Games meet Concurrency: Algorithms and Hardness

by

Michael Joseph Coulombe

S.B., University of California at Davis (2013)
S.M., Massachusetts Institute Of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Michael Joesph Coulombe
Department of Electrical Engineering and Computer Science
May 19, 2023

Certified by: Erik D. Demaine
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Games meet Concurrency: Algorithms and Hardness

by

Michael Coulombe

Submitted to the Department of Electrical Engineering and Computer Science
on May 19, 2023 in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Abstract

Since the turn of the 21st century, seeing the decline of Moore's Law on the horizon, the pursuit of continued software performance gains has led to the prominence of computer architectures with high degrees of parallelism and memory cache hierarchies. However, there are still many challenges to designing efficient algorithms and understanding the complexity of fundamental problems in these new models of computation. Given the similarities of concurrent systems of multiple agents and multiplayer games, this thesis analyzes a spectrum of models connecting these three fields and bridges the gaps between them by building upon techniques from the growing literature studying the complexity of games through gadget motion planning frameworks.

Thesis Supervisor: Erik D. Demaine
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I first thank my advisor, Erik D. Demaine, for his mentorship over the last six years and cultivation of a collaborative and open research environment. I also thank all my colleagues in Erik's group and beyond; their contributions were invaluable to the research presented in this thesis.

I thank all participants of course 6.892 Algorithmic Lower Bounds: Fun with Hardness Proofs (Spring 2019), a flipped class hosted by Erik D. Demaine, Jeffrey Bosboom, and Jayson Lynch, for useful discussions and fruitful collaborations which led to Chapter 1, Chapter 4, and more works outside this thesis. In particular, I thank Lily Chung for contributing to the polynomial-time algorithms in Chapter 1. I also thank Sophie Monahan for editing assistance in Chapter 4.

I thank all the contributors to Zelda Wiki [Zel21b] and Zelda Dungeon [Zel21a] for their comprehensive information on the Legend of Zelda series, and to The Spriters Resource [Mis] and VGMaps.com [roc] for serving as indispensable tools for providing easy and comprehensive access to the sprites used in our figures in Chapter 1. I also thank the original artists and game designers at Nintendo, Capcom, and other associated developers for bringing the timeless classics from the Legend of Zelda series to the world.

I thank Erik D. Demaine, Ryan Williams, and Martin Farach-Colton for their helpful comments, feedback, and assistance towards producing this work as members of my thesis committee.

I especially thank my family for everything they have done to support my journey through life that has led me to where I am today, including my parents, Peter and Kelly; my sister Katie and her husband Michael; my aunts, Trudy and Susan; and my grandparents who live on in our hearts, Doris and Harold Coulombe and Ray and Gloria Rott.

Contents

List of Figures

List of Tables

Introduction

Nearing the end of the Moore's law era of "free" software performance improvements, there has been a call for a focus on algorithms and alternatives to the traditional von Neumann architectures as the source of speedups, in light of existing approaches from increasing parallelism with multicore processors to reducing external memory latency with cache hierarchies [LTE+20]. Unlike before, new algorithms must be designed in order to take full advantage of these new architectures and shape the direction of future hardware development, but the theoretical models of computation that underpin them are not as well understood.

One fruitful strategy for studying problems in one field has been looking for connections with related fields, such as work reducing questions of property testing into communication problems to obtain new bounds and a better understanding of known results [BBM11]. Considering aspects of concurrent models of computation that could be the basis of such a relation, a core conceit is the presence of multiple agents communicating and making decisions in a system to achieve individual, shared, or possibly competing goals. On these grounds, we draw connections between three seemingly-disparate areas of computer science: concurrent models, external memory models, and the complexity of single-player and multiplayer games and puzzles.

The study of games with two or more players in complexity theory has a long history, with classic results characterizing the hardness of board games like Chess [FL81], Go [Rob83], and Checkers [Rob84], as well as abstract games modeled by variants of Turing Machines [Sch78, PR79, PRA01]. The field of game and puzzle research has produced several powerful computational frameworks, such as constraint logic [DH08] and motion-planning-through-gadgets [DGLR18,DHL20], which have also found applications in more recent research on video games [DLL18,DGLR18], modular robot reconfiguration [ADG+21], and global control and shape self-assembly [BMLC+19, CCG+20]. In this thesis, we show the utility of the motion-planning-through-gadgets framework in the study of concurrent models such as surface chemical reaction networks, apply concurrent techniques in understanding motion planning and gadget simulation problems, and advance the state-of-the-art of single-player and multiplayer game analysis.

On the front of cache performance, this thesis introduces the granular external memory model (GEM), generalizing the traditional external memory (EM) model by removing the restriction of block-level access by I/O operations. GEM is inspired by newer solid-state drive (SSD) access protocols that can support noncontiguous operations that would not be efficient on the hard disks and older technology that inspired EM. It is also motivated by the theoretical goal of better understanding barriers to solving open problems in EM by simplifying data layout problems to focus on the challenges with efficient cache management. We develop algorithms and data structures in the GEM model, expose new links to parallel and concurrent techniques, and make the case for full hardware support for these operations in SSDs and elsewhere to surpass EM lower-bounds that can only be broken with granularity.

Thesis Organization

This thesis is organized into three parts.

Part I introduces motion planning frameworks with Chapter 1, where we show their application to single-player video games from across the Legend of Zelda franchise, analyzing many sets of mechanics to prove whether they are polynomial-time solvable or up to PSPACE-complete. In Chapter 2, we investigate the motion planning of multiple robots through gadgets, uncovering connections to Petri nets that reveal ACKERMANN-completeness for reconfiguration problems.

Part II focuses on team multiplayer games. Chapter 3 introduces the Team DFA Game, an undecidable, bounded-space, two-versus-one game, simplifying previous work and providing the first application to real-world games: by using an intermediate motion planning framework, we prove undecidability of Team Fortress 2, Super Smash Brothers, and Mario Kart. Chapter 4 generalizes the Team DFA Game to include limited communication within the two-player team, shows matching upper- and lower-bounds on the threshold of information transfer allowed before the game becomes decidable, and gives similar results for a related Team Formula Game.

Part III explores more-constrained concurrent models of computation. It opens with an analysis of an asynchronous multirobot gadget model in Chapter 5, finding size lower bounds and constructions for gadget-analogues of multiprocessor shared memory objects. In Chapter 6, we look at Surface Chemical Reaction Networks, an asynchronous model of molecular interactions, and prove the hardness of various reconfiguration problems by reduction from motion-planning-through-gadgets problems. Finally, Chapter 7 introduces the granular external memory model, leveraging parallel and concurrent techniques to implement various fundamental and specialized algorithms and data structures beyond the limits of ordinary external memory.

Part I

Games and Gadgets

Chapter 1

Motion Planning and The Legend of Zelda

This chapter presents results from the paper titled "The Legend of Zelda: The Complexity of Mechanics" that the thesis author coauthored with Jeffrey Bosboom, Josh Brunner, Erik D. Demaine, Dylan H. Hendrickson, Jayson Lynch and Elle Najt. This paper appeared in The Thailand-Japan Conference on Discrete and Computational Geometry, Graphs, and Games (TJCDCG^3), 2021 [BBC+22], and has been accepted into the Thai Journal of Mathematics.

Overview

We analyze some of the many game mechanics available to Link in the classic Legend of Zelda series of video games. In each case, we prove that the generalized game with that mechanic is solvable in polynomial time, NP-complete, NP-hard and in PSPACE, or PSPACE-complete. In the process we give an overview of many of the hardness proof techniques developed for video games over the past decade: the motion-planning-through-gadgets framework, the planar doors framework, the doors-and-buttons framework, the "Nintendo" platform game SAT framework, and the collectible tokens and toll roads Hamiltonicity framework.

1.1 Introduction

"It's dangerous to go alone! Take this."

The Legend of Zelda[1] action–adventure video game series consists of 20 main games developed by Nintendo (sometimes jointly with Capcom), starting with the famous 1986 original which sold over 6.5 million copies [Vid21], and most recently with Breath of the Wild, which was a launch title for Nintendo Switch (and is arguably what made the Switch an early success), and its sequel, Tears of the Kingdom, which was just released at time of writing. In each game, the elf protagonist Link explores a world with enemies and obstacles that can be overcome only by specific collectible items and abilities. Starting with nothing, Link must successively search for items that unlock new areas with further items, until he reaches and defeats a final boss enemy Ganon.

Across the over 35-year history of the series, many different mechanics have been introduced, leading to a varied landscape of computational complexity problems to study: what is the difficulty

[1]All products, company names, brand names, trademarks, and sprites are properties of their respective owners. Sprites are used here under Fair Use for the educational purpose of illustrating mathematical theorems.

of completing a generalized Zelda game with specific sets of items, abilities, and obstacles? Reviewing the two Zelda wikis [Zel21b, Zel21a] and playing the games ourselves, we have identified over 80 unique items with unique mechanics, listed in Tables 1.4 and 1.5, and various obstacles in Table 1.6, throughout the first 19 games in the Zelda franchise. More mechanics could likely be identified from the numerous enemy types and other game features.

In tribute to the fun and challenge of the Zelda series, we propose a long-term undertaking where the video-game-complexity community thoroughly catalogs these mechanics and analyzes which combinations lead to polynomial vs. NP-hard computational problems. Toward this goal, we analyze in this chapter the complexity of several new combinations of various items, including Hookshot, Switch Hook, Diamond Blocks, Crystal Switches, Roc's Feather, Pegasus Seeds, Kodongos, Buzz Blobs, Cane of Pacci, Magnetic Gloves, Magnesis, Bombs, Bow, Ice Arrows, Water, Fairies, Magic Armor, Decayed Guardians, Statues, Ancient Orbs, and colored-tile floor puzzles.

Table 1.1 summarizes our results about the Legend of Zelda, and Table 1.2 lists previously known results. The first paper to analyze Legend of Zelda games, from FUN 2014 [ADGV15], showed that Zelda with push-only blocks is NP-hard; Zelda with Hookshot, push-and-pull blocks, chests, pits, and tunnels is NP-hard; Zelda with Small Keys, doors, and ledges is NP-hard; Zelda with ice and sliding push-only blocks is PSPACE-complete; and Zelda with buttons, doors, teleporter tiles, pits, and Crystal Switches that activate raised barriers is PSPACE-complete. More recent work from FUN 2018 [DGLR18] showed that Zelda with spinners is PSPACE-complete. Many more items and mechanics remain to be analyzed; refer to Tables 1.4, 1.5, and 1.6 in Section 1.6.

Our new results also serve to highlight different techniques for proving polynomial-time and NP algorithms, NP-hardness, and PSPACE-hardness of video games involving the control of a single agent. For algorithms, we apply the powerful technique of shortcutting to enable simple searches for solution paths through seemingly complex dungeons, and fixed-parameter tractability analysis to achieve efficiency as dungeons get larger but the game mechanics stay constant. One major category is reductions inspired by Hamiltonian Path, often simplified with Viglietta's Metatheorem 2 [Vig14] concerning collectible items and toll roads. Next is the Nintendo-style SAT reduction from [ADGV15] which later acted as inspiration for the Turrets Metatheorem from [DLL18] and the door-opening gadgets in [DHL20]. For PSPACE-hardness, we use the door-and-button framework of Forišek [For10] and Viglietta [Vig14]. Finally, we use the door gadget from [ADGV15] which, along with the other previous work, inspired the gadgets framework for the complexity of motion-planning problems [DGLR18, DHL20, ABD$^+$20] which we also use here and in Chapters 2, 5, and 6. As a secondary goal, we hope that this chapter offers a nice sampling of proof techniques showing the hardness infrastructure that have been built up in recent years.

We describe our model of generalized Zelda in Section 1.2. The chapter is then organized into sections roughly corresponding to the complexity classes of our results. Section 1.3 gives polynomial-time algorithms for hookshot and pots; and switch hook and diamond blocks. Section 1.4 proves NP-hardness for Zelda with floor puzzles; and hookshots, pots, and keys; and a variety of enemies. We show ways of replacing pots and keys with several other sets of items. Section 1.5 proves PSPACE-hardness for Magnetic Gloves; a more powerful Cane of Pacci (while the original version is in P); the Magnesis rune; minecarts with switchable tracks; and Pedestals with Ancient Orbs or Pressure Places with statures that control doors.

1.2 Zelda Game Model

Each game in the Legend of Zelda series implements a unique two- or three-dimensional variant of a common base of gameplay mechanics, which we extract and model for the purpose of writing

Game Mechanics	Games with Mechanics	Result	Thm
Hookshot, Pots, Pits	ALttP, LA, PH, ALBW	$\in$ P	1.3.1
Hookshot, Pots, Pits, Keys	ALttP, LA, PH, ALBW	NP-hard	1.4.1
Hookshot, Unpushable Pots, Pits, Keys	ALttP, LA, PH, ALBW	NP-complete	1.4.3
Switch Hook, Diamond Blocks, Pits	OoA	$\in$ P	1.3.3
Crystal Switches, Raised Barriers	ALttP, LA, OoA, PH, ALBW	$\in$ P	1.3.4
Roc's Feather, Pegasus Seeds	OoA, OoS, MM	NP-hard	1.4.5
Bombs, Renewing Cracked Walls	OoT, MM, OoA, OoS, TWW, TMC, TP, ST, SS, BotW	NP-hard	1.4.6
Ice Arrows, Water	MM	NP-hard	1.4.7
Healing Items, Unavoidable Damage Region	ALttP, LA, OoT, MM, OoA, OoS, FS, TWW, FSA, TMC, TP, PH, ST, SS, ALBW, BotW	NP-hard	1.4.8
Magic Armor, Unavoidable Damage Region	ALttP, OoT, TWW, TP	NP-hard	1.4.9
Bow or Bombs, and Crystal Switches for Raised Barriers	ALttP, LA, OoA, TP, PH	NP-hard	1.4.10
Colored-tile floor puzzles	LA, OoA, TMC	NP-complete	1.4.11
Kodongos, low walls, sword	ALttP	NP-hard	1.4.12
Buzz Blobs, Master Sword	ALttP, LA, OoA, OoS, TMC, ALBW, TFH	NP-hard	1.4.13
Decayed Guardians, Bombs	BotW	NP-hard	1.4.14
Statues, Pressure Plates, Doors	ALttP, OoT, MM, OoA, OoS, FS, TWW, FSA, TMC, TP, PH, ST, SS, ALBW	PSPACE-complete	1.5.2
Ancient Orbs, Pedestals, Doors	BotW	PSPACE-complete	1.5.3
Magnetic Gloves, metal orbs, ledges, jump platforms	OoS	PSPACE-complete	1.5.4 1.5.5
Cane of Pacci, ground holes, ledges, tunnels	TMC	FPT in duration PSPACE-complete	1.5.6 1.5.7
Magnesis Rune, metal platforms	BotW	PSPACE-complete	1.5.8
Minecarts	OoA, OoS, TMC	PSPACE-complete	1.5.9

Table 1.1: Summary of our complexity results for various game mechanics in Legend of Zelda games, along with a list of specific games with those mechanics abbreviated according to Table 1.3.

Game Mechanics	Games with Mechanics	Result	Prev
Pushable Blocks	Zelda I, LA, OoA, OoS, TMC	NP-complete	[ADGV15]
Pushable/pullable Blocks, hookshot, chests, pits, tunnels	ALttP, LA, OoT, MM, TWW, ALBW	NP-complete	[ADGV15]
Keys, Doors, Ledges	AoL, ALttP, LA, OoT, MM, OoA, OoS, FS, TWW, TMC, TP, PH, ST, SS, ALBW, BotW	NP-complete	[ADGV15]
Pushable Blocks, Ice	OoT, MM, OoS, TMC, TP, ST	PSPACE-complete	[ADGV15]
Buttons, Doors, Teleporters, Pits, Crystal Switches	ALttP, ALBW	PSPACE-complete	[ADGV15]
Spinners	OoA, OoS	PSPACE-complete	[DGLR18]

Table 1.2: Previous complexity results for various game mechanics in Legend of Zelda games, along with a list of specific games with those mechanics abbreviated according to Table 1.3.

Release Year	Game Title or Subtitle	Abbreviation	Dimensions
1986	The Legend of Zelda	LoZ	2
1987	Zelda II: The Adventure of Link	AoL	2
1991	A Link to the Past	ALttP	2
1993	Link's Awakening	LA	2
1998	Ocarina of Time	OoT	3
2000	Majora's Mask	MM	3
2001	Oracle of Ages	OoA	2
2001	Oracle of Seasons	OoS	2
2002	Four Swords	FS	2
2002	The Wind Waker	TWW	3
2004	Four Swords Adventures	FSA	2
2004	The Minish Cap	TMC	2
2006	Twilight Princess	TP	3
2007	Phantom Hourglass	PH	2.5
2009	Spirit Tracks	ST	2.5
2011	Skyward Sword	SS	3
2013	A Link Between Worlds	ALBW	2.5
2015	Tri Force Heroes	TFH	2.5
2017	Breath of the Wild	BotW	3

Table 1.3: List of games studied in this chapter, with the abbreviations used and the number of dimensions. To avoid repetition, we exclude the title prefix "The Legend of Zelda:" present in all games beyond the first two. Game list and release year information from Zelda Wiki [Zel21b].

widely applicable proofs in the remainder of this chapter. These models encompass the 19 Zelda games studied in this chapter, listed in Table 1.3. For brevity, throughout this chapter we use abbreviations for the titles of games in the series, which are listed in the table and are commonly used among players. The table also includes the 2D or 3D classification for each game. Four games are categorized as "2.5D" because, while they are each implemented and visualized as a 3D polygonal world, the top-down gameplay style and item mechanics in many circumstances more closely fit the 2D model described below.

1.2.1 2D

Generalized 2D Zelda is a single-player game in which the player controls an avatar, Link, in a two-dimensional world. (We also use this model for the 2.5D games in Table 1.3.) The world contains ***dungeons***, each consisting of a network of ***rooms***, which are rectangular grids of square ***tiles*** that set the stage for free-moving dynamic ***objects*** (enemies, pots, collectible items, etc.). The goal of the player is to navigate Link from the designated initial room to the designated final room. Each tile may contain an ***obstacle*** (a pit, solid wall, short fence, a door, a chest, grass, water, lava, spikes, etc.) or be empty. Link can collect ***items*** which are recorded in the ***inventory*** and can change the actions available to Link (such as being able to shoot arrows) or how other mechanics affect Link (such as taking less damage).

In some cases, we will refer to categories of mechanics and give examples of specific instances in various games. For example many games have ***pits*** such as water, lava, or holes in the ground which cause Link to take damage and return to a prior location on the map. We may also have ***unavoidable damage regions*** such as spiked floors or blade traps which Link can traverse but will cause damage either on contact with each new tile or over time. For some results we will list these categories or a prototypical example and then list specific instances found in specific games to which it applies.

A ***collision mask*** is a 2D rectangular bitmap representing the space that an object or obstacle occupies at its location, specified with ***pixel*** precision. A ***collision*** occurs when two ***collision masks*** would overlap after updating an object's position, often either preventing that movement, causing damage, or triggering events. A ***sprite*** is the visual graphic representing an object or obstacle.

Link's position (as well as other dynamic objects' positions) is represented as a fixed-point number of pixels within the current room, although the position of the sprite and collision mask are rounded to integer pixel coordinates, and the number of fractional bits in coordinates is taken to be a constant. Time progresses in discrete ***frames***, during which the player sets each input button as pressed or released and then the room's objects are updated. Four directional buttons allow the player to move Link in eight directions at a speed of 1 pixel per frame (diagonal unit-speed motion is approximately $1/\sqrt{2}$ pixels in each dimension). Link will ***face*** in the cardinal direction he has most recently moved in, which determines the direction of his actions. Link's collision mask is a box whose size is a quarter of a tile (half in each dimension), preventing movement through the collision of solid obstacles or other objects.

The game takes place in one room at a time: the room containing Link. When Link leaves a room, some of its objects and tiles may have their current state forgotten or saved globally (temporarily or permanently) for the next time Link enters. Link himself has persistent state, including a ***heart meter***, measuring Link's health in quarter hearts, and an inventory containing ***equipment*** for attacking enemies and traversing obstacles.

Doors between rooms may be defined as ***checkpoints***, and the game stores a record of the most recent checkpoint that Link has passed through. If Link's heart meter becomes empty, Link returns

there with his heart meter refilled to a small number of hearts. The player can also choose to ***save*** the game at any point, which creates a record of the state of the game, but Link's saved position is set to the most recent checkpoint (and certain obstacles or objects may be saved differently as well to accommodate this). If the player chooses to ***quit*** the game at any point, the game resets to the saved state.

1.2.2 3D

In ***Generalized 3D Zelda***, rooms are instead three-dimensional spaces bounded by solid ***polygons*** with fixed-point coordinates, as well as dynamic objects with polygon-defined collision boundaries. Link and other dynamic objects cannot pass through solid polygons, and are pulled downwards by ***gravity***, which means that they will fall if they are not on a polygonal surface and will slide down a sufficiently steep surface. Long falls will cause Link to experience ***fall damage*** proportional to the distance fallen beyond a safe threshold.

The player uses a ***joypad*** to control Link's motion, allowing a choice of a fixed number of angles and magnitudes for Link's velocity in the horizontal plane during each frame. Link has limited ***jump*** abilities: running off the top of a cliff, Link will jump rather than fall, and when at the bottom of a short ledge, Link will do a jump to climb up the ledge. Some items are used by ***aiming*** in Link's first-person view, where the player uses the joypad to adjust the angle the camera points with a fixed-point precision.

Notably, Generalized 2D Zelda reduces to Generalized 3D Zelda by converting the pixels of 2D collision masks of tile obstacles and dynamic objects into polygonal prisms, using a joypad with only four directions and binary magnitude, and aligning everything on top of one large polygon to counter gravity and avoid jumpable ledges. Consequently, we describe our hardness results for Generalized 2D Zelda unless the third dimension is necessary.

1.3 Polynomial-Time Zelda

This section details polynomial-time algorithms for exploring dungeons given certain restricted sets of items and obstacles. The first pair of results are centered around a short-cutting argument guaranteeing that a solution path through a solvable dungeon can be found that never needs to repeatedly visit the same location, due to the locality of Link's abilities. Next, we consider a common mechanic across the series that has global effects on dungeon traversability, and show that in isolation it is easy to overcome.

1.3.1 Hookshot and Switch Hook are in P

The Hookshot was first introduced in The Legend of Zelda: A Link to the Past, a 2D game. On use, Link shoots a hook in the direction he is facing, which travels at a fixed velocity until it collides with something (or reaches a maximum distance) and then retracts. If the hook hits a light-weight object, the object will be carried back to Link, but if the hook hits certain heavy objects, Link will be carried to the collision point.

The hookshot allows Link to collect items or move himself across pits, which are obstacles that usually destroy items and damage Link while teleporting him to where he entered the current room. A common heavy object to hookshot is a pot. Link can also push pots from one tile to an adjacent empty tile, as well as lift a pot over his head and throw it, which destroys it and may damage enemies it hits.

We consider a dungeon containing rooms with only pots and pits, where Link's only item is the hookshot. Starting from the dungeon entrance, Link must push pots, destroy pots, and hookshot onto pots across pits on the path to the dungeon goal room.

Theorem 1.3.1. *Generalized 2D Zelda with the hookshot, pots, and pits is in P.*

Throughout this chapter, each theorem includes a list of games from Table 1.3 and the relevant mechanics from that game to which the theorem applies:

Applicable Games	Hookshot	Pot	Pit
ALttP, LA, ALBW	Hookshot	Pot	Water
PH	Grappling Hook	Barrel	Water

Proof. Given a dungeon, we can construct a directed graph G whose vertices are tiles and whose colored edges indicate the ways Link can visit a tile for the first time: if two tiles are adjacent and neither is a pit, then there are red edges going both directions between them, and if it is possible at tile A to hookshot an existent pot to land on an empty tile B, then there is a blue edge from A to B.

Let a ***platform*** be a connected component of vertices joined by red edges. Notice that Link can always traverse a red edge (by means of lifting and destroying any pot in his way), but may only traverse a blue edge from platform p_1 to p_2 if there exists a pot on a specific tile in p_2. This leads to the following observation:

Lemma 1.3.2. *If there is a solution path, then there is a solution path in which Link visits each platform at most once.*

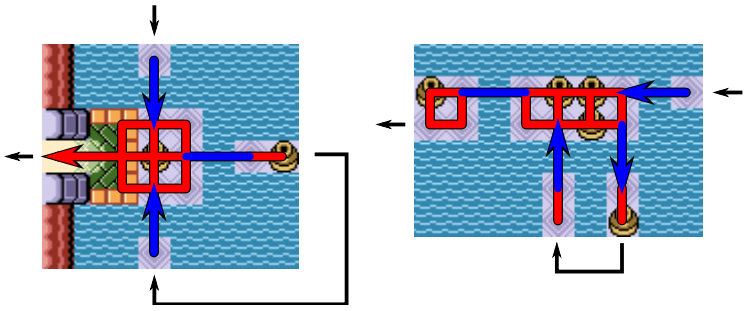


Figure 1.1: Example cases of Lemma 1.3.2: (left) If p holds the goal platform, Link can go directly there along red edges. (right) If p isn't the last platform, the last blue edge out of p is already available from red edges when Link first visits p.

Proof. Refer to Figure 1.1. Consider the last platform p on a given solution path that Link visits more than once (assuming that p exists, otherwise we are done). We can modify the solution path by skipping from the first time Link visits p to the last visit to p.

If the goal tile is on p, then once Link enters p for the first time, he may traverse only red edges directly to the goal. Otherwise, the path after the last visit to p involves visiting only never-visited-before platforms, so any pots on them were not pushed or destroyed before Link visited p. Thus, on Link's first visit to p at tile t_1, the blue edge (t_2, t_3) that Link traverses to leave p on his last visit is immediately available, so we can replace the path from t_1 to t_2 with a path of only red edges in p to get a valid solution that contains strictly fewer platform visits.

By well-ordering, we conclude that a shortest solution path cannot have any platforms that Link visits more than once. □

Using Lemma 1.3.2, we can derive a polynomial-time algorithm to solve the dungeon. First, we construct the graph G, contract its red edges to form a graph G' of platforms with edges corresponding to at least one blue edge in G, then run a breadth-first search from the starting tile's platform to the goal tile's platform to find a simple path q of platforms (or determine the dungeon is unsolvable if q does not exist).

Next, we fill in the gaps between q's blue edges to build a simple path q' in G. For blue edges (u, v) and (x, y), where tiles v and x are on platform p, we run breadth-first search from v to x within the red edges of p and splice the resulting path into q'.

Finally, we translate q' into a solution to the dungeon. For each edge (u, v) in q', if the edge is red then we command Link to lift and throw any pot on v then walk from u to v, and if the edge is blue then we command Link to hookshot in the direction of v. □

A variant of the hookshot was introduced in The Legend of Zelda: Oracle of Ages (also a 2D game), called the Switch Hook. Instead of pulling Link towards a target heavy object, this item swaps their locations, an action which can move unbreakable diamond block obstacles that are otherwise fixed in place. A similar argument to Lemma 1.3.2 and Theorem 1.3.1 proves the following:

Corollary 1.3.3. *Generalized 2D Zelda with the switch hook, diamond blocks, and pits is in P.*

Applicable Games	Switch Hook	Diamond Block	Pit
OoA	Switch Hook	Diamond Block	Lava

Proof. We sketch a proof analogous to Theorem 1.3.1's proof. Given a dungeon, we can construct a directed graph G whose vertices are tiles and whose colored edges indicate the ways Link can visit a tile for the first time: if two tiles are adjacent and neither is a pit, then there are red edges going both directions between them, and if it is possible at tile A to switchhook to an existent diamond block at tile B, where A and B are not adjacent tiles, then there is a blue edge from A to B. See Figure 1.2 for an example.

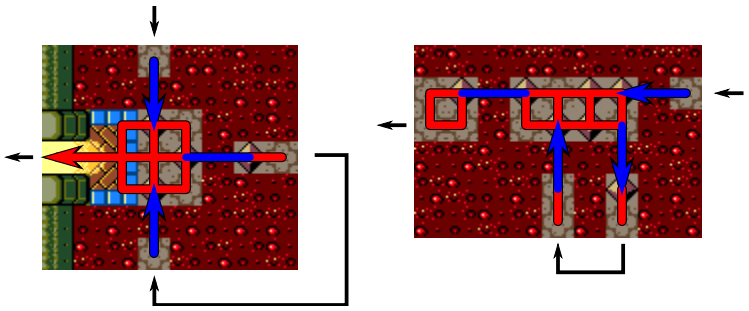


Figure 1.2: Example cases of Corollary 1.3.3, analogous to Figure 1.1.

Like Lemma 1.3.2, here we also get that if there is a solution path, then there is a solution path in which Link visits each platform at most once. Within a platform, Link can reach any tile along red edges by walking through empty tiles and swapping with any adjacent diamond blocks that otherwise block the path. Any unvisited platform must have its diamond blocks in their initial state, since they can only be interacted with using the switchhook, which switches Link's position onto

their platform. This means the same shortcutting argument applies to show platform revisits are unnecessary.

Therefore, constructing the graph G' with all red edges contracted and running a breadth-first search like was described in Theorem 1.3.1's proof will determine whether a solution path exists in polynomial time. □

1.3.2 Crystal Switches with Barriers and Unlimited Activators is in P

In many Zelda games there are raisable barriers controlled by crystal switches. The barriers can either be lowered, allowed Link to freely pass over them, or raised, blocking entry to that tile. These barriers are colored either red or blue. Crystal switches have both red and blue states and hitting them with swords, boomerangs, bombs, or arrows will cause them to switch state. In the blue state, blue blocks are lowered but red ones are raised, and in the red state the red blocks are lowered but the blue ones are raised. Globally there are only two states for the switches to be in, and thus we can search over the whole state space of the game. However, this will be contrasted in Section 1.4.2 where the addition of expendable items which can activate the switches will make the problem NP-hard.

Theorem 1.3.4. *Generalized 2D Zelda with one-ways, Crystal Switches, raised barriers, and an inexhaustible way to activate such switches is in P.*

Applicable Games	One-way	Crystal Switch	Raised Barrier	Activator
ALttP [Tower of Hera], LA [Bottle Grotto], OoA [Crown Dungeon], PH [Temple of Ice], ALBW [Tower of Hera]	Cliff	Crystal Switch	Raised Barrier	Sword or Boomerang

Proof. First, construct a traversability graph for the level in each of the two states for the crystal switches. Next, for each of these graphs examine which locations admit an interaction with a crystal switch. For interactions with bounded range, such as the sword or boomerang, this is a constant for each switch. For something of unbounded range this may be linear in the level size. For each location from which an interaction with a crystal switch is possible, connect the corresponding nodes in the traversability graphs. This new graph is at most quadratic in the level size and we can determine reachability by running a standard graph-search algorithm. □

1.4 NP-Hard Zelda

This section gives NP-hardness results for various mechanics in Zelda. The first set of results uses limited resources needed for traversals to show hardness from Hamiltonian Path in grid graphs. This essentially follows Viglietta's Metatheorem 2 [Vig14], which states that games containing ***collectible cumulative tokens*** and ***toll roads*** that consume these tokens in order to pass them are NP-hard. In Zelda games, one can only carry up to a fixed number of these various items at any given time. We will have to generalize this inventory size for these proofs to apply.

The second set of results details some explicit instances of Hamiltonian Path implemented by the mechanics, and the third set of results, in Section 1.4.4, uses the "Nintendo" platform game NP-hardness framework [ADGV15] to show various combinations of enemies and weapons in the Zelda series are sufficient for NP-hardness.

