 to 15 in a random order. A cell having no block is called a *blank cell*. The aim of the puzzle is to make a board with $\boxed{1}$ to $\boxed{15}$ in cells from 1 to 15 in the order, by sliding blocks into the blank cell. An arrangement of the puzzle is identified with a permutation $\sigma \in S_{15}$ such that $\sigma(i) = j$ if the block \boxed{j} is on the cell i . It is known that a permutation σ has a solution if and only if σ is an even permutation, thus the set of all instances of the 15 Puzzle forms the alternating group A_{15} .

Now we construct a UCF of A_n based on Proposition 5.2 in [10]. If n is odd, A_n is decomposed by $A_n = HK$ where $H \simeq A_{n-1}$ is the stabilizer fixing the point n and $K = \langle (1, 2, 3, \dots, n) \rangle$ is a cyclic group. If n is even (i.e., $n = 2m$), A_n is decomposed by $A_n = HK_1K_2$ where $H \simeq A_{n-1}$ is the stabilizer fixing the point n and $K_1 = \langle (1, 2, \dots, m)(m+1, m+2, \dots, 2m) \rangle$ and $K_2 = \langle (1, m+1)(m, 2m) \rangle$ are cyclic groups.

From the above, we can construct a UCF of A_n for any n . Note that all subgroups correspond to practical shuffles: $K = \langle (1, 2, 3, \dots, n) \rangle$ corresponds to a random cut, $K_1 = \langle (1, 2, \dots, m)(m+1, m+2, \dots, 2m) \rangle$ corresponds to a pile-shifting shuffle, and $K_2 = \langle (1, m+1)(m, 2m) \rangle$ corresponds to a random bisection cut. The above discussion is summarized by the following lemma.

► **Lemma 6.** *Let $n \geq 4$ and A_n be the set of all even permutations in S_n . If $n = 2m$, an A_n -shuffle is equivalent to a sequence of $3m - 3$ shuffles: $m - 1$ random cuts, m random bisection cuts, and $m - 2$ pile-shifting shuffles. If $n = 2m + 1$, it is equivalent to a sequence of $3m - 2$ shuffles: m random cuts, m random bisection cuts, and $m - 2$ pile-shifting shuffles.*

A UCF of A_{15} is given by $A_{15} = H_1H_2 \cdots H_{14}$ as follows:

- $H_1 = \langle (1, 2, 3) \rangle;$
- $H_2 = \langle (1, 3)(2, 4) \rangle;$
- $H_3 = \langle (1, 2)(3, 4) \rangle;$
- $H_4 = \langle (1, 2, 3, 4, 5) \rangle;$
- $H_5 = \langle (1, 4)(3, 6) \rangle;$
- $H_6 = \langle (1, 2, 3)(4, 5, 6) \rangle;$
- $H_7 = \langle (1, 2, 3, 4, 5, 6, 7) \rangle;$
- $H_8 = \langle (1, 5)(4, 8) \rangle;$
- $H_9 = \langle (1, 2, 3, 4)(5, 6, 7, 8) \rangle;$
- $H_{10} = \langle (1, 2, 3, 4, 5, 6, 7, 8, 9) \rangle;$
- $H_{11} = \langle (1, 6)(5, 10) \rangle;$
- $H_{12} = \langle (1, 2, 3, 4, 5)(6, 7, 8, 9, 10) \rangle;$
- $H_{13} = \langle (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11) \rangle;$
- $H_{14} = \langle (1, 7)(6, 12) \rangle;$

- $H_{15} = \langle (1, 2, 3, 4, 5, 6)(7, 8, 9, 10, 11, 12) \rangle$;
- $H_{16} = \langle (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13) \rangle$;
- $H_{17} = \langle (1, 8)(7, 14) \rangle$;
- $H_{18} = \langle (1, 2, 3, 4, 5, 6, 7)(8, 9, 10, 11, 12, 13, 14) \rangle$;
- $H_{19} = \langle (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \rangle$.

4.2 Our Protocol for the 15 Puzzle

Based on the UCF of A_{15} , we can construct a secure random instance generation protocol for the 15 Puzzle. From Lemma 6, our protocol requires 19 practical shuffles consisting of 7 random cuts, 7 random bisection cuts, and 5 pile-shifting shuffles. The protocol proceeds as follows:

1. Arrange 15 blocks as follows:

$$\boxed{1} \boxed{2} \boxed{3} \boxed{4} \boxed{5} \boxed{6} \boxed{7} \boxed{8} \boxed{9} \boxed{10} \boxed{11} \boxed{12} \boxed{13} \boxed{14} \boxed{15}.$$

2. Turn all blocks face-down as follows:

$$\boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?} \boxed{?}.$$

3. Apply a shuffle (shuffle, H_i) for $i = 1, 2, \dots, 19$ as follows:

$$\underbrace{\boxed{?} \dots \boxed{?}}_{15 \text{ blocks}} \xrightarrow{H_1} \underbrace{\boxed{?} \dots \boxed{?}}_{15 \text{ blocks}} \xrightarrow{H_2} \underbrace{\boxed{?} \dots \boxed{?}}_{15 \text{ blocks}} \xrightarrow{H_3} \dots \xrightarrow{H_{19}} \underbrace{\boxed{?} \dots \boxed{?}}_{15 \text{ blocks}}.$$

4. Place the i -th block to the i -th cell of the board for $1 \leq i \leq 15$. The face-down blocks on the board is a random instance of the 15 Puzzle.

5 Secure Random Instance Generation of Rubik's Cube

In this section, we propose a secure random instance generation protocol of Rubik's Cube. In Section 5.1, we give a UCF of the Rubik's Cube group R . In Section 5.2, we construct a protocol in the free permutation model. In Section 5.3, we give a UCF of R whose generators are written by a product of the standard generators. In Section 5.4, we construct a protocol in the restricted permutation model.

5.1 Uniform Cyclic Group Factorization of the Rubik's Cube Group

Rubik's Cube consists of a number of pieces called *cubies*. A face of a cubie is called a *facelet*. There are 6 center cubies, 8 corner cubies, and 12 edge cubies in Rubik's Cube. By fixing the center cubies, any instance can be regarded as a permutation of the 48 facelets, and thus the Rubik's Cube group R is a subgroup of S_{48} . It is generated by the following permutations:

- $F := (17, 19, 24, 22)(18, 21, 23, 20)(6, 25, 43, 16)(7, 28, 42, 13)(8, 30, 41, 11)$;
- $B := (33, 35, 40, 38)(34, 37, 39, 36)(3, 9, 46, 32)(2, 12, 47, 29)(1, 14, 48, 27)$;
- $U := (1, 3, 8, 6)(2, 5, 7, 4)(9, 33, 25, 17)(10, 34, 26, 18)(11, 35, 27, 19)$;
- $D := (41, 43, 48, 46)(42, 45, 47, 44)(14, 22, 30, 38)(15, 23, 31, 39)(16, 24, 32, 40)$;
- $L := (9, 11, 16, 14)(10, 13, 15, 12)(1, 17, 41, 40)(4, 20, 44, 37)(6, 22, 46, 35)$;
- $R := (25, 27, 32, 30)(26, 29, 31, 28)(3, 38, 43, 19)(5, 36, 45, 21)(8, 33, 48, 24)$.

- The subgroup A_{12} of R is given by:

$$A_{12} = \{e(\pi) \mid \pi \in A_{12} \leq S_{12}\}.$$

It represents the group of even permutations over 12 edge cubies. A UCF of A_{12} is given by $A_{12} = H_{28}H_{29} \cdots H_{42}$, where $H_{27+i} = \langle e(\sigma'_i) \rangle$ for $1 \leq i \leq 15$ and σ'_i is a generator of the UCF of A_{12} given in Section 4.1.

- The subgroup Z_2 of R is given by:

$$Z_2 = \langle e((1, 2))c((1, 2)) \rangle.$$

It is a group generated by the composition of exchanging two corner cubies and exchanging two edge cubies. Since Z_2 is a cyclic group, it has a trivial UCF $H_{43} := Z_2$.

From the above and Lemma 2, a UCF of R is given by $R = H_1H_2 \cdots H_{43}$ as follows:

- $H_1 = \langle (2, 34)(4, 10) \rangle;$
- $H_2 = \langle (2, 34)(5, 26) \rangle;$
- $H_3 = \langle (2, 34)(7, 18) \rangle;$
- $H_4 = \langle (2, 34)(12, 37) \rangle;$
- $H_5 = \langle (2, 34)(13, 20) \rangle;$
- $H_6 = \langle (2, 34)(15, 44) \rangle;$
- $H_7 = \langle (2, 34)(21, 28) \rangle;$
- $H_8 = \langle (2, 34)(23, 42) \rangle;$
- $H_9 = \langle (2, 34)(29, 36) \rangle;$
- $H_{10} = \langle (2, 34)(31, 45) \rangle;$
- $H_{11} = \langle (2, 34)(39, 47) \rangle;$
- $H_{12} = \langle (1, 9, 35)(3, 27, 33) \rangle;$
- $H_{13} = \langle (1, 9, 35)(6, 11, 17) \rangle;$
- $H_{14} = \langle (1, 9, 35)(8, 19, 25) \rangle;$
- $H_{15} = \langle (1, 9, 35)(14, 40, 46) \rangle;$
- $H_{16} = \langle (1, 9, 35)(16, 41, 22) \rangle;$
- $H_{17} = \langle (1, 9, 35)(24, 43, 30) \rangle;$
- $H_{18} = \langle (1, 9, 35)(32, 48, 38) \rangle;$
- $H_{19} = \langle (1, 3, 6)(35, 27, 11)(9, 33, 17) \rangle;$
- $H_{20} = \langle (1, 6)(35, 11)(9, 17)(3, 8)(27, 19)(33, 25) \rangle;$
- $H_{21} = \langle (1, 3)(35, 27)(9, 33)(6, 8)(11, 19)(17, 25) \rangle;$
- $H_{22} = \langle (1, 3, 6, 8, 41)(35, 27, 11, 19, 22)(9, 33, 17, 25, 16) \rangle;$
- $H_{23} = \langle (1, 8)(35, 19)(9, 25)(6, 43)(11, 30)(17, 24) \rangle;$
- $H_{24} = \langle (1, 3, 6)(35, 27, 11)(9, 33, 17)(8, 41, 43)(19, 22, 30)(25, 16, 24) \rangle;$
- $H_{25} = \langle (1, 3, 6, 8, 41, 43, 46)(35, 27, 11, 19, 22, 30, 14)(9, 33, 17, 25, 16, 24, 40) \rangle;$
- $H_{26} = \langle (1, 41)(35, 22)(9, 16)(8, 48)(19, 38)(25, 32) \rangle;$
- $H_{27} = \langle (1, 3, 6, 8)(35, 27, 11, 19)(9, 33, 17, 25)(41, 43, 46, 48)(22, 30, 14, 38)(16, 24, 40, 32) \rangle;$
- $H_{28} = \langle (2, 4, 5)(34, 10, 26) \rangle;$
- $H_{29} = \langle (2, 5)(34, 26)(4, 7)(10, 18) \rangle;$
- $H_{30} = \langle (2, 4)(34, 10)(5, 7)(26, 18) \rangle;$
- $H_{31} = \langle (2, 4, 5, 7, 12)(34, 10, 26, 18, 37) \rangle;$
- $H_{32} = \langle (2, 7)(34, 18)(5, 20)(26, 13) \rangle;$
- $H_{33} = \langle (2, 4, 5)(34, 10, 26)(7, 12, 20)(18, 37, 13) \rangle;$
- $H_{34} = \langle (2, 4, 5, 7, 12, 20, 44)(34, 10, 26, 18, 37, 13, 15) \rangle;$
- $H_{35} = \langle (2, 12)(34, 37)(7, 28)(18, 21) \rangle;$

30:12 How to Covertly and Uniformly Scramble the 15 Puzzle and Rubik's Cube

- $H_{36} = \langle (2, 4, 5, 7)(34, 10, 26, 18)(12, 20, 44, 28)(37, 13, 15, 21) \rangle;$
- $H_{37} = \langle (2, 4, 5, 7, 12, 20, 44, 28, 42)(34, 10, 26, 18, 37, 13, 15, 21, 23) \rangle;$
- $H_{38} = \langle (2, 20)(34, 13)(12, 36)(37, 29) \rangle;$
- $H_{39} = \langle (2, 4, 5, 7, 12)(34, 10, 26, 18, 37)(20, 44, 28, 42, 36)(13, 15, 21, 23, 29) \rangle;$
- $H_{40} = \langle (2, 4, 5, 7, 12, 20, 44, 28, 42, 36, 45)(34, 10, 26, 18, 37, 13, 15, 21, 23, 29, 31) \rangle;$
- $H_{41} = \langle (2, 44)(34, 15)(20, 47)(13, 39) \rangle;$
- $H_{42} = \langle (2, 4, 5, 7, 12, 20)(34, 10, 26, 18, 37, 13)(44, 28, 42, 36, 45, 47)(15, 21, 23, 29, 31, 39) \rangle;$
- $H_{43} = \langle (2, 4)(34, 10)(1, 3)(35, 27)(9, 33) \rangle.$

5.2 Our Protocol for Rubik's Cube in the Free Permutation Model

In this section, we construct a secure random instance generation protocol of Rubik's Cube in the free permutation model. It requires color stickers along with a cube.

A *color sticker* is a sticker with one of the six colors as follows:



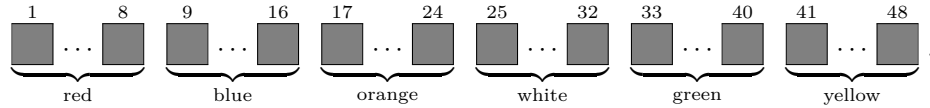
Note that each letter (R, B, O, W, G, and Y) is indicated for clarity and is not actually written. They are the same size as the facelet of the cube and can be placed on facelets. At the beginning, all stickers are concealed by *films* that can be peeled off as follows:



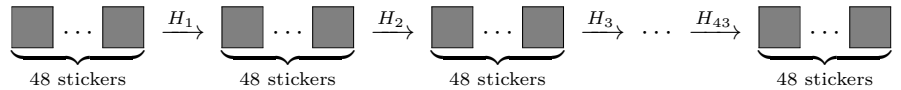
All stickers are assumed to be indistinguishable when their films are not peeled off.

Our protocol requires 48 color stickers, consisting of 8 color stickers of each color, and 43 practical shuffles, consisting of 22 random bisection cuts and 21 pile-shifting shuffles. The protocol proceeds as follows:

1. Arrange a sequence of color stickers as follows:



2. Apply an H_i -shuffle to the sequence for $i = 1, 2, \dots, 43$ as follows:



3. Place the i -th color sticker to the i -th facelet of Rubik's Cube for $1 \leq i \leq 48$. The cube concealed by films is a random instance of Rubik's Cube.