1.4.1 Collectible Objects

In this section we prove the following theorem:

Theorem 1.4.1. *Generalized 2D Zelda with the hookshot, pots, pits, and small keys is NP-hard, if save-and-quit and dying are both prohibited.*

Applicable Games	Hookshot	Pot	Pit	Small Key
ALttP, LA, ALBW	Hookshot	Pot	Water	Small Key
PH	Grappling Hook	Barrel	Water	Small Key

Proof. We reduce from the Hamiltonian s–t path in maximum-degree-3 grid graphs [PV84]. Let $G = (V, E)$ be a maximum-degree-3 plane grid graph, and $s, t \in V$ be two vertices in that graph. The construction in [PV84] can easily attain the additional property that s and t are on the boundary face of G.[2] We construct a dungeon in the following way; refer to Figure 1.3. In our construction, we will describe distances such that the hookshot's length is 10 units.

The setting of the dungeon will be platforms surrounded by deep water tiles, so Link starting with only a quarter heart cannot step off of the platforms without dying. For each vertex $v \in V$ located at (x, y), place a plus shaped tetromino tile centered at $(10x, 10y)$. Place a pot containing a key in the center of block of each tetromino. Link enters the dungeon at s. Following t there is a sequence of $|V|$ doors with an exit at the end. If $|V|$ keys have been collected, then all the doors can be opened.

Link can only move from platform to platform by using the hookshot to move to a platform that has a pot. Due to the hookshot's length, Link can only move between platforms that are adjacent in the grid graph G. Since the pot blocks the path from one end of the platform to the other, Link cannot move off the platform unless the pot has been removed. This prevents Link from using the hookshot to reach this platform a second time. This means that a successful traversal of the dungeon is the same as a Hamiltonian s–t path in G. □

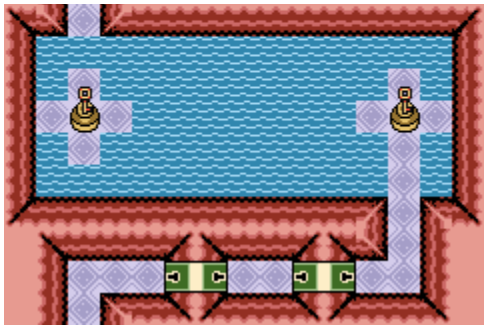


Figure 1.3: Platforms and locked door finale gadgets in the construction of Theorem 1.4.1 from a two-vertex graph.

One common game mechanic we prohibited above is the ability to save the game, quit to a title screen, and reload the game. In most games in the Legend of Zelda series, Link's inventory is preserved but his location is set to the previous outdoor exit used, and many destructible obstacles

[2]The reduction is from Hamiltonian cycle, and vertices s and t (which in fact have degree 1) come from a common vertex in a planar graph, so they belong to a common face, and the embedding can be chosen so that this face is the outside face.

like pots are restored to their original state. If Link quits due to dying with zero hearts, then Link may be reset to 3 hearts.

If we allow the player to die or save and quit, the above construction breaks because that action relocates Link to the beginning of the dungeon and regenerates the pots without removing the collected keys from Link's inventory, which corresponds to allowing the path in the graph to jump to s at any time and reuse edges, violating the Hamiltonian property.

Without prohibiting dying or save and quit, we can show NP-hardness if we can modify the construction to put the game in an unwinnable state if the player uses these mechanics. This can be done by augmenting the dungeon with a one-use door gadget placed at the entrance. One way to achieve this is to place an unavoidable damage region (such as floor spikes or flamethrowers) in which traversal of this region will do three hearts of damage to Link. Thus if Link starts the dungeon with more than three hearts, the region will be passable, but upon dying the level will restart with only three hearts and thus the unavoidable damage region will be impassable.

Corollary 1.4.2. *Generalized 2D Zelda with the hookshot, pots, pits, and small keys is NP-hard, if save-and-quit and/or dying are allowed.*

We also prove membership in NP for the case when pots cannot be pushed, only destroyed.

Theorem 1.4.3. *Generalized 2D Zelda with the hookshot, unpushable pots, pits, and small keys is in NP, if save-and-quit and dying are both prohibited.*

Proof. We give a nondeterministic algorithm that simulates a traversal of the dungeon.

In each phase of the traversal, we first guess Link's next destination: a tile with either a pot to destroy, a key to pick up, a door to unlock (if Link has at least one key), or the exit of the dungeon. Given a destination tile, we next guess the path for Link to take to get there. This path will involve walking and using the hookshot, but not destroying pots or opening doors, so the graph of reachable tiles will be static and the path can be found with breadth-first search in polynomial time (as in Theorem 1.3.1 but without including edges to pot tiles). If no path is found, then either we have guessed incorrectly at some point or the dungeon is impossible to exit, so the algorithm rejects.

If Link can reach the destination, then we have Link perform the corresponding action: destroying the pot (reducing the total number of pots), picking up the key (reducing the number of keys left to find), unlocking the door (reducing the number of locked doors), or exiting the dungeon (ending the traversal, so the algorithm accepts). Because each of these actions decreases the count of an amount of items (pots, keys, doors, or exits) found in the input, the maximum number of phases before the algorithm terminates is polynomially bounded by the input size. □

In some Zelda games, pots can be pushed but only once. By expanding the previous algorithm to remember which pots have been pushed so far and to include the action of pushing an unpushed pot, we obtain membership in NP for this variant as well.

Corollary 1.4.4. *Generalized 2D Zelda with the hookshot, once-pushable pots, pits, and small keys is in NP, if save-and-quit and dying are both prohibited.*

We leave open whether pushable pots (and many other combinations of mechanics in this section) are in NP. Pushable pots in particular are related to Push-1F, which is PSPACE-complete [ACD$^+$22], but Push-1F does not allow destroying pushable objects.

1.4.2 Additional Hamiltonian Path Hardness

Viglietta's Metatheorem 2 [Vig14] applies to a broad range of items beyond the hookshot, pots, and keys. We present a collection of other item sets which fit the framework as well.

- The Roc's Feather is an item that allows Link to jump a distance of one tile,[3] and Pegasus Seeds are consumable items which temporarily give Link the ability to run faster and jump a distance of two tiles. We require Link to collect seeds to jump over two-tile-wide gaps of lava separated by long-enough distances to wear out their effect.

  Corollary 1.4.5. *Generalized 2D Zelda with Roc's Feather, Pegasus Seeds, and lava is NP-hard.*

Applicable Games	Roc's Feather	Pegasus Seed	Pits
OoA, OoS	Roc's Feather	Pegasus Seed	Lava
MM	Goron Mask	Magic jar	Pit with jump ramps
TWW	Deku Leaf	Magic jar	Pit with high watchtowers

- Explosives, such as bombs, bombchus, and bomb arrows, are consumable items which can destroy certain obstacles, including cracked blocks that are regenerated when Link leaves the current room, area, or screen (in the 2D games). We require Link to collect explosives to pass through rooms blocked by cracked blocks.

  Corollary 1.4.6. *Generalized 2D Zelda with regenerating cracked blocks, explosives, and room transitions is NP-hard.*

Applicable Games	Regenerating Cracked Blocks	Explosives	Room
OoA [L3], OoS [L2], TMC [under Hyrule Town]	Cracked Blocks	Bombs	Screen
OoT [Goron City]	Brown Boulders	Bombs	Area
MM [Mountain Village]	Snow Boulders	Bombs	Area
TWW [Rock Spire Isle]	Large Cracked Rocks	Bombs	Area
TP [Snowpeak Ruins]	Large Barrels	Bombs	Room
SS [Lanayru Desert]	Rock Piles	Bombs	Area
BotW [Ja Baij Shrine]	Cracked Concrete Cubes	Bomb Arrows and Bow	Area

[3] If Roc's Feather lands a player on a hole, the mechanics of holes allows the player to escape the hole and end up traveling an additional tile of distance. Our reductions avoid this additional complexity by using lava or water and no holes.

- In The Legend of Zelda: Majora's Mask, Ice Arrows are consumable items that can create temporary ice platforms when shot into water from a Bow with sufficient magic power. We require Link collect arrows and Small Magic Refills to cross pools of water that he cannot climb out of.

  Corollary 1.4.7. *Generalized 3D Zelda with the Bow, Ice Arrows, water pools, and Small Magic Refills is NP-hard.*

Applicable Games	Ice Arrows	Freezable Water	Magic Refill
MM	Ice arrows	Deep Water	Magic jar

- Fairies in bottles are automatic heart-refilling consumable items that refill Link's heart meter when it drops to zero, preventing death. Unavoidable damage regions, such as a hallway with flamethrowers, laser-shooting eyes in the walls, a long fall, or a spiked floor, can be sufficiently large to deal a lethal amount of damage, requiring Link to use one fairy to traverse. While bottles usually occupy limited inventory slots, we consider the case with a number of bottles linear in the dungeon size, so that all required fairies can be carried at once. In Breath of the Wild fairies do not need to be contained in bottles and occupy the inventory like other collectible items.

  Corollary 1.4.8. *Generalized 2D Zelda with fairies, a linear number of empty bottles, and unavoidable damage regions is NP-hard.*

Applicable Games	Healing Item	Unavoidable damage
ALttP, ALBW, TMC	Fairy Bottles	Floor spike trap
OoT, MM, TWW, TP, SS	Fairy Bottles	Long fall
LA	Secret Medicine	Floor spike trap
OoA, OoS	Magic Potion	Floor spike trap
PH, ST	Purple/Yellow Potion	Floor spike trap
FS [Hero's Trial in Anniversary Edition]	Rupees	Blade trap
FSA [Tower of Flames]	Fairies	Fire bars
BotW	Fairies	Malice

- Multiple games in the series have magic invincibility items, including the Magic Cape, the Cane of Byrna, Nayru's Love, and the Magic Armor, which consume magic power (or rupees, in The Legend of Zelda: Twilight Princess) to prevent all damage. Link can collect Small Magic Refills to increase his magic meter and be required to drain it a specific amount to cross unavoidable damage regions.

  Corollary 1.4.9. *Generalized 2D Zelda with a magic invincibility item, Small Magic Refills, and unavoidable damage regions is NP-hard.*

Applicable Games	Magic Invincibility	Magic refill	Unavoidable damage region
ALttP	Magic Cape or Cane of Byrna	Magic jar	Floor spike trap
OoT	Nayru's Love	Magic jar	Blade trap
TWW, TP	Magic Armor	Magic jar and rupees	Blade trap

- Crystal Switches can be activated by limited use ranged weapons such as bombs or bow and arrows. We can construct a toll road by placing a pair of paths blocked by blocks of both colors. From the center of these obstacles, there is a crystal switch which can be reached by our ranged weapon allowing us to switch the color while in between the barriers and thus traverse them.

  Corollary 1.4.10. *Generalized Zelda with Crystal Switches and the bow and arrows or bombs is NP-hard.*

Applicable Games	Crystal Switch	Barriers	Activator
ALttP, LA, OoA, PH	Crystal Switch	Barriers	Bombs
TP [Temple of Time]	Crystal Switch	Shifting Walls	Arrows

1.4.3 Floor Puzzles are NP-Hard

In The Legend of Zelda: Link's Awakening, originally a 2D game, the dungeon Turtle Rock contains puzzles where a flashing block with the ability to replace pits with floor tiles is waiting next to a large set of pits. Link must remotely navigate the block to fill in every pit, which makes a Treasure Chest appear; the challenge being that the block can only traverse over pit tiles. A similar type of 2D puzzle appears in The Legend of Zelda: Oracle of Ages, where multiple dungeons have rooms with blue floors and a single yellow tile that follows Link's movements. If Link moves onto a blue tile, the yellow tile replaces the blue tile and leaves behind a red tile, and the goal is to eliminate all blue tiles.

Theorem 1.4.11. *Generalized 2D Zelda with pits and flashing floor-generating blocks, and Generalized 2D Zelda with colored-tile floor puzzles are both NP-complete.*

Applicable Games	Floor Puzzle
LA [Turtle Rock]	Pits and floor-generating block
OoA [Skull Dungeon], TMC [Dark Hyrule Castle]	Colored-tiles

Proof. Dungeons with either of these puzzle types are in NP, as a nondeterministic algorithm can guess the buttons to press for Link to traverse the tiles (himself or controlling the flashing block) to solve each room and reach the goal. Without leaving the room, each floor tile can only be filled once by the flashing block and a colored tile can only change from blue to yellow and from yellow to red at most once, therefore there need only be a polynomial number of required moves by monotonicity.

A room containing either of these puzzles is effectively an instance of the Hamiltonian Path problem on a grid graph with pits from a fixed start vertex to any end vertex. We reduce from the problem of Hamiltonian Circuit in grid graphs [IPS82] (with no specified endpoints) by laying out the graph using pit tiles (or blue tiles) and replacing one tile on the exterior with the flashing block (or the yellow tile) to specify the starting vertex. □

1.4.4 Fighting Monsters is NP-Hard

The "Nintendo" platform game NP-hardness framework, established in [ADGV15], shows several examples of how to use enemies which can be eliminated from one location, but otherwise block another location to build variables and clauses for a SAT reduction. One-way and crossovers are further needed to establish NP-hardness. In [DLL18] the notion of what enemies and environments are appropriate is generalized. In particular, we need one pathway which is impossible to cross if the enemy is present (likely because that enemy will kill Link) and another pathway which is disjoint from the first, but allows Link to safely eliminate the enemy. Crossovers and one-ways are prevalent in Zelda games. There are numerous pairing of items and enemies that could be used in this construction; we give a few examples below. In this section we assume enemies do not respawn. This is particularly relevant for The Legend of Zelda: Breath of the Wild where all enemies in the game respawn periodically during the Blood Moon.

Theorem 1.4.12. *Generalized 2D Zelda with Kodongos, low walls, and a sword is NP-hard.*

Applicable Games	Kodongos	Low wall	Sword
ALttP [Palace of Darkness]	Kodongos	Low wall	Sword

Proof. Kodongos are enemies which periodically shoot fireballs in a line. For our blocked traversal we will place a Kodongo behind a low wall at the end of a long hallway which is only one tile wide. Low walls prevent Link and Kodongos from walking over them, but do allow Link's sword swipes and the Kodongo's fireballs to pass over the wall. The hallway is long enough that the Kodongo will shoot a fireball down it at least once if Link tries to traverse the hallway while the Kodongo is there. If Link only has half a heart remaining (perhaps from an initial forced traversal of an unavoidable damage region) then this single fireball will kill him.

For our open traversal, we have another long hallway parallel to our blocked traversal but separated by another low wall. This will allow Link to move diagonally adjacent to the Kodongo and safely dispatch it while preventing entry into the other traversal. □

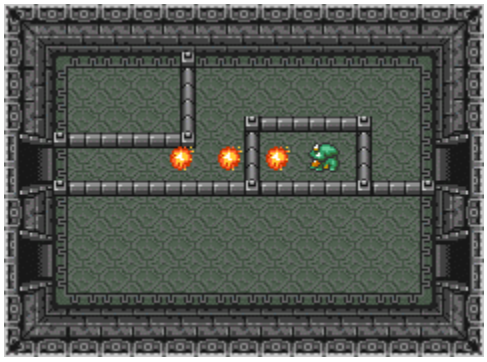


Figure 1.4: Gadget for Theorem 1.4.12

Theorem 1.4.13. *Generalized 2D Zelda with Buzz Blobs and the Master Sword is NP-hard.*

Applicable Games	Buzz Blobs	Sword beam ability
ALttP, OoA, OoS, ALBW	Buzz Blob	Master Sword
LA	Buzz Blob	Koholint Sword
TMC [Palace of Wind]	Electric Chu Chu	Sword Beam or Peril Beam technique
TFH	Buzz Blob	Sword Suit

Proof. Buzz Blobs are enemies which are not injured by sword swipes. If Link attempts to directly hit a Buzz Blob with a sword, Link will take damage instead.[4] The Master Sword allows Link to shoot a ranged attack if he is at full hearts. This ranged attack is able to damage the Buzz Blobs as long as Link is not adjacent to the Buzz Blob.

For our blocked traversal, we will place a Buzz Blob between two ledges. If Link is on top of the first ledge, the ranged attack from the Master Sword will go over the Buzz Blob. If Link jumps down while the Buzz Blob is present, he will take damage. At the very end of the dungeon, we will construct a hallway which is single tile wide filled with Buzz Blobs. If Link is still at full health when reaching this last hallway, Link will be able to dispatch the Buzz Blobs with ranged attacks from the Master Sword. Otherwise it will be impassable.

For the open traversal, we provide a pathway at the same height as the Buzz Blob which is separated by a pit. Link can safely blast the Buzz Blob from across the pit, but cannot cross the pit himself. □

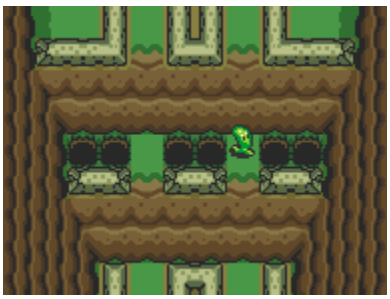


Figure 1.5: Gadget for Theorem 1.4.13.

Theorem 1.4.14. *Generalized 3D Zelda with Decayed Guardians and bombs is NP-hard.*

Applicable Games	Decayed Guardian	Bombs
BotW	Decayed Guardian	Bomb Rune

Proof. A Decayed Guardian is an enemy from The Legend of Zelda: Breath of the Wild that has a fixed location but has a laser which can rotate. If Link goes within range the Guardian will take a few seconds to "lock on" to Link as a target and then shoot a powerful laser. If Link is unarmored, this deals more than three hearts of damage, the starting maximum number of hearts in the game. However, the Guardian must have a clear line of sight to use its laser.

For our blocked traversal we will have a long, narrow hallway with a Guardian at the end. This hallway will be long enough so that Link will not be able to reach the Guardian before it fires. Further, the hallway is narrow preventing Link from being able to dodge the laser, thus rendering it impassible while the Guardian is there.

[4]This is not true of the Master Sword Lv.3, The Golden Sword, which can safely attack Buzz Blobs.

For our open traversal we will have another hallway which goes next to the Guardian but is separated by a wall which is slightly taller than Link. This allows Link to safely throw bombs over the wall while not being targeted by the Guardian and not being able to jump over the wall into the blocked traversal. Link could potentially try to bomb-jump over this barrier; however, with only three hearts and no armor attempting a bomb-jump would be lethal. □

1.5 PSPACE-Complete Zelda

In this section we show various mechanics in Zelda are sufficient for PSPACE-hardness. To do so, we make use of two common frameworks for proving the hardness of games involving motion planning: the "doors-and-buttons" framework and the "gadgets" framework.

In the ***doors-and-buttons framework*** [Vig14, For10], an agent traverses an environment consisting of "doors" and "buttons" connected by traversable pathways. A ***door*** can be ***open*** or ***closed***, and prevents passage when closed. Each ***button*** is connected to a set of doors. The agent can ***press*** a button it visits, which potentially changes the state of the doors to which it is connected. In particular, the result used in this chapter involves a button called a ***switch***, which swaps the state of all doors to which it is connected (from open to closed, or closed to open). A version of this framework, which we call ***1-switch-2-doors***, in which each button is connected to two doors and pressing the button swaps the openness of both doors, was shown to be PSPACE-complete in [VDZB15]. Proofs using this framework can be found in Section 1.5.1.

In the ***motion-planning-through-gadgets framework*** [DGLR18, DHL20], an agent traverses an environment consisting of "gadgets" connected by traversable pathways. Each ***gadget*** has ***locations*** where the agent can enter/exit the gadget, ***traversals*** which describe pairs of locations that can be traveled between in the current state (within the gadget), and ***states*** which describe which traversals are currently possible. Doing a traversal may change the state of that gadget in addition to changing the agent's location. Navigating a planar system of connected gadgets from one location to another is known to be PSPACE-complete when every gadget is a locking 2-toggle [DHL20], a door gadget [ABD$^+$20], or a self-closing door [ABD$^+$20]. These gadgets have the additional property that the set of possible traversals over all states are always a subset of ***tunnels***, which are disjoint pairs of locations (a perfect matching on the locations).

Figure 1.6 gives state diagrams for two of these gadgets. The ***locking 2-toggle*** has two directed tunnels where, after going down either one, the only allowed traversal is to return along the same tunnel in the opposite direction, resetting the gadget to the prior state. A ***door gadget*** has three tunnels: traverse, open, and close. The open and close tunnels are always traversable, which sets the state of the gadget to "open" or "closed" respectively, while the traverse tunnel can be traversed only when the gadget is in the open state. A ***self-closing door gadget*** is a modified door gadget with two tunnels, open and traverse, where traversing the traverse tunnel also closes it.

Before discussing the various particular mechanics, we first show the following general claim which applies to all our PSPACE-completeness results in this section.

Lemma 1.5.1. *Generalized 2D and 3D Zelda are in PSPACE.*

Proof. Savitch's Theorem [Sav70] shows that PSPACE equals NPSPACE, so we give the following simple algorithm to show containment within NPSPACE. As rooms are specified by a polynomial number of tiles with pixel-resolution collision masks or by polygons with fixed-point coordinates, the only varying quantities in the game are polynomially bounded states of Link, his items, enemies, and obstacles (including fixed-point positions and velocities), therefore a polynomial-space nondeterministic simulator could guess which player inputs to make to find a path to the goal, if a path exists. □

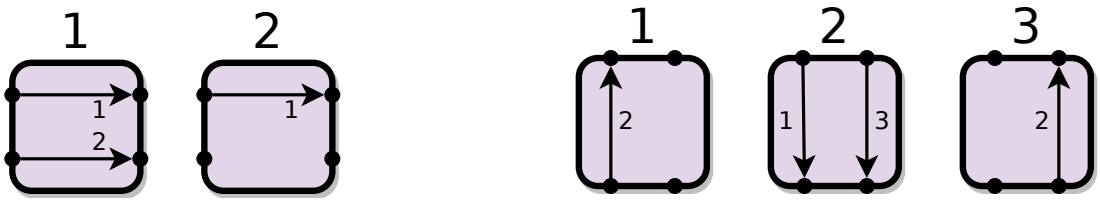


Figure 1.6: Gadget state diagram for the self-closing door (left) and the locking 2-toggle (right). Each box represents the gadget in a different state, in this case labeled with the numbers $1, 2, 3$. Dots represent the four locations of the gadget. Arrows represent transitions in the gadget and are labeled with the states to which those transitions take the gadget. For example, in state 1 of the self-closing door, the bottom traversal (the traverse tunnel) changes the state to 2, preventing further bottom traversal until the top traversal resets the state back to 1.

1.5.1 Statues and Pressure Plates are PSPACE-Complete

Many games in the Legend of Zelda series include dungeon puzzles where, for example, a statue needs to be pushed onto a pressure plate that opens a nearby door as long as the it is pressed down, but the statue can be pushed off of the pressure plate to be used for another use. More recently, in The Legend of Zelda: Breath of the Wild, many shrines contain Ancient Orbs, which can be carried around and placed in Ancient Pedestals to activate other nearby ancient technology.

These statues and orbs act as temporary, reusable keys that exists as objects in the world, as opposed to a collected inventory item like a Small Key that is permanently consumed to open a locked door. As long as we also have barriers to prevent the arbitrary transportation of these objects, this type of puzzle mechanic is PSPACE-hard to solve.

Theorem 1.5.2. *Generalized 2D Zelda with pushable heavy statues, pressure plates, doors, and stairs is PSPACE-complete.*

Applicable Games	Pushable Statue	Pressure Plate	Stairs
ALttP, OoA, OoS, FS, TWW, FSA, TMC, ALBW	Statue	Floor Button	Stairs
OoT, MM	Wooden Box	Small Pressure Plate	Stairs
TP [Temple of Time]	Pot	Pressure Plate	Stairs
PH, ST, SS [Pirate Stronghold]	Metal Box	Floor Button	Stairs
BotW (see Corollary 1.5.3)	Ancient Orb	Ancient Pedestal	Ladder

Proof. We reduce from motion planning with 1-switch-2-doors gadgets [VDZB15]. Shown in Figure 1.7, the gadget consists of a tri-partitioned room, one part with one statue and two pressure plates, each controlling a door in the other two parts. The statue partition's entrance is raised up from the floor by stairs, preventing Link from pushing the statue outside, and the other two parts each have two entrances on opposite sides of their inner door. Link may choose to open either door by pushing the statue onto the corresponding pressure plate, but with the one statue, at most one door can be opened at a time. □

Corollary 1.5.3. *Generalized 3D Zelda with Ancient Orbs, Pedestals, and Doors, along with ladders, is PSPACE-complete.*

Proof. We use the same construction as Theorem 1.5.2, replacing statues with Ancient Orbs, pressure plates with Ancient Pedestals, and short steps with ladders. Link is unable to carry an orb while climbing up ladders. □

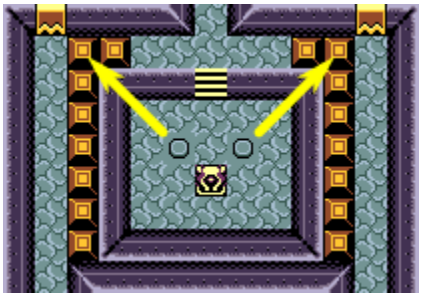


Figure 1.7: Construction for Theorem 1.5.2, in Oracle of Ages. The floor buttons open the corresponding shutter doors (signified with arrows) when the statue is pushed on them.

1.5.2 Magnetic Gloves is PSPACE-Complete

The Magnetic Gloves are an item introduced in The Legend of Zelda: Oracle of Seasons, a 2D game, that projects a north or south magnetic force in any of the four cardinal directions. Among other interactions, they allow Link to remotely attract or repel metal "N" orbs, which are polarized north. Two important properties are the fact that multiple metal objects in range of the force are affected simultaneously, and that metal orbs are affected at any distance, even when off-screen. Since there are no rooms in the game larger than 15×11 tiles or containing more than one metal orb, we make the assumptions that the force would affect multiple metal orbs simultaneously and that orbs cannot overlap other orbs, and consider the cases where it has an infinite range and when it has a finite range of up to 15 tiles from Link.

Theorem 1.5.4. *Generalized 2D Zelda with infinite-range Magnetic Gloves, metal orbs, ledges, and jump platforms is PSPACE-complete.*

Proof. We show PSPACE-hardness via reduction from motion planning with door gadgets [ADGV15]. Figure 1.8 shows our construction of a door gadget. In the center of the gadget is a metal orb that always blocks the traverse path (when closed) or the close path (when open). To open the door from the closed state, Link must be in the open path and repel the central metal orb with north magnetic force while facing down. To use the close path while in the open state, Link must use north magnetic force to repel the central metal orb while facing up. If Link tries to attract the central metal orb with south magnetic force, then one of the two ledge orbs will fall and permanently block the traverse path.

In an effort to embed the graph into a single room, we must prevent Link from using the Magnetic Gloves to manipulate a metal orb inside a gadget from far away. This is solved by entirely surrounding the room with a path with metal orbs on ledges leading to the goal, as in Figure 1.9. By selectively removing orbs (that would otherwise be dropped to block this path) in rows or columns which we intend the Magnetic Gloves to be used with a certain polarity, and placing our gadgets on disjoint sets of rows and columns, any unintended magnetic manipulations will permanently block the outer path and prevent the goal from being reached. □

Theorem 1.5.5. *Generalized 2D Zelda with at least 15-tile range Magnetic Gloves, metal orbs, ledges, and jump platforms is PSPACE-complete.*

Proof. Compared to infinite range, having a maximum force distance permits black-box gadget constructions, as we prevent external interference by laying-out gadgets far apart in the dungeon. However, the construction in Theorem 1.5.4 is not self-sufficient because we protected the central metal orb from the left or right by using a single, distant hallway with orbs poised to block traversal to the goal.

We bring these two aspects together by compacting the door gadget enough to run blocking hallways on both sides, as shown in Figure 1.10. With this construction, the metal orbs above the side hallways are within the 15-tile distance from anywhere in the gadget where horizontal magnetic glove usage could affect the central metal orb. Rather than running the goal hallway around the outside of the room, we thread it past every gadget on both sides, completing the reduction. □

1.5.3 Cane of Pacci is PSPACE-Complete

The Cane of Pacci is an item introduced in The Legend of Zelda: The Minish Cap, a 2D game, that shoots a bolt of magic that can enchant a circular hole tile, which will launch Link up an adjacent ledge if he enters the hole. As a pseudo-3D effect, the bolt ignores hole tiles that are not "vertically aligned" with Link's feet: if the bolt travels down a ledge, then the bolt will remember that it is now high above the floor. The bolt also ignores already-enchanted holes. In the game, the hole stays enchanted for a significant but limited time, so we consider both the finite- and infinite-duration generalizations.

Theorem 1.5.6. *Generalized 2D Zelda with fixed-duration Cane of Pacci, ground holes, ledges, and tunnels is fixed-parameter tractable with respect to cane duration.*

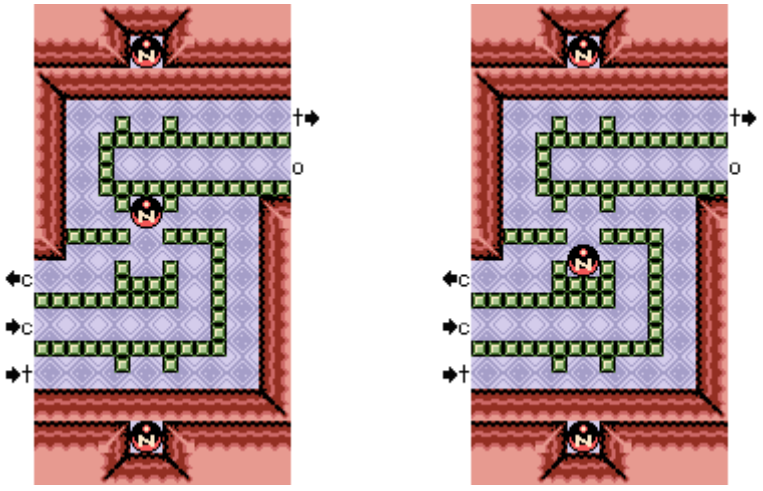


Figure 1.8: Construction of a door gadget using metal orbs, in the closed (left) and open (right) configuration. The open, traverse, and close paths (implementing the gadget's tunnels/traversals) are marked with directions. Link can move through the small gaps between the green quarter-tiles, but the orb cannot.

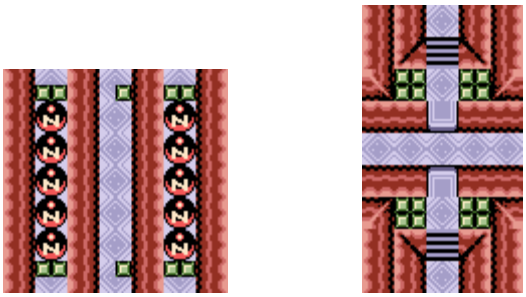


Figure 1.9: (left) Path lined with metal orbs to prevent Link from using the Magnetic Gloves while facing perpendicular into the path. (right) Crossover using jump platforms.

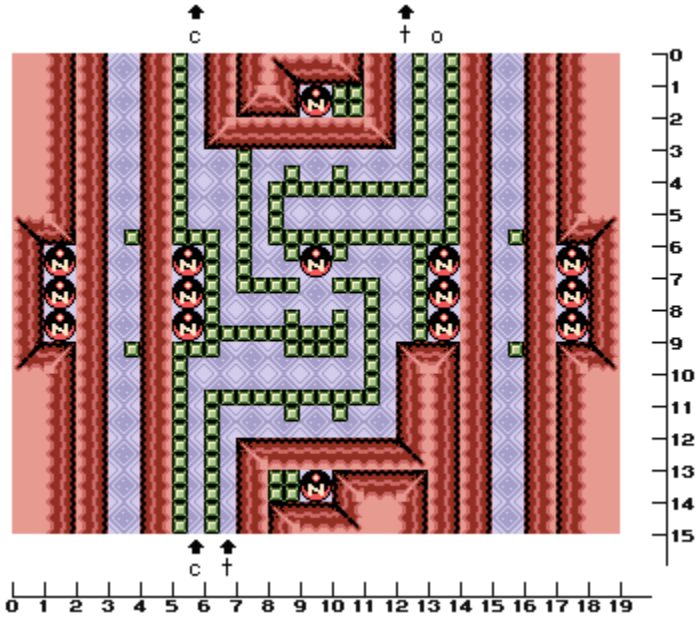


Figure 1.10: Compact construction of a door for 15-tile-range Magnetic Gloves, in the closed state. Hallways on the left and right are traversed at the end to reach the goal.

Applicable Games	Cane of Pacci (fixed-dur.)	Ground holes	Ledges	Tunnels
TMC	Cane of Pacci (fixed-dur.)	Ground holes	Ledges	Tunnels

Proof. Let the Cane of Pacci enchant holes for t frames before they automatically unenchant, and let Link's running speed be at most $v \leq 1$ tiles per frame, which is slower than the bolt's travel speed u.

For Link to use an enchanted hole, he must be within a circle of radius vt tiles centered at the hole from the duration of the enchantment. Symmetrically, all holes that are beyond vt tiles from his location cannot be enchanted and used, so without loss of generality no strategy for beating the dungeon ever has more than $h = O(v^2t^2) = O(t^2)$ holes that are enchanted at any point.

Supposing that there are n square tiles in the world and Link moves at a speed of 1 pixel per frame, he can be at $O(n/v^2)$ possible positions. Link can fire at most one bolt per frame, and each bolt that enchants a reachable hole travels for at most $vt/u < t$ frames. Under efficient play, where bolts are only ever shot at reachable holes, the total number of game configurations would be

$O(n/v^2 \times ht \times (t+1)^h) = n\,(t+1)^{O(t^2)}$.

Therefore, we can create a graph in linear time for fixed t, where each node is such a configuration of enchanted holes and to-be-enchanted holes around Link's location, connected by edges representing the effects of possible player inputs on the next frame: Link moving, Link shooting a bolt at a hole in view, or a bolt enchanting a hole. There will be a strategy to get to the end of the dungeon if and only if this graph has a path from the starting configuration node and an ending configuration node. □

Theorem 1.5.7. *Generalized 2D Zelda with infinite-duration Cane of Pacci, ground holes, ledges, and tunnels is PSPACE-complete.*

Applicable Games	Cane of Pacci (∞-dur.)	Ground holes	Ledges	Tunnels
TMC	Cane of Pacci (∞-dur.)	Ground holes	Ledges	Tunnels

Proof. To show PSPACE-hardness, we reduce from planar motion planning with self-closing doors [ABD+20]. Figure 1.11 shows our design for a self-closing door gadget. Link opens the door by entering the open path and firing the Cane of Pacci over the stone barrier at the hole below the ledge. When open, Link can later traverse by hopping from hole to hole, and the last hole will launch Link up the ledge, disabling the enchantment and thus closing the door behind him. The walls surrounding the holes, and the fact that the cane's bolt does not travel down to lower height levels when shot from the top of a ledge, prevent Link from opening the door anywhere except the open path. Because the enchantment does not have a finite duration, Link may be required to open a door but not return to use the door for an arbitrarily long time.

To lay out the graph of self-closing door gadgets in the game, we can make use of the crossover gadget, also shown in Figure 1.11, if the graph is not planar. Link can freely travel north or south on the upper level, and another path may run left and right by going down stairs and using a tunnel on the lower level. □

1.5.4 Magnesis Rune is PSPACE-Complete

In The Legend of Zelda: Breath of the Wild, a 3D game, Link obtains the multi-purpose Sheikah Slate, a tool that can be equipped with magical abilities called Runes. Among them is the Magnesis rune, which grants Link telekinetic power over metallic objects within a fixed distance. Compared to the Magnetic Gloves described in Section 1.5.2, Magnesis provides full 3D control of exactly

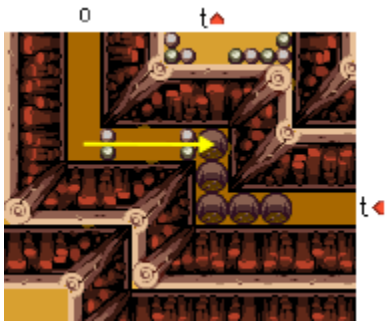


Figure 1.11: Gadgets in The Minish Cap: a self-closing door using holes for the Cane of Pacci (top). The path of the bolt for the Cane of Pacci is shown by the yellow arrow.

one targeted metal object in a world with more-advanced simulated physics, although Link cannot target objects that are out of his line-of-sight or that he is standing on.[5]

Theorem 1.5.8. *Generalized 3D Zelda with the Magnesis rune and large metal plates is PSPACE-complete.*

Applicable Games	Magnesis ability	Large metal plates
BotW [Great Plateau]	Magnesis Rune	Large metal plates

Proof. We reduce from motion planning with self-closing doors [ABD+20], using the gadget illustrated in Figure 1.12. Within a closed room, we construct two paths of platforms over pits: the traverse line, with two gaps that can only be crossed by placing a large metal plate as a bridge, and the open line, raised above the first close enough to use Magnesis on the plate but too far to use it as a bridge to cross paths. Both paths connect to the outside with small exit doors to keep the large metal plate inside.

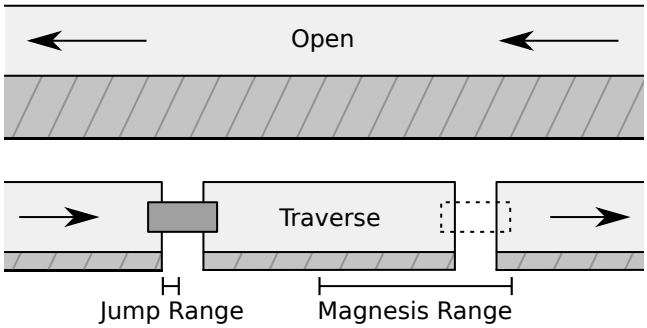


Figure 1.12: Construction of a door gadget using a large metal plate and platforms over pits, shown in the open state. The open line is raised above the traverse line. The layout was inspired by a puzzle in the Oman Au Shrine where the Magnesis rune is unlocked in The Legend of Zelda: Breath of the Wild.

The self-closing door starts closed, where the large metal plate is not within Magnesis reach of the start of the traverse line. To open the door, Link must use Magnesis from the open line to relocate the plate so that when Link later enters the traverse line, he can use the plate as a bridge across both gaps. Carrying the plate from the first gap to the second gap puts it out of Magnesis range of the entrance of the traverse line, which closes the door upon traversal. □

1.5.5 Minecarts Navigation

Minecarts are an environmental feature appearing in a variety of Zelda games which pose unique navigational challenges. In these games, Link can ride minecarts along paths of minecart tracks fixed onto the ground (or raised in the air), ending at minecart stop pads. Tracks may also pass through special doors which only open to let a minecart through. Levers can be used to change the state of sections of track, which can open new paths or create dead-ends that reflect the minecart backwards. We give PSPACE-completeness proofs that address the types of minecarts found in The Minish Cap, Oracle of Ages, and Oracle of Seasons.

[5]This mechanic intends to prohibit using Magnesis to fly by riding the object being controlled, but there is a glitch that involves stacking multiple specific metal objects to build a "flying machine" [Pup]. Our constructions place only a single metal object in a room so that flight cannot be achieved.

In The Legend of Zelda: Oracle of Ages and Seasons, Link can ride on minecarts which automatically transport him slowly along a track to a destination with no control during the ride beyond the use of some items, such as the sword or bombs. There are also levers Link can switch to rotate certain sections of track 90° to toggle the available path through a T-junction. In The Legend of Zelda: The Minish Cap, the dungeon Cave of Flames has fast minecarts that act similarly, although no items may be used during transport, and there are also four way junctions which can be switched between connecting opposite pairs of tracks. Falling off the end of the track or crashing into other minecarts is not a possible situation in the setups in any of these games, so we avoid that situation in our proofs. However, an intermediary simplyfying step considers a model where minecarts bounce off of each other when they collide.

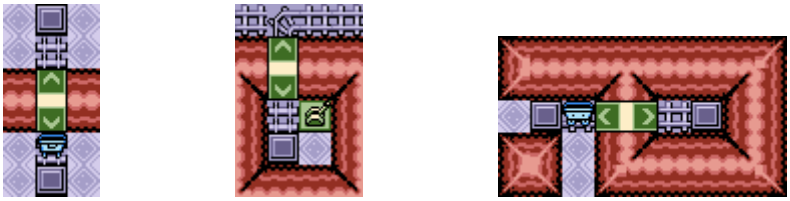


Figure 1.13: Gadgets for Oracle of Ages/Seasons: 1-Toggle while walking (left), 1-Toggle while riding a minecart (center), Diode while walking (right)

To introduce our minecart construction techniques, we describe the basic gadgets shown in Figure 1.13. The left image shows a 1-toggle while walking, which consists of a single minecart which can through a minecart-only door. When a minecart is present on Link's side of the door, he can ride it to the other side, and otherwise Link can't do anything else. The 1-toggle while riding a minecart consists of a single T-junction which leads to a minecart stop with a lever controlling the junction. When Link is riding a minecart toward the junction, if it is rotated toward him, he will end up in the minecart stop. At this point, he can flip the lever, and get back in the minecart to continue out the other side. If he enters the junction when it is rotated away from him, he bounces off and returns to where he came from.

Although the construction for a diode is not directly used in our proof, we present it here because it may be useful in future constructions, it demonstrates the rules of exiting a minecart, and it shows that Zelda with minecarts is not reversible, a fact that initially surprised us. When Link enters the minecart from any direction, he is taken into the dead-end room, but riding back out will force Link to exit onto the minecart stop pad on the left side. Thus, if Link entered from the bottom, he must exit on the left, and there is no way to traverse from the left to the bottom.

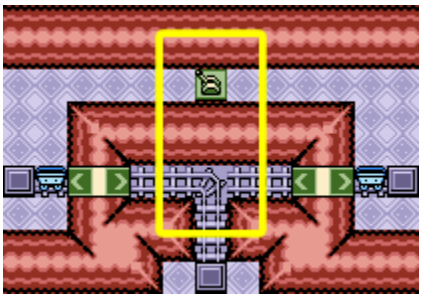


Figure 1.14: Minecart 2-to-1 Toggle for Oracle of Ages/Seasons, simplified under the assumption that minecarts bounce off of stationary minecarts. Adding minecart 1-toggles at each stop would achieve the same bouncing effect.

Theorem 1.5.9. *Generalized 2D Zelda with a sword, minecarts with tracks, levers to switch T-junctions, and minecart-only doors is PSPACE-complete.*

Applicable Games	Minecarts	Switches	T-junction	Minecart door
OoA, OoS, TMC	Minecarts	Levers	T-junction tile	Minecart door

Proof. We reduce from motion-planning with Locking 2-Toggles [DHL20].

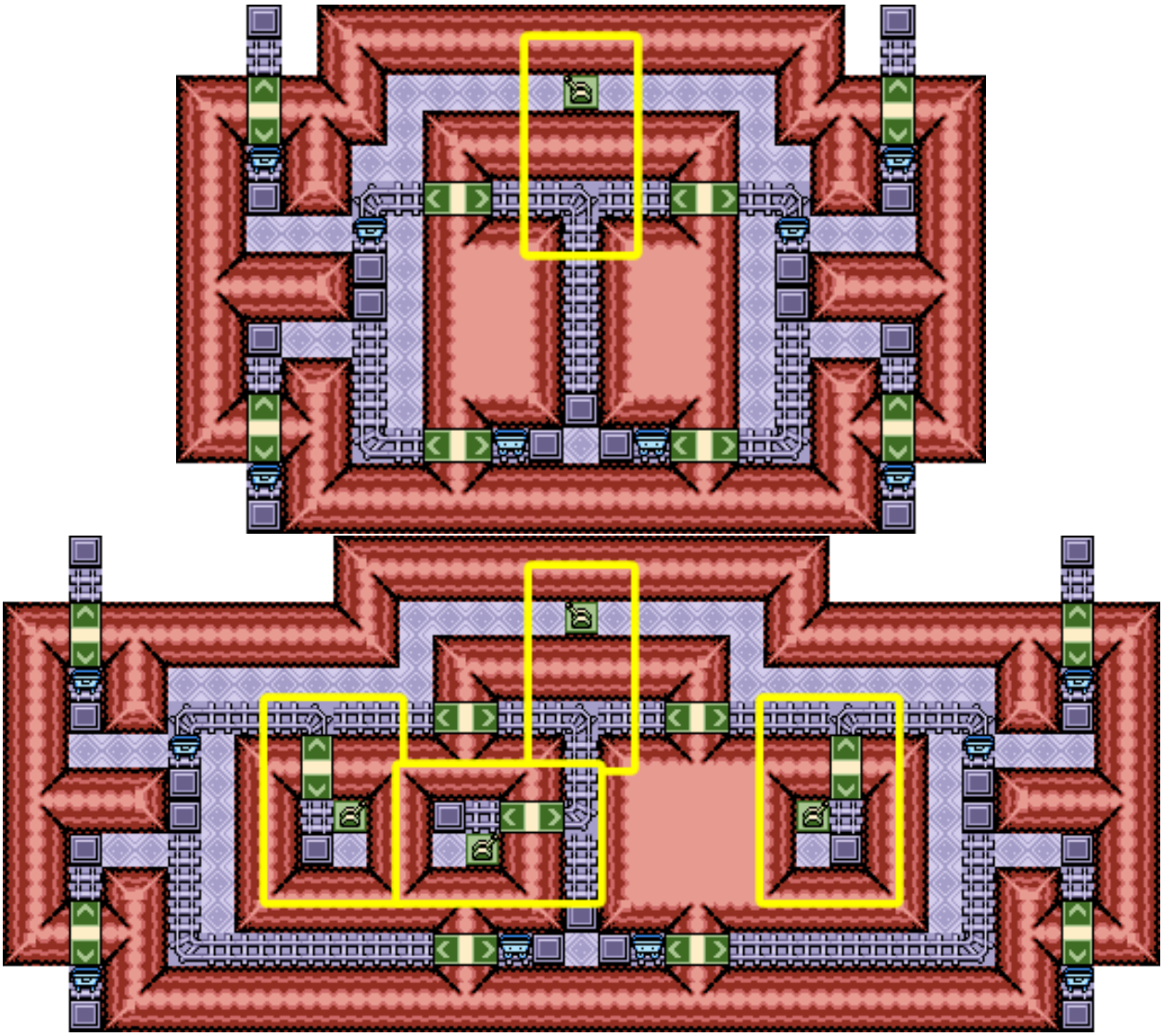


Figure 1.15: Minecart-based Locking 2-Toggle gadget for Oracle of Ages/Seasons. The simplified top figure assumes minecarts bounce off of stationary minecarts, while the bottom adds minecart 1-toggles to get the same effect. Levers and the junctions they switch are highlighted in yellow. Shown in the open state. The traversal lines go from bottom to top on the left and on the right.

Figure 1.15 shows our construction of a locking 2-toggle. Without loss of generality, we assume that if a moving minecart collides with a stationary minecart at a minecart stop, it will bounce backwards in the same way that it will bounce off of the dead-end side of a junction. The top half of Figure 1.15 depicts the simplified construction using this assumption. By adding minecart 1-toggles

in front of every minecart stop, as we also show in the bottom half of Figure 1.15, this simplifying assumption can be dropped.

The construction is centered around the 2-to-1 toggle gadget displayed in Figure 1.14 under our simplifying assumption that minecarts which collide simply bounce off of each other and return the direction they came from. This initially allows Link to enter from the left or the right side and exit from the bottom. Afterwards, Link is only able to go from the bottom to the side from which he last came. Thus this looks like a locking 2-toggle with two of the locations merged.

Now consider the entire gadget which has four entrances made of 1-toggles in the four corners of the gadget. To traverse from the bottom-left to the top-left, first Link uses the 1-toggle to enter the left section. The corridor to the top-left exit is blocked by a minecart, and the only way to move it is to ride it to the central minecart stop and return through the bottom 1-toggle. If the central junction was set to turn right, then Link must first flip the lever, which is easily accessible from both the left and right sections. Since Link can only exit a minecart onto a stop pad, this is the only traversal that gets past the top-left minecart.

In the open state, the central minecart stop is unoccupied, so Link can successfully relocate the cart blocking his path and proceed by using the 1-toggle at the top-left exit. If Link tries later to use the right traversal line, he will not be able to relocate the minecart blocking the top-right exit because the central minecart stop will be occupied. To undo this traversal and restore the open state, Link just needs to retrace his steps, and because the minecart brought from the central area blocks the bottom-left entrance (due to the stop pad location), that is Link's only available choice.

By symmetry, the above arguments also apply to right-line traversals. □

The above proof made clever use of the fact that minecarts can block Link's path by carefully placing the stop pads, forcing link to enter the minecart. It would be interesting to know whether this is necessary for hardness.

Question 1. Can one show PSPACE-hardness for Zelda minecarts with T-junctions without using the fact that minecarts can be used to block paths?

It may also be of interest, or enjoyable to find a simpler reduction for the four way junction. It is tempting to say hardness should follow from the 1-switch-2-doors result of [VDZB15], however, the minecarts create a 1-toggle like constraint on the pathways. This suggests a reduction from a toggle-lock or locking 2-toggle will be more appropriate.

1.6 Open Problems

Tables 1.4, 1.5, and 1.6 list many of the items, mechanics, and obstacle types from across all of the Zelda games, along with known complexity results. This table gives a sense of the significant work left to complete the quest of Zelda complexity.

It appears the first two Zelda games, The Legend of Zelda and The Adventure of Link, are the only Zelda games which have not been shown to be PSPACE-complete. Resolving the NP versus PSPACE gap for these oldest examples is a remaining challenge.

In The Legend of Zelda: Oracle of Ages, the Crown Dungeon has a collection of block-pushing puzzles with an interesting twist: all blocks with the same color move simultaneously (if there is an empty tile to move into) when any one of them is pushed. This global manipulation is similar to a discretization of the uniform global control for swarm robotics studied in [BHW$^+$13].

The Iron Boots are a common item in 3D Zelda games, first introduced in The Legend of Zelda: Ocarina of Time to allow Link to sink underwater further than he could swim, and was

expanded upon in The Legend of Zelda: The Wind Waker as a means to walk against strong winds and activate springboards, and even further in The Legend of Zelda: Twilight Princess by adding interactions with magnetic forces. While the Iron Boots are a nonconsumable inventory item with no inherent motive force, the fact that Link must choose whether or not to wear them to traverse a variety of hazardous terrain gives them the potential to be useful when combined with other items.

Items and Mechanics	Known
Small Key	1.4.1
Sword	1.4.12
Bow, Slingshot, Seed Shooter	1.4.10
Shield, Mirror Shield	
Bombs	1.4.6, 1.4.10, 1.4.14
Boomerang	
Heart Container	
Fairy, Secret Medicine	1.4.8
Health Refill Potion	
Flippers, Zora Armor Set	
Piece of Heart, Spirit Orbs	
Hookshot, Grapple Hook, Clawshot, Gripshot	1.3.1, 1.4.1, [ADGV15]
Power Bracelet	
Sword Beams	1.4.13
Warp Song, Warp Seeds, Warp Bell	
Blue Ring, Blue Mail	
Hammer	
Pegasus Boots, Pegasus Seeds	1.4.5
Shovel, Digging Mitts, Mole Mitts	
Magic Rod, Fire Rod, Fire Gloves	
Roc's Feather, Roc's Cape	1.4.5
Candle, Lamp	
Fire Arrows, Ember Seeds	
Gust Jar, Deku Leaf, Whirlwind, Gust Bellows	
Magic Refill Potion	
Bombchu	
Four Sword, Dominion Rod, Command Melody	
Bombos, Ether, Quake, Din's Fire	
Remote Boomerang	
Deku Stick, Boku Stick	
Fire Resist, Lava Resist, Heat Resist	
Ice Arrows	1.4.7
Cane of Byrna, Magic Cape, Nayru's Love, Magic Armor	1.4.9
Ember Seeds, Magic Powder, Oil Lantern	
Red Ring, Red Mail	
Timed Expiring Items	
Bomb Arrows	
Bug Net	
Ice Rod, Cryonis Rune	
Magic Mirror, Harp of Ages, Rod of Seasons	
Magnetic Gloves, Magnesis	1.5.2, 1.5.8
Super Bomb, Powder Keg	
Silver Scale, Golden Scale, Zora Tunic, Zora Armor, Mermaid Suit	

Table 1.4: All Items and Mechanics (and associated known results) from across all the Zelda games, as documented on [Zel21a] and [Zel21b], part 1 (continued in Table 1.5).

Items and Mechanics	Known
Cane of Somaria	
Chain Chomp	
Deku Nuts	
Farore's Wind, Travel Medallion	
Hover Boots	
Iron Boots	
Moon Pearl, Portals, Stumps	
Paraglider, Deku Leaf	1.4.5
Remote Bomb	
Sand Wand, Sand Rod	
Tornado Rod	
Whip	
Air Potion	

Items and Mechanics	Known
Axe	
Ball and Chain	
Beetle, Hook Beetle	
Cane of Pacci	1.5.6
Lightning Rod	
Minish Cap, Gnat Hat	
Phantom Hourglass, Sand of Hours	
Ravio's Bracelet	
Song of Time 3-day reset	
Spinner (item)	
Stasis Rune	
Switch Hook	1.3.3
Tingle Tuner	
Water Rod	

Table 1.5: All Items and Mechanics (and associated known results) from across all the Zelda games, as documented on [Zel21a] and [Zel21b], part 2 (continuing from Table 1.4).

Obstacles	Known
Raised red & blue barriers	1.3.4, 1.4.10, [ADGV15]
Pots	1.3.1, 1.4.1
Pits, unswimmable water or lava	1.3.1, 1.3.3, 1.4.5, 1.4.11, 1.4.13, 1.5.8

Obstacles	Known
Minecarts	1.5.5
Floor tile puzzles	1.4.11
Spinners (obstacle)	[DGLR18]
Metal 3D Physics Objects	1.5.8
Floor spikes, walkable lava or fire, long falls	1.4.8, 1.4.9

Table 1.6: Known results for some Obstacles from across all the Zelda games, as documented on [Zel21a] and [Zel21b].

Chapter 2

Motion Planning of Arbitrarily Many Robots

This chapter presents results from the paper titled "Complexity of Motion Planning of Arbitrarily Many Robots: Gadgets, Petri Nets, and Counter Machines" that the thesis author coauthored with Joshua Ani, Erik D. Demaine, Yevhenii Diomidov, Timothy Gomez, Dylan Hendrickson, and Jayson Lynch. At time of writing, this paper has been accepted to the Symposium on Algorithmic Foundations of Dynamic Networks (SAND), 2023 [ACD+23].

Overview

We extend the motion-planning-through-gadgets framework to several new scenarios involving multiple robots, and analyze the complexity of the resulting motion-planning problems. While past work considers just one robot or one robot per player, most of our models allow for one or more locations to *spawn* new robots in each time step, leading to arbitrarily many robots. In this one-player context, where the player can choose how to move the robots, we prove equivalence to Petri nets, EXPSPACE-completeness for reaching a specified location, PSPACE-completeness for reconfiguration, and ACKERMANN-completeness for reconfiguration when robots can be destroyed in addition to spawned.

2.1 Introduction

In this chapter, we build upon the ***motion-planning-through-gadgets framework*** that we used in Chapter 1. It has had significant study, primarily in the one-player setting, since its introduction [DHL20, ABD+20, ADHL22, ADD+22, DHHL22, ACD+22, Lyn20, Hen21]. The goal in that setting can be for the robot to traverse from one specified location to another (***reachability***) or for the system of gadgets to reach a desired state (***reconfiguration***) [ADD+22]. Existing results characterize in many settings which gadgets (in many cases, one extremely simple gadget) result in NP-complete or PSPACE-complete motion-planning problems, and which gadgets are simple enough to admit polynomial-time motion planning. This framework has already proved useful for analyzing the computational complexity of motion-planning problems involving modular robots [ADG+21], swarm robots [BMLC+19, CCG+20], and chemical reaction networks [AFG+22], the latter of which we investigate further in Chapter 6. These applications all involve naturally multi-agent systems, so it is natural to consider how the complexity of the gadgets framework changes with more than one robot.

In Section 2.2.2, we consider a generalization of this one-player gadget model to an arbitrary number of robots, and the player can move any one robot at a time. By itself, this extension does not lead to additional computational complexity: such motion planning remains in PSPACE, or in NP if each gadget can be traversed a limited number of times. To see the true effect of an arbitrary number of robots, we add one or two additional features: a ***spawner*** gadget that can create new robots, and optionally a ***destroyer*** gadget that can remove robots. For reachability, only the spawning ability matters — it is equivalent to having one "source" location with infinitely many robots — and we show that the complexity of motion planning grows to EXPTIME-complete with a simple single gadget called the ***symmetric self-closing door*** (previously shown PSPACE-complete without spawners [ABD+20]). For reconfiguration, we show that motion planning with a spawner and symmetric self-closing door is just PSPACE-complete (just like without a spawner), but when we add a destroyer, the complexity jumps to ACKERMANN-complete (in particular, the running time is not elementary). These results follow from a general equivalence to ***Petri nets*** — a much older and well-studied model of dynamic systems — whose complexity has very recently been characterized [Ler22, CO22].

In the original paper this chapter is based on [ACD+23], we also studied other settings which are beyond the scope of this thesis. A summary of those results are given below.

Zero-player with arbitrarily many robots. We considered the same concepts in a zero-player setting, where every robot has a forced traversal during its turn, and spawners and robots take turns in a round-robin schedule. zero-player motion planning in the gadget framework with one robot was considered previously [ADHL22, DHHL22], with the complexity naturally maxing out at PSPACE-completeness. With spawners and a handful of simple gadgets, we proved that the computational complexity of motion planning increases all the way to RE-completeness. In particular, the reachability problem becomes undecidable. This is a surprising contrast to the one-player setting described above, where the problem is decidable.

Impartial two-player with a shared robot. We considerd changing the number of robots in the downward direction. Past study of two-player motion planning in the gadget framework [DHL20] considers one robot per player, with each player controlling their own robot. What happens if there is instead only one robot, shared by the two players? This variant results in an ***impartial*** game where the possible moves in a given state are the same no matter which player moves next. To prevent one player from always undoing the other player's moves, we introduce a ***ko rule***, which makes it illegal to perform two consecutive transitions in the same gadget. In this model, we showed that two-player motion planning is EXPTIME-complete for a broad family of gadgets called "reversible deterministic interacting k-tunnel gadget", matching a previous result for two-player motion planning with one robot per player [DHL20]. In other words, reducing the number of robots in this way does not affect the complexity of the problem (at least for the gadgets understood so far).

2.2 The Gadget Model and Petri Nets

This section begins with gives a more detailed definition of the gadget model of motion planning, introduced in [DGLR18], generalizing the one-player setting used in Chapter 1, and defines another model: Petri nets.

2.2.1 Motion Planning Through Gadgets

In general, a ***gadget*** consists of a finite number of ***locations*** (entrances/exits) and a finite number of ***states***. Each state S of the gadget defines a labeled directed graph on the locations, where a directed edge (a, b) with label S' means that a robot can enter the gadget at location a and exit at location b, changing the state of the gadget from S to S'. Equivalently, a gadget is specified by its ***transition graph***, a directed graph whose vertices are state/location pairs, where a directed edge from (S, a) to (S', b) represents that the robot can traverse the gadget from a to b if it is in state S, and that such traversal will change the gadget's state to S'. Gadgets are ***local*** in the sense that traversing a gadget does not change the state of any other gadgets.

A ***system of gadgets*** consists of gadgets, their initial states, and a ***connection graph*** on the gadgets' locations. If two locations a and b of two gadgets (possibly the same gadget) are connected by a path in the connection graph, then a robot can traverse freely between a and b (outside the gadgets). (Equivalently, we can think of locations a and b as being identified, effectively contracting connected components of the connection graph.) These are all the ways that the robot can move: exterior to gadgets using the connection graph, and traversing gadgets according to their current states.

Previous work has focused on the robot reachability[1] problem [DGLR18, DHL20]:

Definition 2.2.1. For a gadget G, ***robot reachability for G*** is the following decision problem. Given a system of gadgets consisting of copies of G, the starting location(s), and a win location, is there a path a robot can take from the starting location to the win location?

Gadget reconfiguration, which had target states for the gadgets to be in, was considered in [ADD+22] and [Hen21]. We additionally investigate a problem where we have target states and multiple locations which require specific numbers of robots.

Definition 2.2.2. For a gadget G, the ***multi-robot targeted reconfiguration problem for G*** is the following decision problem. Given a system of gadgets consisting of copies of G, the starting location(s), and a target configuration of gadgets and robots, is there a sequence of moves the robots can take to reach the target configuration?

[DHL20] also defines two-player and team analogues of this problem. In this case, each player has their own starting and win locations, and the players take turns making a single transition across a gadget (and any movement in the connection graph). The winner is the player who reaches their win location first. The decision problem is whether a particular player or team can force a win. When there are multiple robots, we are asking whether any of them can reach the win location.

We will consider several specific classes of gadgets.

Definition 2.2.3. A ***k-tunnel*** gadget has $2k$ locations, which are partitioned into k pairs called ***tunnels***, such that every transition is between two locations in the same tunnel.

Most of the gadgets we consider are k-tunnel.

Definition 2.2.4. The ***state-transition graph*** of a gadget is the directed graph which has a vertex for each state, and an edge $S \to S'$ for each transition from state S to S'. A ***DAG*** gadget is a gadget whose state-transition graph is acyclic.

[1] In [DGLR18, DHL20], "reachability" refers to whether an agent/robot can reach a target location. Here we refer to it as ***robot reachability*** since for models such as Petri-nets the Reachability problem refers to whether a full configuration is reachable.

DAG gadgets naturally lead to bounded problems, since they can be traversed a bounded number of times. The complexity of the reachability problem for DAG k-tunnel gadgets, as well as the two-player and team games, is characterized in [DHL20].

Definition 2.2.5. A gadget is ***deterministic*** if every traversal can put it in only one state and every location has at most 1 traversal from it. More precisely, its transition graph has maximum out-degree 1.

2.2.2 Multi Robot Motion Planning with Spawners and/or Destroyers

In this chapter, we investigate one-player motion planning with multiple robots, where a single player controls a set of robots, with the ability to separately command each, moving any one robot at a time. There is no limit to the number of robots that can be at a given location. We include a ***spawner*** gadget which the player can use to produce a new robot at a specific location, providing an unlimited source of robots at that location. We optionally also include a ***destroyer*** gadget, which deletes any robot that reaches a specified sink location; such removal plays a role when we consider the ***targeted reconfiguration*** problem where the goal is to achieve an exact pattern of robots at the locations. If a system of gadgets only has a single spawner gadget we call that gadget the ***source*** and if the system only has a single destroyer gadget we call that the ***sink***.

We will show an equivalence between this one-player motion planning problem and corresponding problems on Petri nets. Through these connections, we establish EXPSPACE-completeness for reachability; PSPACE-completeness for reconfiguration with a spawner; and ACKERMANN-completeness for reconfiguration with a spawner and a destroyer.

2.2.3 Petri Nets

Petri nets are used to model distributed systems using tokens divided into dishes, and rules which define possible interactions between dishes. This is a natural model since many equivalent models have been defined, including Vector Addition Systems as well as Chemical Reaction Networks, which we discuss from a different perspective in Chapter 6.

Definition 2.2.6. A ***Petri net*** $\{D, R\}$ consists of a set of dishes D and rules R. A configuration t is a vector over the elements of D which represents the number of tokens in each dish. Each rule $(u, v) \in R$ is a pair of vectors over D. A rule can be applied to a configuration d_0 if $d_0 - u$ contains no negative integers to change the configuration to $d_1 = d_0 - u + v$. The volume of a configuration denoted $|d|$ is the sum of all its elements.