5.3 Uniform Cyclic Group Factorization with the Standard Generators

For each $O \in \{F, B, U, D, L, R\}$, we denote O^3 by O' . All generators of the UCF in Section 5.1 can be expressed by a product of the standard generators F, B, U, D, L, and R. For example, $(2, 34)(4, 10) = \text{RLFU}^2\text{F}'\text{RL}'\text{UB}^2\text{U}'\text{F}^2\text{L}^2\text{U}'\text{F}^2\text{R}^2\text{B}^2\text{D}'$. All generators of the subgroups H_i ($1 \leq i \leq 43$) are expressed by a product of the standard generators as follows:

- $H_1 = \langle \text{RLFU}^2\text{F}'\text{RL}'\text{UB}^2\text{U}'\text{F}^2\text{L}^2\text{U}'\text{F}^2\text{R}^2\text{B}^2\text{D}' \rangle;$
- $H_2 = \langle \text{RUR}'\text{U}'\text{R}'\text{U}'\text{RURB}'\text{U}'\text{R}^2\text{URB} \rangle;$
- $H_3 = \langle \text{U}^2\text{LFUF}'\text{UL}'\text{B}'\text{R}'\text{U}^2\text{RBU}'\text{LU}'\text{L}' \rangle;$
- $H_4 = \langle \text{LBDL}^2\text{D}'\text{L}'\text{BD}^2\text{R}^2\text{F}^2\text{U}^2\text{F}^2\text{R}^2\text{DB}^2\text{D} \rangle;$

- $H_5 = \langle LB'D'B^2DLBDL^2DF^2R^2U^2R^2F^2D^2 \rangle;$
- $H_6 = \langle RL'FU^2F'R'LDR^2U^2R^2F^2R^2U^2R^2D' \rangle;$
- $H_7 = \langle RB'D'R^2DRB'U^2F^2D'L^2DF^2UB^2U \rangle;$
- $H_8 = \langle URLF'R^2F'RL'UR^2U^2B^2L^2F^2DL^2B^2U' \rangle;$
- $H_9 = \langle UBU'B'U'B'UBUR'B'U^2BUR \rangle;$
- $H_{10} = \langle RLF'U^2FRL'U^2R^2L^2DF^2D'R^2L^2U^2R^2 \rangle;$
- $H_{11} = \langle B^2LUBU'BL'D'R'B^2RDB'LB'L' \rangle;$
- $H_{12} = \langle R^2U'F^2UB^2U'F^2UB^2DR'URD'R'U'R' \rangle;$
- $H_{13} = \langle L^2D'L^2DF^2U'FUL^2UL^2U'F \rangle;$
- $H_{14} = \langle F^2U'F^2U'R^2DR^2DB^2D^2FDB^2D'F'U^2 \rangle;$
- $H_{15} = \langle R^2D'B^2U'F^2UB^2U'F^2UR'URDR'U'R' \rangle;$
- $H_{16} = \langle L^2D'B^2DB^2U'L^2UL^2D^2RD'L^2DR'D^2 \rangle;$
- $H_{17} = \langle R^2UF^2U'L^2UF^2U'R^2UFD'B^2DF'U' \rangle;$
- $H_{18} = \langle B^2DB^2D'R^2UR^2U'B^2U^2F'UB^2U'FU^2 \rangle;$
- $H_{19} = \langle R^2DB^2D'F^2DB^2D'F^2R^2 \rangle;$
- $H_{20} = \langle RBR'F^2RB'RB^2U'F^2DL^2B^2D'F^2UF^2 \rangle;$
- $H_{21} = \langle RU^2R^2DR^2U^2RB^2UR^2UR^2U^2B^2D'R^2 \rangle;$
- $H_{22} = \langle UR^2UF^2U^2F^2U'R^2U'LR'F^2L'R \rangle;$
- $H_{23} = \langle URLD^2R'L'F^2U'L^2U'B^2UL^2DR^2D' \rangle;$
- $H_{24} = \langle U^2F^2R^2F^2R^2UR^2F^2R^2F^2U \rangle;$
- $H_{25} = \langle L^2UR'LF^2R'L'F^2UB^2L^2D^2R^2DF^2DB^2 \rangle;$
- $H_{26} = \langle RLU^2RL'U^2R^2F^2U^2B^2D^2R^2B^2U^2L^2 \rangle;$
- $H_{27} = \langle URLU^2D^2RLU'F^2R^2UF^2D^2R^2UB^2L^2 \rangle;$
- $H_{28} = \langle R^2U^2F^2L^2B^2DB^2L^2F^2U'R^2 \rangle;$
- $H_{29} = \langle URL'U^2R'LUF^2UF^2UF^2UF^2UF^2 \rangle;$
- $H_{30} = \langle UFB'U^2F'BUL^2UL^2UL^2UL^2UL^2 \rangle;$
- $H_{31} = \langle L^2FL^2U^2L^2FUL^2B^2R^2U'R^2UDR^2B^2 \rangle;$
- $H_{32} = \langle R'U^2F^2U^2RU^2F^2R^2U^2F^2U^2F^2R^2U^2F^2 \rangle;$
- $H_{33} = \langle L^2U^2F^2R^2B^2U'L^2B^2R^2UFUF'D'ULU'L \rangle;$
- $H_{34} = \langle B^2U'B^2L^2UL^2B^2L^2D'U^2L'B'LDL^2BLB \rangle;$
- $H_{35} = \langle R^2F^2U^2FU^2D^2BU^2D^2F^2D^2B^2L^2U^2D^2 \rangle;$
- $H_{36} = \langle DUB^2L^2R^2D^2F^2UL^2DBDF'L'FLBR^2F' \rangle;$
- $H_{37} = \langle U^2L^2DR^2F^2R^2UF^2R^2U^2FD'LR'FL'FL'RU' \rangle;$
- $H_{38} = \langle RDBD'R'U'L'B'LUBU'DL'D'U \rangle;$
- $H_{39} = \langle R^2D'R^2UF^2L^2D'B^2R^2DFL'U'B^2UL'R^2BR^2U^2 \rangle;$
- $H_{40} = \langle L^2B^2L^2D^2UR^2U'B^2U^2B'L^2D'U'B^2R'D^2R^2U'F^2U' \rangle;$
- $H_{41} = \langle D'RU^2F^2U^2R'F^2U^2L^2B^2U^2F^2D^2R^2B^2D' \rangle;$
- $H_{42} = \langle DF^2U'F^2L^2U^2F^2U^2F^2UFR^2U'L^2R^2UF^2RF^2U' \rangle;$
- $H_{43} = \langle UR^2U'R^2DR^2D'F^2UF^2R^2 \rangle.$

We compute them by Online Rubik's Cube Solver [24]. We remark that all of the generators as above are of length at most 20, which is God's Number of Rubik's Cube.

5.4 Our Protocol for Rubik's Cube in the Restricted Permutation Model

In this section, we construct a secure random instance generation protocol of Rubik's Cube in the restricted permutation model.

Our protocol uses a *piece of cloth*, which is sufficiently large to hide a cube completely. A player can hold a cube through his/her hand from under the cloth, and can apply a sequence of the standard generators. During this operation, which permutation is applied to the cube is completely hidden from other players.

Now we introduce a repetitive shuffle, which is a shuffling operation for a cube. Let $\text{seq} \in \{F, B, U, D, L, R\}^*$ be a sequence of the standard generators such that the order of seq is k , i.e., applying seq k times is equal to the identity permutation. For a cube covered with a piece of cloth, a *repetitive shuffle* of seq covertly applies seq^r to the cube for a uniformly random number $r \in \{0, 1, \dots, k-1\}$. Here, r is completely hidden from all players.

We give two physical implementations of repetitive shuffles. The first method is performed by a single player: The player holding a cube through his hand from under the cloth applies seq a sufficiently large number of times until he loses the number of times. The second method is performed by (at least) two players: The first player holding a cube through his hand from under the cloth applies seq^{r_1} to the cube for a uniformly random number $r_1 \in \{0, 1, \dots, k-1\}$ which is generated by his mind; Then the cube is passed to the second player, and she applies seq^{r_2} to the cube for a uniformly random number $r_2 \in \{0, 1, \dots, k-1\}$ which is generated by her mind. Since $r := r_1 + r_2 \bmod k$ is distributed uniformly at random and r is completely hidden from the both players, the second method correctly implements the repetitive shuffle.

Our protocol requires 43 repetitive shuffles. The protocol proceeds as follows:

1. Prepare a solved cube. Cover it with a piece of cloth.
2. Apply 43 repetitive shuffles of the generators of cyclic groups given in Section 5.3. The cube covered with a piece of cloth is a random instance of Rubik's Cube.

6 Conclusion

In this paper, we introduced the secure random instance generation problem for the first time and designed three protocols for the 15 Puzzle and Rubik's Cube. We left as an open problem to achieve the same task with a smaller number of practical shuffles. Note that our protocols are based on UCFs with multiplicity 1, but it may be possible to construct a shorter UCF with multiplicity 2 or more. Another research direction is to design a secure random instance generation protocol for other combination puzzles.

References

- 1 Adrian Brünger, Ambros Marzetta, K. Fukuda, and Jürg Nievergelt. The parallel search bench ZRAM and its applications. *Ann. Oper. Res.*, 90:45–63, 1999.
- 2 Yang Chu and Robert Hough. Solution of the 15 puzzle problem, 2019. [arXiv:1908.07106](https://arxiv.org/abs/1908.07106).
- 3 Claude Crépeau and Joe Kilian. Discreet solitary games. In *Advances in Cryptology—CRYPTO'93*, volume 773 of *LNCS*, pages 319–330. Springer, 1994.
- 4 Erik D. Demaine, Sarah Eisenstat, and Mikhail Rudoy. Solving the rubik's cube optimally is NP-complete. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018*, volume 96 of *LIPIcs*, pages 24:1–24:13, 2018.
- 5 Bert Den Boer. More efficient match-making and satisfiability the five card trick. In *EUROCRYPT 1989*, volume 434 of *LNCS*, pages 208–217. Springer, 1990.

- 6 Yuji Hashimoto, Koji Nuida, Kazumasa Shinagawa, Masaki Inamura, and Goichiro Hanaoka. Toward finite-runtime card-based protocol for generating a hidden random permutation without fixed points. *IEICE Trans. Fundam.*, E101.A(9):1503–1511, 2018.
- 7 Yuji Hashimoto, Kazumasa Shinagawa, Koji Nuida, Masaki Inamura, and Goichiro Hanaoka. Secure grouping protocol using a deck of cards. In *Information Theoretic Security*, volume 10681 of *LNCS*, pages 135–152. Springer, 2017.
- 8 Takuya Ibaraki and Yoshifumi Manabe. A more efficient card-based protocol for generating a random permutation without fixed points. In *Mathematics and Computers in Sciences and in Industry (MCSI)*, pages 252–257, 2016.
- 9 Rie Ishikawa, Eikoh Chida, and Takaaki Mizuki. Efficient card-based protocols for generating a hidden random permutation without fixed points. In *Unconventional Computation and Natural Computation*, volume 9252 of *LNCS*, pages 215–226. Springer, 2015.
- 10 Kazuki Kanai, Kengo Miyamoto, Koji Nuida, and Kazumasa Shinagawa. Uniform cyclic group factorizations of finite groups. *Communications in Algebra*, 2023.
- 11 Alexander Koch and Stefan Walzer. Foundations for actively secure card-based cryptography. In *Fun with Algorithms*, volume 157 of *LIPICs*, pages 17:1–17:23, 2020.
- 12 Kengo Miyamoto and Kazumasa Shinagawa. Graph automorphism shuffles from pile-scramble shuffles. *New Gener. Comput.*, 40:199–223, 2022.
- 13 Takaaki Mizuki, Yoshinori Kugimoto, and Hideaki Sone. Secure multiparty computations using the 15 puzzle. In *Combinatorial Optimization and Applications*, volume 4616 of *LNCS*, pages 255–266. Springer, 2007.
- 14 Takaaki Mizuki and Hiroki Shizuya. A formalization of card-based cryptographic protocols via abstract machine. *Int. J. Inf. Secur.*, 13(1):15–23, 2014.
- 15 Takaaki Mizuki and Hideaki Sone. Six-card secure AND and four-card secure XOR. In *Frontiers in Algorithmics*, volume 5598 of *LNCS*, pages 358–369. Springer, 2009.
- 16 Bruce Norskog and Morley Davidson. The fifteen puzzle can be solved in 43 “moves”, 2010. URL: <http://cubezzz.duckdns.org/drupal/?q=node/view/223>.
- 17 Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the $N \times N$ extension of the 15-puzzle is intractable. In *Proceedings of the 5th National Conference on Artificial Intelligence. Volume 1: Science*, pages 168–172, 1986.
- 18 Daniel Ratner and Manfred K. Warmuth. $N \times N$ puzzle and related relocation problem. *J. Symb. Comput.*, 10(2):111–138, 1990.
- 19 Tomas Rokicki, Palo Alto, Herbert Kociemba, Morley Davidson, and John Dethridge. God’s number is 20. URL: <https://www.cube20.org/>.
- 20 Takahiro Saito, Daiki Miyahara, Yuta Abe, Takaaki Mizuki, and Hiroki Shizuya. How to implement a non-uniform or non-closed shuffle. In *Theory and Practice of Natural Computing*, volume 12494 of *LNCS*, pages 107–118. Springer, 2020.
- 21 Kazumasa Shinagawa and Kengo Miyamoto. Automorphism shuffles for graphs and hypergraphs and its applications. *IEICE Trans. Fundam.*, E106.A(3):306–314, 2023.
- 22 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 *Provable Security*, volume 9451 of *LNCS*, pages 127–146. Springer, 2015.
- 23 Itaru Ueda, Daiki Miyahara, Akihiro Nishimura, Yu-ichi Hayashi, Takaaki Mizuki, and Hideaki Sone. Secure implementations of a random bisection cut. *Int. J. Inf. Secur.*, 19(4):445–452, 2020.
- 24 Online rubik’s cube solver. URL: <https://rubiks-cube-solver.com/>.

A Programming Language Embedded in *Magic: The Gathering*

Howe Choong Yin ✉

Independent Researcher, Singapore

Alex Churchill ✉ 

Independent Researcher, Cambridge, UK

Abstract

Previous work demonstrated that the trading card game *Magic: The Gathering* is Turing complete, by embedding a universal Turing machine inside the game. However, this is extremely hard to program, and known programs are extremely inefficient. We demonstrate techniques for disabling *Magic* cards except when certain conditions are met, and use them to build a microcontroller with a versatile programming language embedded within a *Magic* game state. We remove all choices made by players, forcing all player moves except when a program instruction asks a player for input. This demonstrates *Magic* to be at least as complex as any two-player perfect knowledge game, which we demonstrate by supplying sample programs for Nim and the Collatz conjecture embedded in *Magic*. As with previous work, our result applies to how real *Magic* is played, and can be achieved using a tournament-legal deck; but the execution is far faster than previous constructions, generally one cycle of game turns per program instruction.