Definition 2.2.7. A reachable set for a Petri-net configuration, denoted $REACH_P(\{D, R\}, t)$, is the set of configurations of a Petri net reachable starting in configuration t and applying rules from R.

We can view a system of gadgets with multiple robots as a set of gadget states Γ and a vector l indicating the counts of robots at each location. We can define the set of reachable targeted configurations as $REACH(\Gamma, l)$ similarity to Petri nets.

2.3 Equivalence between Petri Nets and Gadgets

We present transformations that turn Petri nets into gadgets, and gadgets into Petri nets. We use these simulations to prove the complexity of robot reachability and reconfiguration with arbitrarily many robots.

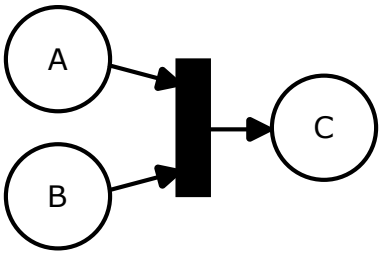


Figure 2.1: General Petri-net rule (u, v), where u's nonzero dishes are shown on the left side and v's nonzero dishes are shown on the right side.

Gadgets to Petri Nets. We can transform a set of gadgets into a Petri net where each location, besides the source and sink, is represented as a ***robot dish***. Each gadget besides the spawner and destroyer is given a number of ***state dishes*** equal to its states, and each transition of the gadget is represented by a ***rule***. The set of dishes D is $D_{STATE} \cup D_{LOCT}$, the union of state and robot dish sets, respectively.

A configuration of robots and gadgets is represented by a Petri-net configuration t satisfying the following:

- Each k-state gadget is simulated by k unique dishes in D_{STATE}, one per state. The state of the gadget is represented by a single token which is contained in the corresponding dish, and the other $k - 1$ dishes are empty.
- Each location in the system of gadgets is simulated by a unique dish in D_{LOCT}. The number of tokens in that dish is equal to the number of robots at that location.

A Petri net $\{D, R\}$ simulates a system of gadgets G if for any configuration $\{\Gamma, l\}$ of G represented by Petri-net configuration t, each configuration in $REACH_G(\Gamma, I)$ is represented by a configuration $REACH_P(\{D, R\}, t)$ and each configuration in $REACH_P(\{D, R\}, t)$ represents a configuration in $REACH_G(\Gamma, I)$.

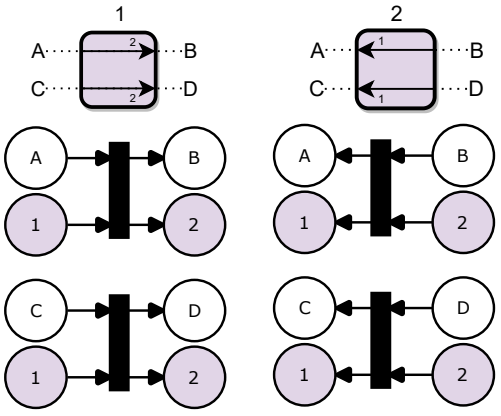


Figure 2.2: Petri-net rules which simulate a 2-tunnel toggle gadget

Lemma 2.3.1. *For any set of deterministic gadgets S, any system of multiple copies of gadgets in S with a spawner (and optionally, a destroyer) can be simulated by a Petri net.*

Proof. We first explain how to create the rules for gadgets that are not connected to the source or sink locations. Each gadget transition will be represented by a unique rule. For example the 2-tunnel toggle gadget is shown in Figure 2.2 and has four transitions. It can be traversed:

- from A to B in state 1,
- from C to D in state 1,
- from B to A in state 2, and
- from D to C in state 2.

The four corresponding rules for the gadget are drawn in Figure 2.2 as well. Each rule takes in one token from a robot dish and one from a state dish, and places one token in a robot dish and one in a state dish. The token being moved between robot dishes models moving one robot across a gadget, and the token being moved between state dishes models the state change of the gadget.

If a gadget is connected to the source, any transition from the source is represented by a rule that only takes in a state token, producing two tokens. One token is output to a location dish and one to a state dish. If a transition is connected to the sink then the rule takes in two tokens and outputs only a state token. These special cases are shown in Figure 2.3. Note that we do not have an actual dish for the source so the player may spawn multiple robots at the source but they do not appear in the simulation until they traverse a gadget.

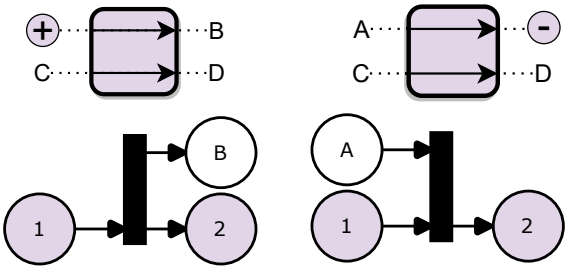


Figure 2.3: Left: Rule we include when a gadget can be traversed from the source. Right: Rule we include when a traversal leads to the sink.

For each configuration of a system of gadgets, there exists a configuration of the Petri net with dishes that represent the gadgets and locations. Each rule of the Petri net acts as a traversal of a robot changing the state of a gadget. The rules need the gadgets state token to be in the correct dish, and a robot token in the location dish representing the start traversal. □

Petri Nets to Gadgets. We simulate a Petri net with symmetric self-closing doors using a location for each dish, where each rule is represented by multiple gadgets. We also have a single ***control robot*** which starts in a location we call the ***control room***. The other robots are ***token robots*** which represent the tokens in each dish. At a high level, our simulation works by "consuming" the input tokens to a rule to open a series of tunnels for the control robot to traverse. The control robot then opens a gadget for each output to allow token robots to traverse into their new dishes. We use the source and sink to increase and decrease rules as needed. Figure 2.5 gives an overview.

Symmetric self-closing door. The ***symmetric self-closing door*** is a deterministic 2-state 2-tunnel gadget shown in Figure 2.4. The states are $\{1, 2\}$ and the traversals are

- in state 1 from A to B changing state to 2, and

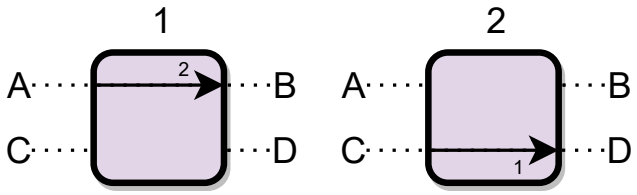


Figure 2.4: Symmetric self-closing door

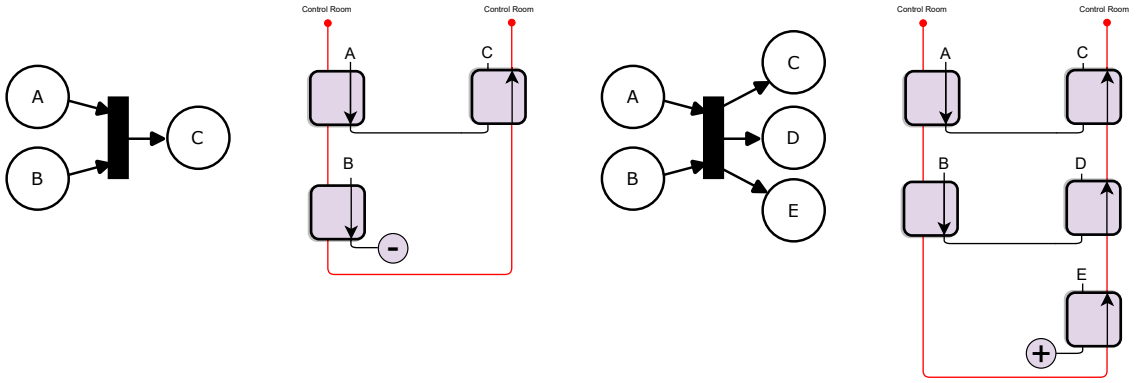


Figure 2.5: How to simulate a rule which decreases volume (Left) and a rule which increases volume (Right).

- in state 2 from C to D changing state to 1.

Using this simulation we prove two problems in Petri-nets are polynomial time reducible to the gadgets problems we are interested in. [Esp05] lists many problems including the ones we describe here[2]. First is production, this problem asks given a Petri-net configuration and a target dish, does there exist a reachable configuration which contains at least one token in the target dish. Configuration reachability asks given an initial and target configuration, is the target reachable from the initial configuration.

Lemma 2.3.2. *Production in Petri nets is polynomial time reducible to robot reachability with the symmetric self-closing door and a spawner. Configuration reachability in Petri nets is polynomial-time reducible to multi-robot targeted reconfiguration with the symmetric self-closing door and a spawner.*

Proof. For a rule (a, b) we include $|a| + |b|$ copies of the gadgets. There is a gadget for each input to the rule; these gadgets can be traversed from the location representing an input dish to an intermediate location, opening another tunnel for the control robot to traverse. The control robot must traverse all the input gadgets the goes through the tunnels of the output gadgets. The control robot opens the doors of these gadgets allowing the robots moving from an intermediate wire to traverse to a location representing the output dishes.

If a rule would increase the volume, the surplus output gadgets will allow traversal from the spawn location instead of an input gadget. If a rule decreases the volume, then the surplus input gadgets send robots to a "sink" location instead of an output gadget. We do not require a true sink in this case because we can add an extra location which robots can be held instead of being deleted. If we do not connect this location to any other gadget, then the robots can never leave and can be thought of as having left the system.

[2]Problems names may differ.

Production reduces to robot reachability since a robot can reach a location if and only if a token can reach the corresponding dish. If token is placed in a dish, it must have moved through a rule gadget. The robot can only move through a rule gadget if the number of robots in the dishes are at least the number of tokens of the left hand side of the rules to open the tunnels for the control robot to move through.

Configuration reachability in Petri nets reduces to multi-robot targeted reconfiguration. The target and initial states of the gadgets are the same. The only difference between the initial configuration and the target is the number of robots at each location, equal to the counts in the instance of Configuration reachability for Petri nets. The number of robots at each location is equal to the number of tokens in each dish. The targets for each intermediate wire is 0 and in the control room 1. Thus, it is never beneficial to partially traverse a rule gadget. □

2.4 Complexity of Reachability

The reachability problem for a single robot is very similar to the well-studied problem in Petri nets called coverage. The input to the coverage problem is a Petri net and a vector of required token amounts in each dish, and the output is yes if and only if there exists a rule application sequence to reach a configuration with at least the required number of tokens in each dish.

Definition 2.4.1 (Coverage Problem)**. Input:** A Petri net $\{D, R\}$, and vectors d_0 and d_c.

Output: Does there exist a reachable configuration $d \in REACH(\{D, R\}, d_0)$ such that $d[k] \geq d_c[k]$ for all $0 \leq k < |D|$.

Theorem 2.4.1. *Robot reachability is EXPSPACE-complete with symmetric self-closing doors, a spawner, and optionally a destroyer.*

Proof. We can solve robot reachability by converting the system of gadgets to a Petri net which simulates it as in Lemma 2.3.1. In this simulation, a token can be placed in a location dish if and only if a robot can reach that location represented by that dish. Determining if a single token can be placed in a target dish, the production problem, is a special case of coverage problem where the target dish is labeled with 1 and all others labeled with 0. We can use the exponential-space algorithm for Petri-net coverage shown in [Rac78] to solve robot reachability. When simulating the sink we require rules that decrease the volume of a Petri net. This algorithm works for general Petri nets so it implies membership with a sink.

For hardness, we first reduce Petri-net coverage to Petri-net production by adding a target dish T starting with 0 tokens and a new rule. This rule takes as input the number of tokens equal to the goal of the coverage problem and produces one token to the t dish. This token can only produced if the reach a configuration that has at least the target number of each species. We then use Lemma 2.3.2 to reduce production to robot reachability with the self-closing symmetric door and a spawner. It is relevant to note the first reduction does not work when exactly the target numbers are required. The reduction works even when not allowing the sink as described in Lemma 2.3.2.

□

2.5 Complexity of Reconfiguration

The reconfiguration problem has been studied in the single-robot case as the problem of moving the robot through the system of gadgets so that each gadget is in a desired final state. Targeted reconfiguration not only asked about the final states of the gadgets, but the location of the robot as

well. Here, we study multi-robot targeted reconfiguration which requires both that all gadgets are in specified final states and that each location contains a target number of robots.

Definition 2.5.1. For a gadget G, the ***multi-robot targeted reconfiguration problem for G*** is the following decision problem. Given a system of gadgets consisting of copies of G and the starting location(s) a target configuration of gadgets and robots, is there a sequence of moves the robots can take to reach the target configuration?

The complexity of multi-robot targeted reconfiguration depends on whether we allow a destroyer. If we do not allow for a destroyer, the complexity is bounded by polynomial space since we can never have more robots than the total target size. If we allow for the ability to destroy robots, then the reconfiguration problem is the same as the configuration reachability problem in Petri nets from our relations between the models above. This is a fundamental problem about Petri nets and was only recently shown to be ACKERMANN-complete [Ler22, CO22].

Theorem 2.5.1. *Multi-robot targeted reconfiguration is ACKERMANN-complete with symmetric self-closing doors, a spawner, and a destroyer.*

Proof. For membership we can solve multi-robot target reconfiguration by converting the gadgets to the Petri net using Lemma 2.3.1. The target configuration is a state token for each gadget in the dish of its target state, and a number of tokens in each location dish as the number of robots in the target configuration. We can then call the ACKERMANN algorithm for configuration reachability in Petri nets shown in [LS19].

For hardness we can reduce from configuration reachability. It was shown in [CO22] that configuration reachability is ACKERMANN-hard. □

The reduction presented in [CO22] vitally uses the ability of Petri nets to delete tokens, so we must use a sink in our simulation. Without a sink, we have PSPACE-completeness for multi-robot targeted reconfiguration.

Theorem 2.5.2. *Multi-robot targeted reconfiguration for symmetric self-closing doors and a spawner is PSPACE-complete.*

Proof. Consider the input to the reconfiguration problem: two configurations of a system of gadgets. Namely, the start and end state of all the gadgets, and a start and end integer for each location. Since we can never destroy a robot once it is spawned, it always exists, so the player cannot spawn more robots than the total number of robots in the target configuration. We can then solve this problem in NPSPACE by nondeterministically selecting a robot to move, either from the source or another location. If we ever increase the total number of robots above the target we may reject. If we ever reach the configuration with the correct gadget states and robots at each location accept. Since PSPACE = NPSPACE we get membership.

We inherit hardness from the one-player single-robot case by not including the source or connecting it to an unreachable location. □

2.6 Open Problems

For one-player multi-agent motion planning, we investigated robot reachability and multi-agent targeted reconfiguration. The hardness for both these problems relies on simulating Petri nets with a symmetric self-closing door. Do there exist reversible gadgets for which the problem is the same complexity? How does this relate to reversible Petri nets?

Part II

Team Games and Communication

Chapter 3

Undecidability of Team Multiplayer Games

This chapter presents results from the paper titled "Cooperating in Video Games? Impossible! Undecidability of Team Multiplayer Games" that the thesis author coauthored with Jayson Lynch. This paper appeared in the International Conference on Fun with Algorithms (FUN), 2018 [CL18], and was published in Theoretical Computer Science (TCS), 2020 [CL20].

Overview

We show the undecidability of whether a team has a forced win in a number of well known video games including: Team Fortress 2, Super Smash Brothers: Brawl, and Mario Kart.To do so, we give a simplification of the Team Computation Game [HD09] and use that to give an undecidable abstract game on graphs. This graph game framework better captures the geometry and common constraints in many games and is thus a powerful tool for showing their computational complexity.

3.1 Introduction

Multiplayer videogames account for a large portion of the video game market and yet the additional computational complexity added by coordinating different team members has not seen much study from a theoretical standpoint. We finally bridge the gap between known theoretical models where imperfect information team games are known to be much more computationally complex and popular, commonly played video games.

In a series of papers [Rei79, PR79, Rei84, PRA01], Reif and Peterson explored the computational complexity of games of imperfect information. One surprising result was a proof that unbounded team multiplayer games with imperfect information can be undecidable, despite having a bounded configuration space in the game itself. This work has been expanded to include formula and constraint logic games [HD09]; however, to the best of our knowledge, no commonly played game has been shown to be undecidable using this framework.

The computational complexity of video games has started becoming a popular topic of inquiry. Past research includes the study of classic arcade games like Pac-Man [Vig14], classic Nintendo games such as Mario and the Legend of Zelda [ADGV15], to more modern games like Candy Crush [GLN14], Portal [DLL18], and Angry Birds [SRG17]. However, all of these papers considered single-player, perfect information versions of the game. These are both aspects that, when altered to team multiplayer and imperfect information, intuitively and theoretically should make the games

much more computationally challenging. The work in this chapter critically utilizes these properties to show far stronger hardness results than usually appears. At the time of publishing the original paper this chapter is based on [CL20], we were aware of only one other video game, Braid [Ham14], which had been shown to be undecidable. However, it was shown by the construction of a counter machine using enemy units and thus playing such a level will require unbounded computational resources. The game Recursed has since been proven undecidable as well [DKL20] using a more space-efficient – but still unbounded – construction based on the Post Correspondence Problem. The ability for a bounded game state to be able to lead to an undecidable problem has been remarked on by others are a fascinating feature of this type of problem [HD09].

In addition, much of the past work on video games has focused on environmental obstacles such as toggles for moving platforms and locking doors, rather than more central mechanics of the game. An aesthetic advantage of our proofs are that they focus on player vs player interaction and use the central combat mechanics of the game as core elements in the reduction. This focus complements our work on models such as motion-planning-through-gadgets in Chapters 1, 2, 5, and 6.

Organization This chapter is organized into two parts. The first half deals with abstract games and builds a framework for later reductions. In particular, Section 3.2 details the kind of gadgets involved in our team multiplayer graph game. Section 3.3 reduces the Team Computation Game to the Team Graph Game using our simplification of the former, the Team DFA Game. The second half, Section 3.4, applies this framework to show the undecidability of several popular multiplayer games.

3.2 Team Graph Game Components

In this section we describe the different components of our undecidability framework which will be instantiated in the Team Graph Game which we define and show to be undecidable in Section 3.3. Roughly speaking, it is a multi-player game with two teams, which we will refer to as blue and red, on a graph where each team wants to get one of their players to one of the win nodes. Players take time moving from node to node and from a node other nodes may be visible, allowing the player to determine if another player is there. In addition, some nodes will allow a player to guard an edge. A player attempting to cross a guarded edge will be eliminated and no longer be able to perform any useful actions. In our reduction we want to simulate a DFA which takes input from blue and red players and changes state based on this input. The state of the DFA will be encoded in the location of one player on the blue team, called the runner, and we call the other blue team members executors. The DFA entering an accept state will correspond to the runner being on a path which leads freely to a win node. The red team will supply their inputs by guarding some of the possible paths of the executors, while the executors will provide the blue team's inputs by choosing among unguarded paths to take. Both teams' inputs will force the runner to take a certain path through the region representing the DFA transition function. This section of the chapter will describe these gadgets and their function in detail and Section 3.3 will formalize and complete the proof.

We break this framework down into several important gadgets each given their own subsection. We require a state transition gadget to manage the state of a deterministic finite automaton. This is described in Subsection 3.2.3. Both teams need to set variables which are taken as input to the DFA which is done with the choice gadgets described in Subsection 3.2.2. We need to synchronize all of the players so that the variable choices and DFA execution all occur in the proper order. This is done with a delay gadget described in Subsection 3.2.1. Finally, there is an optional initializer gadget which forces players from initial locations to the pathways needed in the gadgets. This is

described in Subsection 3.2.4. These gadgets are put together in Section 3.3, as shown in Figure 3.5.

In this chapter we use the following diagram conventions. Edges and nodes in the graph potentially containing red Team players are red and use square for nodes. Edges and nodes potentially containing blue Team players except for the runner are blue with circles as nodes. Edges and nodes potentially containing the runner are black with diamonds for nodes. The graph contains both directed and undirected edges. Bold edges represent many different paths which serve similar function but are only accessed by one player. They are often accompanied by a label of how many edges are represented. Triple dots denote the continuation of a pattern, often many of the same type of edge. In contrast to bold edges, a different player will generally occupy each of these. Combat zones are pairs of nodes and edges and are denoted by a lightly colored red or blue triangle. The color dictates which team is posing a threat in the combat and always involves a node guarding an edge. If relevant, the combat zone is labeled with the length of time an enemy must spend traversing a guarded edge to be eliminated. These zones also imply visibility; however, we do not explicitly label visibility in all of our diagrams. Labeled boxes are used to refer to unrepresented gadgets, and dotted boxes are used to delineate different gadgets whose internal details are in the figure. An encircled W is a win node. Other labels and notation will hopefully be clear from context. Some of these conventions are used more liberally in the diagrams in Section 3.4 along side more representative pictures for the games.

3.2.1 Delay Gate

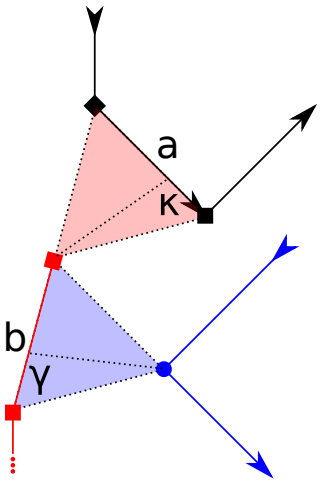


Figure 3.1: Delay Gate, a gadget to delay the runner until a blue executor arrives to remove the red attacker.

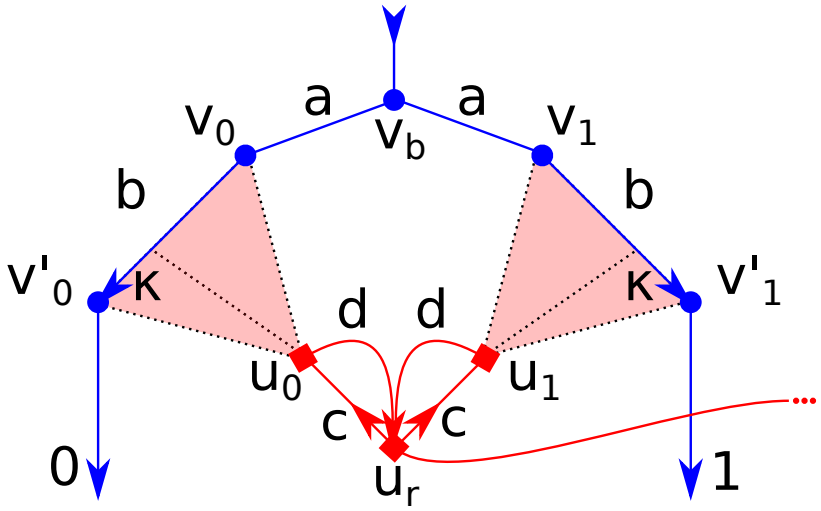


Figure 3.2: Red Choice Gadget, a gadget for a red player to force a blue player to take exit 0 or 1.

The simplest gadget is the Delay Gate, as seen in Figure 3.1. The blue runner moves through the maze and is frequently blocked from making progress by a red player guarding a combat zone (edge) from an attack node. To progress, one of the blue executors must arrive at its own attack node which threatens the red guard, who must escape outside the combat zone (and far from its attack node) or be eliminated. As long as the red-beats-blue time $\kappa < a$ and the blue-beats-red time $\gamma < b$, the Delay Gate achieves this goal.

3.2.2 Red Team Choice Gadget

The Red Team Choice Gadget gives the red team the ability to influence the path of a blue team player's movement. Detailed in Figure 3.2, a blue team member starts at node v_b and wants to exit out of v'_0 or v'_1, and a red team chooser at u_r (or its neighbors) will be able to force the outcome without fully preventing progress.

The graph is symmetric, so suppose without loss of generality that the red chooser wants the blue player to exit out of v'_1. Given their choice of where to start among the subgraph $\{u_0, u_r, u_1\}$, they can successfully block the v'_0 exit by simply waiting at u_0 and attacking if the blue player tries to traverse edge (v_0, v'_0). If $c > a + b$, no starting location of the red chooser allows them to prevent the blue player from reaching both exits: the red chooser must start at least $d = a + b - \kappa$ time units away from u_0 to block v'_0, which means starting $c + (c - d) > a + b$ away from u_1 which is too far to block v'_1 as well.

An optimal strategy for the blue player to guarantee progress is thus to immediately move towards v'_0. Either the red chooser is blocking v'_1 and the blue player will leave through the preferred exit, or red chooser is blocking v'_0 and the blue player will have time to turn around and reach v'_1 (the preferred exit) before the red chooser can reach u_1.

3.2.3 State Transition Gadget

Whereas the Red Team Choice Gadget is used to allow red team to influence a blue executor's path, the State Gate gadget is used to allow blue team executors to influence the blue runner's path. The "core" of a State Gate is essentially two Delay Gates sharing the same red guard who, unlike the Red Team Choice Gadget, is able to simultaneously block both exits for the blue runner. Depending on which of the two paths the blue executor is on, it will be able to safely open one of two exit paths for the blue runner.

Looking ahead to our undecidability proof for TGG, we generalize the core into a State Gate by first allowing for two independent hallways per blue executor "input" and second to allow for multiple independent hallways for the blue runner. Detailed in Figure 3.3, the first can be constructed using two cores (each with one hallway of each "input" type) or with one core modified such that the red guard's edges are the target of two blue executor attack nodes at once. The second generalization is simply constructed using multiple instances of the first in series along the blue executor's paths, one per required blue runner hallway.

The core works correctly as long as the red guard has visibility on the blue runner and executor and $\gamma < b < a - \kappa$. When safe, the red guard can mimic the blue runner's movement and always reach the closer attack node fast enough to block the path, but when the blue executor arrives on one side, the red guard must vacate the corresponding attack zone and can only safely block the opposite path. Thus, the blue runner strategy of repeatedly attempting to go in either direction until the red guard stops following to block will allow for guaranteed safe passage without visibility between the two blue team players. As a side note, the core could also be implemented with two separate, unmodified Delay Gates, thus using two red guards instead of one but having no additional timing constraints.

3.2.4 Initialization

In many games we are modeling with TGG, all players on each team start in their team's single spawn room. In order to force the team members into separate hallways, they are coerced into guarding a set of paths, one per player (besides the runner), which all lead to the victory node w. Figure 3.4 shows the initializer gadget with spawn nodes s_b or s_r, where first blue must split into

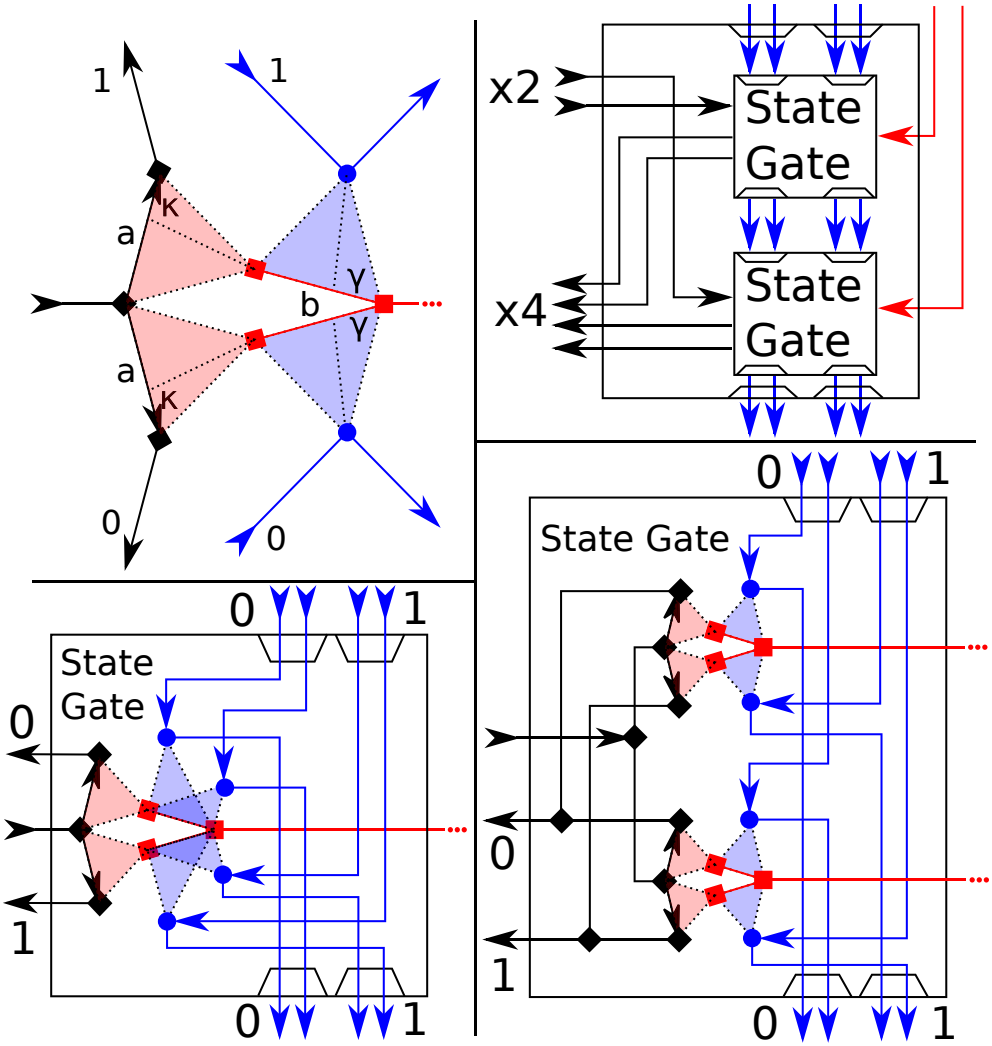


Figure 3.3: "State Gate" gadget schema for a blue executor to branch the blue runner. The core of player interaction (top-left) is generalized first allowing two blue paths per input (two possible constructions on bottom) then allowing multiple runner paths (top-right).

three hallways to block any red players from reaching w and force the red players to make progress and split up in order to block the blue runner from reaching w.

Specifically, to incentivize the blue team to fully split up, two red team "win paths" are placed and each guarded by a series of n_r blue attack zones of length $b_2 > \gamma$, so that even if the red team sends all of its players down one win path, the defending blue player could eliminate all of them by the end. If the blue team tries to send multiple players out the same hallway from s_b, they will either allow red team to win through the other win path in the initializer gadget, or have no player in the blue runner path, which is designed in our undecidability construction to be the only path to w.

If blue team does split up and guard the red team win paths, then red team must then prevent the blue runner from reaching w by going down a third path that splits into n_r branches, each responsible for guarding a different path for the blue runner. This forces the red team to separate and block every path until the blue runner gives up and exits the Initializer Gadget, at which point all other now-separated players can safely exit as well.