2012 ACM Subject Classification Theory of computation → Representations of games and their complexity

Keywords and phrases Programming, computability theory, *Magic: the Gathering*, two-player games, tabletop games

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.31

Related Version *Full Version*:

https://www.toothycat.net/~hologram/Magic/Magic_Microcontroller_full.pdf [5]

Supplementary Material *InteractiveResource (Magic Microcontroller Simulator)*:

<https://www.toothycat.net/~hologram/Magic/MTGProgSimulatorText.html> [3]

1 Introduction and Previous Work

Magic: The Gathering (also known as *Magic*) is the world’s largest tabletop collectible card game, played by hundreds of thousands of players in tournaments and by millions more players casually. In 2020, Churchill, Biderman & Herrick published an embedding of a universal Turing machine inside *Magic* [2]. This is the first widely played tabletop game to be shown Turing complete in the format in which it is usually played, as opposed to some infinite generalisation. For example, chess is EXPTIME-complete with infinite board and pieces, but has a finite number of states in the 8x8 board used for tournament play. Churchill et al. showed that the question “will this *Magic* game ever terminate” cannot be answered in the general case, even for two-player *Magic* played with all the usual tournament restrictions. However, this paper did not contain any concrete example computations.

The author of [1] investigated the runtime performance of this Universal Turing Machine (UTM) embedded within *Magic*. He established a compilation sequence from an arbitrary Turing machine with N states, into a 2-tag system, into the 2-state UTM(2,18), and thence into *Magic*. He supplied a simple Turing machine to compute $2+3$ in a unary adder. However, he found that no simulation was able to establish how long it would take to compute $2+3$



© Howe Choong Yin and Alex Churchill;
licensed under Creative Commons License CC-BY 4.0
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 31; pp. 31:1–31:19

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

in the UTM. The straightforward compiler's output from this simple Turing machine for computing $2+3$ results in a tape over 40 million symbols long (15 million in the program and 25 million in the data). The UTM simulation of the Turing machine needs to constantly move between the program and data sections, resulting in absurdly inefficient computation times. No simulation was able to establish how long it would take to compute $2+3$ in the *Magic* game.

The same author created an optimised unary adder Turing machine with just 2 states, and compiled a simplified computation of $1+1$. This completed in a mere 3,958,876,878 game cycles (of each of the two players taking a turn). Even after nearly 8 billion game turns, because of the multiple steps in translating the calculation into the UTM(2,18), the output consisted of hundreds of creature tokens, which needed to be interpreted by carefully counting how many tokens of type Myr were mixed in among the hundreds of tokens with type Aetherborn, all in order to retrieve the output of "2".

In this paper we set out a different construction, which embeds a full microcontroller within *Magic: The Gathering*. In Section 2 we describe at a high level some key features of the construction, and Section 3 specifies the programming language supported by the microcontroller. Section 4 provides the details of we implement the framework, and we work through an example instruction in Section 5. Section 6 contains our conclusions and discussion of the implications. Finally, Appendix A provides several sample programs, Appendix B explains which cards we use to modify other cards, and Appendix C supplies a decklist that could be brought to a *Magic* tournament to assemble the construction.

2 Outline of the Construction

The rest of this paper describes a construction that simulates a fully general programming language within *Magic: The Gathering*. Compared to [2] this is also Turing complete, but is much more efficient and easy to program, and reports its outputs much more clearly. It also allows reading input from each of the players of the two-player *Magic* game in which it is embedded, and can be programmed to terminate in a win for either player or a draw, or of course can keep running indefinitely.

As with the Turing machine construction, we start by assuming one player, Alice, draws a combination of cards that allows her to take over the game, draw all the rest of her deck, and remove all cards from the hand of the opponent, Bob. After the initial setup is completed, she removes all player choices, so that neither player has any option but to let the program execution continue, short of conceding the game. Thus the outcome of this tournament-legal *Magic* game is entirely determined by the result of the program.

The program is written in a language of 12 symbols, represented by basic land cards, which are allowed to occur any number of times in a player's deck. Alice's deck needs to start off containing a lot of other cards, but once the microcontroller is set up and she has drawn all her cards, she returns to her deck a sequence of cards from among these 12, which encodes the program to be executed.

The program is read one card at a time. During Alice's combat step we put one of these cards, which we call the "program permanent", onto the battlefield for long enough to read it, then move it to the bottom of Alice's deck. We in fact make Alice have three combat steps each turn, and in each one, a program permanent is read and possibly other game actions are (automatically) taken. Each instruction in the programming language is a sequence of three symbols, interpreted by some set of "instruction permanents" that do extra things during Alice's or Bob's turn.

All instruction permanents that don't apply to the current instruction are "inactive", i.e. have all their abilities removed. To do this we turn them into creatures with a certain creature type (e.g. Angel), have a card which makes all Angels also Saprolings, make all Saprolings lands, and remove abilities from all lands. Crucially, we can conditionally allow a certain group of instruction permanents to regain their abilities by temporarily "phasing out" the card which makes (e.g.) Angels into Saprolings. (Objects in *Magic* which are "phased out" are treated by the rules as if they don't exist.) We do this during one of Alice's combat steps, so those instruction permanents have their abilities during the rest of Alice's turn (including her later combat steps) and all of the other player Bob's turn.

Permanents in *Magic* have zero or more colours, from the set of white, blue, black, red and green. Permanents that are creatures have zero or more creature types, also drawn from a well-defined set, but this set has over 200 types in it. We use both of these characteristics extensively. All "inactive" permanents are made green Saproling creatures, and all green creatures are given protection from certain creature types. We make extensive use of the capabilities *Magic* offers to edit existing cards by changing colour words (using the card **Mind Bend**), creature type text (using **Artificial Evolution**), colours (using **Prismatic Lace**), and creature types (using a variety of cards according to circumstances).

In particular, we carefully apply restrictions so that any time an ability triggers that would normally let its controller choose a target, there is precisely one legal target (or occasionally no targets, which prevents the ability from going on the stack at all). We ensure that all creatures attack and block where possible using **Grand Melee**, but make most creatures unable to attack or block using **Stormtide Leviathan**. Any time a creature is able (and thus forced) to attack, we arrange that either it can't be blocked at all, or there is precisely one creature forced to block it.

3 The Programming Language

The programming language we implement has the following features:

- Twelve registers $r_0 \dots r_{11}$, each able to contain an arbitrarily large nonnegative integer.
- An unlimited number of memory slots, each addressed by a nonnegative integer address; each memory slot can hold a single arbitrarily large nonnegative integer.
- A single Boolean flag that is set by certain instructions such as comparisons. The flag can be read by certain instructions; most notably, jump instructions can be made conditional on whether the flag is true or false.

The program is written in a language of 12 symbols, and each instruction is a sequence of three symbols. For example, the sequence 0 1 2 (represented by cards Plains Island Swamp) encodes the instruction "Add 1 2", which will result in increasing the value of register r_1 by the value of r_2 . We provide the following instructions in the language:

5 Y Z ($Y \neq Z$)	Set $r_Y r_Z$	Set r_Y to the value of r_Z .
5 Y Y	Zero r_Y	Set r_Y to zero.
0 Y Z	Add $r_Y r_Z$	Set r_Y to $r_Y + r_Z$.
4 1 Z	Add1 r_Z	Set r_Z to $r_Z + 1$.
4 2 Z	Halve r_Z	Set r_Z to half the value of r_Z , rounding down. Set the flag to the remainder from the division.
1 Y Z ($Y \neq Z$)	SubCond $r_Z r_Y$	If $r_Z \geq r_Y$, set r_Z to $r_Z - r_Y$ and set the flag to 0. Otherwise, set the flag to 1.
1 Z Z	Sub1Cond r_Z	If $r_Z \geq 1$, set r_Z to $r_Z - 1$ and set the flag to 0. Otherwise, set the flag to 1.
6 Y Z	Mult $r_Z r_Y$	Set r_Z (note, not r_Y) to $r_Y \times r_Z$.

31:4 A Programming Language Embedded in *Magic: The Gathering*

7 Y Z	DivCeil $r_Y r_Z$	Set r_Y to $\lceil r_Y/r_Z \rceil$. If the division was exact, set the flag to 0, otherwise set it to 1. If $r_Z = 0$ or $Y = Z$, this is undefined behaviour.
2 Y 0	AInput r_Y	Set r_Y to a nonnegative integer of Alice's choice.
2 Y 1	BInput r_Y	Set r_Y to a nonnegative integer of Bob's choice.
2 Y 2	SetF r_Y	Set r_Y to the flag's value.
2 Y 3	SetNF r_Y	Set r_Y to the Boolean negation of the flag's value (1 if it's 0 and vice versa).
2 Y 4	Rand6 r_Y	Set r_Y to a random nonnegative integer less than 6.
2 Y 5	Rand20 r_Y	Set r_Y to a random nonnegative integer less than 20.
10 Y Z	NumBuild $12Y+Z$	Set r_0 to $12Y+Z$, except if the last instruction that was executed was also a NumBuild instruction, in which case multiply r_0 by 144 and add $12Y+Z$ to it. As its name suggests, this instruction can be used repeatedly to build any nonnegative integer value in r_0 , two base-12 digits at a time.
8 Y Z	Store $r_Z r_Y$	Store the value of r_Y at memory address r_Z .
9 Y Z	Load $r_Y r_Z$	Load the value at memory address r_Z into r_Y .
3 0 Z'	JumpFwd Z'	Jump forward by Z' instructions. If $Z = 0$, Z' is r_0 , and otherwise, Z' is Z . Thus instead of a useless command to jump 0 instructions, we gain the ability to jump an arbitrary or computed distance.
3 1 Z'	JumpBwd Z'	Jump backward by Z' instructions. Backwards jumps by more than the length of the program ¹ do nothing.
3 2 Z	JumpFwdNF Z'	Jump forward by Z' instructions if the flag is 0/false.
3 3 Z	JumpBwdNF Z'	Jump backward by Z' instructions if the flag is 0/false.
3 4 Z	JumpFwdF Z'	Jump forward by Z' instructions if the flag is 1/true.
3 5 Z	JumpBwdF Z'	Jump backward by Z' instructions if the flag is 1/true.
3 6 0	CallFwd r_0	Call a function r_0 instructions ahead: Jump forward $3r_0$ cards and push $P - 3r_0$ onto the return stack. P is the length of the program in cards ¹ . If $r_0 = 0$ or $3r_0 > P$, this is undefined behaviour.
3 7 Z	Return Z'	Return from a function Z' instructions long: Pop a value S from the return stack, and jump forward $\max(0, S - 3Z')$ cards. If the return stack was empty, end the game in a draw.
3 6 1	CallBwd r_0	Call a function r_0 instructions behind: Jump backward $3r_0$ cards and push $3r_0$ onto the return stack. If $3r_0 \geq P$, this is undefined behaviour. May not be used to call a function from within itself.
3 6 2	CallBwdR r_0	Call a function r_0 instructions behind (direct-recursion-capable): Same as CallBwd, but push $P + 3r_0$, so that this may be used to call a function from within itself.

¹ If the program is less than 6 cards long, P is the first multiple of the length that is at least 6.

11 Y Z ($Y \neq Z$)	FLess $r_Z r_Y$	Set the flag to 1 if $r_Z < r_Y$, 0 otherwise. Flag-combining.
11 Z Z	FIsZero r_Z	Set the flag to 1 if $r_Z = 0$, 0 otherwise. Flag-combining.
4 0 0	HaltD	End the game in a draw.
4 0 1	HaltA	End the game with Alice winning.
4 0 2	HaltB	End the game with Bob winning.

“Flag-combining” is a property used by some instructions whose only purpose is to set the flag. It means that flag values given by subsequent instructions are combined by logical OR instead of replacing the flag’s value. This state ends when any of the following is executed:

- Any instruction that uses the flag’s value.
- Any jump instruction (including calls and returns, and regardless of whether the jump is taken or not).

For example, this can be used to check whether r_0 and r_1 are equal, with FLess $r_0 r_1$ followed by FLess $r_1 r_0$.

The program is cyclic, which is to say it wraps around: after passing the final instruction (by executing it or jumping past it) execution continues with the first instruction. Similarly you can jump backwards beyond the start of the program and end up near the end of the program.

Encountering a sequence that does not match any of the instructions above is undefined behaviour.

These instructions are a superset of those required for a random-access register machine such as Melzak’s Q-machine, which is Turing complete.[9]

4 Implementation of the Microcontroller

In this section we describe the various gadgets that make up the microcontroller as described in the previous section. We defer discussion of how to set up the board state (including modifying card types, creature types, colours, etc) to Appendix B.

Note: The following explanation is easier to read at [6] or [5], where the same information is presented with interactive tooltips giving reminders of the significance of various creature types, providing card rules text, and indicating which specific card modification techniques from Appendix B are being used.

4.1 The program

The program is a sequence of the following cards: **Plains**, **Island**, **Swamp**, **Mountain**, **Forest**, **Wastes**, **Snow-Covered Plains**, **Snow-Covered Island**, **Snow-Covered Swamp**, **Snow-Covered Mountain**, **Snow-Covered Forest**, and **Snow-Covered Wastes**². We call these cards “symbol cards” and assign them numbers 0, 1, 2, ..., 11 in the order listed.

² To be released on 7th June 2024. To play the Microcontroller before that date, instead use Persistent Petitioners and Infinite Reflection as described at [6].

Most of the time, one of the cards in the program will be on the battlefield under Alice's control; we call this card the *program permanent*. The rest of the cards will be in Alice's library, with the next symbol in the program on top of the library, continuing in program order from top to bottom, then continuing from the start of the program until the symbol before the current symbol.

The program is made up of instructions that each consist of 3 cards. X, Y, and Z shall refer to the numbers of the three cards that form the instruction currently being executed, in that order. For example, the instruction "Add $r_1 r_2$ " is represented by 0 1 2 (Plains Island Swamp), and X is 0, Y is 1 and Z is 2.

4.2 Global environment control

On the battlefield is a **Grand Melee** ("All creatures attack each combat if able. All creatures block each combat if able") and a **Stormtide Leviathan** ("Creatures without flying or islandwalk can't attack"). Most creatures are unable to attack; when we want a creature to be able to attack, we will give it flying or islandwalk. Bob's creatures with islandwalk will be unblockable because Alice controls the program permanent which is given type Island.

Both players have life total 1, and each has a **Worship** keeping their life totals at 1 through the damage they will be dealt. (Note that **Worship** only modifies the *result* of damage; the damage itself is still dealt, so effects triggered on combat damage still trigger.)

4.3 Advancing through the program

Alice has a **Vaevictis Asmadi, the Dire**, whose rules text reads "Whenever Vaevictis Asmadi, the Dire attacks, for each player, choose target permanent that player controls. Those players sacrifice those permanents. Each player who sacrificed a permanent this way reveals the top card of their library, then puts it onto the battlefield if it's a permanent card."

We use the techniques in Appendix B to make **Vaevictis Asmadi, the Dire** into a 1/1 Sliver Beast Reflection. It attacks in Alice's combat phase. Its ability is forced (see subsection 4.6) to target the program permanent for Alice, and one particular permanent for Bob. When it resolves, Alice sacrifices the program permanent, and **Wheel of Sun and Moon** enchanting Alice sends it to the bottom of Alice's library; then the next card (on top of Alice's library) is put onto the battlefield, becoming the new program permanent. Meanwhile, Bob has been given a **Tajuru Preserver** stopping Bob's permanent from being sacrificed, and thus Bob does not reveal the top card of his library.

Alice controls **Tetsuko Umezawa, Fugitive** ensuring that **Vaevictis Asmadi, the Dire**, and all her other creatures with power or toughness 1 or less, can't be blocked.

4.4 Disabling and conditionally enabling permanents

For each of the 12 program cards, letting n be its number, we choose a creature type X_n . X_0 is Aetherborn, X_1 is Beeble, and so on through Cephalid, Drake, Eldrazi, Faerie, Gremlin, Homarid, Illusion, Juggernaut, Kavuu, and Lhurgoyf.

We have Alice control twelve token copies of **Dralnu's Crusade**, whose printed rules text reads "All Goblins get +1/+1. All Goblins are black and are Zombies in addition to their other creature types." However we use the techniques in Appendix B to edit each one to instead to affect creatures with a different one of the 12 creature types X_n ($n \in \{0, 1, \dots, 11\}$), and add type Saproling. **Life and Limb** makes Saprolings into lands, and also makes them green, and then **Blood Sun** removes their abilities. So all creatures with any of the 12 creature types X_i have their abilities removed.

Each of these **Dralnu's Crusades** is made into a creature with creature type Sliver, with power and toughness 2/1, and given flying. For each of the 12 basic land cards, the corresponding **Dralnu's Crusade** is equipped with a **Strata Scythe** imprinted with that basic land card ("Equipped creature gets +1/+1 for each land on the battlefield with the same name as the exiled card").

All twelve **Dralnu's Crusades** are forced to attack, and the one corresponding to the current program permanent will be boosted to 3/2; the previously mentioned **Tetsuko Umezawa, Fugitive** ensures that the others can't be blocked.

Bob controls a **Dream Fighter** ("Whenever Dream Fighter blocks or becomes blocked by a creature, Dream Fighter and that creature phase out") with creature type Sliver and granted reach (so it can block creatures with flying). This blocks the 3/2 **Dralnu's Crusade**, and they both phase out, so that creatures of the type corresponding to the current program permanent are no longer made Saprolings. We have a **Shadow Sliver** ensuring that only Slivers can block these **Dralnu's Crusades**.



■ **Figure 1** After Vaevictis Asmadi, the Dire puts a Wastes onto the battlefield, the only one of Alice's attacking **Dralnu's Crusades** which Bob's **Dream Fighter** can block is the one given +1/+1 by a **Strata Scythe** imprinted with Wastes.

This reads the first card of an instruction; creatures of the corresponding type will regain their abilities after **Dream Fighter's** ability resolves. We also choose a thirteenth creature type (Monkey) denoted X_A , and have another **Dralnu's Crusade** making X_A creatures also Saprolings, but this one is not a creature; rather we make it an artifact so **Bludgeon Brawl** makes it an Equipment, and attach it to Bob's **Dream Fighter**, so it will phase out when the **Dream Fighter** does. This means that creatures of type X_A get to regain their abilities after the first card is read, no matter what it is, but they will lose their abilities again when Bob's turn starts and the **Dream Fighter** phases back in.

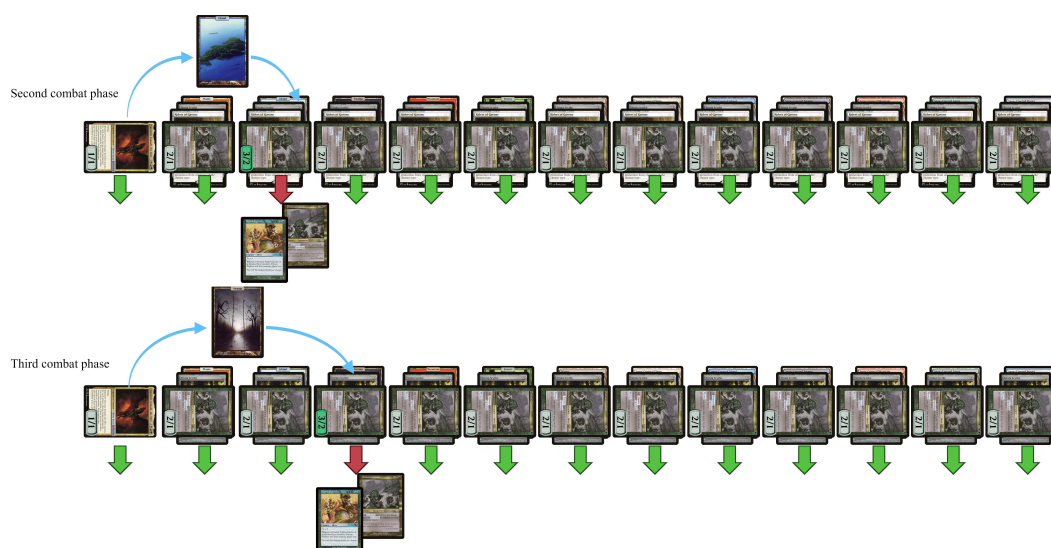
We will also often want to use this conditional mechanism on noncreature permanents; to do that, we make token creature copies of them using **Urza, Prince of Kroog**, combined with **Memnarch** if necessary.

4.5 Reading the second and third cards

Alice controls a **Bloodthirster** ("Whenever Bloodthirster deals combat damage to a player, untap it. After this combat phase, there is an additional combat phase. Bloodthirster can't attack a player it has already attacked this turn.") It is made a 1/1 Sliver Beast Reflection and given double strike. This also attacks, can't be blocked because of **Tetsuko Umezawa, Fugitive**, and adds a second combat phase. The second combat phase is used to read the second card, using another copy of the above setup, but with all the creatures involved granted an additional creature type of X_A so that they do not attack or block in the first combat phase (because they don't have their abilities at that point). So there is a second

Vaevictis Asmadi, the Dire to advance through the program, given types Sliver Beast Reflection X_A ; a second **Dream Fighter** with reach and types Sliver X_A ; and another batch of **Dralnu's Crusades** for another 13 creature types Y_0, Y_1, \dots, Y_{11} , and Y_A (Antelope, Basilisk, Camel, Dauthi, Efreet, Fox, Gnome, Hippo, Inkling, Jellyfish, Kor, Lammasu, and Metathran), the first 12 having **Strata Scythes**.

Similarly, since the **Bloodthirster** has double strike, it will also give Alice a third combat phase, which is used by another copy of the setup (with creature type Y_A) to make another 13 creature types Z_0, Z_1, \dots, Z_{11} , and Z_A conditional on the third card of the instruction (Aurochs, Brushwagg, Camarid, Druid, Elephant, Ferret, Graveborn, Hamster, Imp, Jackal, Kithkin, Ligid, and Masticore).



■ **Figure 2** More Dralnu's Crusades attack in the second and third combat phases, and another of each is blocked and phases out. They couldn't attack before because of their types X_A or Y_A .

4.6 Constraining targets

Almost all creatures will be green; any creature that isn't naturally green and isn't specified to be differently coloured is made green by **Prismatic Lace**. A few creatures will be red or blue instead, but even those are made green while inactive by **Life and Limb**.

A **Masked Gorgon** edited to give green and blue creatures protection from Reflections means that only red creatures are legal targets for Reflections' abilities. Similarly, a **Masked Gorgon** edited to give green and red creatures protection from Beasts means that only blue creatures are legal targets for Beasts' abilities.

We will refer to the creature types Reflection and Beast as tR and tU respectively, indicating their function. ("U" is the usual abbreviation for "blue" in *Magic*.)

As mentioned earlier, each **Vaevictis Asmadi, the Dire** has been made both a Beast and a Reflection, so that green creatures, blue creatures, and red creatures are all illegal targets for it; the intended target under Alice's control is a noncreature land. The same types are applied to the **Bloodthirster** for a different purpose, to stop it from blocking on Bob's turn (which is something else protection does), because it untaps itself and stays untapped (as its own ability prevents it attacking in the later combat phases).

Spectral Guardian makes noncreature artifacts illegal targets for anything. Alice and Bob both control a **Sterling Grove** to make other enchantments illegal targets for anything. Alice's is made an artifact so it gains shroud from **Spectral Guardian**. Bob's one does not itself have shroud, and thus it is the only legal target under Bob's control for **Vaevictis Asmadi, the Dire**, but Bob has a **Tajuru Preserver** so he does not sacrifice anything. We do however give Bob's **Sterling Grove** protection from blue, which will be useful later to stop some other things from targeting it.

Alice and Bob both have **Ivory Mask**, making both players illegal targets for anything.

4.7 Order of continuous effects, and one more of them

The *Magic Comprehensive Rules* [10] specify a system of “*layers*” for working out what happens when multiple effects apply to the same permanent. For example, effects that make one permanent a copy of another object apply in layer 1. All effects that change a permanent's type (such as creature, land, etc) or subtype (Angel, Goblin, etc) apply in layer 4. Anything that adds or removes abilities applies in layer 6, and so on. Within a layer, if multiple effects try to affect the same permanent, each object or effect has a “*timestamp*”, generally when that object or effect was created. Within this document, we denote timestamps with circled numbers: an effect with timestamp ① will take effect earlier than timestamp ②.

The continuous effects mentioned so far are timestamped as follows:

- ①: **Stormtide Leviathan**
- ②: **Dralnu's Crusades** and **Blood Sun**
- ③: **Life and Limb** and protection-granting and shroud-granting effects

Any other continuous effect is timestamped ① unless otherwise stated.

The exception to timestamp order is “*dependency*”: if two effects would apply within the same layer, but one will change the existence of the second or which objects the second acts on or what it does to them, the first one applies first even if the second has an earlier timestamp. This applies to our construction where abilities that will be removed by the **Blood Sun** wait for it to be applied (and thus end up not being applied themselves).

The program permanent is a land so it is granted type Island by the **Stormtide Leviathan**, but it may be a Forest as well. We do not want **Life and Limb** to make the program permanent a creature. So we also have Alice control an **Illusionary Terrain**, made a creature with type Z_A , within timestamp ① after the **Stormtide Leviathan**, turning all Islands to Islands. This is not as ineffective as it sounds: rule 305.7 [10] says that this removes all other types, so that the program permanent's subtype is set to Island and no others. Note that, because there are no Saprolings before ②, the **Stormtide Leviathan** does not have a dependency on the **Life and Limb**.

By being made a Z_A , this **Illusionary Terrain** has its abilities removed by **Blood Sun** most of the time, most importantly during Alice's upkeep. Because the effect of setting basic lands' types is applied in layer 4, before the abilities are removed in layer 6, it still functions despite the abilities being removed. However, the removal of abilities does shut off its cumulative upkeep.

4.8 Registers

There are twelve registers named r_0, r_1, \dots, r_{11} , each of which holds a nonnegative integer value.

31:10 A Programming Language Embedded in *Magic: The Gathering*

Each register r_n is a token copy of **Joraga Warcaller** under Bob’s control, given copiable creature type Z_n and additionally given creature type Rabbit, given indestructible and vigilance, coloured red at timestamp ④, with base power and toughness noncopiously set to 2/2 but with two -1/-0 counters, and a number of +1/+1 counters on it equal to its register value. (As always, see Appendix B or the tooltips in [6] for how all these changes are accomplished.) **Joraga Warcaller**’s rules text says “Other Elf creatures you control get +1/+1 for each +1/+1 counter on Joraga Warcaller”. Thus, after Z is read from the program, the active register r_Z regains its ability and adds its value to the power and toughness of each other Elf that Bob controls. (This does not include the other registers; their Elf type is overwritten.)

Note that each register’s power is equal to its value, whether it is r_Z (having base power 2 and two -1/-0 counters) or not (having base power 1 from the **Life and Limb**, +1 power from a **Dralnu’s Crusade**, and two -1/-0 counters).

All the registers are made red by **Prismatic Lace** at timestamp ④, later than that of the **Life and Limb**, so that they are always red even when inactive. But the inactive registers are still Saprolings even though they’re red.

r_0 also has Rhino added to its creature types; this will be useful for some instructions that use specifically this register.



■ **Figure 3** The first three registers. In this case r_0 has value 3, r_1 value 0 and r_2 value 2. If $Z = 0$, the **Dralnu’s Crusade** making Aurochs into Saprolings is phased out and Bob’s Elves get +3/+3.

For each $n \in \{0, 1, \dots, 11\}$, Bob has a **Riders of Gavony** giving Z_n creatures protection from Yetis. Each of these is made into a noncreature artifact with mana value 0 and attached to the **Dralnu’s Crusade** that applies to Y_n , so that they phase out together. As a result, this means that each register except for r_Y has protection from Yetis. Then, a creature can be given the types Reflection and Yeti so that it can only target r_Y ; we call this combination tr_Y .

Bob also has another **Riders of Gavony** giving Saprolings protection from Zubera creatures. This means the inactive registers (those other than r_Z) can’t be targeted by any creature that’s a Zubera. Then, as above, creature types Reflection and Zubera together mean that a creature can only target r_Z ; we call this combination tr_Z .

4.9 Memory

We provide an unlimited number of memory slots, each addressed by a nonnegative integer address. Each memory slot can hold a single arbitrarily large nonnegative integer.

A nonzero value V at a memory address A is represented by a Mouse token with base power and toughness V/V under Alice's control with $A +1/+1$ counters on it. A zero value is represented by an absence of such a token.

4.10 The flag

There is a Boolean flag that some instructions use. It is represented by a card, being in Bob's library for 0/false and in Bob's hand for 1/true; this card must not have any abilities that function in those zones other than characteristic-defining abilities. We assume that Bob has at least one such card in his deck (most lands, planeswalkers, instants and sorceries would be suitable). We remove all other cards from Bob's hand, library and graveyard.

Wheel of Sun and Moon enchanting Bob allows us to set the flag to 0 by making Bob discard the card. We give Bob a **Tomorrow, Azami's Familiar**, allowing us to set the flag to 1 by making Bob draw a card, while not making Bob lose the game if the flag was already 1.

To stop Bob from playing this card, we give Bob a **Nevermore** and/or an **Aggressive Mining** made into an artifact as appropriate.

4.11 Further environment control

To prevent any player choice involving the program permanent's abilities, we use **Root Maze** to make each new program permanent enter the battlefield tapped and **Choke** to keep them tapped (recall that they are all made Islands). Also, Bob's **Sterling Grove** has its activated ability shut off by **Suppression Bonds** attached to it. **Stony Silence** and **Cursed Totem** shut off activated abilities of artifacts and creatures.

Both players control a copy of **Recycle**, skipping both player's draw steps. **Mirror Gallery** disables the "legend rule". And we give both players a **Corrosive Mentor** so that black creatures controlled by either player have wither.

4.12 Instructions

Sadly length constraints prevent us including here the details of how each instruction is implemented. See [4] for the full implementation details.