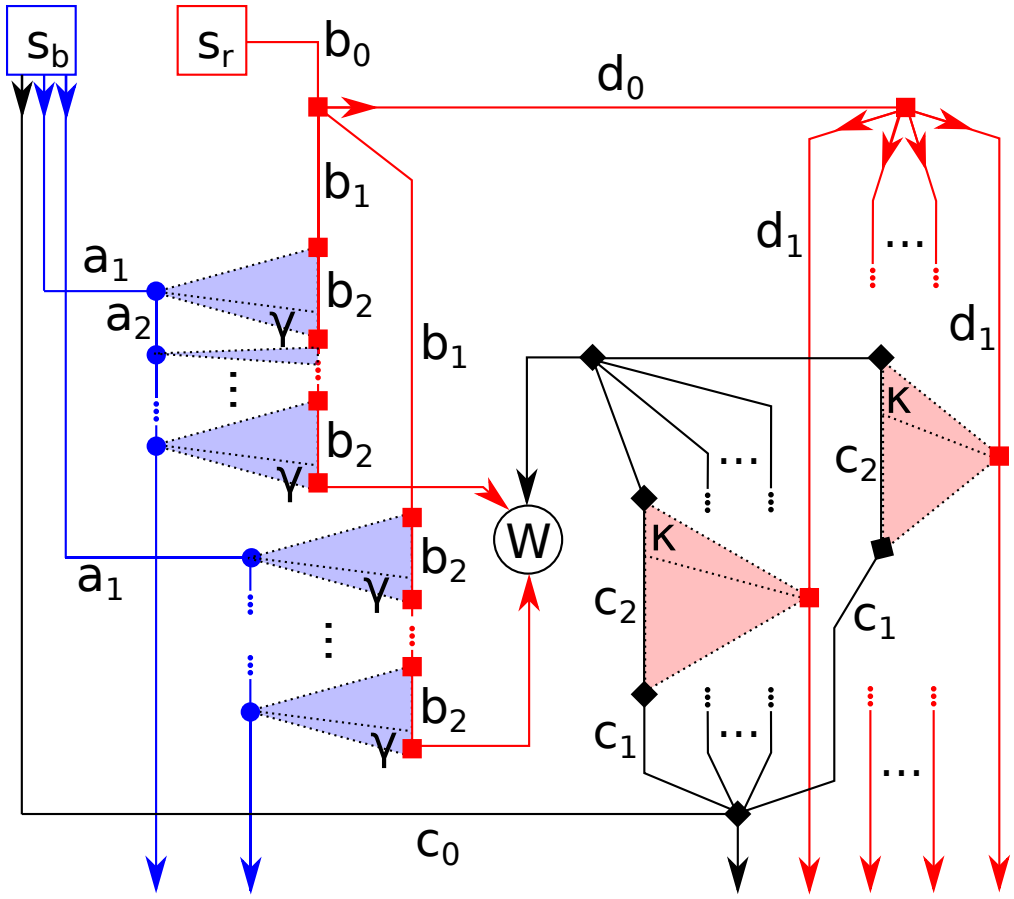


Figure 3.4: Initializer Gadget to separate players that must start together in team spawn rooms.

The constraints on the Initializer Gadget are light beyond the need for visibility so each player can learn when it is safe to stop guarding an attack zone and make progress. No information needs to be private at this point so full visibility is allowed within the gadget, although a set of hallways at the exit for the blue runner to pass within visibility range of every other player would be a sufficient signal for games being modeled by TGG with occlusion or view distance constraints. For the blue players to have time to block the red players, the attack nodes should be close enough together such that $\forall i \in [0, n_r) : a_1 + ia_2 < b_0 + b_1 + (i+1)(b_2 - \gamma)$. So that the red players have time to block the blue runner, it must be that $b_0 + d_0 + d_1 < c_0 + c_1 + c_2 - \kappa$.

3.3 Reductions

The Team Computation Game (TCG), as defined in [DH08], is a game about two teams ($\exists$ and $\forall$) whose players alternate writing symbols onto certain cells of a finite-length tape of a Turing machine, which takes a fixed number of steps during each round and if it halts then the game ends and one team wins based on whether it accepts or rejects. A simplifying insight is that this Turing machine is effectively a DFA that teams are alternatively feeding input symbols into until it ends up in a final state that determines which team wins. The following modified definition will use this terminology instead for the purposes of the later reduction. We also present reductions establishing the equivalence of TDA with TCG and thus its undecidability.

Definition 3.3.1. The Team DFA Game (TDG) is a two-versus-one team game. An instance of the game is a DFA $D = (\Sigma = \{0,1\}, Q, q_0, \delta, F = F_\exists \Delta F_\forall)$. The existential team $\{\exists_1, \exists_2\}$ competes against the universal team $\{\forall\}$. The game starts with D in state q_0 and each round proceeds as follows:

1. If D's state $q \in F_\exists$ then team existential wins. If $q \in F_\forall$ then team universal wins.
2. $\forall$ learns the state q of D then inputs two bits b_1, b_2 into D.
3. $\exists_1$ learns b_1 then inputs one bit m_1 into D. $\forall$ learns m_1.
4. $\exists_2$ learns b_2 then inputs one bit m_2 into D. $\forall$ learns m_2.

Lemma 3.3.1. *TDG is reducible from and to TCG. Namely, $\exists f : \langle D \rangle \to \langle I \rangle$ and $\exists g : \langle I \rangle \to \langle D \rangle$ which map between instances $\langle D \rangle$ of TDG and instances $\langle I \rangle$ of TCG which both preserve the predicate of whether or not the existential team has a forced win.*

Proof. We prove both directions separately.

$\Longleftarrow$ Consider an instance $I = \langle S, O, k, \Gamma \supset O \cup \{\mathsf{A}, \mathsf{B}\}\rangle$ of the TCG.

The TDG on the corresponding DFA D will directly simulate the TCG on I. The state space $Q(D)$ is the configurations of S as well as additional counters for input tracking. The first $\forall$ turn runs S without input from the existential team, thus $q_0(D)$ is the result of immediately applying δ_S k times (or until termination) from its initial configuration. After that, both games check for termination in the same way (accept states of S are win states of existential team, reject for universal), then begin writing to S's tape or feeding bits into D. The only significant difference is that the existential moves O must be input to D in binary over $2\lfloor \log_2 |O| \rfloor$ rounds where the universal player's moves are ignored by D. The transition function δ_D simply writes the appropriate bits of the moves from $\forall, \exists_1, \exists_2$ onto the tape of the current configuration, and once everything is input then it updates the configuration by applying δ_S k times (or until termination).

$\Longrightarrow$ Consider an instance D of the TEAM DFA COMPUTATION GAME.

The TCG on the corresponding instance $I = \langle S, O, k, \Gamma\rangle$ will similarly be a direct simulation of the TDG. Using $k = 6$ and $\Gamma = O = \{0,1\}$, the tape of S is just the cells for each input bit b_1, b_2, m_1, m_2 plus unused space at the end. Its state space simply augments $Q(D)$ with input reading states. The first k steps, S will be in $q_0(D)$ and move nowhere, but each following time S is simulated for k steps, starting at tape position 0, S will read each bit, applying δ_D to update its DFA state for each read (unless it has entered a final state), then just return to position 0.

At the start, TCG runs S for k steps, which does nothing. The termination check for each game is the same, as before, then each player will input their move onto the appropriate cell of the tape (in the same order in both games) then run S again, which will simulate the same inputs being given to D and updating its state. □

Theorem 3.3.2. *The Team DFA Game is undecidable.*

Proof. If TDG were decidable, then TCG would be decidable using f from Theorem 3.3.1 to get a homomorphic instance, but since TCG is undecidable [DH08], TDG cannot be either. □

We now go on to define the Team Graph Game and show it is undecidable by a reduction from the Team DFA Game.

Definition 3.3.2. The Team Graph Game is a team multiplayer game. Let the TGG of red team vs blue team consist of:

- Directed Graph $G = (V, E)$ with edge weights $\in \mathbb{N}$
- Designated team start nodes $s_r, s_b \in V$ and win node $w \in V$
- Directed visibility relation $S \subseteq V^2$
- (Uni)Directed attack relation $A \subseteq V^2$
- Initial number of players per team $n_r, n_b \in \mathbb{N}$

The execution of the Team Graph Game starts with n_r red player tokens at node s_r and n_b blue player tokens at node s_b. Blue team wins if either every red token is eliminated or any blue token reaches the node w. Red team wins similarly.

The game proceeds as a sequence of time steps, or frames. Each frame, all active players simultaneously commit to their action and then all effects are triggered and handled before the frame ends. The action of a player consists of a node $n \in N[v]$ to move towards (or none to signify not moving). Once players have performed their moves, each player whose token can "see" another player's token learns of said token's position and team. Visibility zones are defined at nodes by S and on edges by union of the visibilities of the endpoints; combat zones are defined similarly.

Theorem 3.3.3. *TDG reduces to the Team Graph Game (TGG). Namely, $\exists h : \langle D \rangle \mapsto \langle I \rangle$ which maps instances $\langle D \rangle$ of TDG and instances $\langle I \rangle$ of TGG such that the existential team has a forced win in the TDG on D iff the blue team has a forced win in TGG on I.*

Proof. Figure 3.5 gives an overview of the structure of $I = h(D)$. Once the initializer gadget distributes each blue and red player into their proper hallways, each loop of the blue team in the graph simulates one round of TDG. The universal team's decisions b_1, b_2 are made (cooperatively) by the two decision-making red team members in the red choice gadgets, and the existential team's decisions m_1, m_2 are made (independently of each-other) by the decision-making blue team members directly after exiting the red choice gadgets. The blue runner's location corresponds directly to the state of the DFA, and their teammates open paths inside state gates which allows the runner to implement the DFA transition function δ.

Each state $q \in Q \setminus F_\forall$ of the DFA has an "arena" with two sides: the right side with a series of four state gates of increasing arity and a left side with a series of two Delay Gates. When the blue runner enters the right side of the arena for q before the first state gate, the DFA is in state q. If $q \in F_\exists$ then there will also be a hallway here leading directly to the win node. The four state gates encode the tree of states reachable from q in up to 4 transitions, outputting the runner in one of 16 hallways each corresponding to a state $q' = \mathsf{foldl}(\delta, q, [b_1, b_2, m_1, m_2])$ and leading to the left side of the arena for q'. Once the runner passes through the Delay Gates, they enter the right side of the arena for q'. Lastly, if $q \in F_\forall$, then all hallways entering its arena lead to a dead-end.

As we showed in Section 3.2.4, each team has a course of action which will prevent any players on the other team from reaching the Win node. Further, this puts every player on a path whose only way forward is out of the initializer gadget. At that point there is no incentive to stay in the initializer gadget and we may as well assume they continue into the rest of the map.

$\implies$ Suppose the existential team has a forced win in TDG on D. This means that there are optimal strategy functions $\mathsf{s}_i : ([b_{i,1}, b_{i,2}, ..., b_{i,j-1}], [m_{i,1}, ..., m_{i,j-1}], b_{i,j}) \mapsto m_{i,j}$ which produce

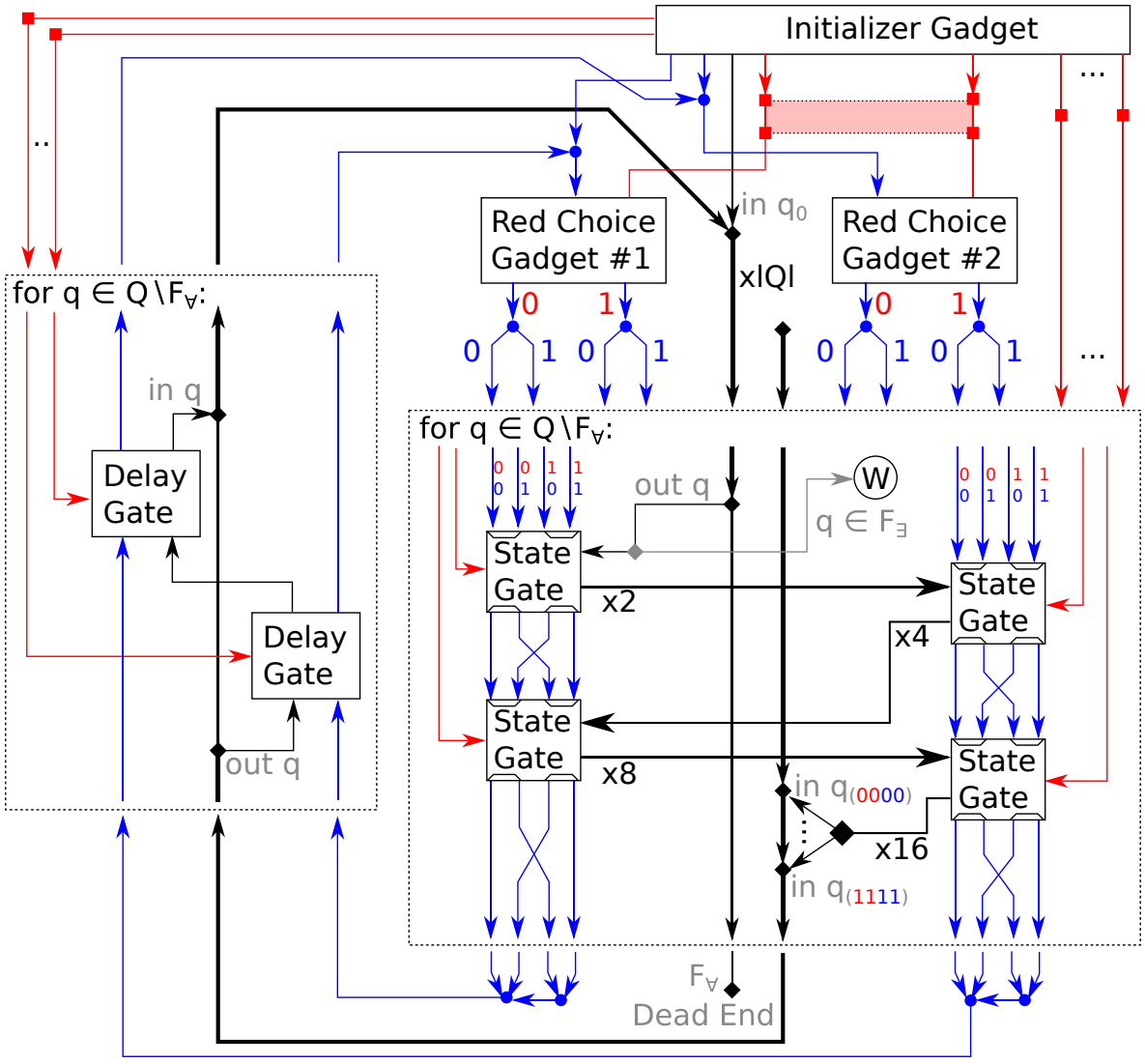


Figure 3.5: A diagram of how the gadgets are put together.

a win-preserving move for $\exists_i$ in round j given $\forall$'s move and what they learned in the past $j-1$ turns.

For decision-making blue player i, on the j^{th} time they pass through red choice gate i, let $b_{i,j} = 0$ if they exit on the A side else let $b_{i,j} = 1$ if they exit on the B side, and let $m_{i,j} = \mathsf{s}_i([b_{i,1}, ..., b_{i,j-1}], [m_{i,1}, ..., m_{i,j-1}], b_{i,j})$. At the upcoming branch, they take path $m_{i,j}$. The blue runner should follow the hallways and wait until combat zones are safe before passing through, and the decision-making blue team members should open combat zones long enough for the runner to pass through safely and to defeat the red team member there if necessary. By the structure of the graph, the path of the runner will lead to a $q \in F_\exists$ no matter what choices red team makes in the red choice gadgets, and every attack zone along the way will be opened up for the blue runner by their teammates, thus blue team has a forced win in TGG on I.

$\Longleftarrow$ Now suppose blue team has a forced win in TGG on I. Since only the blue runner can reach

win node (outside the initializer gadget), any winning execution entails a path through the graph that the runner took which starts by entering the right side of the q_0 arena, passes through n arena right sides and left sides (as described earlier), and ends at the entrance of the right side of an arena for some $q_n \in F_\exists$.

In order for the runner to pass through the combat zones in the gates along the path, the decision-making blue teammates must have dealt with the attacking red team members. Since blue team has a forced win, they still have a forced win even if red team attackers always leave their attack zone before the decision-making blue team member has a chance to defeat them, thus that strategy forces the blue runner at the entrance of the right side of an arena to take a path through the state gates determined by the red and blue teams' choices at the start of the loop.

This implies the existence of functions $\mathsf{s}_i : ([b_{i,1}, b_{i,2}, ..., b_{i,j-1}], [m_{i,1}, ..., m_{i,j-1}], b_{i,j}) \mapsto m_{i,j}$ which produce a win-preserving branch for decision-making blue team member i to take on the loop j after exiting red choice gate i from exit $b_{i,j}$ and what they learned in the past $j-1$ loops. By the structure of the graph, s_i is also an optimal strategy function for $\exists_i$ in TDG on D, thus the existential team has a forced win. □

Corollary 3.3.4. *The Team Graph Game is undecidable.*

Proof. If Team Graph Game were decidable, then TDG would be decidable using h from Theorem 3.3.3 to get a homomorphic instance, but since TDG is undecidable by Theorem 3.3.2, Team Graph Game cannot be either. □

3.4 Applications

We now show how to apply the TEAM GRAPH game to generalized versions of several popular video games. In particular we will show that it is undecidable to determine whether a team can force a win in the following games: Team Fortress 2, Mario Kart, and Super Smash Bros. Brawl. For all of these games we generalize the map size and number of players able to participate in a single game. In addition, we assume that players on the same team have no way of communicating with each other beyond their actions in the game. This means players are not co-located, there is no screen-sharing, and any sort of team or global chat is disabled.

The following are the essential components needed in the game to fit the TGG framework. 1) The game needs a 3D map or crossover gadgets in 2D because the TGG graph used in our reduction is non-planar. 2) One-way Doors. 3) Visibility zones such that we can have two players communicate their location without being able to reach each others path, and ways of blocking visibility so communication can only occur in specific regions. 4) Combat zones which allow the attacker a guaranteed strategy to eliminate or disable the defender and which has no path between the attacker and defender. 5) A win condition that can be activated by one player in a limited location.

3.4.1 Team Fortress 2 and many other team FPS games

Like many others of its kind, Team Fortress 2 is a first person shooter with 3D environments (1), one-way doorways (2), clear unbreakable glass/fences and opaque walls (3) made out of polygons, grenades and sniper rifles (4), and a capture point where one team can win by standing on it (5). These features allow TF2 and others to directly simulate TGG, leading to their undecidability. Note: only the base TF2 game with default loadouts are considered.

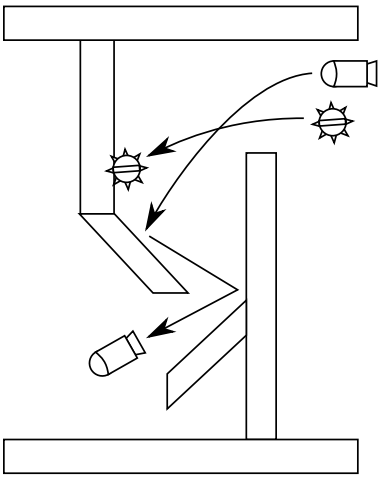


Figure 3.6: Grenade-only Attack Gadget (vertical 2D slice)

The nodes and edges of the graph are generally represented as hallways made of opaque walls connecting at intersections, possibly lengthened or bent-out-of-shape to enforce a required minimum traversal time. Visibility is limited by the first-person view, and visibility zones are constructed by making walls out of glass that gives a line-of-sight between desired locations and possibly additional walls to block view elsewhere.

The combat zones are constructed based on which team the attacker is on. A blue team member attacking a red team member will be faced with a room with a wall that only Demomen grenades can be shot over and succeed at damaging the defender. Figure 3.6 shows how to construct a hole which only physics-enabled grenades can tumble through and sticky bombs and other weapons cannot penetrate. A red team member attacking a blue team member will be faced with a small hole in the wall at Sniper-eye-level which gives a long-distance view of the defender's head such that only a Sniper's sniper rifle can kill the defender before they can pass through the attack zone at optimal speed.

In order to further enforce desired class choices, the red and blue teams are incentivized to choose the Sniper and Demoman classes (respectively) by the map design. The blue team spawn room is separated by a deadly chasm that can only be crossed using the Demoman's unique ability to sticky bomb jump long distances through the air without touching a surface (as a Soldier requires). Health pack pick-ups and distance-based fall damage may be used to force the health of players down so one sniper shot or grenade explosion will defeat any opponent.

By playing in a king-of-the-hill match with unlimited-time and with text and voice chat disabled, this map structure will exactly simulate TGG.

3.4.2 Super Smash Brothers

Super Smash Brothers is a popular Nintendo fighting game series. Out of the series' five releases, the most recent three (Super Smash Bros. Brawl, Super Smash Bros. for 3DS, and Super Smash Bros. for Wii U, henceforth referred to as Brawl, SSB4 3DS, and SSB4 Wii U, respectively) share a number of gameplay elements which we will shortly show result in undecidability.

We consider a generalized Super Smash Bros. game, where an arbitrary number of players on red or blue team control fighters (who are followed by the players' personal, local cameras, as in SSB4 3DS Smash Run mode) which fight on a stage (a bounded 2D plane with gravity, solid polygonal ground, and other obstacles) in Stamina mode (where each player starts with a given number of hit points and dies when they are depleted). Fighters are selected among a set of characters, each with unique traits, and can walk, run, jump off the ground and jump in the air finitely-many times before landing, and fight using aerial and ground attacks (which may create hitboxes which damage and knockback other characters, may move the attacker, and may provide defense), and defensive maneuvers such as shielding (a bubble around character which blocks attacks at the expense of temporary shrinkage), air and ground dodging (temporary invincibility at the cost of short vulnerability before and afterwards) and rolling (a ground dodge with fixed motion left or right). Due to close-quarters, we also consider obtrusive stage background music such that all character sound effects are drowned-out.

Theorem 3.4.1. *In generalized Super Smash Bros. match between two teams of Pikachus on some stage, it is undecidable whether Player 1's team has a forced win.*

Proof. Reducing from TGG constrained to graphs constructed from DFA as in Theorem 3.3.3, we consider only the character Pikachu due to its unique Thunder attack that temporarily spawns a damaging cloud and lightning strike at a fixed position above Pikachu, even if there are obstacles in between. Instead of 3D hallways, our construction of the stage simulating the graph only needs to bound 2D areas with strings of solid blocks (as in Brawl's and SSB4 Wii U's stage builder) that are thin enough in certain areas for Thunder to attack other characters through ceilings. We also use thin floors, which allow for jumping upwards through but do not allow for falling through, to construct one-way doors.

The most striking problem for this 2D fighting game is the need for a crossover gadget. We make use of the barrel cannon stage obstacle, as seen in the Kongo Jungle stage from the first Super Smash Bros. as well as all future titles in some form, which captures a player upon contact and, when activated by the player inside, launches them along a fixed path without the player having aerial control until the end. Notably, we consider the original design of the cannon where a launched player does not hurt others via collision. By using two barrels and two one-way floors, a section of the stage as in Figure 3.7 can allow for crossovers without player interaction, although it does provide visibility. Because the constrained TGG graphs can be embedded in the plane where all edge crossings are either outside of the main loop before the simulation begins, same-player crossings, or between players who are allowed to know where the other's token is located, visibility does not transmit information that is useful for making red or blue team "choices."

As mentioned, attack zones are built around Pikachu's Thunder attack, which unconditionally creates a hitbox at a fixed distance high above the character. For attack zones that guard the traversal of an edge, the idea is to force the defending Pikachu to predictably position itself in a vulnerable state above the attacker, so that the attacking Pikachu can always hit them with Thunder if traversal is attempted. In Delay Gates, such as in Figure 3.8, where the red attacker of the blue runner is under attack themselves, the blue attacker is able to Thunder the only location at which the red attacker can use Thunder to hit the blue runner, so as to open the path safely. The Red Team Choice Gadget can be implemented in Brawl similarly to the Delay Gate, and the State Gates directly out of Delay Gates, thus the given TGG graph is fully representable.

When the blue runner reaches the win node, they can themselves open a path for the other blue Pikachus and all go into a new series of pathways that lead underneath every red team player so they can work together to eliminate them all, as properly-timed Thunders by multiple players can break shields and hit for longer than dodge invincibility. and end the match with a blue victory. This path-opening can be a Delay Gate or even compactly implemented using Brawl's Falling Block object, which is a solid obstacle that temporarily falls and disappears after a player (the blue runner, in this case) stands on it, reappearing at its original position after a short period of time. □

3.4.3 Mario Kart

In an earlier paper, two player, perfect information Mario Kart was shown to be PSPACE-complete [BDH$^+$15]. It also did not consider the commonly enjoyed Battle game type. Here we show that a generalized version of Mario Kart in team Balloon Battle mode is undecidable by a reduction from TGG.

Mario Kart takes place in a 3D environment where each player has a personal third-person camera view of their character; when playing online or on local wireless, players cannot see other players' screens. In Balloon Battle, the players are placed in an enclosed, obstacle-filled Battle Course with a small number of balloons that pop when the player is damaged, eliminating the player if none remain. By searching the course for item boxes (in fixed, reusable spawn locations),

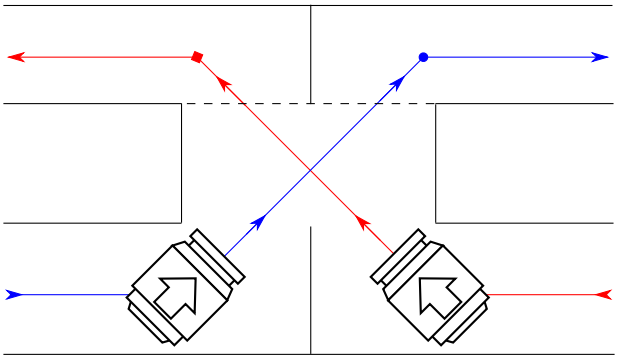


Figure 3.7: Super Smash Bros Crossover Gadget using Barrel Cannons

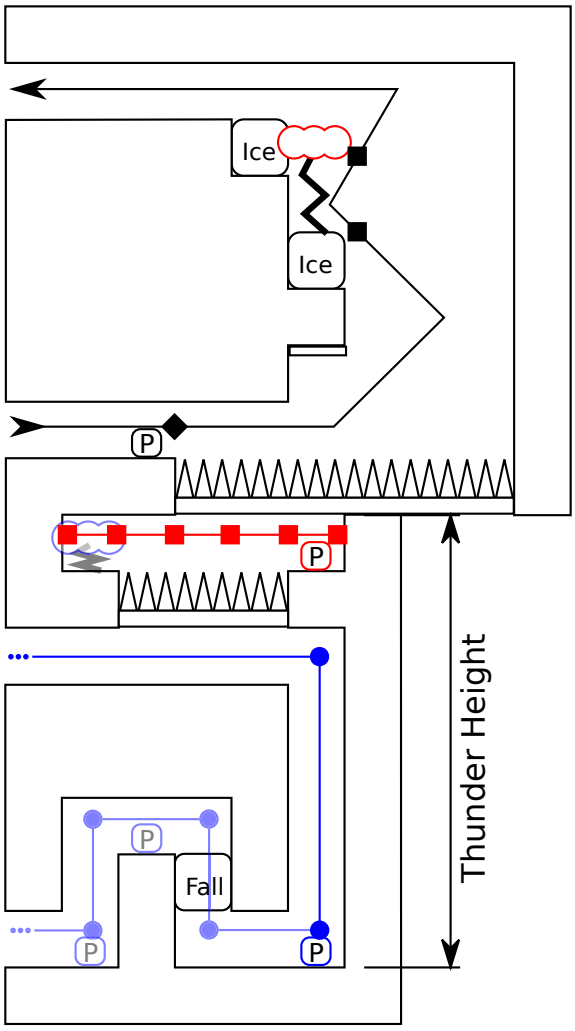


Figure 3.8: Delay Gate constructed using Brawl's Custom Stage Builder parts. A single player's screen is approximately 5 blocks tall, so the blue executor can never see the runner. Each "P" is an example location of a Pikachu, "Ice" is a block with no edge to hang onto, and "Fall" represents a Falling Block. Shaded blue figures are only relevant during the blue victory phase. Example Thunder clouds and associated lightning strikes are also shown.

players can get items from a given distribution to damage other players and avoid attacks against themselves. There is a blue team and a red team, and if one team is completely eliminated, the other team wins.

Theorem 3.4.2. *In generalized Mario Kart Balloon Battle with the Bob-ombs Only item distribution, it is undecidable whether or not the blue team has a forced win.*

Proof. We reduce from TGG constrained to graphs constructed from DFA as in Theorem 3.3.3, which involves building a Battle Course that simulates the graph. Mario Kart courses are polygonal 3D environments with a finite maximum movement speed, one-way jumps, clear glass, and opaque walls, so the primary complexity is describing the attack zones and how to win.

A player using a Bob-omb item causes a Bob-omb to be thrown from the character's kart in an arc. It can bounce off walls and will explode into a large, temporary, damaging sphere on contact with another player or after a short time interval. One common obstacle in Mario Kart is the Thwomp, which are large spike-covered boxes which can move along fixed paths.

To construct an attack zone where the attacker is preventing the defending character from traversing an edge, said edge is a short, thin hallway with exits guarded by Twomps that alternate moving up and down between the ceiling and ground such that at least one is always on the ground blocking the path and the space between is smaller than the diameter of a Bob-omb explosion. The attacking character is spawned in a raised hallway with an item box and an uncrossable pit such that a Bob-omb can be thrown by the attacker and create an explosion to eliminate any player between the Thwomps but no Bob-omb can be thrown back high enough to reach the attacker. In an attack zone where the defender is itself an attacker in a dead-end hallway, there need only be one Thwomp guarding the single exit and trapping the defender for a period of time such that the attacker in an even-more-raised hallway could safely throw down a Bob-omb to eliminate them. Figure 3.9 gives an overview of this construction.

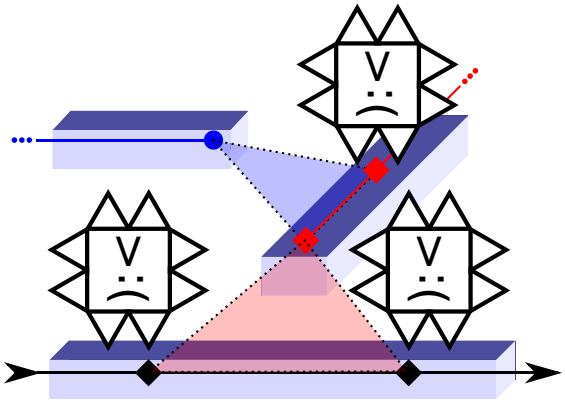


Figure 3.9: The Mario Kart Delay Gate's 3D Layout with Thwomps (opaque walls not shown).

When the Mario Kart character simulating blue runner is supposed to reach the win node, they are first able to open a path for their blue teammates (normally blocked by a red attack zone) to join them into a set of hallways above the rest of the course which lead to attack zones spanning each red team character's small region of the graph. With plentiful item boxes, the blue team characters can thus trap and eliminate each red team member using coordinated Bob-omb threats and throws, winning them the game. □

3.5 Conclusion and Open Problems

Our Team Graph Game framework has proven useful in showing the undecidability of more natural team multi-player games, as shown in our application to various video games. We currently wonder how far this framework can go. Can we capture other popular genera of video-games with teams such as MMORPGs like World of Warcraft and Guild Wars, real time strategy games like Starcraft or Age of Empires, MoBAs like DotA and Heroes of the Storm, or others? Each of these has their own challenges in adapting to our framework, but given our success with Super Smash Brothers which was a 2D game that lacked vision blockers and a location based victory condition, we believe a lot can be done with a little work. We also pose the question of whether this framework can be used to understand the complexity of any real world multi-agent coordination scenarios.

There are also a number of interesting questions about imperfect information team multi-player games, many of which would be very useful in allowing broader application of this framework. For example, is the Team Graph Game still undecidable when there are only a constant number of players on each team? The Team DFA Game needs only three; however, we find it useful to assign different players to many of our gadgets, leading to a linear scaling.

In the following Chapter 4, we will tackle another question this work raises. In Section 3.4, we had to impose unrealistic isolation between players on the same team to prevent private information sharing. Was this necessary? Are these team games, both the abstract games and video games like those we studied in this chapter, still undecidable if we allow a limited amount of communication between players on the same team?

Chapter 4

Decidability of Team Games with Communication

This chapter presents results from the paper titled "Characterizing the Decidability of Finite State Automata Team Games with Communication" that the thesis author coauthored with Jayson Lynch. This paper appeared in Games, Automata, Logics, and Formal Verification (GandALF), 2022 [CL22]. This chapter also includes extended results under submission at the time of the writing of this thesis.

Overview

In this chapter, we define a new model of limited communication for multiplayer team games of imperfect information. We prove that the Team DFA Game and Team Formula Game, which have bounded state, remain undecidable when players have a rate of communication which is less than the rate at which they make moves in the game. We also show that meeting this communication threshold causes these games to be decidable.

4.1 Introduction

Deciding optimal play in multiplayer games of incomplete information is known to be an undecidable problem [PR79, PRA01]. This includes games where the state space is bounded, a surprising result first shown of a collection of abstract computation games [DH08] that we extended in Chapter 3 to generalized versions of real games, like Team Fortress 2 and Super Smash Bros. However, these works have relied on the complete inability of teammates to communicate during the game, which is often not a realistic assumption. In this chapter, we study deterministic models of communication between players in two of these computation games, the Team DFA Game and the Team Formula Game, and show a sharp change in computational complexity based on whether players are able to eventually communicate all of their moves or only able to communicate a constant fraction.

One motivation for this model is a better understanding of real world games. Many team games played in-person naturally permit free form communication between teammates to coordinate their actions, and online multiplayer video games often provide communication channels such as voice-chat, text, and emotes to simulate this in-person environment. These include many FPS games such as Team Fortress and Left4Dead, MOBAs such as DOTA2 and League of Legends, and RTS games such as Starcraft and Age of Empires. Some of these examples have drawn research interest in AI/ML [VBC+19, BBC+19] as well as computational complexity [Vig14]. The real-time nature

of these games ensures that communication channels are bounded; however, modeling free form communication, as well as efficiently implementing meaningful player choices in a real-time setting, makes it difficult to analyze these games with these communication features enabled.

Outside of the team setting, communication is a central aspect of many other games. For example, in the cooperative card game Hanabi players are unable to see their own hands of cards, but this information is visible to everyone else. In addition, players are not allowed to communicate except through actions in the game, one of which allows players to reveal partial information about what is in another players hand. A perfect information version of Hanabi was shown to be NP-complete [BCD$^+$17]. The Crew: Quest for Planet Nine is a cooperative trick taking card game which uses limited communication between players as a core mechanic. The complexity of this game was also studied in the perfect information setting [Rei21]. Under the limited information setting, containment in NP for both of these games is not obvious, and we see a need for models of player communication in games. Other examples of cooperative boardgames with limited communication channels between players include Mysterium, The Mind, and Magic Maze.

Other games may limit communication simply with time pressure in the game. Examples of fully cooperative games with imperfect information that use time pressure to limit coordination and communication include Space Alert, 5-Minute Dungeon, Keep Talking and Nobody Explodes, and Spaceteam.

Multi-agent, imperfect information games are also a topic of interest in Reinforcement Learning. In [CCBG19] algorithms are developed to address team extensive form games of imperfect information where communication is allowed at certain points during gameplay, with Bridge and collusion among some players between hands in poker being the motivating examples. Other work considers Sequential Social Dilemmas, a type of iterated economic game where in any given instant a player is incentivized towards non-cooperative behavior, but cooperative strategies can obtain higher payoff over the game as a whole. Learning algorithms for these models both with and without explicit bounded communication channels were studied in [JLH$^+$19]. Purely cooperative settings have also been of interest [FAdFW16].

One major achievement was human level performance on a limited version of DOTA2, a MOBA-style video game [BBC$^+$19]. These are real-time, team games with partially observable state. Although both text and voice chat are typically allowed in professional play, the AI system did not utilize these explicit communication mechanisms. The board game Diplomacy, while not explicitly a team game, features coordination and temporary alliances between players as a core game dynamic. This game has also seen interest as a new challenge in the AI community, but focusing on No-Press Diplomacy which does not allow explicit communication between players [PLB$^+$19, BWLB21].

These examples of both AI and computational complexity research which considers games with cooperation and communication motivate, but frequently ignore the important role of communication in these games, motivates our work in this chapter.

Chapter Organization. In Section 4.1.2 we formally define our model of communication for the Team DFA Game. In Section 4.2 we prove undecidability for a few simple communication patterns to help build intuition for the techniques used in the next section. In Section 4.3 we prove our main undecidability result for Team DFA Games with Communication. In Section 4.4 we prove the game becomes decidable when either both players can communicate all information about their moves, or one player receives no information but can communicate all of their moves to their teammate. In Section 4.5 we show that analogous algorithms and undecidability results hold for Team Formula Games.

4.1.1 Team DFA Game

As in Chapter 3, the problem we consider in this chapter is: given an instance of the game, does a particular team have a ***forced win***? More formally, does there exist a strategy function s_i for each player i on the team, specifying on each turn which move to make based on any information they have learned so far, that when followed will guarantee that this team will win? We define the complexity of a game as the complexity of whether a specified team has a forced win in the game.

We will continue studying the Team DFA Game defined in Chapter 3 and proven to be undecidable in Theorem 3.3.2 in the case without communication. A number of variations of this game, all undecidable in the general case, exist. These include Team Computation Game where players give inputs to a Turing machine on a bounded tape [PR79], Team Constraint Logic Game where players make moves in a partially observable constraint logic graph [DH08]; and Team Formula Game where players flip the values of Boolean variables trying to satisfy different formulas [PRA01, DH08].

4.1.2 Communication Model

We model communication in the Team DFA Game with a policy that specifies the bandwidth of a dynamic information channel, as one might have due to natural factors (e.g. playing a real-time game with voice chat) or intentional game design (e.g. Hanabi's card-revealing moves) allowing a predictable but bounded amount of player-to-player communication between moves. Specifically, a policy P is a DFA over a unary alphabet with functions $P_{\text{MID}}, P_{\text{END}}$ over its states. In a round of the game in policy state p, $P_{\text{MID}}(p)$ is the number of bits which are exchanged simultaneously between $\exists_0$ and $\exists_1$ after (b_0, b_1) are revealed but before (m_0, m_1) must be determined, and similarly $P_{\text{END}}(p)$ is the number of bits to exchange after (m_0, m_1) are submitted but before the next round starts.

Definition 4.1.1. The Team DFA Game with Communication (TDGC) is a game of the existential team $\{\exists_0, \exists_1\}$ versus the universal team $\{\forall\}$, extending the Team DFA Game. An instance of the game is a pair of a game DFA $D = (\{0,1\}, Q, q_0, \delta, F_\exists \cup F_\forall)$ and a policy P, which consists of a policy DFA $(\{1\}, \Pi, p_0, \pi, \emptyset)$ and functions $P_{\text{MID}}, P_{\text{END}} : \Pi \to \mathbb{N} \times \mathbb{N}$. The game starts with D in state q_0 and the policy DFA in state p_0, and each round proceeds with added communication steps as illustrated in Figure 4.1.

There are two beneficial aspects of studying policies as DFAs on unary alphabets: bounding the state space allows for the policy to be computable by the mechanics of a bounded-state game (such as the DFA in the Team DFA Game), and giving every state exactly one next state (for the

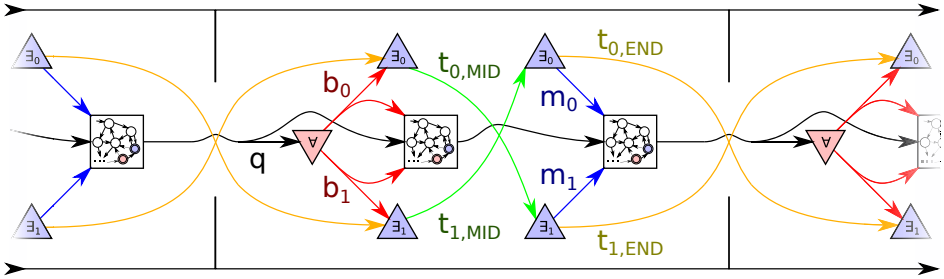


Figure 4.1: Information flow graph of one round of the Team DFA Game with Communication, including from the previous round and into the next round. New to this game are the mid-round transmissions, $t_{0,\text{MID}}$ and $t_{1,\text{MID}}$, and the end-of-round transmissions, $t_{0,\text{END}}$ and $t_{1,\text{END}}$, which have sizes determined by P_{MID} and P_{END} applied to the policy state.

Algorithm 1 The execution of one round of TDGC, given D is in state q and P is in state p.

function TEAM-DFA-GAME-COMMUNICATION-ROUND(q, p)
    If $q \in F_\exists$, then the existential team wins.
    If $q \in F_\forall$, then the universal team wins.
    $\forall$ learns q, then inputs two bits b_0, b_1 into D. ▷ $q \leftarrow \delta(\delta(q, b_0), b_1)$
    $\exists_0$ learns b_0, and $\exists_1$ learns b_1.
    EXCHANGE($P_{\text{MID}}(p)$)
    $\exists_0$ inputs one bit m_0 into D. ▷ $q \leftarrow \delta(q, m_0)$
    $\exists_1$ inputs one bit m_1 into D. ▷ $q \leftarrow \delta(q, m_1)$
    $\forall$ learns m_0 and m_1.
    EXCHANGE($P_{\text{END}}(p)$)
    Advance the policy state. ▷ $p \leftarrow \pi(p, 1)$

function EXCHANGE((n_{01}, n_{10}))
    $\exists_0$ privately defines message t_0, consisting of n_{01} bits.
    $\exists_1$ privately defines message t_1, consisting of n_{10} bits.
    $\exists_0$ and $\forall$ learn t_1.
    $\exists_1$ and $\forall$ learn t_0.

next round of the game) means the bandwidth every round will be known in advance when building our constructions, rather than being dependent on player actions. As a result of this choice, it is important to note that the shape of its state transition graph will always have the form: from the start state, there is an initial chain of unique states (possibly of length zero) that leads to a cycle of periodically-repeating states. Also shown in Figure 4.2.

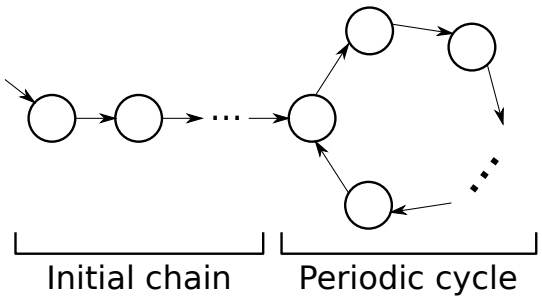


Figure 4.2: General form of a policy DFA: an initial chain followed by a cycle.

4.2 Undecidability of Simple Communication Games

In this section, we will explore some basic classes of policies that preserve the undecidability of the Team DFA Game with Communication. Our proof technique is to reduce from the zero-communication Team DFA Game, where we compensate for the message passing by "clogging the channel" with the forced transmission of garbage bits that do not facilitate information sharing. Section 4.3 builds upon these examples to obtain more general results, however proving the special cases in this section allows us to introduce ideas needed in the full proof and discuss some of the techniques more concretely.

For each class of policies $\mathcal{P}$ below, we will show that given any policy $P \in \mathcal{P}$ and DFA D for playing TDG, we can produce a DFA D' for playing TDGC under P such that the $\exists$ team has a

forced win on D with no channel iff they have a forced win on D' given a channel following policy P. As TDG is undecidable, so will be TDGC under any policy $P \in \mathcal{P}$. For simplicity, we consider policies with DFA C_r, a length-r cycle of states $\Pi = \{0, 1, \ldots, r-1\}$ with no initial chain, for arbitrary values of r.

4.2.1 Mid-round Communication

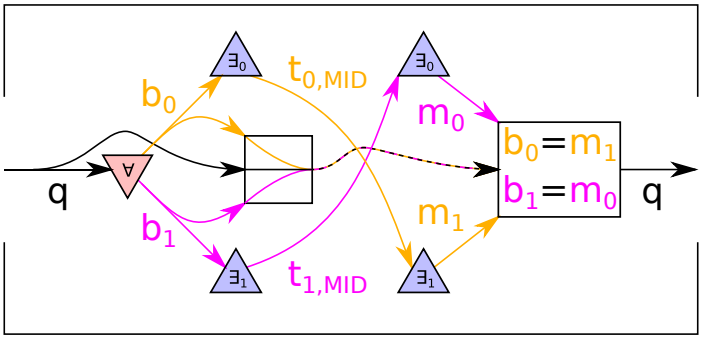


Figure 4.3: Mid-round 1-bit channel clogging technique. Values with the same color must be equal, namely $t_i = b_i = m_{1-i}$, or else the DFA permanently enters $F_\forall$.

Theorem 4.2.1. *TDGC is undecidable with a 1 bit mid-round exchange every $r \geq 2$ rounds: policies P where $P_{\text{MID}}(p) = (1,1)$ if $p \equiv 0 \mod r$, $P_{\text{MID}}(p) = (0,0)$ otherwise, and $P_{\text{END}}(p) = (0,0)$.*

Proof. We construct a DFA D' by first augmenting the state q of D with the state p of C_r. When $p \not\equiv 0 \mod r$, D' simply simulates D for one round. However, when $p \equiv 0 \mod r$, D' instead takes the inputs (b_0, b_1, m_0, m_1) and tests $b_0 = m_1 \wedge b_1 = m_0$. If the test fails, then D' enters a final state for $\forall$.

How D' clogs the channel is diagrammed in Figure 4.3. By tracking the round number, it knows exactly when $\exists_0$ and $\exists_1$ will exchange bits, and in that round D' expects $\exists_0$ to guess b_1, a bit that $\exists_0$ does not learn by the game procedure, and vice-versa. $\exists_0$ and $\exists_1$ are forced to spend their single bit of communication on exchanging b_0 and b_1 to their teammate, in order to guarantee survival against any $\forall$ strategy for choosing b_0 and b_1.

Since $\exists_0$ and $\exists_1$ do not learn anything from each other or alter the simulated D's state in the rounds with communication, they have a winning strategy on D' playing TDGC under P if and only if they have a strategy for the non-exchanging rounds (which happen infinitely-often since $r \geq 2$) that would give a winning strategy on D playing TDG. □

Theorem 4.2.2. *TDGC is undecidable with n rounds of 1-bit mid-round exchanges across $r > n$ rounds: policies P where $P_{\text{MID}}(p) \in \{(0,0),(1,1)\}$ with pre-image size $|P_{\text{MID}}^{-1}((1,1))| = n$ and $P_{\text{END}}(p) = (0,0)$.*

Proof. We generalize Theorem 4.2.1 by constructing a DFA D' which clogs the channel on any round $p \pmod r$ in which $P_{\text{MID}}(p) = (1,1)$ and simulates D in the other $r - n > 0$ out of r rounds. By the same argument, this prevents communication between $\exists_0$ and $\exists_1$ while playing TDGC beyond the corresponding play of TDG taking place during non-exchanging rounds, and thus preserves forced win-ability. □

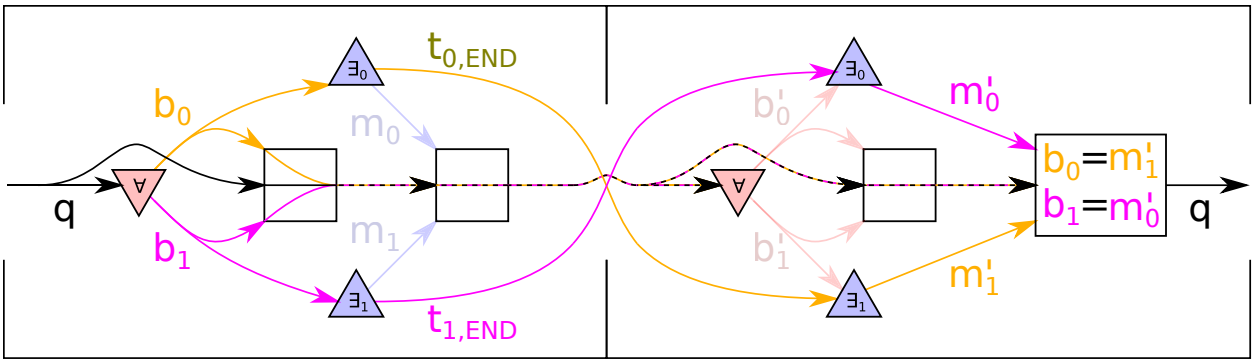


Figure 4.4: End-round channel clogging technique when $r \geq 3$, showing two rounds. The faded-out edges represent messages (m_0, m_1, b'_0, b'_1), which are not used. D' simulates D on other rounds.

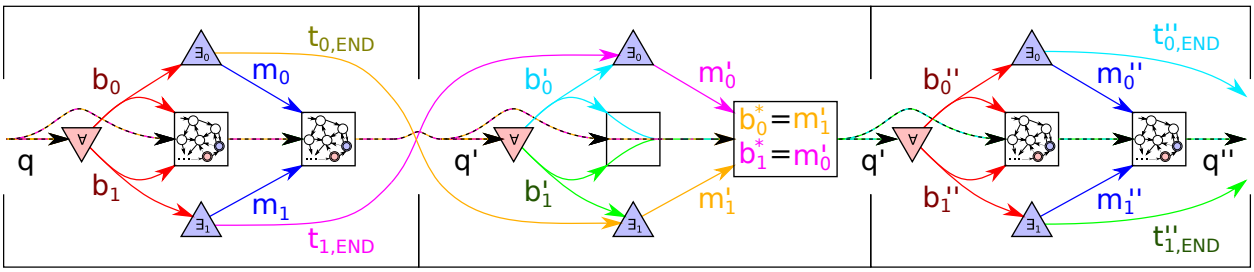


Figure 4.5: End-round channel clogging technique when $r = 2$, showing three rounds. Bits (b'_0, b'_1) created in odd rounds get checked two rounds later, labeled as (b^*_0, b^*_1). D' simulates D in even rounds before $\exists$ players get a chance to exchange those bits.

4.2.2 End-round Communication

Theorem 4.2.3. *TDGC is undecidable with a 1 bit end-round exchange every $r \geq 3$ rounds: policies P where $P_{\text{MID}}(p) = (0,0)$ and $P_{\text{END}}(p) = \begin{cases} (1,1), & \text{if } p \equiv 0 \mod r \\ (0,0) & \text{otherwise} \end{cases}$.*

Proof. We construct a DFA D' like before, by augmenting D with C_r's state as well as two bits of storage, initialized to $(0,0)$. When $p \equiv 0 \mod r$, D' stores the inputs (b_0, b_1) from $\forall$ and ignores $\exists_0$ and $\exists_1$ in that round. In the next round, when $p \equiv 1 \mod r$, it will ignore $\forall$ in that round and use those stored bits to test $b_0 = m_1 \wedge b_1 = m_0$. If the test fails, then D' enters a final state for $\forall$. In all other rounds $\{2, \ldots, r-1\} \mod r$, D' simply simulates D.

The exchange occurring between round 0 and 1 (mod r) requires this new clogging method, shown in Figure 4.4. Compared to the constructions in Theorems 4.2.1 and 4.2.2, we separate the bit choices of $\forall$ and the bit guesses of $\exists_0$ and $\exists_1$ into two rounds to leave time for communication of the bits, without using up all messages that are otherwise used to simulate D (since $r \geq 3$).

By similar arguments to previous theorems, D' clogs all significant channels thus this reduction preserves forced win-ability. □

Theorem 4.2.4. *TDGC is undecidable with a 1 bit end-round exchange every $r = 2$ rounds: policies P where $P_{\text{MID}}(p) = (0,0)$ and $P_{\text{END}}(p) = \begin{cases} (1,1), & \text{if } p \equiv 0 \mod 2 \\ (0,0) & \text{if } p \equiv 1 \mod 2 \end{cases}$.*

Proof. The construction in Theorem 4.2.3 for $r \geq 3$ fails when $r = 2$ because it requires two rounds

without D' simulating D. To address this, we give the modified construction shown in Figure 4.5. We augment D's state with two pairs of bits, to remember the two most recent transmissions. In an odd round, D' stores (b_0, b_1) from $\forall$ as the most recent pair, and then expects the $\exists$ team to submit bits equal to the least recent pair. In an even round, D' simulates D. To deal with the first two rounds, where there is no previous transmission to validate, we initializing all stored bits to 0 so the $\exists$ players can just submit 0 to pass the check.

Although this technique does have in-flight clogging bits during the simulation rounds, every transmission occurs directly before a validation check for those bits, which guarantees that the $\exists$ players must exchange the clogging bits rather than attempt to communicate other information to gain an advantage in the TDG on D. Therefore, as before, this reduction implies undecidability for $r = 2$ as well. □

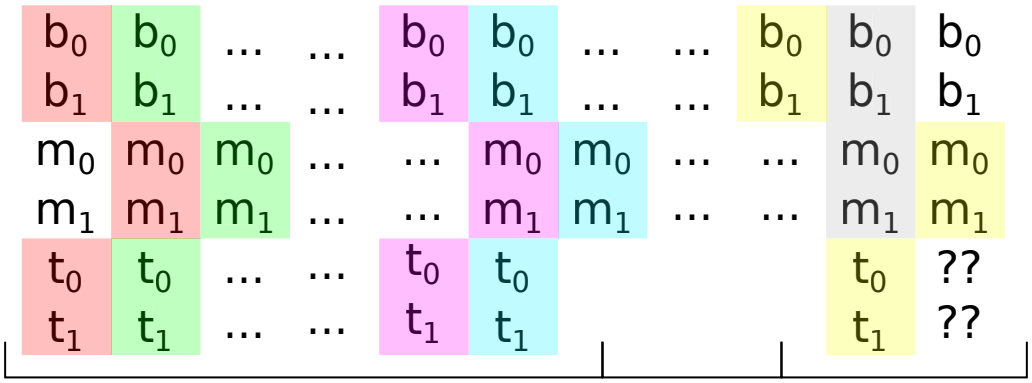


Figure 4.6: Pattern for the end-round channel clogging gadget. Matching colors denote the flow of clogging bits. The first round's (m_0, m_1) and last round's (b_0, b_1) are unused, and the last round may or may not include an end-exchange.

Theorem 4.2.5. *TDGC is undecidable with n bits worth of only end-round exchanges across $r > n$ rounds: where $P_{\text{MID}}(p) = (0, 0)$, $P_{\text{END}}(p) \in \{(0, 0), (1, 1)\}$, and pre-image size $|P_{\text{END}}^{-1}((1, 1))| = n$.*

Proof. Again reducing from TDG, since $r > n$, we have at least one round per period with no end-exchange, and assuming $n \geq 1$ at least one round with an end-exchange (otherwise this is equivalent to TDG). We combine the techniques from Theorems 4.2.3 and 4.2.4 to construct a D' which handles each stretch of contiguous rounds with end-round exchanges and those without.

Our gadget which does this is described by the pattern in Figure 4.6. Leading up to a round with no end-exchange, we repeat the technique from Figure 4.4 (possibly zero times), using the previously "unused" bits to overlap. In the last round in the stretch of no-end-exchange rounds (which may be the only such round), the pattern finishes with the technique from Figure 4.5, guaranteeing one round is available to simulate D per period. Following this, another instance of the gadget begins, repeating ad infinitum.

The clogging guarantees from Theorems 4.2.3 and 4.2.4 still hold of this combined gadget, since the validation of each clogging bit still occurs directly after it can be first exchanged. Inductively assuming the previous checks successfully clogged earlier transmissions, the $\exists$ players cannot communicate any strategically beneficial information to each other, so whether or not they have a winning strategy is undecidable. □

4.3 Undecidability of General Communication Games

This section proves our main result: that a broad class of policies with sufficiently low communication rate remain undecidable for the Team DFA game. We now define this more general notion.

Definition 4.3.1. A policy is (r, x_0, x_1, N)**-rate-limited** if, after a fixed number of rounds N, the rate of transmission from player $\exists_i$ to $\exists_{1-i}$ is x_i during every period of r rounds. Specifically, it must satisfy $x_i = \sum_{k=k_0}^{k_0+r-1} P_{\text{MID}}(p_k)[i] + P_{\text{END}}(p_k)[i]$ for $k_0 = N + \ell r$, where $\ell \in \mathbb{N}$ and p_k is the policy state on round k.

This now allows us to state the main theorem.

Theorem 4.3.1. *TDGC is undecidable under all (r, x_0, x_1, N)-rate-limited policies where $x_0 < r$ and $x_1 < r$.*

4.3.1 Properties of Rate-Limited Policies

Before proceeding to the proof of Theorem 4.3.1, we will establish useful lemmas about rate-limited policies. First, we have the following two simple observations:

Lemma 4.3.2. *Any policy implemented as a unary-alphabet DFA with $n > 1$ states is (r, x_0, x_1, N)-rate-limited for some $1 < r \leq n$ and some initial segment of length $N \leq n$.*

Proof. Refer back to the diagram in Figure 4.2. Consider the unary-alphabet DFA as a directed graph on states, each with exactly one outgoing edge representing the next transition. Any path starting at the start state p_0 cannot have length at least n without repeating some state p_N, so all sufficiently-long paths must first traverse an initial segment $(p_0, \ldots, p_{N-1})$ of $N \leq n$ states then go around the simple cycle $(p_N, \ldots, p_{N+r}, p_N)$ of $r \leq n$ states forever after. If the simple cycle is a self-loop, then instead use the non-simple cycle composed of going around the self-loop twice to have $r > 1$.

Going around this cycle of r states once will permit transmission of a fixed number of bits $x_i = \sum_{k=N}^{N+r} P_{\text{MID}}(p_k)[i] + P_{\text{END}}(p_k)[i]$ from player $\exists_i$ to $\exists_{1-i}$, thus defining a period that makes the policy (r, x_0, x_1, N)-rate-limited. □

Lemma 4.3.3. *Any (r, x_0, x_1, N)-rate-limited policy is also $(2r, 2x_0, 2x_1, N)$-rate-limited if $r > 1$.*

Proof. Given a cycle of length r starting after N steps, repeating the cycle twice results in length $2r$ periods with $2x_0$ and $2x_1$ bits of transmissions also starting after N steps. □

Next, we will need the following property bounding the partial sums of certain repeated finite sequences for analyzing the transmission rates in each part of a round across a period.

Definition 4.3.2. Let a be any sequence of $2n$ natural numbers $(a_0, a_1, \ldots, a_{2n-1})$ with sum at most $n - 1$, and let i be an index into a. ROTATE-BOUNDED(a, i) is the predicate that holds when the infinite sequence $b_j^{(i)} = a_{(i+j \mod 2n)}$ with partial sums $B_j^{(i)} = \sum_{k=0}^{j-1} b_k^{(i)}$ satisfies $\forall j > 0.\ B_j^{(i)} < \frac{j}{2}$.

Lemma 4.3.4. *For any such a,* ROTATE-BOUNDED*(a, i) holds for some even index i.*

Proof. Consider the procedure in Algorithm 2. If the procedure returns i, then $b^{(i)}$ satisfies the given condition not just for $j \in [1, 2n]$ but for all $j > 0$ because it is an infinitely-repeating sequence: if $B_j^{(i)} < \frac{j}{2}$ then $B_{j+2n}^{(i)} = B_j^{(i)} + B_{2n}^{(i)} < \frac{j}{2} + n = \frac{j+2n}{2}$, where $B_{2n}^{(i)} = \sum_{k=0}^{2n-1} a_k < n$ is the sum of the original sequence.

Algorithm 2 Procedure to find a satisfying $b^{(i)}$ for a sequence a (Lemma 4.3.4)

```
Initialize i ← 0
loop
    Find the minimum j ∈ [1, 2n] where B_j^(i) ≥ j/2.
    if no such j exists then return i.
    i ← the even number among {i + j, i + j + 1}.
```

To show the procedure halts, observe that there are only n even-indexed a_i where $b^{(i)}$ can start, thus an infinite loop must get stuck in a "cycle" $(i_0, i_1, \ldots, i_m)$ with $i_0 \equiv i_m \mod 2n$, finding non-zero minimum lengths $(j_0, j_1, \ldots, j_{m-1})$, for some $m \in [1, n]$. Let $w \in [1, n]$ be the number of times the cycle wraps back around modulo $2n$, meaning $i_m - i_0 = 2nw$.

Suppose there were an infinite loop that finds m too-large sums $B_{j_\ell}^{(i_\ell)} \geq \frac{j_\ell}{2}$ of disjoint intervals $\left(b_0^{(i_\ell)}, \ldots, b_{j_\ell-1}^{(i_\ell)}\right) = \left(b_{i_\ell}^{(0)}, \ldots, b_{i_\ell+j_\ell-1}^{(0)}\right)$ of the sequence $b^{(0)}$. Since $i_{\ell+1} = i_\ell + j_\ell + p$, where $p \in \{0, 1\}$ makes $i_{\ell+1}$ even, we have $B_{j_\ell}^{(i_\ell)} \geq \frac{j_\ell}{2} = \frac{i_{\ell+1} - i_\ell - p}{2} \geq \frac{i_{\ell+1} - i_\ell}{2} - \frac{1}{2}$ thus $B_{j_\ell}^{(i_\ell)} \geq \frac{i_{\ell+1} - i_\ell}{2}$ since $\frac{1}{2} < 1$ is the only non-integer term. This means the total sum of all intervals in the cycle is $\sum_{\ell=0}^{m-1} B_{j_\ell}^{(i_\ell)} \geq \sum_{\ell=0}^{m-1} \frac{i_{\ell+1} - i_\ell}{2} = \frac{i_m - i_0}{2} = nw$. However, if we sum all $2nw$ natural numbers $b_{i_0}^{(0)} + \ldots + b_{i_m-1}^{(0)}$, not just those covered by these intervals, then we cannot exceed $(n-1)w$ because the sum of the original sequence is at most $n - 1$. With this contradiction, we conclude that the procedure must halt at some i that satisfies the lemma. □

We also developed another proof of Lemma 4.3.4, which is presented below to give another perspective on this critical component of our main theorem.

Proof. Let $C_j^{(i)} = B_j^{(i)} - \frac{j}{2}$. Let us find an even index i such that $C_j^{(i)} < 0$ for all $j > 0$, and consider the largest length $j^* < 2n$ which maximizes $C_{j^*}^{(0)}$. If $C_{j^*}^{(0)} < 0$ then $i = 0$ satisfies the claim, so suppose $C_{j^*}^{(0)} \geq 0$.

Because the sum of a is at most $n - 1$, notice that for any j, $j + 2n$ in the next period satisfies $C_{j+2n}^{(0)} = B_{j+2n}^{(i)} - \frac{j+2n}{2} < \left(B_j^{(i)} + n\right) - \left(\frac{j}{2} + n\right) = C_j^{(0)}$. Since j^* is the largest maximizer in the first period, for all $j > j^*$:

$$0 > C_j^{(0)} - C_{j^*}^{(0)} = \left(B_j^{(0)} - \frac{j}{2}\right) - \left(B_{j^*}^{(0)} - \frac{j^*}{2}\right) = B_{j-j^*}^{(j^*)} - \frac{j - j^*}{2} = C_{j-j^*}^{(j^*)}$$

and therefore $\forall j > 0$. $C_j^{(j^*)} < 0$, thus $i = j^*$ is an index for which a is rotate bounded. If j^* is even we are done. If j^* is odd, then we know that $j^* < 2n - 2$ because we supposed that $C_{j^*}^{(0)} \geq 0$ whereas $C_{2n-1}^{(0)} < 0$:

$$C_{2n-1}^{(0)} = B_{2n-1}^{(0)} - \frac{2n-1}{2} \leq (n - 1 - a_{2n-1}) - \left(n - \frac{1}{2}\right) = -a_{2n-1} - \frac{1}{2} < 0$$

Thus, consider the even j^*+1 and let $j' \in [j^*+1, 2n)$ be the maximum length such that $C^{(0)}_{j'} = C^{(0)}_{j^*+1}$. $0 > C^{(0)}_{j^*+1} - C^{(0)}_{j^*} = C^{(j^*)}_1 = b^{(j^*)}_0 - \frac{1}{2}$ therefore $b^{(j^*)}_0 = 0$ and $C^{(0)}_{j'} = C^{(0)}_{j^*+1} = C^{(0)}_{j^*} - \frac{1}{2}$. Since $B^{(0)}_{j^*}$ is an integer and j^* is odd, j' must also be even, and by similar arguments $\forall j > j'$. $C^{(0)}_{j'} > C^{(0)}_j$, thus $\forall j > 0$. $C^{(j')}_j < 0$, so $i = j'$ satisfies the claim.