5 Example Instruction

For demonstration purposes, here is how an example turn cycle looks. Let us say the next three cards on the top of Alice's library are Wastes, Plains, Swamp. This triplet encodes symbols 5 0 2, a Set instruction.

At the start of Alice's turn, most creatures are Saprolings and therefore have no abilities. Recall that all creatures are forced to attack and block where able, but only creatures with flying or islandwalk are allowed to attack. In Alice's first combat phase, twelve flying 2/1 **Dralnu's Crusades** attack along with the **Bloodthirster** and **Vaevictis Asmadi, the Dire**, whose ability puts the Wastes onto the battlefield. This makes the **Dralnu's Crusade** affecting X_5 get $+1/+1$ from its **Strata Scythe**, and so it gets blocked by Bob's first **Dream Fighter** and phases out. Creatures with type X_5 or X_A regain their abilities (unless they also have another type making them a Saproling such as Y_n).

31:12 A Programming Language Embedded in *Magic: The Gathering*

In the second combat phase (granted by **Bloodthirster**), another **Vaevictis Asmadi, the Dire** and twelve more **Dralnu's Crusades** attack, as their type X_A is no longer causing their flying ability to be removed. This Vaevictis's ability puts the Wastes onto the bottom of Alice's library and the next card of the program in its place, the Plains. The **Dralnu's Crusade** affecting Y_0 gets +1/+1 from its **Strata Scythe** and gets blocked by Bob's second **Dream Fighter**, whose type X_A is no longer having its reach ability removed. Creatures with type Y_0 or Y_A regain their abilities.

In the third combat phase, Alice's **Archpriest of Iona** with types $X_5 Y_A tr_Y$ Cleric has finally regained its abilities. Its ability triggers, and is forced to target r_0 , because its types tr_Y mean it can't target any green or blue creatures or any of the other registers. r_0 gains flying, so it'll be able to block, and gets a temporary +1/+1.

When the third set of **Dralnu's Crusades** and the third **Vaevictis Asmadi, the Dire** attack, Alice's **Shape Stealer** with types $X_5 Y_A$ is also forced to attack. The **Dralnu's Crusades** all have shadow, so r_0 can't block any of them; the only creature r_0 can block is the **Shape Stealer**. **Shape Stealer's** ability gives it base power equal to r_0 's value +1, which is why it has the -1/-0 counter so its actual power is r_0 's value. It is given wither because it is black, so the damage is dealt as -1/-1 counters, cancelling out all the +1/+1 counters on the register and setting r_0 's value to 0.

In Bob's combat phase, **Halana and Alena, Partners** triggers. Because it is an Elf, its power is equal to r_2 's value. And because it has types tr_Y as well, just like with Alice's **Archpriest of Iona**, the only legal target for its trigger is r_0 . So it adds r_2 +1/+1 counters to r_Y . Then nothing else happens on the rest of Bob's turn, and we move back to Alice's turn, when the three copies of **Vaevictis Asmadi, the Dire** will read three more cards from the program.



■ **Figure 4** The five steps of instruction 5 0 2, Set r_0 r_2 .

6 Implications and Conclusion

6.1 Readability and programmability

In sharp contrast to the impenetrable millions of tokens produced by the Turing machine in [2], the game state will be clearly readable when a program in this construction terminates. After the computation of sample program 1 “Calculate 10 cubed” (see Appendix A), there will be one Mouse creature token with power and toughness 1000/1000. When sample program 2 “Prime Factors” halts, for each prime factor of the input number, there will be one Mouse creature token with power and toughness equal to that factor. When sample program 4 “Nim” halts, the result of the *Magic* game will be victory for Alice or Bob according to who won the embedded game of Nim.

The programming language provided is comparable to other microcode programming languages and assembly languages. It has some quirks but is perfectly usable to write moderate-sized programs. Readers are invited to write their own programs in the simulator we wrote to test the sample programs [3].

6.2 Tournament playability

The construction uses many different *Magic* cards, far more than are normally included in tournament decks. But it is legal to bring a deck with more than 60 cards to a tournament; players sometimes play decks with over 200 cards [7]. The only restriction is that you must be able to physically shuffle the deck in a reasonable amount of time [11].

Appendix C contains a decklist of a 360-card deck which could be brought to a Legacy tournament. The deck’s composition breaks down as 160 land cards to be used for the program; 136 distinct named cards used in the microcontroller; 40 cards used during setup to edit the text and characteristics of the cards used on the microcontroller; and 24 cards used to generate an unbounded amount of mana, draw all the remaining cards, set up the construction and remove all Bob’s cards.

With the correct draw, a player can take control of the game as early as the first turn, and set up the construction. Getting that correct draw is much less likely than with a 60-card deck, but this is a theoretical result anyway; the difference between a one in a million chance and a one in several trillion is not particularly relevant.

There are minimal constraints on Bob’s deck (one card to serve as the flag must have no abilities that function in the graveyard or hand), which will easily be satisfied by any normal deck. So it is perfectly possible for a hapless player to sit down expecting a tournament *Magic* game, have the opponent take over and set up the Microcontroller, and find that they can only win the game by winning (say) a game of chess instead.

6.3 Computational implications

The previous construction in [2] was Turing complete, so this does not increase the amount of computation possible inside *Magic*. However, the addition of input commands during program execution adds a lot to the programs that can be usefully written, in terms of ability to simulate multi-player games involving choices – see e.g. sample program 4 which implements Nim. The language is clearly powerful enough to similarly write programs for chess, checkers, go, or any similar two-player perfect knowledge game.

A common joke upon the publication of [2] was “Now we can write *Magic Online* [a digital implementation of *Magic*] in *Magic*”. With the Turing machine-based construction, all players would have had to pre-register all their moves before computation started. By contrast, if a digital card game were implemented using the construction in this paper, players could choose their moves during gameplay in response to the moves made by their opponent.

Similarly, this result shows that the complexity of *Magic* includes any game or algorithm which involves a finite (but potentially unbounded) sequence of choices, including responses to another player's choices. It is of course still not an especially practical environment to perform any real computation.

We include sample program 3, which searches for a counterexample to the Collatz conjecture, as a concrete demonstration of the possibility brought up in discussions of [2]. If Alice should set up the microcontroller and start this program, the game is a victory for Bob as soon as the program finds a cycle of numbers that is a counterexample to the Collatz conjecture. If (as is widely suspected) no such counterexample exists [8], or if it instead finds a sequence that goes on forever without repeating, the game is a draw by infinite loop. This provides an explicit *Magic* game state where all player choices have been removed but the end result of the game is unknown to current mathematics.

6.4 Further research

It is clear that *Magic* is as computationally complex as it's possible for a perfect knowledge game to be. But not all two-player games are perfect knowledge, and *Magic* contains many cards and mechanics that use hidden information. Our construction doesn't use any of these, but it's possible future constructions could. This would allow embedding a wider variety of games into *Magic*, such as two-player games where both players choose their moves simultaneously.

It is also possible that there exist other tabletop games which support embedding this kind of construction. But any such game would need to be in the small subset of tabletop games which allow all of the following:

- An unlimited number of player actions rather than a fixed number of turns
- An unlimited number of at least one resource
- Some way to constrain the actions players can perform
- Enough flexibility in player choices to allow forcing one action to result in another

It may well be the case that *Magic* is the only widely played tabletop game meeting these criteria which has enough depth of rules and scope for player creativity to allow this kind of construction. If that is the case, we are very grateful to Wizards of the Coast for providing such a versatile set of building blocks for us to play with.

References

- 1 Jan Biel. How to run any program in a Magic: The Gathering Turing machine, 2019. URL: <https://www.youtube.com/watch?v=YzXoF1dEux4>.
- 2 Alex Churchill, Stella Biderman, and Austin Herrick. Magic: The Gathering is Turing complete. In *10th International Conference on Fun with Algorithms (FUN 2020)*, 2020.
- 3 Alex Churchill and Choong Yin Howe. Magic microcontroller simulator, April 2024. URL: <https://www.toothycat.net/~hologram/Magic/MTGProgSimulatorText.html>.
- 4 Alex Churchill and Choong Yin Howe. Magic microcontroller website, April 2024. URL: <https://www.toothycat.net/~hologram/Magic/>.
- 5 Choong Yin Howe and Alex Churchill. Magic microcontroller full version, April 2024. https://www.toothycat.net/~hologram/Magic/Magic_Microcontroller_full.pdf.
- 6 Choong Yin Howe and Alex Churchill. A more easily programmable system constructible in Magic: The Gathering, April 2024. URL: <https://cyh31.neocities.org/amepscintg/explanation>.

- 7 William “Huey” Jensen. Retro deck highlight: Huey’s 2002 battle of wits, June 2023. URL: <https://strategy.channelfireball.com/home/retro-deck-highlight-hueys-2002-battle-of-wits/>.
- 8 Jeffrey C. Lagarias. The $3x+1$ problem: An overview. *The Ultimate Challenge: The $3x+1$ Problem*, pages 3–29, 2010. doi:10.48550/arXiv.2111.02635.
- 9 Z.A. Melzak. An informal arithmetical approach to computability and computation. *Canadian Mathematical Bulletin*, 4(3):279–293, 1961. doi:10.4153/CMB-1961-031-9.
- 10 Wizards of the Coast. Magic: The Gathering comprehensive rules, January 2024. URL: <https://magic.wizards.com/en/rules>.
- 11 Wizards of the Coast. Magic: The Gathering tournament rules, January 2024. URL: <https://wpn.wizards.com/en/rules-documents>.

A Sample Programs

Readers are encouraged to explore the functionality of these sample programs firsthand by executing them within the provided simulator interface [3] where they are preset options.

Sample Program 1 10 Cubed.

Symbols	Instruction	Comments
10 0 10	NumBuild 10	Initialise r_0 to 10
5 1 0	Set $r_1=r_0$	r_1 is the output
6 0 1	Multiply $r_1 r_0$	Now r_1 is 100
6 0 1	Multiply $r_1 r_0$	Now r_1 is 1000
4 0 0	HaltD	We’re done.

The complete program is: 10 0 10 5 1 0 6 0 1 6 0 1 4 0 0 – or in cards: Snow-Covered Forest, Plains, Snow-Covered Forest, Wastes, Island, Plains, Snow-Covered Plains, Plains, Island, Snow-Covered Plains, Plains, Island, Forest, Plains, Plains.

After 5 of Alice’s turns and 4 of Bob’s, the game will end in a draw. Register r_1 will have 1000 $+1/+1$ counters on it, the result of the calculation.

Sample Program 2 Prime Factors.

Symbols	Instruction	Comments
<i>Register usage: r_0: constant 2. r_1: input number remaining to be factorised. r_2: copy of r_1 for divisions. r_3: current divisor. r_4: how many factors found so far.</i>		
2 1 0	AInput 1	Read the input number into r_1
4 1 3	Add1 r_3	Initialise divisor to 1
10 0 2	NumBuild 2	Initialise r_0 to constant 2
<i>Main loop: test the next number</i>		
4 1 3	Add1 r_3	Increment the divisor we’re testing
5 2 1	Set $r_2 r_1$	Prepare to test r_1
7 2 3	DivCeil $r_2 r_3$	Divide and check remainder
3 5 4	JumpBwdF 4	If flag, r_3 is not a factor
<i>Found a factor: store it and the quotient</i>		
8 3 4	Store $r_3 r_4$	It is. Save r_3 to a new memory slot,
4 1 4	Add1 r_4	and increment number of factors found
5 1 2	Set $r_1 r_2$	Remember the new divided total
11 0 2	FLess $r_2 r_0$	Is r_2 now 1?
3 3 8	JumpBwdNF 8	If not, continue. Could be another factor of r_3 so recheck it.
4 0 0	HaltD	If so, halt

31:16 A Programming Language Embedded in *Magic: The Gathering*

After execution finishes, there will be one memory entry for each prime factor in the input number. For example, if Alice chooses 120, the program finishes after 57 turn cycles, and memory consists of {2, 2, 2, 3, 5}.

■ Sample Program 3 Collatz ($3n + 1$).

Symbols	Instruction	Comments
<i>Register usage: r_0: built numbers. r_1, r_2, r_3: constants 1, 2, 3. r_4: source of the current chain. r_6: temp read memory. r_7: current number being checked. r_8: either half r_7 or $3r_7 + 1$.</i>		
4 1 1	Add1 r_1	Initialise r_1 to 1
10 0 2	NumBuild 2	Create constant 2
5 2 0	Set r_2 r_0	Store constant 2 in r_2
10 0 3	NumBuild 3	Create constant 3
5 3 0	Set r_3 r_0	Store constant 3 in r_3
10 4 2	NumBuild $12 \times 4 + 2$	Start searching at 50
5 4 0	Set r_4 r_0	Initialise current search root
5 7 4	Set r_7 r_4	Start checking at current search root
<i>Label 0: we have a new r_7 to investigate</i>		
11 2 7	FLess r_7 r_2	Is $r_7 = 1$?
10 1 4	NumBuild $12 \times 1 + 4$	Create longjump distance 16
3 4 0	JumpFwdF r_0	If so, jump to label 3
9 6 7	Load r_6 r_7	Load memory r_7 into r_6
11 6 6	FIsZero r_6	Is this a new number?
3 4 3	JumpFwdF 3	If so, go to label 1
11 2 6	FLess r_6 r_2	Is this a number that we know gets to 1?
3 4 11	JumpFwdNF 1	If so, jump to label 3
4 0 2	HaltB	If not, we found a loop & disproved the Collatz conjecture!
<i>Label 1: r_7 is a number we've not seen before</i>		
5 8 7	Set r_8 r_7	Prepare to halve r_8
4 2 8	Halve r_8	Halve r_8 . Did that leave remainder?
3 2 3	JumpFwdNF 3	If not, r_8 is what we want at Label 2
5 8 7	Set r_8 r_7	Set r_8 to $r_7 \dots$
6 3 8	Mult r_8 r_3	$\dots \times 3 \dots$
4 1 8	Add1 r_8	$\dots + 1$.
<i>Label 2: r_8 is the next number in the sequence</i>		
8 8 7	Store r_7 r_8	Store r_7 in memory r_8
5 7 8	Set r_7 r_8	Now investigate r_8
10 1 7	NumBuild $12 \times 1 + 7$	Create longjump distance 19
3 1 0	JumpBwd r_0	Go back to label 0
<i>Label 3: A number r_4 got down to 1. Label the chain with 1s.</i>		
5 7 4	Set r_7 r_4	Restart at r_4
9 6 7	Load r_6 r_7	Load memory r_7 into r_6
11 2 6	FLess r_6 r_2	Is $r_6 = 1$?
3 4 3	JumpFwdF 3	If so, skip to end of the loop
8 1 7	Store r_7 r_1	Save 1 into r_7
5 7 6	Set r_7 r_6	Set r_7 to the number we read
3 1 6	JumpBwd 6	Go back 6
4 1 4	Add1 r_4	We're done with r_4 's chain. Next number!
3 0 7	JumpFwd 7	Loop around to just before label 0

Space constraints prevent us from including Sample Program 4 “Nim” here. See [3] to see and run this sample program and all the others.

B Card Modification Techniques

Here we detail the techniques used while setting up the microcontroller to accomplish the various modifications to cards described in Section 4 and in the instructionse (whose implementation is omitted from this paper for space reasons, but can be seen at [4]). We assume Alice has generated an arbitrarily large amount of mana and drawn all the cards she needs using **Dimir Guildmage**. We are able to repeatedly cast the instants and sorceries used below by repeatedly casting **Archaeomancer** and bouncing it with **Capsize**.

- Editing creature types: **Artificial Evolution**
- Editing colour words: **Mind Bend**
- Copiably setting creature type and/or colour and making creatures 1/1: **Croaking Counterpart** combined with **Artificial Evolution** and **Spectral Shift**
- Adding copiable creature types: **Glasspool Mimic** in conjunction with **Artificial Evolution**
- Non-copiablely setting creature type: Use **Blade of Shared Souls** to temporarily make the creature a copy of **Proteus Machine**. Use **Backslide** to turn it face down, then turn it face up and set its creature type.
- Adding non-copiable creature types: **Olivia Voldaren**, modified by **Artificial Evolution** to change Vampire to another type.
- Copiably setting power and toughness, for positive toughness: **Saw in Half** after adjusting as necessary with **Belbe’s Armor**, **Enrage**, and/or **Drana, Kalastria Bloodchief**.
- Copiably setting power and toughness to 0/0 and adding type artifact: Have an **Engineered Plague** on Shapeshifter creatures. Cast **Hulking Metamorph** prototyped, and decline to copy anything; it’s now 2/2. Cast **Saw in Half** on it, producing token copies that are base 1/1, net 0/0, and then they can copy other creatures while setting base P/T to 0/0. **Grumgully, the Generous** adds a +1/+1 counter to keep it alive.
- Non-copiablely setting power and toughness to the same number: **Gigantoplasm**
- Copiably setting mana value to 0: **Vizier of Many Faces** copiablely removes the mana cost, making the mana value 0. (This is usually done so that **Bludgeon Brawl** does not make this give a power boost.) We can repeat this if necessary using **Lithoform Engine** to copy the Embalm ability, untapped by **Twiddle**.
- Copiably setting card type to (only) artifact: **Imposter Mech**, after targeting the original with **Donate**.
- Adding type artifact: **Memnarch**
- Copiably adding type creature: **Urza, Prince of Kroog**, in conjunction with **Memnarch** if necessary, and with the creature type edited by **Artificial Evolution**.
- Copiably adding flying: **Irenicus’s Vile Duplication** copiablely adds flying.
- Adding keyword ability counters – flying, indestructible, reach, first strike, double strike, lifelink, deathtouch, vigilance, trample: **Kathril, Aspect Warper**, having used **Dimir Guildmage** to discard other cards used in the construction: **Healer’s Flock**; **Darksteel Myr**; **Halana and Alena, Partners**; **Sylvia Brightspear**; **Questing Beast**; and **Quartzwood Crasher**. **Regrowth** gets back the discarded cards afterwards.
- Adding protection from a colour: Have a **Council Guardian** enter the battlefield, and use **Ballot Broker** to make sure the right colour wins the vote. Then use **True Polymorph** to turn it into what it should be.

31:18 A Programming Language Embedded in *Magic: The Gathering*

- Adding islandwalk: **Fishliver Oil**, copied with **Mythos of Illuna**
- Adding “Whenever this creature deals combat damage to a player, draw a card.”: Use **Blade of Shared Souls** to temporarily make the creature a copy of an **Ascendant Spirit**, which itself is copiable an Angel by **Glasspool Mimic**+**Artificial Evolution**, and then activate its last ability to add this ability. Generate the snow mana with a Snow-Covered land repeatedly untapped by **Twiddle**.
- Removing flying: Use **Blade of Shared Souls** to temporarily make the creature a copy of **Mist Dragon**, and use its ability to remove flying.
- Adding +1/+1 counters: **Kathril**, **Aspect Warper** and **Resourceful Defense**
- Adding -1/-0 counters: **Jabari’s Influence** to get the first one, and multiple copies of **Resourceful Defense** to get more.
- Adding +1/+0 counters: **Dwarven Armorer** produces +1/+0 counters. Multiple copies of **Resourceful Defense** multiply the counters and then move them, as necessary.
- Creating arbitrary tokens: **Rotlung Reanimator** with **Artificial Evolution** creates tokens of arbitrary types, and **Saw in Half** sets their sizes after adjusting with **Belbe’s Armor**.
- Setting colours: **Prismatic Lace**
- Giving Bob control of cards: **Donate**

Example: We make each **Vaevictis Asmadi, the Dire** into a 1/1 Sliver Beast Reflection by casting **Croaking Counterpart** targeting a real **Vaevictis Asmadi, the Dire**, then responding by casting **Artificial Evolution** targeting **Croaking Counterpart** to change Frog to Sliver. Once the 1/1 Sliver is present, we return **Artificial Evolution** to our hand using **Archaeomancer**. We cast **Glasspool Mimic**; edit it with **Artificial Evolution** replacing Shapeshifter with Beast; cast **Capsize** with buyback on the **Archaeomancer**; recast **Archaeomancer** using **Leyline of Anticipation** to get back **Artificial Evolution** again; then use **Artificial Evolution** one more time to replace Rogue with Reflection. Then **Glasspool Mimic** resolves and becomes a 1/1 Sliver Beast Reflection copy of **Vaevictis Asmadi, the Dire**. We can create further token copies of this with **Mythos of Illuna** (all the creature type additions are copiable), and then use **Capsize** to return the original **Glasspool Mimic** to our hand for the next time we need to use it.

C Decklist

Table 4 on the facing page contains a decklist suitable for bringing to a Legacy tournament which could set up the microcontroller. Adjust the number of basic lands according to the program you wish to write; the following decklist contains enough to write the Collatz sample program. You can also leave out cards that are only used in instructions that aren’t present in the program you wish to write. E.g. **Oubliette** is only used in Call and Return instructions.

■ **Table 4** Decklist to play the *Magic* Microcontroller in a Legacy tournament.

Card	Purpose	Card	Purpose
4 Ancient Tomb	Bootstrap	4 Lotus Petal	Bootstrap
4 Grim Monolith	Infinite mana	4 Power Artifact	Infinite mana
4 Gemstone Array	Infinite mana	4 Dimir Guildmage	Draw rest of deck
13 Plains	Program	13 Snow-Covered Plains	Program
13 Island	Program	18 Snow-Covered Island	Program
13 Swamp	Program	13 Snow-Covered Swamp	Program
18 Mountain	Program	8 Snow-Covered Mountain	Program
18 Forest	Program	10 Snow-Covered Forest	Program
13 Wastes	Program	10 Snow-Covered Wastes	Program
1 Memnarch	Make token copies	1 Mythos of Illuna	Make token copies
1 Capsize	Set up	1 Archaeomancer	Set up
1 Artificial Evolution	Edit cards	1 Mind Bend	Edit cards
1 Prismatic Lace	Edit cards	1 Spectral Shift	Edit cards
1 Glasspool Mimic	Add types	1 Olivia Voldaren	Add types
1 Proteus Machine	Set creature types	1 Backslide	Set creature types
1 Argent Mutation	Set up	1 Leyline of Anticipation	Set up
1 Gigantoplasm	Edit power/toughness	1 Croaking Counterpart	Set up
1 Saw in Half	Edit power/toughness	1 Belbe's Armor	Edit power/toughness
1 Enrage	Edit power/toughness	1 Drana, Kalastria Bloodchief	Edit power/toughness
1 Engineered Plague	Edit power/toughness	1 Grungully, the Generous	Edit power/toughness
1 Hulking Metamorph	Edit power/toughness	1 Vizier of Many Faces	Edit mana value
1 Lithoform Engine	Edit mana value	1 Twiddle	Edit mana value
1 Imposter Mech	Edit types	1 Donate	Edit control
1 Astral Dragon	Add types and abilities	1 Irenicus's Vile Duplication	Add abilities
1 Urza, Prince of Kroog	Add types	1 Kathril, Aspect Warper	Add abilities
1 Council Guardian	Add abilities	1 Ballot Broker	Add abilities
1 True Polymorph	Add abilities	1 Fishliver Oil	Add abilities
1 Blade of Shared Souls	Change types and abilities	1 Ascendant Spirit	Add abilities
1 Mist Dragon	Remove abilities	1 Resourceful Defense	Add counters
1 Jabari's Influence	Add counters	1 Fishliver Oil	Add abilities
1 Dwarven Armorer	Add counters	1 Sealock Monster	Add types
1 True Polymorph	Add abilities	1 Grand Melee	Control combat
1 Stormtide Leviathan	Control combat	1 Tetsuko Umezawa, Fugitive	Control combat
1 Worship	Control combat	1 Vaevictis Asmadi, the Dire	Advance program
1 Wheel of Sun and Moon	Advance program	1 Tajuru Preserver	Advance program
1 Dralnu's Crusade	Conditional mechanism	1 Life and Limb	Conditional mechanism
1 Blood Sun	Conditional mechanism	1 Strata Scythe	Conditional mechanism
1 Healer's Flock	Keyword abilities	1 Dream Fighter	Conditional mechanism
1 Shadow Sliver	Conditional mechanism	1 Bludgeon Brawl	Conditional mechanism
1 Bloodthirster	Advance program	1 Masked Gorgon	Constrain targets
1 Spectral Guardian	Constrain targets	1 Sterling Grove	Constrain targets
1 Ivory Mask	Constrain targets	1 Illusionary Terrain	Control types
1 Joraga Warcaller	Registers	1 Riders of Gavony	Registers
1 Nevermore	Control choices	1 Aggressive Mining	Control choices
1 Root Maze	Control choices	1 Tomorrow, Azami's Familiar	Flag
1 Chokey	Control choices	1 Suppression Bonds	Control choices
1 Stony Silence	Control choices	1 Cursed Totem	Control choices
1 Recycle	Control state	1 Mirror Gallery	Control state
1 Corrosive Mentor	Add abilities	1 Halana and Alena, Partners	Instructions
1 Archpriest of Iona	Instructions	1 Shape Stealer	Instructions
1 Necrogen Mists	Instructions	1 Halfdane	Instructions
1 Tanuki Transplanter	Instructions	1 Furtive Homunculus	Instructions
1 Omnath, Locus of Mana	Instructions	1 Wrathful Red Dragon	Instructions
1 Belligerent Brontodon	Instructions	1 Smog Elemental	Instructions
1 Hornet Nest	Instructions	1 Phantom Steed	Instructions
1 War Elemental	Instructions	1 Tocatli Honor Guard	Instructions
1 Ward Sliver	Instructions	1 Godhead of Awe	Instructions
1 Wandering Wolf	Instructions	1 Behind the Scenes	Instructions
1 Spinneret Sliver	Instructions	1 Quartzwood Crasher	Instructions
1 Arwen, Weaver of Hope	Instructions	1 Aether Flash	Instructions
1 Charisma	Instructions	1 Skeleton Key	Instructions
1 Serpent of Yawning Depths	Instructions	1 Field Marshal	Instructions
1 Questing Beast	Instructions	1 Vigor	Instructions
1 Toralf, God of Fury	Instructions	1 Gideon's Intervention	Instructions
1 Progenitor Mimic	Instructions	1 Volcano Hellion	Instructions
1 Artificer Class	Instructions	1 Syr Elenora, the Discerning	Instructions
1 Slithering Shade	Instructions	1 Suntail Hawk	Instructions
1 Strength-Testing Hammer	Instructions	1 Ancient Gold Dragon	Instructions
1 Rat Colony	Instructions	1 Goblin Pyromancer	Instructions
1 Celestial Convergence	Instructions	1 Alert Heedbonder	Instructions
1 Spiritual Sanctuary	Instructions	1 Abyssal Specter	Instructions
1 Catacomb Dragon	Instructions	1 Taii Wakeen, Perfect Shot	Instructions
1 Excruciator	Instructions	1 Aegar, the Freezing Flame	Instructions
1 Khorvath Brightflame	Instructions	1 Sylvia Brightspear	Instructions
1 Mangara's Equity	Instructions	1 Darksteel Myr	Instructions
1 Ojutai, Soul of Winter	Instructions	1 Angrath's Marauders	Instructions
1 Kangee, Aerie Keeper	Instructions	1 Shimmer	Instructions
1 Fiery Emancipation	Instructions	1 Chains of Mephistopheles	Instructions
1 Captain's Claws	Instructions	1 Steely Resolve	Instructions
1 Reaper King	Instructions	1 Akron Legionnaire	Instructions
1 Discordant Spirit	Instructions	1 Blinding Angel	Instructions
1 Meishin, the Mind Cage	Instructions	1 Bower Passage	Instructions
1 Darksteel Myr	Instructions	1 Moonsilver Spear	Instructions
1 Spellbane Centaur	Instructions	1 Melira's Keepers	Instructions
1 Spiteful Shadows	Instructions	1 Empyrial Archangel	Instructions
1 Lich	Instructions	1 Justice	Instructions
1 Rotlung Reanimator	Instructions	1 Oubliette	Instructions
1 Willbreaker	Instructions	1 Razorjaw Oni	Instructions
1 Sosuke, Son of Seshiro	Instructions	1 Syphon Sliver	Instructions
1 Dormant Sliver	Instructions	1 Skanos Dragonheart	Instructions
1 Corpsejack Menace	Instructions	1 Okk	Instructions
1 Bishop of Binding	Instructions	1 Tamiyo, Collector of Tales	Instructions
1 Shimmer	Instructions	1 Sporemound	Instructions
1 Polyraptor	Instructions	1 Chief of the Scale	Instructions
1 Gruul Ragebeast	Instructions	1 Sliver Hivelord	Instructions

Eating Ice-Cream with a Colander

Kien Huynh

Communications and Transport Systems, ITN, Linköping University, Sweden

Valentin Polishchuk

Communications and Transport Systems, ITN, Linköping University, Sweden

Abstract

k -order α -hull is a generalization of both k -hull and α -shape (which are generalizations of convex hull); since its introduction in a 2014 IPL paper (which also established its combinatorial properties and gave efficient algorithms to compute it), it was used in a variety of applications (as witnessed by 38 citations in Google Scholar) ranging from computer graphics to hydrology to seismology. The subject must have been so rich and complex that it took more than a year to review the submission at IPL (which was chosen as the venue “Devoted to the Rapid Publication”), as may be witnessed by the timeline in the paper header. Nonetheless it was not rich enough to warrant publication at SODA 2009 and WADS 2009 (the reviews saying it is not yet ready for the prime time – cited from memory) nor in FUN 2010 to which the paper was submitted under the title “Eating Ice-Cream with Colander”

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases computational geometry, alpha-shape, k-hull, robust shape reconstruction

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.32

Category Salon des Refusés

Funding This work was supported by the Swedish Research Council and Swedish Transport Administration.