Plotted examples of the even and odd cases can be seen in Figure 4.7. □

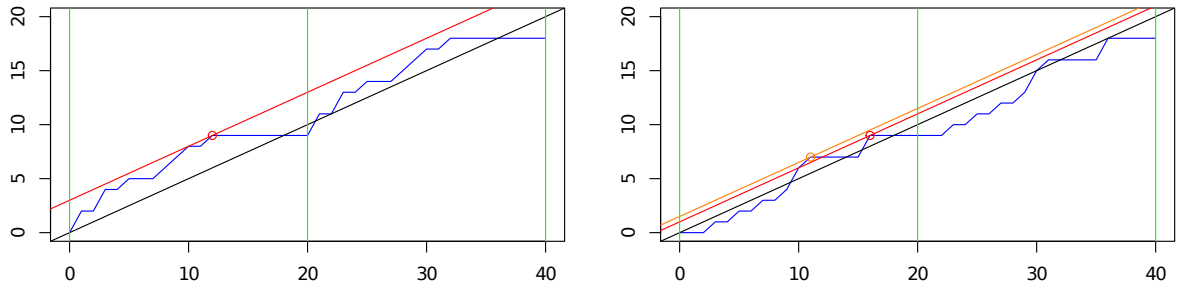


Figure 4.7: Two examples of Lemma 4.3.4 with $n = 10$. The blue line shows two periods of the partial sums $B^{(0)}_j$, separated by vertical green lines, the black line shows $y = x/2$, which was shifted up to the red line to pass through the circled point $(i, B^{(0)}_i)$ for $i = j^*$ on the left and $i = j'$ on the right (with the orange line showing the odd j^* we couldn't use).

Corollary 4.3.5. *For any such a,* ROTATE-BOUNDED(a, i) *holds for some odd index i.*

Proof. Consider $a' = (a_1, a_2, \ldots, a_{2n-1}, a_0)$. By Lemma 4.3.4, there is an even index i' which satisfies ROTATE-BOUNDED(a', i'). Thus $i \equiv i' + 1$ is an odd index such that ROTATE-BOUNDED(a, i). □

4.3.2 Construction Outline

First, we introduce our reduction from the Team DFA Game to the Team DFA Game with Communication. Given an (r, x_0, x_1, N)-rate-limited policy P and an underlying DFA D, we create a DFA D' for playing the TDGC under P which simulates playing the TDG on D while completely clogging the communication between the $\exists$ team to nullify any advantage such communication could bring. This lets us conclude that a winning strategy for TDGC on D' exists exactly when a winning strategy exists for TDG on D.

The reduction applies when each $x_i < r$, meaning the communication rate defined by P is eventually below an average of one bit per round. We also assume $r > 1$ and each $x_i > 0$: if there is no communication at all then TDGC is identical to TDG, and if communication only occurs in one direction then the aspects of the construction that deal with the silent direction may be omitted. Lastly, by Lemma 4.3.3, we take period length r to be even without loss of generality.

The code for D' is fully shown in Algorithm 3. The behavior of D' is designed around what we call the ***honest*** strategy for the $\exists$ team. We will show that it is the only strategy that guarantees the $\exists$ team will pass validation checks by D', but also that it requires using all available transmission bits, resulting in no information transfer between players for their additional benefit in the simulated TDG.

Along with the current state of D in the TDG, D' maintains two queues X_0, X_1 of clogging bits that have been given to each $\exists$ player by the adversary $\forall$ in specific rounds. These bits are expected to be submitted by the opposite $\exists$ player to D' for validation in later rounds in order to avoid losing the game, so the players are forced to use transmission bandwidth to exchange this

information. The honest $\exists_i$ player sends these bits directly and as soon as possible to $\exists_{1-i}$, who maintains a "knowledge" queue K_{1-i} of all bits sent from $\exists_i$ but not yet validated by D'. We note that $X_i \setminus K_{1-i}$ is thus the set of yet-to-be-transmitted private bits known only to $\exists_i$.

4.3.3 Build-up Phase

D' begins the build-up phase after N rounds, once P has started to repeat its policy states. This phase lasts for exactly r^2 rounds, or r periods of P's cycle. D' starts with empty X_0 and X_1, and every round D' simply enqueues b_0 and b_1 into the appropriate queues.

During these rounds, $\forall$ can send one bit per round to $\exists_i$, who can transmit those bits to $\exists_{1-i}$, for each $i \in \{0, 1\}$. Because the rate of transmission can vary above or below one bit per round, there is some maximum amount $x'_i \leq x_i$ out of r bits that can reach $\exists_{1-i}$ in the first r rounds. Each subsequent r rounds, x_i out of the r new bits can be sent (by the rate-limitedness of P), thus after r^2 rounds at most $x'_i + (r-1)x_i \leq rx_i < r(r-1)$ bits in X_i can be sent to $\exists_{1-i}$ and thus at least r are not known. By this argument, at the end of the build-up phase we can say that an honest player's knowledge queue has size $|K_i| = x'_{1-i} + (r-1)x_{1-i} \in [r-1, r(r-1)]$, since we assume $x_{1-i} \geq 1$.

4.3.4 Clogging Phase

In the clogging phase, D' simulates playing TDG on D while clogging the transmissions between $\exists$ players at a steady rate to keep $|X_i|$ and $|K_i|$ constant on period boundaries. In the last round of every period of r rounds, D' alternates between (1) having $\forall, \exists_0, \exists_1$ play one round of TDG, and (2) forcing $\forall$ to tell the $\exists$ team if they have won in the TDG yet and therefore if D' is going to start the next phase: the tear-down phase.

In the first $r-1$ rounds of each period in this phase, D' clogs the transmissions between $\exists$ players by requiring that bits given to $\exists_i$ by $\forall$ (placed into queue X_i) are sent to $\exists_{1-i}$. This is done by dequeuing the oldest bit b from X_i and checking for $\exists_{1-i}$ to submit $m_{1-i} = b$, otherwise they will lose the game. Specifically, to preserve the size of X_i and keep up with the rates at which the $\exists$ players can transmit information to each other, D' will do $\textsc{enq}(X_i, b_i)$ then validate $\textsc{deq}(X_i) = m_{1-i}$ for the first x_i rounds of each period.

Across the whole period, K_i will gain x_{1-i} new bits transmitted from $\exists_{1-i}$ (by the rate-limitedness of P). New available bits always exist because the number of private bits available to be sent is $|X_{1-i} \setminus K_i| \geq r > x_{1-i}$ at the start of the period. Additionally, across the first x_{1-i} rounds of the period, K_i will lose x_{1-i} bits submitted by $\exists_i$ to D', which are always known because $|K_i| \geq (r-1)x_{1-i} \geq x_{1-i}$ at the start. Overall, this means $|K_i|$ is preserved on period boundaries and honest players will always be able to submit the correct bit and pass the validation.

Labeling the first clogging period with index 0, at the end of every odd-indexed period, D' will simulate TDG by forwarding the inputs of all players directly to D. However, at the end of even-indexed periods, D' will ignore $\exists_0, \exists_1$ and expect both $\forall$ bits to state whether or not the $\exists$ team has won in TDG, specifically requiring that $b_0 = b_1 = [q \in F_\exists]$. If this validation fails, then D' will halt with a $\exists$ team victory, so the $\forall$ player must give the correct information to both $\exists$ players to avoid losing, which it is always able to do.

Assuming validation never fails, which is achieved by the honest strategy, the clogging phase continues until the simulated Team DFA Game ends. If the $\exists$ players lose in the simulation, they lose immediately, otherwise after the even-indexed period when the $\exists$ players learn they have won, D' moves onto the tear-down phase to perform the final validation checks.

4.3.5 Tear-down Phase

The tear-down phase starts at the beginning of a period, so by the previous arguments for queue size preservation, it starts with $|X_i| = r^2$ and $|K_{1-i}| = x'_i + (r-1)x_i$. In order to ensure the $\exists$ team's transmissions have been completely clogged all the way until the simulated victory, D' must validate that the remaining bits in K_i have actually been sent by this point.

This phase is split into two parts, with a boosting sub-phase to adjust the size of X_i and K_{1-i} for the following draining sub-phase that empties them. Once each queue has been drained and all validation checks have been passed, then D' will halt with an $\exists$ team victory. We will need the following fact:

Lemma 4.3.6. *There exists a k_{end} such that, in every round up to the k^{th}_{end} round of a period, the cumulative number of bits $\exists_0$ will transmit to $\exists_1$ before $\exists_1$ submits a bit to D' in the k^{th}_{end} round is always upper-bounded by the cumulative number of bits $\exists_1$ will submit to D' in that time (from round N onwards).*

Proof. Say the period begins in round $m \geq N$, and recall that we can assume the period length r is even. Consider the sequence a_k of the number of bits transmitted from $\exists_0$ to $\exists_1$ in the k half-rounds starting in round m, so $a_k = P_{\text{MID}}(p_{m+k/2})[0]$ when k is even and $a_k = P_{\text{END}}(p_{m+(k-1)/2})[0]$ when k is odd. Since policy states repeat, $\forall k \geq 0$. $a_{k+r} = a_k$, and $a_0 + \ldots + a_{r-1} = x_0 < r$, so we can apply Corollary 4.3.5 to the reversed sequence $(a_{r-1}, \ldots, a_0)$ to get an odd index i such that $\forall j > 0$. $B^{(i)}_j < \frac{j}{2}$.

Since $B^{(i)}_j$ is the cumulative number of bits transmitted from $\exists_0$ to $\exists_1$ across the j half-rounds ending when $\exists_1$ submits a bit to D' in round $m + \frac{r-1-i}{2}$, and $\lceil \frac{j+1}{2} \rceil \geq \frac{j}{2}$ is the cumulative number of bits $\exists_1$ submits to D' across the same set of j half-rounds, then round offset $k_{end} = \frac{r-i-1}{2}$ satisfies Lemma 4.3.6. □

Draining Sub-Phase Given k_{end} from Lemma 4.3.6 (by symmetry, the lemma applies in both directions), let $t_{end} \leq k_{end}$ be the total number of bits transmitted from the beginning of a period until the bit submission in the k^{th}_{end} round. If a period starts with $|X_i| \leq k_{end}$ and $|X_i \setminus K_{1-i}| = t_{end}$, then we can have D' validate bits in the $|X_i|$ rounds before the k^{th}_{end} round and reach $|X_i| = |K_{1-i}| = 0$ where each of the t_{end} transmitted bits are clogging bits from $X_i \setminus K_{1-i}$ with no room for extra communication from $\exists_i$ to $\exists_{1-i}$.

In order to ensure some period starts with $|X_i| \leq k_{end}$ and $|X_i \setminus K_{1-i}| = t_{end}$ we use some n_i periods beforehand to drain each queue appropriately. Since in each period there are x_i transmission bits (fixed) and up to r validated bits (based on D'), it suffices to have $|X_i| \leq n_i r + k_{end}$ and $|X_i \setminus K_{1-i}| = n_i x_i + t_{end}$.

Boosting Sub-Phase The tear-down phase must start with $|X_i \setminus K_{1-i}| = r^2 - (x'_i + (r-1)x_i) \geq r-1$, but this may not be $n_i x_i + t_{end}$ for any n_i, so before $n_i + 1$ draining periods, we will have additional periods to increase the number of private bits by $\delta_i = (n_i x_i + t_{end}) - (r^2 - (x'_i + (r-1)x_i))$. So for $\delta_i \geq 0$, we can choose any sufficiently-large n_i.

After one period where $\forall$ gives c_i new clogging bits to $\exists_i$ and D' validates v_{1-i} bits from $\exists_{1-i}$, we would have $\Delta|X_i| = c_i - v_{1-i}$ and $\Delta|K_{1-i}| = x_i - v_{1-i}$ (given that $\exists_i$ initially has $|X_i \setminus K_{1-i}| \geq x_i$ private bits to transmit to $\exists_{1-i}$), thus $\Delta|X_i \setminus K_{1-i}| = c_i - x_i$. Therefore, if we set $c_i = x_i + 1 \leq r$ and $v_{1-i} = x_i$, then we get $\Delta|X_i| = +1$, $\Delta|K_{1-i}| = 0$, and $\Delta|X_i \setminus K_{1-i}| = +1$. If $\delta_i < \delta_{1-i}$, then to delay we also need "filler" rounds with no change to the sizes of any queues, which can be achieved by setting $c_i = v_{1-i} = x_i$.

To ensure δ_i is positive and $|X_i| \leq n_i r + k_{end}$ at the end of this sub-phase, we need to choose an n_i that satisfies the following constraints at the start of the tear-down phase:

$$\begin{aligned} 0 \leq \delta_i &= (n_i x_i + t_{end}) - |X_i \setminus K_{1-i}| \\ n_i &\geq (|X_i \setminus K_{1-i}| - t_{end}) / x_i \end{aligned}$$

and

$$\begin{aligned} n_i r + k_{end} &\geq r^2 + \delta_i \\ n_i r + k_{end} &\geq |X_i| + (n_i x_i + t_{end}) - |X_i \setminus K_{1-i}| = |K_{1-i}| + (n_i x_i + t_{end}) \\ n_i &\geq (|K_{1-i}| + t_{end} - k_{end}) / (r - x_i) \end{aligned}$$

We pick n_i to be the smallest natural number satisfying both lower bounds:

$$\begin{aligned} n_i &= \left\lceil \max \left\{ \frac{|X_i \setminus K_{1-i}| - t_{end}}{x_i}, \frac{|K_{1-i}| + t_{end} - k_{end}}{r - x_i} \right\} \right\rceil \\ &= \left\lceil \max \left\{ \frac{r^2 - (x_i' + (r-1)x_i) - t_{end}}{x_i}, \frac{(x_i' + (r-1)x_i) + t_{end} - k_{end}}{r - x_i} \right\} \right\rceil \end{aligned}$$

Since $0 \leq x_i' \leq x_i < r$ and $0 \leq t_{end} \leq k_{end} < r$, we can upper bound $n_i = O(r^2)$.

Putting it all together At the beginning of the tear-down period, D' will run a set of δ_i periods where $\forall$ produces $x_i + 1$ new bits and D' validates x_i bits, followed by $\max\{\delta_0, \delta_1\} - \delta_i$ periods of x_i new and validated bits. After $\delta_{\max} = \max\{\delta_0, \delta_1\}$ rounds, we will have $|X_i| = r^2 + \delta_i$ and $|X_i \setminus K_{1-i}| = n_i x_i + t_{end}$, preserving $|K_{1-i}| = x_i' + (r-1)x_i$. D' will then run n_i periods plus k_{end} rounds ignoring $\forall$ and validating the remainder of X_i (starting $|X_i|$ rounds before the end).

4.3.6 Proof of Undecidability

Theorem 4.3.1. *TDGC is undecidable under all (r, x_0, x_1, N)-rate-limited policies where $x_0 < r$ and $x_1 < r$.*

Proof. We reduce from the Team DFA Game. For any (r, x_0, x_1, N)-rate-limited policy P where $x_0, x_1 < r$, given an input DFA D for playing the TDG, we construct the DFA D' described in the previous sections (given in full detail in Algorithm 3) for playing the TDGC under policy P. Since determining whether or not the $\exists$ team has a forced win in the TDG is undecidable, this reduction will show that the same question of the TDGC under policy P is undecidable as well.

Given the analysis of D' up to this point, we first note that D' is indeed a finite automaton: the waiting counter takes on N values; each queue X_i has maximum size $r^2 + \delta_i$ bits, where $n_i = O(r^2)$ so $\delta_i = O(r^3)$; the state q of D has $|Q|$ possible values; and the various other counters require $O(\log r)$ bits each. From beginning to end, the maximum memory requirement is $O\left(\max\left\{\log N, r^2 + \log|Q|, r^3\right\}\right)$ bits, summarizing Table 4.1.

If there is a winning strategy S for the $\exists$ team on D in the TDG, then the corresponding honest strategy described above that plays the simulated TDG using S will be a winning strategy for the $\exists$ team on D' in the TDGC under policy P.

If there is a winning strategy for the $\exists$ team on D' in the TDGC under policy P, then consider any winning execution γ. Since winning requires termination, let C be the number of periods in the clogging phase.

If γ reaches HALT($\exists$) in the clogging phase because $\forall$ did not correctly tell the $\exists$ team whether or not $q \in F_\exists$, then $\forall$ did not play optimally. Since $\forall$ has perfect information and is allowed to give

State Category	**Space Needed (bits)**
HALT($winner$)	$\Theta(1)$
WAITING(w)	$\Theta(\log N)$
BUILD-UP(X_0, X_1)	$\Theta(r^2)$
CLOG(X_0, X_1, q, p, k, c_{01}, c_{10})	$\Theta(r^2 + \log \lvert Q \rvert)$
BOOST(X_0, X_1, d_{01}, d_{10}, k, c_{01}, c_{10})	$\Theta(r^2 + \delta_{\max})$
DRAIN(X_0, X_1, c_{01}, c_{10})	$\Theta(r^2 + \delta_{\max})$

Table 4.1: Memory Requirements of D' over the course of the TDGC.

either 0 or 1 by the game rules, there is an alternate execution γ' where $\forall$ gives the correct answer instead and the game continues, so no $\exists$ team strategy can force a win in this way.

The only other way for the $\exists$ team to win is for γ to reach HALT($\exists$) at the end of the draining phase, which means they must pass all of the validation checks by D'.

Phase	ENQ(X_i) Count	DEQ(X_i) Count	Information $\exists_i \to \exists_{1-i}$
Build-up	r^2	0	$x'_i + (r-1) \times x_i$
Clogging	$C \times x_i$	$C \times x_i$	$C \times x_i$
Boosting	$\delta_{\max} \times x_i + \delta_i$	$\delta_{\max} \times x_i$	$\delta_{\max} \times x_i$
Draining	0	$r^2 + \delta_i$	$n_i \times x_i + t_{end_i}$

Table 4.2: Accounting of ENQ(X_i), DEQ(X_i), and Information Transfer between players in each phase

Table 4.2 details the value of three quantities in each phase of the game: the number of bits enqueued into X_i, the number of bits dequeued from X_i, and the amount of meaningful bits of information that can be transmitted from $\exists_i$ to $\exists_{1-i}$. By the definition of δ_i and some algebra, it can be seen that each column has the same sum; let I be this total quantity of bits.

Because D' validates the value of each dequeued bit, in order for $\exists_i$ to guarantee they pass all validation checks, they must send I bits of information to $\exists_{1-i}$. However, because I is the maximum amount of information $\exists_i$ can send to $\exists_{1-i}$, no further information can be sent, which means that in every round in which D' simulates the TDG on D', $\exists_i$ has the same amount of information about the state q of D as it would when actually playing TDG on D. Therefore, if the $\exists$ team has a winning strategy for playing TDGC on D' under policy P, within it is a winning sub-strategy for them to play the TDG on D. □

4.4 Decidability

We show that our general construction from the previous section is tight with respect to the transmission rate between $\exists$ players.

For our precise bounds, we assume the straightforward encoding of the input DFA D with n states as a table for δ containing $2n$ states, a state q_0, and the states in $F_\exists$ and $F_\forall$, thus the input size is $\Theta(|Q|)$.

Algorithm 3 Pseudocode for the D' internal update function per round

```
q' ← WAITING(1)                                                          ▷ Initial state
function DFA-ROUND-UPDATE(q', b_0, b_1, m_0, m_1)
    switch q'
        case HALT(winner)                                                ▷ Game is over, with q' ∈ F'_winner
            return HALT(winner)
        case WAITING(w)                                                  ▷ Waiting Phase, delaying until policy starts repeating
            if w < N then return WAITING(w + 1)
            return BUILD-UP([], [])
        case BUILD-UP(X_0, X_1)                                          ▷ Build-up Phase, filling up X_i queues
            ENQ(X_0, b_0)
            ENQ(X_1, b_1)
            if LENGTH(X_0) < r^2 then return BUILD-UP(X_0, X_1)
            return CLOG(X_0, X_1, q_0, EVEN, r, x_0, x_1)
        case CLOG(X_0, X_1, q, p, k, c_01, c_10) given k > 1             ▷ Clogging Phase, boosting
            for all i ∈ {0, 1}
                if c_{i,1−i} > 0 then
                    ENQ(X_i, b_i)
                    if DEQ(X_i) ≠ m_{1−i} then return HALT(∀)
                    c_{i,1−i} ← c_{i,1−i} − 1
            return CLOG(X_0, X_1, q, p, k − 1, c_01, c_10)
        case CLOG(X_0, X_1, q, ODD, 1, 0, 0)                             ▷ Clogging Phase, simulating D
            q ← δ(δ(δ(δ(q, b_0), b_1), m_0), m_1)
            return CLOG(X_0, X_1, q, EVEN, r, x_0, x_1)
        case CLOG(X_0, X_1, q, EVEN, 1, 0, 0)                            ▷ Clogging Phase, testing for ∃ win
            if ¬ (b_0 = b_1 = [q ∈ F_∃]) then return HALT(∃)
            if q ∈ F_∃ then return BOOST(X_0, X_1, δ_0, δ_1, r, x_0, x_1)
            if q ∈ F_∀ then return HALT(∀)
            return CLOG(X_0, X_1, q, ODD, r, x_0, x_1)
        case BOOST(X_0, X_1, d_01, d_10, k, c_01, c_10) given k > 1      ▷ Boost Phase, clogging
            for all i ∈ {0, 1}
                if c_{i,1−i} > 0 then
                    ENQ(X_i, b_i)
                    if DEQ(X_i) ≠ m_{1−i} then return HALT(∀)            ▷ Return from caller
                    c_{i,1−i} ← c_{i,1−i} − 1
            return BOOST(X_0, X_1, d_01, d_10, k − 1, c_01, c_10)
        case BOOST(X_0, X_1, d_01, d_10, 1, 0, 0)                        ▷ Boost Phase, new boost bits
            for all i ∈ {0, 1}
                if d_{i,1−i} > 0 then
                    ENQ(X_i, b_i)
                    d_{i,1−i} ← d_{i,1−i} − 1
            if d_01 + d_10 > 0 then return BOOST(X_0, X_1, d_01, d_10, r, x_0, x_1)
            return DRAIN(X_0, X_1, r × n_0 + k_{end_0}, r × n_1 + k_{end_1})
        case DRAIN(X_0, X_1, c_01, c_10) given c_01 + c_10 > 0           ▷ Drain Phase, emptying queues
            for all i ∈ {0, 1}
                if c_{i,1−i1} > 0 then
                    if |X_i| = c_{i,1−i} ∧ DEQ(X_i) ≠ m_{1−i} then return HALT(∀)
                    c_{i,1−i1} ← c_{i,1−i1} − 1
            return DRAIN(X_0, X_1, c_01, c_10)
        case DRAIN([], [], 0, 0)                                         ▷ Drain Phase, finished!
            return HALT(∃)
```

First, we demonstrate $(r, r, r, 0)$-rate-limited policies under which the Team DFA Game with Communication is not only decidable but in PSPACE. Later we will show more restrictive communication patterns are in EXPSPACE. Recall $(r, r, r, 0)$-rate-limited policies are the case where both players are allowed to exchange r bits over the course of a period of length r.

Theorem 4.4.1. *TDGC is decidable in PSPACE with a 1-bit, mid-round exchange in both directions every round: policies P with $P_{\text{MID}}(p) = (1, 1)$ and $P_{\text{END}}(p) = (0, 0)$ for all $p \in \Pi$.*

Proof. Under such a policy, TDGC becomes a perfect information game. In each round of the game, the optimal play for $\exists_i$ is to send b_i to $\exists_{1-i}$ immediately after receiving it, meaning $\exists_{1-i}$ will know both b_0 and b_1 before it chooses m_{1-i}. Since the $\exists$ team knows the initial state q_0 of D, we can consider strategy functions $s : (q, b_0, b_1) \mapsto (m_0, m_1)$, which both players can use to decide their own next move and know what move their teammate will perform as well, letting them use δ to learn the state q of D in the next round and beyond.

Note that it suffices for the $\exists$ team to have a memoryless strategy because the policy P is constant per round, DFA transitions do not depend on the history of the game, and the adversarial $\forall$ player's choices are not bound by the history either. It also suffices to have a deterministic strategy: if there exists a non-deterministic winning strategy s', then we can fix $s(q, b_0, b_0)$ to be some (m_0, m_1) with $\Pr[s'(q, b_0, b_1) = (m_0, m_1)] > 0$ because all game executions in which the $\exists$ team plays with deterministic strategy s are possible executions when playing with strategy s', thus must also be winning.

We show that deciding whether or not the $\exists$ team has a forced win in TDGC under policy P is in PSPACE by giving a brute-force search algorithm. For every strategy s among the $4^{4|Q|}$ possible strategy functions, we construct a game graph G_s where each state $q \in Q \setminus F_\exists$ is a vertex and for all $b_0, b_1 \in \{0, 1\}$, q has an edge to $q' = \delta(\delta(\delta(q, b_0), b_1), m_0), m_1)$ where $(m_0, m_1) = s(q, b_0, b_1)$ as long as $q' \notin F_\exists$. This means s is a winning strategy if and only if all $q \in F_\forall$ and all cycles are not reachable from q_0 in G_s, since otherwise the traversal corresponds to a losing execution or the start of a potentially non-terminating execution of the game that the $\forall$ player can force to occur. We can thus perform an exhaustive depth-first search from q_0 for a counterexample (of length at most $|Q|$) to decide whether or not s is a winning strategy. Since we only need $\Theta(|Q|)$ space to store the current s, G_s, and depth-first search stack, this algorithm runs in PSPACE. □

Since it is sufficient to send only one bit of useful information mid-round, we can extend Theorem 4.4.1 to higher transmission rates.

Corollary 4.4.2. *TDGC is decidable in PSPACE with at least a 1-bit, mid-round exchange in both directions every round: policies P with $P_{\text{MID}}(p)[i] \geq 1$ for all $p \in \Pi$ and each $i \in \{0, 1\}$.*

Next, we consider the decidability of TDGC under $(r, r, 0, 0)$-rate-limited policies, which is tight given the undecidability of $(r, r - 1, 0, 0)$-rate-limited policies. This shows that only one member of the team needs to have perfect information.

Theorem 4.4.3. *TDGC is decidable in EXPSPACE with a 1-bit, mid-round exchange every round from $\exists_0$ to $\exists_1$, but none from $\exists_1$ to $\exists_0$: policies P with $P_{\text{MID}}(p) = (1, 0)$ and $P_{\text{END}}(p) = (0, 0)$ for all $p \in \Pi$.*

Proof. As described in the proof of Theorem 4.4.1, $\exists_0$ can and should send b_0 to $\exists_1$ each round to give $\exists_1$ perfect information, but $\exists_0$ themself can learn nothing about b_1. Using the terminology from [PR79], this asymmetry makes TDGC under P a hierarchical team game. To decide the existence of a winning strategy, we adapt ideas from the proof of Theorem 4 in the same paper

that shows $\text{DTIME}\left(2^{2^{2^{cS(n)}}}\right) \supseteq \text{MPA}_2\text{-SPACE}(S(n))$, the languages decided by hierarchical 2-vs-1 private alternation Turing machines in $S(n)$ space.

Consider the set of all possible mid-round configurations (q, b_0, b_1) of the game, which are fully known to $\forall$ and $\exists_1$. Define C be the set of possible configurations (b_0, u) of $\exists_0$'s mid-round knowledge: the known b_0 and the set $u \in \mathcal{P}(Q \times \{b_0\} \times \{0,1\})$ of possible mid-round configurations given the history of the game thus far. Since two game states with the same $c \in C$ are strategically equivalent from the perspective of $\exists_0$ (and thus $\exists_1$ too), a winning strategy only needs to account for the $|C| = 2^{2|Q|+1}$ knowledge configurations in its decision-making.

Given this, we can do a brute-force search as in Theorem 4.4.1 over the space of deterministic $\exists$ team strategies $s : c \in C \mapsto (m_0, m_1)$ of size $4^{|C|}$. For each s, we construct the game graph G_s, where $c \in C$ has an outgoing edge representing the outcome of each b_0, b_1 choice of $\forall$ after the $\exists$ players use s to make their moves and $\exists_0$ updates their knowledge, and then search for counter-example game executions with length up to $|C|$ to decide whether s is a winning strategy. Therefore, TDGC under P is decidable in $\Theta(|C|)$ space, which is exponential in $|Q|$. □

As before, Theorem 4.4.1 extends to higher transmission rates from $\exists_0$ to $\exists_1$ (or vice versa), as long as the receiver stays silent.

Corollary 4.4.4. *TDGC is decidable in PSPACE with at least a 1-bit, mid-round exchange in one direction every round, but none in the other direction: policies P with $P_{\text{MID}}(p)[i] \geq 1$ and $P_{\text{MID}}(p)[1-i] = P_{\text{END}}(p)[1-i] = 0$ for all $p \in \Pi$ and some $i \in \{0,1\}$.*

Given the results of Theorems 4.4.1 and 4.4.3 and their corollaries, we conjecture that r is a tight bound in all other cases.

Conjecture 4.4.1. *TDGC is decidable under all (r, x_0, x_1, N)-rate-limited policies where $x_0 \geq r$ or $x_1 \geq r$.*

4.5 Team Formula Games with Communication

Formula games model many types of games. The Team Formula Game was defined and proven undecidable in [DH08]. We define a communication version of this game and prove results analogous to the ones for TDGC.

Definition 4.5.1. A ***Team Formula Game*** (TFG) instance consists of sets of Boolean variables X, X', Y_1, Y_2 and their initial values; variables $h_0, h_1 \in X$; and Boolean formulas $F(X, X', Y_0, Y_1)$, $F'(X, X')$, and $G(X)$ such that F implies $\neg F'$. The TFG problem asks whether $\{W_0, W_1\}$, team White, has a forced win against $\{B\}$, team Black, in the game that repeats the following steps in order ad infinitum:

1. B sets X to any values. If F and G are true, then Black wins. If F is false, White wins.
2. B sets X' to any values. If F' is false, then White wins.
3. W_0 sets Y_1 to any values.
4. W_1 sets Y_2 to any values.

where B has perfect information but W_i can only see the values of Y_i and h_i.

Definition 4.5.2. ***Team Formula Game with Communication*** (TFGC) is TFG along with a policy P which specifies a number of bits to be transmitted between W_0 and W_1 mid-round (before each step 3) and at the end of the round (after each step 4)

Theorem 4.5.1. *TFGC is undecidable under all (r, x_0, x_1, N)-rate-limited policies where $x_0, x_1 < r$.*

Proof. For any such policy P, we reduce from the Team DFA Game with Communication under the same policy P, adapting the reduction done in Theorem 8 of [DH08] from the Team Computation Game to the Team Formula Game. In the reduction, the White team plays as the $\exists$ team and B plays as $\forall$ while also facilitating the simulation of TDGC in TFGC.

Given a DFA D to play TDGC under P, we first augment D so it will be suitable for the simulation. To each state, we add a 3-value counter to eliminate any four-edge cycles in the transition graph ($t \xrightarrow{\delta} (t+1) \xrightarrow{\delta} (t+2) \xrightarrow{\delta} t \xrightarrow{\delta} (t+1) \neq t$). Also, we add four new states in a path $q_0 \xrightarrow{\delta} q_0^{(1)} \xrightarrow{\delta} q_0^{(2)} \xrightarrow{\delta} q_0^{(3)} \xrightarrow{\delta} q_0^{(4)}$ from a new initial state q_0 to the original initial state $q_0^{(4)}$ in order to delay the first meaningful state transitions until the start of the second round, which is when the first set of player inputs are available.