1 The surveyed paper

This note surveys the paper [6]

D. Krasnoshchekov and V. Polishchuk. Order- k α -hulls and α -shapes.

Information Processing Letters, 114(1-2):76–83, 2014.

<https://doi.org/10.1016/j.ipl.2013.07.023>

Publicly available version: <https://www.itn.liu.se/~valpo40/pages/ka.pdf>

Convex hull of a point set $S \subset \mathbb{R}^2$, which may be defined as the complement of the union of all halfplanes that contain no points of S , gives a rough description of the “shape” of S (Fig. 1, left). α -hull (the complement of the union of all disks of radius α that contain no points of S ; in particular, convex hull is α -hull for $\alpha = \infty$) and α -shape (a “straight-line” version of α -hull) [2] of S outline the shape more precisely (Fig. 1, right).

As rightly stated in [6], “Often the set of points representing a geometric object is obtained by sampling the object in the presence of noise, which can introduce outliers in the data”; the outliers may distort the shape (Fig. 2a,b). The surveyed paper defined generalizations of α -hull and α -shape: k -order α -hull (the complement of the union of all disks of radius α that contain less than k points of S) and k -order α -shape (a “straight-line” version of k -order α -hull – the exact definition is somewhat technical; see [6, Definition 2.6]). The generalizations are “capable of ignoring a certain amount of outliers, which results in a more robust shape reconstruction” (Fig. 2c,d).

Another generalization of convex hull of S is k -hull (the locus of points such that any halfplane through a point in the k -hull contains at least k points from S) [1], widely used in statistics where it is known under the name of k -depth contour because it is the level set



© Kien Huynh and Valentin Polishchuk;
licensed under Creative Commons License CC-BY 4.0

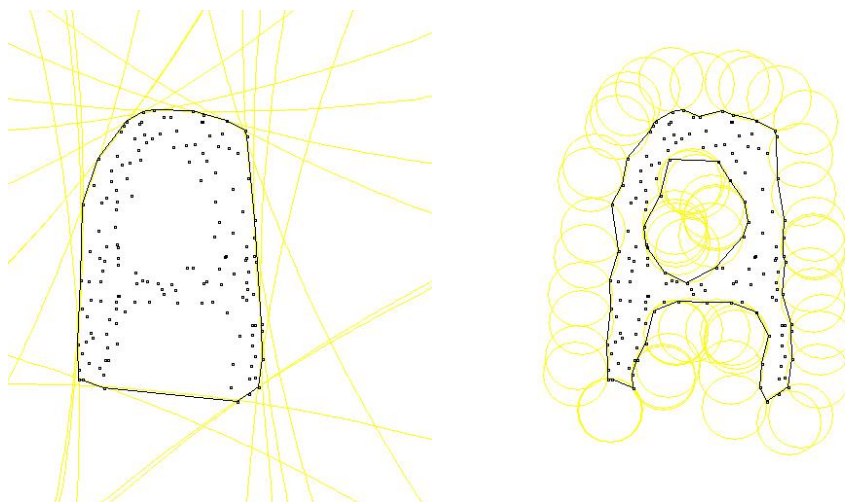
12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 32; pp. 32:1–32:4

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** Convex hull (left) misses important features of the shape (e.g., the hole). α -shape (right) does a better job. Yellow are the empty circles; for the convex hull, the circles are infinite-radius (halfplanes). The images are generated with the applet [7] (downloadable from [11]) accompanying the surveyed paper.

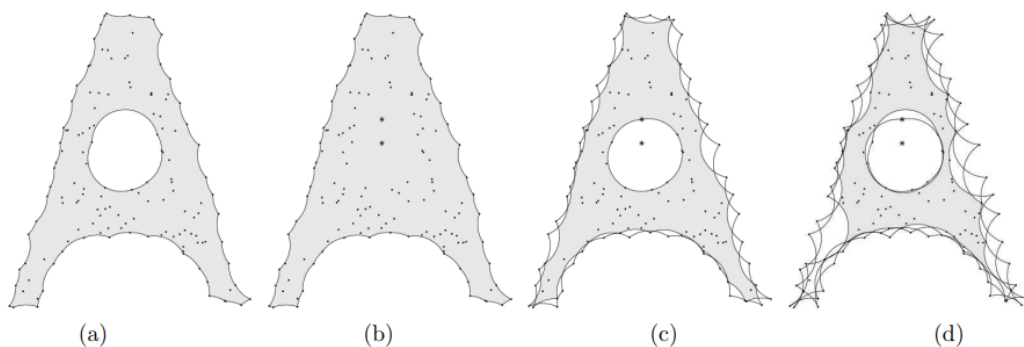


Figure 3: Order- k α -hull ignores the outliers. (a): An α -hull. (b): Two outliers change the picture. (c) and (d): Order- k α -hull with $k = 2, 3$ brings it back.

■ **Figure 2** Figure 3 from [6], together with its caption.

of the “location depth” (aka halfspace depth). The rich field of statistical data depth has produced a number of depth functions (many of which generalize convex hull in the sense that the points on the convex hull have depth 1) which are unimodal functions of distance to the “center” – the deepest point defined by S (the center may not belong to S). k -order α -hull, which is a generalization of k -hull, allows the depth to have local maxima, allowing each cluster of data to have its own depth function (Fig. 3).

Fischer [4], inspired by Edelsbrunner and Mücke [3], had a fun view of the space as an infinite piece of icecream, with points S being chocolate chips in it; then α -hull is the icecream that remains after a person, allergic to chocolate, eats as much icecream as possible with a scoop of radius α . The surveyed paper extended this fun view of α -hulls, suggesting how convex hull, k -hull and k -order α -hull may be obtained by eating as much icecream as possible with various kitchenware (Fig. 4).

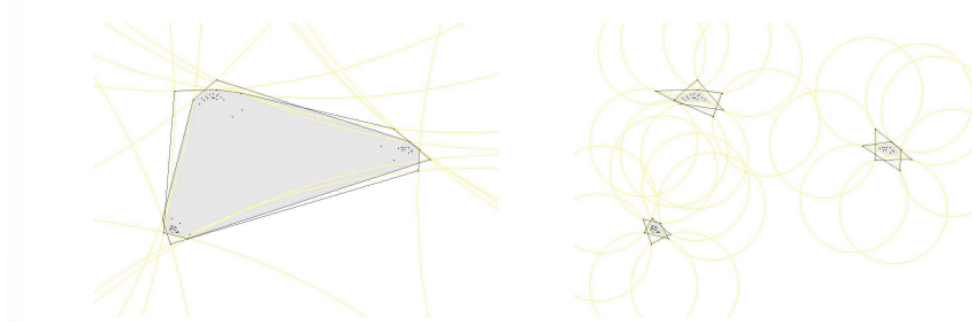
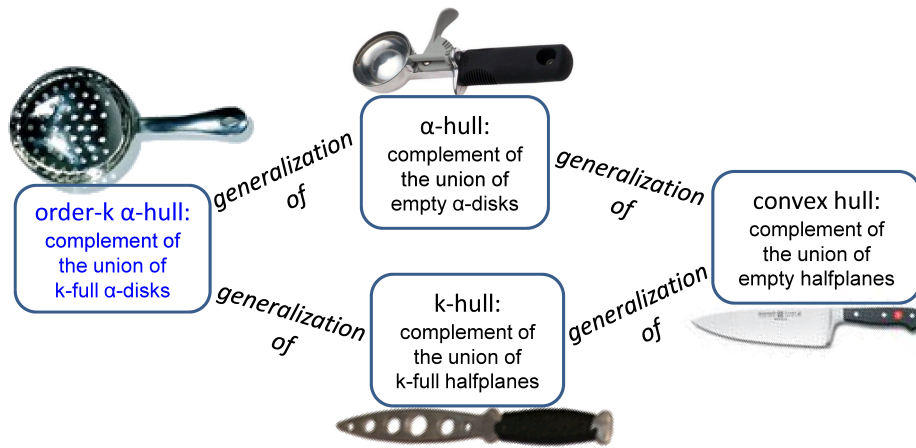


Figure 6: Points come in 3 clusters. Left: The 2-hull (2-depth contour) oversmooths the data. Right: Order-2 α -hull (for a suitable $\alpha < \infty$) better fits the data by identifying 2-depth contours independently within each cluster.

■ **Figure 3** Figure 6 from [6], together with its caption.



■ **Figure 4** Figure 2 from [6]: The family picture.

2 The paper history

The paper was born in discussions between geophysicists (who needed to do robust shape analysis of seismological data) and algorithmists (computational geometers who were interested in developing extensions of α -shapes and k -hulls); the results of the joint work were presented in a poster already in 2008 [5]. The algorithmic/combinatorial results on k -order α -hulls and α -shapes were submitted to SODA 2009 and WADS 2009 none of which saw the potential of the introduced notions; FUN 2010 (to which the paper was submitted as “Eating Ice-Cream with Colander”) did not find the kitchenware amusing enough.




Due to lack of luck with algorithmic conferences, the obtained results (both seismic data analysis and algorithmic/combinatorial results on k -order α -hulls and α -shapes) were submitted to Geophysical Journal International which liked the seismology results, but fairly suggested that the algorithmic part should be reviewed by experts in an algorithmic journal (to get a “stamp of correctness”). By this time, the seismology colleagues were rightly anxious to get the results out, so IPL was chosen as the algorithms venue “Devoted to the Rapid Publication” [10]. After a 1.5-year review (see the paper header: “Received 20 January 2012, Revised 24 July 2013, Accepted 27 July 2013”) the paper was accepted, and it was

no problem to publish the results from application of the techniques immediately [8, 9]. By now, the surveyed paper gathered 38 citations on Google Scholar, in subjects ranging from computer graphics to hydrology to seismology to urban airspace optimization. Overall, the subject is quite pictorial, see, e.g., the video [7] that accompanied the presentation of interactive applet [11] at SoCG 2010 (the web version of the applet requires Java in the browser, but the standalone version, downloadable from [11], may be run on any machine).

References

- 1 Richard Cole, Micha Sharir, and Chee K Yap. On k-hulls and related problems. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 154–166, 1984.
- 2 Herbert Edelsbrunner, David Kirkpatrick, and Raimund Seidel. On the shape of a set of points in the plane. *IEEE Transactions on information theory*, 29(4):551–559, 1983.
- 3 Herbert Edelsbrunner and Ernst P Mücke. Three-dimensional alpha shapes. *ACM Transactions On Graphics (TOG)*, 13(1):43–72, 1994.
- 4 Kaspar Fischer. Introduction to alpha shapes. *Utrecht University*, 2000.
- 5 Dmitry Krasnoshchekov and Valentin Polishchuk. Robust curve reconstruction with k-order α -shapes. In *2008 IEEE International Conference on Shape Modeling and Applications*, pages 279–280. IEEE, 2008.
- 6 Dmitry Krasnoshchekov and Valentin Polishchuk. Order-k α -hulls and α -shapes. *Information Processing Letters*, 114(1-2):76–83, 2014.
- 7 Dmitry N Krasnoshchekov, Valentin Polishchuk, and Arto Vihavainen. Shape approximation using k-order alpha-hulls. In *Symposium on Computational geometry*, pages 109–110, 2010.
- 8 Mikko Nikkilä, Valentin Polishchuk, and Dmitry Krasnoshchekov. Robust estimation of seismic coda shape. *Geophysical Journal International*, 197(1):557–565, 2014.
- 9 Eli Packer, Peter Bak, Mikko Nikkilä, Valentin Polishchuk, and Harold J Ship. Visual analytics for spatial clustering: Using a heuristic approach for guided exploration. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2179–2188, 2013.
- 10 A. Tarlecki. Information processing letters. URL: <https://www.sciencedirect.com/journal/information-processing-letters>.
- 11 A. Vihavainen. k-order alpha-shapes. URL: <https://www.cs.helsinki.fi/group/compgeom/kapplet/>.

Retrospective: Avoiding the Disk Bottleneck in the Data Domain Deduplication File System

Kai Li   

Department of Computer Science, Princeton University, NJ, USA

Abstract

The paper titled “Avoiding the Disk Bottleneck in the Data Domain Deduplication File System” [3] describes several fundamental ideas behind the file system that drives Data Domain’s deduplication storage products. Initially submitted to the 2007 ACM SIGOPS Symposium on Operating System Principles (SOSP), the paper was rejected by its program committee. It was subsequently submitted and accepted for publication at the USENIX Conference on File And Storage Technologies (FAST) in 2008. Twelve years later, it was honored with the USENIX Test-of-Time Award. This retrospective explores the paper’s historical significance and impact, analyzes the reasons behind its initial rejection, and suggests methods to enhance the paper review process in the academic community.

2012 ACM Subject Classification Hardware → Communication hardware, interfaces and storage

Keywords and phrases Deduplication, file systems, compression

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.33

Category Salon des Refusés

1 Innovation History

Founded in October 2001, Data Domain, Inc. aimed to replace traditional tape libraries in data centers with advanced disk-based storage products for backup and disaster recovery purposes. This vision was inspired by similar transitions happening at the time in the consumer electronics sector, where MP3 players (such as Apple’s iPod) and Digital Video Recorders (DVRs) were supplanting music cassettes and VCR tapes, respectively. Both leveraged lossy compression methods (MP3 and MP4), achieving compression ratios of at least 10X, thus rendering disk-based devices economically competitive with their tape-based counterparts.

There were two significant challenges in developing disk-based storage systems to supplant tape libraries in data centers. The first challenge involved developing a compression method capable of achieving a 10X compression ratio to make disk-based systems economically competitive with similar capacity tape libraries. However, unlike the lossy compressions for audio and video data used in consumer products, the compression needed for this purpose had to be lossless. Traditional lossless compression methods, such as the Ziv-Lempel algorithm [4], typically only managed 2-3X compression, which varied according to the nature of the data.

The second challenge was to ensure high deduplication throughput while maintaining low costs. This was necessary to ensure that backups could be completed within a limited window of a few hours, thereby preventing any disruption to the normal daytime operations of the primary storage systems. Achieving this balance of high-throughput and cost-effectiveness was imperative. Although there were proposals about deduplication file systems [2, 1], none of them addressed this challenge.

In January 2002, we developed the key ideas for a high-throughput deduplication system designed to overcome both major challenges. We quickly implemented these strategies into a basic prototype and conducted tests using production backup data from three data centers.



© Kai Li;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 33; pp. 33:1–33:4

Leibniz International Proceedings in Informatics



Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

33:2 Avoiding the Disk Bottleneck

The encouraging results from these tests validated our ideas to create a high-throughput deduplication file system on cost-effective, standard hardware, achieving 20-30X lossless compression for backup data.

The technology validation convinced us to move forward with the design and development of a deduplication storage product. Additionally, we designed a data protection ecosystem that utilized these deduplication storage systems for storing local backups and efficiently transferred deduplicated backup data to the cloud for disaster recovery, eliminating the need for tape libraries in data centers and transporting tapes to remote sites. By 2004, we had successfully completed and released the products.

2 Paper Submission, Rejection and Publication

After five years of developing and successfully launching three generations of products in the market, we decided to document and share some of the key ideas behind the Data Domain file system. We submitted our paper to the ACM SIGOPS Symposium on Operating System Principles (SOSP) in 2007.

Regrettably, the SOSP program committee did not accept our submission, citing a lack of detailed evaluations as the primary reason. This decision came as a surprise to us, given that our deduplication file system was considered state-of-the-art in the storage industry, and our product line had generated over \$100 million in revenue with over an 80% gross margin that year.

The reviewers did not recognize that their expected detailed comparative evaluations would require significant modifications to the file system, an unrealistic demand for a complex and sophisticated industry product. Therefore, after making minor edits, we resubmitted our paper to the USENIX Conference on File and Storage Technologies (FAST), where it was published in February 2008 [3].

3 Impacts

The Data Domain product line, powered by the deduplication file system described in our paper, dominated the backup storage market, capturing over 65% of the market share since its launch. Its revenue soared to \$570 million in 2009 and surpassed \$1 billion in 2010, supplanting the traditional tape libraries in data centers. The system's efficient high-compression ratio enabled the product line to sustain a gross margin exceeding 80%.

In the academic sphere, the paper has been extensively cited within the storage systems research community. In recognition of its lasting influence, the paper was awarded the USENIX Test-of-Time Award at the FAST Conference in 2020¹.

4 What's Not Included

The paper does not cover several key components of the deduplication file system:

- A concurrent GC (Garbage Collection) component to reclaim storage space of deleted data. The physical space of a data segment can be reclaimed only when it is not used by any file. The challenge is to accomplish this on-disk garbage collection using a small amount of memory and to keep up with the high deduplication throughput.

¹ <https://www.usenix.org/conferences/test-of-tiemsme-awards>

Concurrent Garbage Collection (GC): This component is essential for reclaiming storage space from deleted data. It only frees up the physical space of a data segment when it is no longer used by any file. On average, a data segment is shared by over 10 files in a deduplication file system. The challenge lies in performing this on-disk garbage collection efficiently using minimal memory while maintaining high deduplication throughput.

- Physical Data Replication: This component handles the replication of physical data containers over the Internet without the need to rebuild metadata structures remotely. It is designed for high-throughput, 1-to-1 data replications across high-speed network links.
- Logical Data Replication: Unlike physical data replication, this file-level protocol transfers individual physical segments. It is particularly useful for many-to-1 data replications to prevent the transfer of duplicated data segments at the destination, significantly reducing network bandwidth requirements.
- Software RAID: This component implements an abstraction layer for block storage, ensuring reliability in the event of disk failures, power outages, OS software malfunctions, and during the replacement of failed disks.
- Error Detection and Correction: This component regularly scans the storage space to identify and correct data corruptions using stored error codes. Because corruption in a single data segment could compromise many files, its role is essential for maintaining data integrity.

To fully comprehend the Data Domain deduplication file system, one must be familiar with these components. It would have been advantageous for the community to read papers about them.

5 Reflections

Why was a high-impact paper rejected by a reputable conference? Several hypotheses can be considered:

- Expertise of Reviewers: The reviewers may not have been experts in storage systems, potentially lacking an understanding of the significance of advancements in the field.
- Expectations for Evaluations: Academic reviewers are trained to expect detailed comparative evaluations, feasible with a simulator but unrealistic with a complex industry product.
- Credibility of Claims: Reviewers could have found it difficult to believe that Data Domain's deduplication file system could achieve a 10X higher compression ratio and operate 10X faster than optimized, known compression tools.

Every program committee strives to select the best papers for its conference and aims to avoid overlooking high-impact submissions. To mitigate these issues, program committees could ensure that reviewers are chosen for their expertise in the relevant subject matter and that they have realistic expectations for evaluations of industry products. Additionally, a blend of anonymous and open review processes might help the committee better understand the credibility of the systems discussed in the papers during their final deliberations.


How can we determine if a systems paper has made a significant impact? Alan Perlis once remarked, "The proof of a system's value is its existence."² This insightful quote underscores that the real-world application and longevity of a system attest to its value and impact.

² <http://www.cs.yale.edu/homes/perlis-alan/quotes.html>

References

- 1 Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, 2001.
- 2 Sean Quinlan and Sean Dorward. Venti: A new approach to archival data storage. In *Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. USENIX Association. URL: <https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage>.
- 3 Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Fast*, volume 8, pages 1–14, 2008.
- 4 Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.

Short Programs for Functions on Curves: A STOC Rejection

Victor S. Miller  

Computer Science Laboratory, SRI, Menlo Park, CA, USA

Abstract

In 1986 I submitted a note “Short Programs for functions on curves” to the STOC conference. It was rejected. Since it seemed to be a paper that would only be interesting to a very small group of people, I didn’t try to publish it, but instead circulated it among people who, I thought, would be interested in it. However, about 11 years later I was contacted by Dan Boneh, to whom I had given a copy a few years previously, who said that the algorithm in my paper had important applications. Since then it has become a core algorithm in the field of “Pairing Based Cryptography”.

2012 ACM Subject Classification Computing methodologies → Number theory algorithms; Security and privacy → Public key encryption

Keywords and phrases Elliptic Curves, Finite Fields, Weil Pairing, Straight Line Program

Digital Object Identifier 10.4230/LIPIcs.FUN.2024.34

Category Salon des Refusés

Related Version Short Programs for Functions on Curves: unpublished

Unpublished: <https://crypto.stanford.edu/miller/miller.pdf>

Full Version: <https://link.springer.com/article/10.1007/s00145-004-0315-8>

1 Prehistory

The paper that I want to discuss is “Short Programs for functions on curves” [16] which I wrote in 1986, while I was a member of the Mathematics Department at the IBM TJ Watson Research Center, and submitted to the STOC conference. It was rejected. Since it seemed to be a paper that would only be interesting to a very small group of people, I didn’t try to publish it, but instead circulated it among people whom I thought would be interested.

The main result of the paper was an efficient algorithm for the calculation of the **Weil Pairing** for Elliptic Curves over Finite Fields. At the time of writing, the field of Elliptic Curves was considered a very arcane branch of Number Theory. In 1985 I presented a paper “Use of Elliptic Curves in Cryptography” [15] at the annual Crypto conference in Santa Barbara. At the time, and for many years thereafter, using Elliptic Curves for cryptography seemed to be a very obscure niche. However, it eventually had great effect. For example a large percentage of the use of public key now uses Elliptic Curves.

The Weil Pairing was introduced by Andre Weil in 1940 [23], and has essential use in the Number Theoretic analysis of the arithmetic of Elliptic Curves. In response to a challenge of Manuel Blum, I tried to relate the discrete logarithm problem on Elliptic Curves to the more familiar problem in the multiplicative group of finite fields (which is what’s used in the original Diffie-Hellman key distribution protocol), I realized that the Weil pairing might be able to relate the two groups if it could be computed efficiently. Although it could be calculated as a ratio of two polynomials in the coordinates of the points, the degrees of the numerator and denominator were exponential in the size of the inputs. After seeing a talk by Erich Kaltofen [11], I realized that the needed result could be calculated by a short straight line program whose length was linear in the size of the inputs. The paper that I



© Victor S. Miller;

licensed under Creative Commons License CC-BY 4.0

12th International Conference on Fun with Algorithms (FUN 2024).

Editors: Andrei Z. Broder and Tami Tamir; Article No. 34; pp. 34:1–34:4

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

wrote described the algorithm. It also gave a partial answer to a question raised by Rene Schoof [19] (in another influential papers that was rejected by the FOCS conference in 1984) about the structure of an Elliptic Curve group.

2 Influence

The first mention of it was in a talk entitled “Elliptic Curves and Number Theoretic Algorithms” [12] given by Hendrik Lenstra at the International Congress of Mathematicians in Berkeley a few months later. I gave a copy of the paper to Burt Kaliski, then a graduate student at MIT, who used it in an essential way in his Ph.D. thesis “Elliptic Curves and Cryptography: A Pseudorandom bit generator and other tools” [10], and was the first (that I know of) to implement the algorithm. I gave a copy to Scott Vanstone at University of Waterloo who then used the algorithm in his influential paper with Menezes and Okamoto “Reducing elliptic curve logarithms to logarithms in a finite field” [14, 13].

I also gave a copy of the paper to Dan Boneh in 1994, when he was a graduate student at Princeton. A few years later, Dan got in touch and asked if it was ok with me for the Stanford CS department to make of copy of the paper available on the web. He told me that my algorithm had important applications. Indeed, using the algorithm was a crucial step in the realization of Shamir’s idea of “Identity Based Encryption” [21] described in the paper of Boneh and Franklin “Identity-based encryption from the Weil pairing” [4]. Antoine Joux in “A one round protocol for tripartite Diffie–Hellman” [8] also used my algorithm in an essential way. The three authors shared the Gödel prize [2] for their work.

3 Pairing Based Cryptography

Starting with the papers of Joux, and Boneh-Franklin, the field of “Pairing Based Cryptography” blossomed. For a number of years there was a conference on the subject “Pairing Based Cryptography” [22, 7, 20, 9, 1, 5] I gave the keynote address at the 2009 conference. Pairing based cryptography is still is a very lively field.

Because of its importance of my paper, Arjen Lenstra asked me in 2003 to write an extended version of the original paper for a special issue of the Journal of Cryptology, which appeared the next year [17]. According to Google Scholar, the unpublished manuscript has 483 citations in the published literature, and the extended version has 805 citations. Both are still being cited, with 28 citations in 2023.

4 Lasting Influence

Because it is now such an important part of the literature of cryptography, it has stopped being cited by many papers. A google search for “Miller’s algorithm” “pairing” produces 6250 hits. It has become standard terminology to speak of the “Miller loop” (which gets 5300 hits on google when “pairing” is included). The annual CFAIL conference [6], in 2019, gave me the inagural “Distinguished Failure Award” for keeping the original manuscript unpublished for so long. In 2008 NIST hosted a conference on Pairing Based Cryptography [18]. In the video “Pairings in Cryptography” [3] by Dan Boneh, at the 36 minute mark, there’s a discussion of my paper being rejected from STOC and its influence.

5 Intended Survey

I'll discuss the applications of the original paper, in particular give a survey of the field of pairing based cryptography.

References

- 1 Michael Abdalla and Tanja Lange, editors. *Pairing-Based Cryptography – Pairing 2012*, volume 7708 of *Lecture Notes in Computer Science*. Springer, May 2012. doi:10.1007/978-3-642-36334-4.
- 2 ACM. Gödel prize. URL: <https://www.acm.org/media-center/2013/may/acm-group-presents-godel-prize-for-advances-in-cryptography>.
- 3 Dan Boneh. Pairings in cryptography, July 2015. URL: <https://youtu.be/8WDOpzzxpTE?si=sXCoj8UFVBzzhxyF&t=2143>.
- 4 Dan Boneh and Matt Franklin. Identity-based encryption from the Weil pairing. In *Annual international cryptology conference*, pages 213–229. Springer, 2001. doi:10.1007/3-540-44647-8_13.
- 5 Zhenfu Cao and Fangguo Zhang, editors. *Pairing-Based Cryptography–Pairing 2013*, volume 8365 of *Lecture Notes in Computer Science*. Springer, November 2013. doi:10.1007/978-3-319-04873-4.
- 6 CFAIL. The conference for failed approaches and insightful losses in cryptology, 2019. URL: <https://www.cfail.org>.
- 7 Steven D. Galbraith and Kenneth G. Paterson, editors. *Pairing-Based Cryptography – Pairing 2008*, volume 5209 of *Lecture Notes in Computer Science*. Springer, September 2008. doi:10.1007/978-3-540-85538-5.
- 8 Antoine Joux. A one round protocol for tripartite Diffie–Hellman. In *International algorithmic number theory symposium*, volume 1838 of *Lecture Notes in Computer Science*, pages 385–393. Springer, 2000. doi:10.1007/10722028_23.
- 9 Marc Joye, Atsuko Miyaji, and Akira Otsuka, editors. *Pairing-Based Cryptography–Pairing 2010*, volume 6487 of *Lecture Notes in Computer Science*. Springer, December 2010. doi:10.1007/978-3-642-17455-1.
- 10 Burton S. Kaliski. *Elliptic curves and cryptography: A pseudorandom bit generator and other tools*. PhD thesis, Massachusetts Institute of Technology, 1988. URL: <https://dspace.mit.edu/bitstream/handle/1721.1/14709/18494044-MIT.pdf>.
- 11 Erich Kaltofen. Computing with polynomials given by straight-line programs I: greatest common divisors. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing - STOC '85*. ACM Press, 1985. doi:10.1145/22145.22160.
- 12 Hendrik Willem Lenstra. Elliptic curves and number-theoretic algorithms. In Andrew M. Gleason, editor, *Proceedings of the International Congress of Mathematicians 1986*. Universiteit van Amsterdam Mathematisch Instituut, 1986. URL: www.math.leidenuniv.nl/~hw1/PUBLICATIONS/1988a/art.pdf.
- 13 A.J. Menezes, T. Okamoto, and S.A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, 1993. doi:10.1109/18.259647.
- 14 Alfred Menezes, Tatsuaki Okamoto, and Scott Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 80–89, 1991. doi:10.1145/103418.103434.
- 15 Victor S Miller. Use of elliptic curves in cryptography. In *Conference on the theory and application of cryptographic techniques*, pages 417–426. Springer, 1985. doi:10.1007/3-540-39799-X_31.
- 16 Victor S. Miller. Short programs for functions on curves. STOC 1986 submission, May 1986. URL: <https://crypto.stanford.edu/miller/miller.pdf>.

34:4 Short Programs for Functions on Curves: A STOC Rejection

- 17 Victor S Miller. The Weil pairing, and its efficient calculation. *Journal of cryptology*, 17(4):235–261, 2004. doi:10.1007/s00145-004-0315-8.
- 18 NIST. Pairing based cryptography, June 2008. URL: <https://csrc.nist.gov/Projects/pairing-based-cryptography/events>.
- 19 René Schoof. Elliptic curves over finite fields and the computation of square roots mod p . *Mathematics of computation*, 44(170):483–494, 1985. doi:10.2307/2007968.
- 20 Hovav Shacham and Brent Waters, editors. *Pairing-Based Cryptography-Pairing 2009*, volume 5671 of *Lecture Notes in Computer Science*. Springer, August 2009. doi:10.1007/978-3-642-03298-1.
- 21 Adi Shamir. Identity-based cryptosystems and signature schemes. In *Advances in Cryptology: Proceedings of CRYPTO 84*, pages 47–53. Springer, 1985. doi:10.1007/3-540-39568-7_5.
- 22 Tsuyoshi Takagi, Tatsuaki Okamoto, Eiji Okamoto, and Takeshi Okamoto, editors. *Pairing-Based Cryptography-Pairing 2007*, volume 4575 of *Lecture Notes in Computer Science*. Springer, July 2007. doi:10.1007/978-3-540-73489-5.
- 23 André Weil. Sur les fonctions algébriques a corps de constantes fini. *Comptes Rendus Acad. Sci. Paris*, 210(1940):592–594, 1940.