In the instance of TFGC, we will have (1) variables $h_i = b_i \in X$ and $b_i' \in X'$, representing the $\forall$ player's chosen bits in the current and previous round; (2) $Y_i = \{m_i\}$, containing the $\exists_i$ player's message bit each round; (3) sets of $\Theta(\log |Q|)$ variables $\langle q' \rangle \subset X'$ and $\langle q \rangle \subset X$ that encode the previous state q' and current state q; (4) and two parity bits $p \in X$ and $p' \in X'$ which B will be required to flip each round. We also choose the initial value of q' to be q_0 so that in step 1 of the first round B will be forced to set q to $q_0^{(4)}$; other initial values are arbitrary.

In step 1, formula F holds if B sets X so $q = \delta(\delta(\delta(\delta(q', b_0'), b_1'), m_0), m_1)$, $q \notin F_\exists$, and $p' \neq p$. Formula G will be true if the current state $q \in F_\forall$. Thus, when F and G are both true, then in the TDGC the state transition function was correctly implemented and led to a final state where $\forall$ has won, and thus Black wins the TFGC. On the other hand, if F is false, then either B violated the simulation or the TDGC led to a final state where $\exists$ team has won, and thus White wins the TFGC.

In step 2, formula F' will be true if B sets X' such that $q' = q$ and $p' = p$, updating the previous state for the next round to the new state. If F' is false, then B violated the simulation, and thus White wins the TFGC. Additionally, the parity bit checks guarantee that F implies $\neg F'$.

Since this is a faithful simulation where each round of TFGC corresponds exactly to one round of TDGC, and by Theorem 4.3.1 it is undecidable whether or not there exists a winning strategy for the $\exists$ team playing TDGC under P, it is also undecidable whether or not there exists a winning strategy for White playing TFGC under the same policy P. □

The strategy for proving decidability results of Team DFA Game with Communication also be used to give the following tight decidability results on the Team Formula Game with Communication.

Theorem 4.5.2. *TFGC is decidable in 2-EXPSPACE with a 1-bit, mid-round exchange in both directions every round: policies P with $P_{\text{MID}}(p) = (1,1)$ and $P_{\text{END}}(p) = (0,0)$ for all $p \in \Pi$.*

Proof. Since player B gives one bit h_i to player W_i each round, under such a policy P, h_i can be immediately transmitted to W_{1-i}. Thus W_0 and W_1 always have the same information when choosing to set Y_0 and Y_1. This effectively makes the TFGC a two-player game, but not a perfect information game: the values of $X' \cup X \setminus \{h_0, h_1\}$ are not visible to White, and between each move of White, Black is allowed to set X and X' to any values. However, White does gain information on their turns from knowing h_0, h_1 and that the game did not end, as it must have been that F was true and G was false after step 1, and F' was true after step 2.

Therefore, as before, we can check over the entire space of strategies $s : (c, h_0, h_1) \mapsto (Y_0, Y_1)$ for White, where $c \in C$ specifies the set of possible hidden values for $X' \cup X \setminus \{h_0, h_1\}$, by creating a

strategy graph G_s on C with degree 4 to search for counterexample paths of length at most $|C|$ to s being a winning strategy. Since $|C| = 2^{2^{|X|+|X'|-2}}$, we only require space doubly-exponential in the input size, which is $\Omega(|X| + |X'|)$. □

Theorem 4.5.3. *TFGC is decidable in 3-EXPSPACE with a 1-bit, mid-round exchange every round from W_0 to W_1, but none from W_1 to W_0: policies P with $P_{\text{MID}}(p) = (1, 0)$ and $P_{\text{END}}(p) = (0, 0)$ for all $p \in \Pi$.*

Proof. By the same argument as in Theorem 4.4.3, we can extend Theorem 4.5.2 by adding the set of possible knowledge states of W_0 to the game configuration. If we take $C = \mathcal{P}(X' \cup X \setminus \{h_0, h_1\})$ then we must consider triplets (h_0, c_0, c) for actual configurations $c \in C$ the game can be in, and $c_0 \in \mathcal{P}(C)$ specifying what W_0 thinks c could be.

This allows us to bound the space required to search (like before) for a winning strategy $s : (h_0 \in \{0, 1\}, c_0 \in \mathcal{P}(C)) \mapsto (Y \cup Y')$ to $2^{2^{2^{O(|X|+|X'|)}}}$, triply-exponential in the input size, which is $\Omega(|X| + |X'|)$. □

As with TDGC, these results for TFGC suggest that r may be a tight bound in all other cases.

Conjecture 4.5.1. *TFGC is decidable under all (r, x_0, x_1, N)-rate-limited policies where $x_0 \geq r$ or $x_1 \geq r$.*

4.6 Open Problems

One exciting question is whether we can prove computational complexity results about real games with communication. It seems plausible that TDGC may be sufficient for applications to games with highly structured communication. We present a number of questions that we think may help strengthen results to allow their application to more real world scenarios or questions we find particularly interesting for their own sake.

One of the main technical questions left open by this work is the complexity for rate-limited policies with $x_0 \geq r$ and $r > x_1 > 0$. We conjecture this case is decidable but our current arguments rely on both players either having full information or no information.

Looking further, there are many interesting variations and extensions of this model to study. Our arguments rely heavily on communication policies having some bounded period which is useful both for algorithms to bound the uncertainty in the game and for undecidability to allow for constructions that simulate a step in a zero information game after a bounded number of rounds. What happens if our policy is described by something more general than a DFA, such as a sequence recognizable by a pushdown automaton?

Similarly, some of our arguments rely on the fact that the game is played on something with bounded state, such as a DFA or Boolean Formula. What happens with team games on more general systems, such as a pushdown automaton or a bounded space Turing Machine?

Many realistic scenarios have noisy communication channels. How does the computational complexity change under different models of noise? We conjecture that there will again be a cutoff based on whether the information capacity of the channel is sufficiently high. However, it is also possible that the small probability of error will compound over these games of unbounded length resulting in different behavior. It would also be interesting to understand what happens when other sources of inherent randomness are introduced to these games.

It is also often the case that one's ability to communicate depends on the state of the environment and potentially the actions of the people involved. Thus having communication policies that depend on player actions or the game state would be another interesting generalization.

We also only consider two players on the Existential Team. We believe that when more players are added, undecidability will emerge if at least two players have imperfect information. However, this should be verified and the details around more complex communication patterns may lead to richer behavior.

Finally, there is an issue when trying to apply these results to real games or real world problems. Our characterization in some sense relies on communication being high or low compared to critical or meaningful choices in the games. Many natural scenarios have a much larger action space than communication rate, however many of those choices may be essentially equivalent or strategically inadvisable. Our undecidability proofs in Chapter 3 based on the Team Graph Game have very inefficient reductions and require significant numbers of in-game actions to simulate one move in the DFA game. This makes a direct application of our results in this chapter difficult.

Part III

Concurrency and External Memory

Chapter 5

Atomic Gadget Simulations for Asynchronous Motion Planning

This chapter presents results by the thesis author in collaboration with Erik D. Demaine.

Overview

We study the motion-planning-through-gadgets framework with multiple robots through the lens of asynchronous concurrency and traditional shared memory algorithms. We give a universal simulation of any bounded-memory shared memory algorithm by gadgets and a construction of MRMW atomic multivalued register gadgets from SRSW safe boolean register gadgets, as well as an impossibility result for simulating the task of consensus in gadgets. As robots are anonymous, we also develop multiplexers and demultiplexers to temporarily merge single-robot paths or separate multi-robot paths through the gadget system. Finally, along the way, we present a small set of gadgets that can efficiently provide mutual exclusion of a shared region of the system for any number of robots.

5.1 Introduction

In this chapter, we describe an asynchronous multi-robot variant of the motion-planning-through-gadgets model; see our other discussions of this model in Chapters 1, 2, and 6 for more details. When possible, we borrow standard terminology from the asynchronous shared memory model for parallel concepts.

A ***robot*** is an independent player-controlled agent that traverses a system of ***gadgets***, which are objects with finite internal states, connected at external ***ports*** at ***locations*** in the system. A gadget is static over time except when acted upon via a ***transition*** by a robot; we call a sequence of transitions through the system by one robot a ***traversal***. A system is ultimately built from ***base gadgets***, which are black-box gadgets where all transitions are indivisible, occurring in one step, as defined in other chapters. In this chapter, we give special attention to ***composite gadgets***, which are a subsystem of other gadgets intended to be a simulation of some base gadget. Simulations have been a powerful tool for showing the power and hardness of single-robot motion planning from the beginning [DGLR18, DHL20], but we will need to generalize the notion of a simulation from those works due to new challenges that arise in the multirobot setting. In particular, we will need to specify what correctness means during periods with concurrent traversals by multiple robots which have no quiescent states to correspond to the state of some other gadget, and we will need to give progress guarantees that robots performing a traversal can eventually succeed.

In our asynchronous model, an execution proceeds in a sequence of ***turns***. On a robot's turn, it observes the state of the system and may choose to take a step by performing any available transition from its location of any adjacent base gadget, or pass and do nothing. In most of this chapter, we assume an ***adversarial scheduler***, meaning that any robot may be chosen to take the next turn in an execution, nondeterministically. This is in contrast with the multi-robot setting in Chapter 2 where the single player controls both the turn order and actions of robots.

We define the correctness of a composite gadget's simulation in terms of histories. A ***history*** is a sequence of entrances and exits from gadgets occurring in a possible execution. A robot's traversal through a gadget thus represents an interval of the history between its entrance and exit; any two traversals may be concurrent (if their intervals overlap) or sequential (if their intervals are disjoint). A history is called ***linearizable*** [HW90] if the interval of every traversal in the history contains a turn (its ***linearization point***) such that if all traversals were performed sequentially in the order of the linearization points, the outcome of every traversal would be the same. If every history restricted to a gadget's traversals are linearizable, then we call that gadget linearizable. Note that all base gadgets are linearizable since each traversal is just a single transition taking place in one turn; no two traversals of a base gadget are ever concurrent. The benefit of a linearizable composite gadget is that its histories cannot be distinguished from the histories of a base gadget whose transitions occur at the linearization points.

Linearizability alone does not suffice without progress guarantees for traversals. A gadget is ***wait-free*** [Her88] if there is a deterministic strategy a robot can follow such that, in every execution in which a robot attempts a traversal, there is a bounded number of turns it can be scheduled before it will complete the traversal, independent of the scheduler and the other robots' choices. A gadget is called ***atomic*** if it is linearizable and wait-free. Again, note that all base gadgets are wait-free and thus atomic by definition. An atomic composite gadget is equivalent to a base gadget since all traversals eventually complete (given enough steps), so it is a correct simulation that can function as a drop-in replacement for the base gadget for the purpose of motion planning, as defined in previous work [DGLR18].

Before continuing, we offer some observations about our choice of definitions.

Lemma 5.1.1. *A bound on the number of actions (i.e. turns that do not pass) before a robot exits in a traversal strategy is not sufficient to obtain bound on the number of turns scheduled to the robot before it exits.*

Proof. For any composite gadget, consider the strategy "pass on every turn." Any robot that follows this strategy will take a bounded number of actions (zero), but will not succeed in completing a traversal in any bounded number of turns. □

Lemma 5.1.2. *If there is a wait-free strategy that takes a bounded number of turns, there is also one in which passing is never dictated.*

Proof. Take any wait-free strategy s with bound B on the number of turns for a gadget g. Suppose in some execution a robot r is inside g, scheduled in turn i, and s dictates that r must pass. If r is scheduled repeatedly in turns $i+1, i+2, \ldots, i+B$, then s cannot dictates that r passes on each of these turns because it must exit g after at most B turns, so r must take a step on some turn $i+j \leq i+B$. Thus, there must have been a step available and an alternative wait-free strategy s' that tells r to take that step on turn i. Since there are only a finite number of configurations of g for s to account for, these "unnecessary" passes can be eliminated repeatedly to obtain a strategy that always dictates to step. □

Corollary 5.1.3. *In a wait-free gadget, following the wait-free strategy can never put a robot into a location from which it has no available transitions.*

5.1.1 Shared Memory Objects

This section will review three common objects in the shared memory model that will be analyzed through gadget-analogues in this chapter, as well as standard terminology surrounding them that we will be using. See resources such as [HSLS20] for an in-depth introduction to the field.

A ***register*** is an object that stores an integer value, supporting methods READ() to get the value and WRITE(v) to set the value to v. A register is ***safe*** if it behaves atomically except that a read that overlaps a write may return any value. It is ***regular*** if it behaves atomically except that a read that overlaps a write may return either the old value or the new value. A register may support one (SR) or more (MR) concurrent readers and one (SW) or more (MW) concurrent writers, and may be able to store values $\{0, 1\}$ (boolean) or $\{0, 1, \ldots, v_{\max} - 1\}$ (multivalued) for fixed $v_{\max} > 1$.

A ***mutex*** (or ***lock***) enables mutually-exclusive access to a resource, called the ***critical section***, supporting methods LOCK() and UNLOCK(). A thread can only access the critical section if it is holding the lock, and at most one thread can hold the lock concurrently. A mutex provides ***deadlock-freedom*** if a thread is unable to get a lock only if some other thread is holding the lock, and any thread holding the lock can always unlock.

Consensus is a task in which m threads start with private input values $V = \{v_1, ..., v_m\}$ and must all output the same value $v^* \in V$. An m-consensus object has one method DECIDE(v) which supports up to m threads calling it at most once and has the sequential specification of always returning the input of the first call to DECIDE(v). An execution state of DECIDE(v) is ***univalent*** if all possible executions after that point have the same output value, or ***bivalent*** if there are two executions with different outputs, and a ***critical state*** is a bivalent state in which all successor states are univalent.

5.2 Bounded Shared Memory Simulation

In this section, we give a universal simulation by gadgets of any shared memory algorithm on concurrent objects with bounded total memory usage, assuming we have a diode gadget as shown in Figure 5.1.

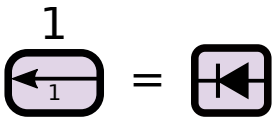


Figure 5.1: The specification of a Diode gadget (left) and its compact notation (right).

For simplicity, we take our concurrent objects to be in a canonical defunctionalized form. For an object a, let there be a single method a.INVOKE(i, f, x) which takes the thread index i, a name f from among a finite set of methods, and the list of arguments x for f of a fixed size, and runs the algorithm $a.f(x)$ as thread i, returning a value if necessary. For example, a stack data structure would permit calls such as a.INVOKE(1, "PUSH", [5]) and a.INVOKE(4, "POP", []), which can be implemented as in Algorithm 4.

We specify the algorithm A for the INVOKE method of a bounded-space concurrent object using the following components: the number of supported threads m, the finite set S of base shared objects, the finite local state space of a thread Q, input states $q_{0,(i,f,x)} \in Q$ that a thread enters

Algorithm 4 Defunctionalization Example: a FIFO Stack

```
struct DefuncStack
    Stack s;
    struct invoke(i, f, x)
        switch f
            case "push" return s.push_i(x[0])
            case "pop" return s.pop_i(i)
```

upon calling INVOKE(i, f, x), output states $F \subseteq Q$ that threads may enter when returning from a call to INVOKE, and a next-step function δ that specifies the next action of a thread in state $q \in Q$.

Theorem 5.2.1. *Any bounded-space concurrent algorithm A on a set of base shared objects can be simulated by a composite gadget G including diodes and gadget simulations of those base objects, where the history of* INVOKE(i, f, x) *calls and responses of all objects used in A correspond to a history of G with the same output behavior, and a traversal through G has the same progress guarantees as the corresponding call to* INVOKE(i, f, x).

Proof. We show how to convert A into G. As memory is bounded, each component or temporary object $a \in S$ will be represented by a gadget g_a in G; if a is a base shared object, the design of g_a is provided, otherwise g_a is recursively built by this conversion procedure. The local state $q \in Q$ of a thread will be represented by a location ℓ_q in G, and we use diodes to have "directed edges" from one location to another. Thread i is represented by a robot that starts at location $\ell_{q_{0,i}}$.

When a thread in state $q \in Q$ has its next step to call a.INVOKE(i, f, x) on some object a with states Q_a, we connect location ℓ_q through a unique diode to the location of the port of g_a representing its input state $q_{0,(i,f,x)} \in Q_a$. Simularly, when a thread returns from the call to object a in output state $q_a \in F_a$ and enters state $q \in Q$, we connect the location of the port for q_a through a unique diode to the location for q. We do not connect an output state $q \in F$ with any outgoing edges; it may act as an output port.

It is clear from the direct one-to-one correspondence of every state and step of A with locations and transitions of G that the input-output behavior and progress guarantees of all method calls will be preserved as claimed. □

5.3 Atomic Registers

In this section, we present a construction of a multi-reader, multi-writer atomic multivalued register gadget by wait-free single-reader, single-writer safe boolean register gadgets. In line with the requirements of the universal simulation from Section 5.2, and for wider applicability, each individual reader and writer corresponds to a distinct subset of ports and transitions that only needs to support use by at most one robot at a time. The construction is done through multiple layers of simulation, adapting proofs presented in [HSLS20] to the gadget model as well as tackling the model differences such as our exclusivity requirements for reader and writer areas with a solution found in a multiplexer gadget construction.

5.3.1 SRSW Safe Boolean Register

Our starting gadget is the wait-free, single-reader, single-writer, safe boolean register gadget, based on the SRSW boolean register object from the concurrent setting. The single-robot gadget

specification is shown in Figure 5.2, but our weaker requirements mean we only assume the gadget follows the specification in the multi-robot model up to these exceptions:

- More than one robot performing a read traversal at a time results in undefined behavior.
- More than one robot performing a write traversal at a time results in undefined behavior.
- A read traversal concurrent with a write traversal results in the reading robot ending up in either the 0 or 1 output location, nondeterministically.

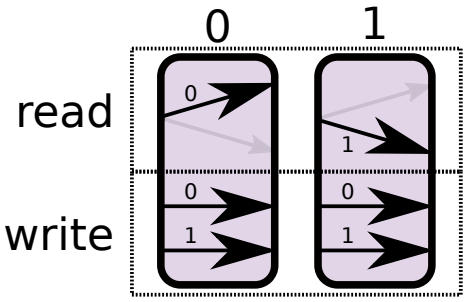


Figure 5.2: Specification of a boolean register gadget, with reader and writer areas outlined.

5.3.2 MRSW Safe Boolean Register

The first layer of the construction is to add multi-reader support. Each reader gets its own SRSW gadget, and a write traversal just involves writing to each individual SRSW. This is shown in Figure 5.3.

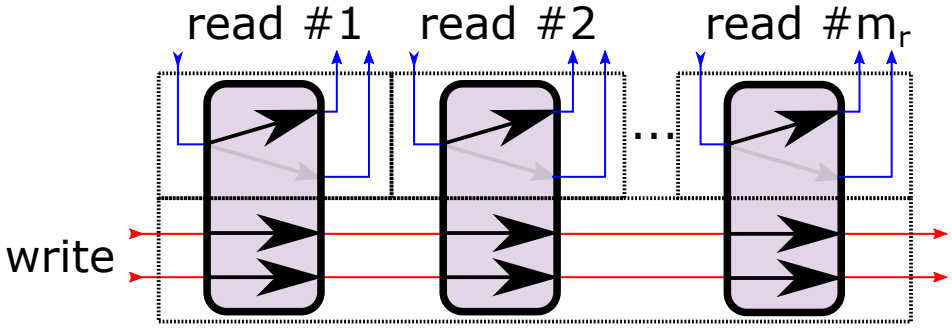


Figure 5.3: MRSW safe boolean register from the single-reader version, with the writer and each reader areas outlined.

5.3.3 MRSW Regular Boolean Register

The second layer of the construction is to improve our correctness guarantee from safe to regular, which means that a read that is concurrent to a redundant write, where the old and new value are the same, will always output that unchanging value. We do this by adding two new reading areas so the writer can branch on whether or not the old value is the same as the new value, allowing it to skip the underlying write transition unless it actually wants to change the value. This is shown in Figure 5.4.

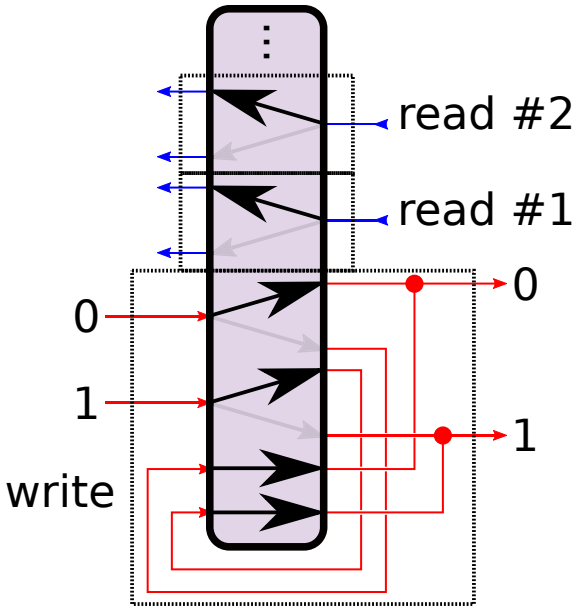


Figure 5.4: MRSW regular boolean register from the safe version, with the writer and each reader areas outlined.

5.3.4 MRSW Regular Multivalued Register

The third layer of the construction is to extend the register from boolean to any range of values. This can be achieved by chaining the boolean gadgets into an "array of bits" where the value of the register is determined by the first gadget from the beginning set to 1. A reader merely scans the array and exits from the first 1 it finds, which will always succeed as long as the end of the array is initialized a 1. A writer on the other hand will first write a 1 at the gadget at the new value's index, then writes 0 to all lower-indexed gadgets, before performing its own scan to exit out the corresponding output. An example with four values is shown in Figure 5.5.

5.3.5 Multiplexers and Single- to Exclusive-Writer Registers

So far, "single-writer" has only meant that there is one location for the single writing robot to be able to perform the write for each value, which differs from the concurrent object setting where a register can be written to in various contexts by the same writer. We can now unify these differences with a multiplexer. Analogous to the digital logic device of the same name, given multiple areas that are collectively exclusive to a single robot, a multiplexer creates a single "multiplexed" area accessible from each where the robot can eventually return to exactly the area it originated from. A multiplexer gadget can be constructed with a rewiring of a multivalued register gadget, as shown in Figure 5.6.

Using a multiplexer, we can have multiple writing locations that are still exclusive to the single writing robot. This fifth layer of the construction, of a single-reader, exclusive-writer multivalued register gadget, is shown in Figure 5.7. Each write tunnel of the MRSW gadget is placed within its own multiplexer, so each can be accessed by the writing robot from multiple areas.

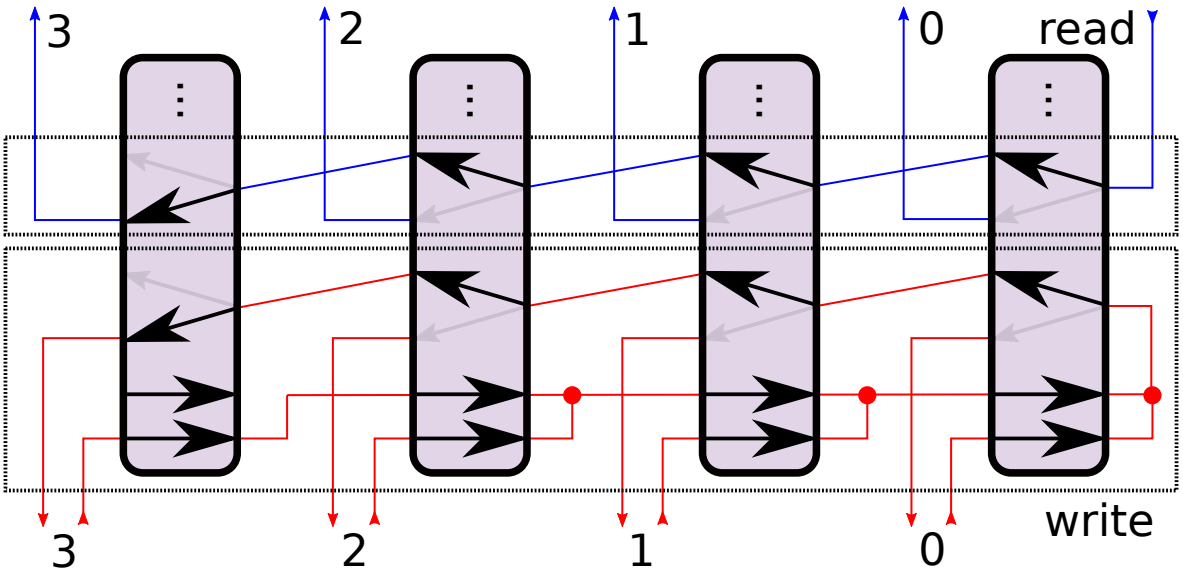


Figure 5.5: MRSW regular multivalued register gadget from the boolean version, with the writer and reader areas outlined. A four-valued register is chosen as an example.

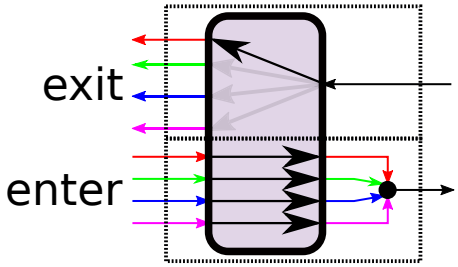


Figure 5.6: Single-robot multiplexer gadget from a multivalued register gadget. Four-way multiplexing is chosen as an example.

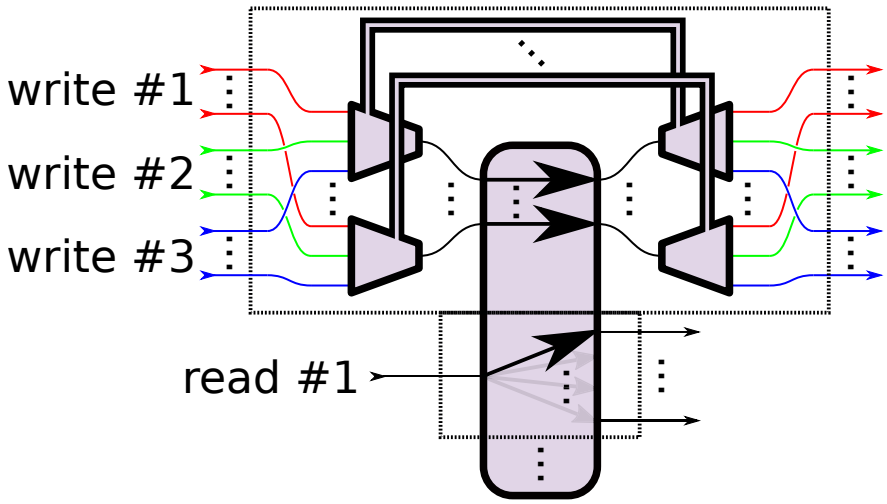


Figure 5.7: Multi-reader, exclusive-writer regular multivalued register gadget from the single-writer version and multiplexers, with the writer and reader areas outlined. Three writing areas is chosen as an example.

5.3.6 Timestamps and the MRMW Atomic Multivalued Register

With the single-writer vs exclusive-writer discrepancy solved, we now have the power to use the universal simulation from Theorem 5.2.1, which respects all properties such as single- or multi-reader, exclusive- or multi-writer, safe, regular or atomic, and boolean or multivalued.

The remaining steps to a multi-reader, multi-writer, atomic multivalued register require one final technique: a bounded timestamp system. Such a system maintains a fixed collection of objects with timestamp labels, and supports operations LABELING, which gives an object the "latest" timestamp among all objects, and SCAN, which returns a snapshot of the order of objects by their timestamps. To implement a timestamp system in the gadget model, we first apply Theorem 5.2.1 to the result of [HV95] to obtain MREW atomic multivalued register gadgets using our MREW regular multivalued register gadgets. Using these, we can apply the theorem again to the result of [DS97] to obtain a bounded timestamp system gadget implemented from MREW atomic multivalued register gadgets.

Given the existence of a bounded timestamp system, it is easy to see that all further layers of simulation presented in [HSLS20] are also bounded state, so they can be directly implemented through Theorem 5.2.1 to obtain multi-reader, multi-writer, atomic multivalued registers.

Theorem 5.3.1. *Wait-free, single-reader, single-writer, safe boolean register gadgets can simulate wait-free multi-reader, multi-writer, atomic multivalued register gadgets, where each reader and writer area is single-robot exclusive.*

5.4 Atomic Multivalued Concensus

In this section, we consider the task of consensus in the gadget model, which we can represent as a gadget like in Figure 5.8. This gadget implements ***binary one-shot consensus***: robots may perform one of two exclusive transitions to input either 0 or 1, the first robot to perform any transition fixes the state of the gadget to correspond to their input, and every robot exits the same one of their two exclusive outputs based on the state.

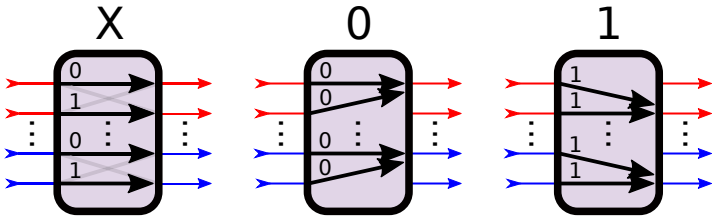


Figure 5.8: Binary Consensus Gadget

First, we consider consensus that is ***obstruction-free*** [HS11], a concurrent progress condition where a thread is guaranteed to complete a method call when given a bounded-length interval during which no other threads are scheduled.

Theorem 5.4.1. *There is a one-shot, obstruction-free implementation of a binary consensus gadget for each number m of robots with exclusive input/output lines from a fixed set of finite-sized base gadgets.*

Proof. By [Bow11], obstruction-free consensus can be implemented in shared memory from MRMW atomic registers, of bounded amount and magnitudes as a function of m. Therefore, by Theorems 5.2.1 and 5.3.1, that implementation can be simulated with the fixed-sized SRSW safe boolean registers for any m. □

Second, we consider wait-free consensus. Concurrent objects can be classified by a ***consensus number*** [Her91], which is the maximum number of threads for which it can simulate a wait-free one-shot consensus object (under certain conditions; see [GKSW17]). In the gadget model, we propose a similar analysis of gadgets and their ability to simulate wait-free one-shot consensus gadgets. The following theorem takes the first step by giving a space lower-bound and impossibility result for a restricted type of simulation. This is in contrast to single-player motion-planning, where we have universal simulator gadgets such as doors [ABD+20].

Theorem 5.4.2. *Any bounded-step (every execution has a finite number of gadget transitions), one-shot, atomic implementation of a binary consensus gadget for $m \geq 2$ robots with exclusive input/output lines requires a base gadget with at least $2m$ ports.*

Proof. Given an execution tree with vertices as states, edges as turns, and maximum height B from its bivalent root down to its univalent leaves, there must exist a execution γ ending in critical state, a bivalent state in which any step performed by any robot results in a univalent state.

In a critical state, there must exist a robot r_0 whose next step ρ_0 would lead to a 0-univalent state and another robot r_1 whose next step ρ_1 would lead to a 1-univalent state. These steps must be transitions through a single gadget g; otherwise $\gamma\rho_0\rho_1$ and $\gamma\rho_1\rho_0$ both would end in the same state due to the lack of interaction, contradicting the different univalencies. Because every robot's next step leads to a univalent state, all robots must be adjacent to the same gadget g and have at least one transition available through g. Finally, because the input/output lines are exclusive, we see that (1) no two robots can be in the same port since it would allow them to switch places, and (2) no two robots can have available transitions to the same ports or to the other's ports for the same reason, thus g must have at least $2m$ distinct ports. □

Corollary 5.4.3. *No finite-sized gadget can simulate atomic one-shot consensus gadgets for an arbitrary number of robots without permitting an infinitely-long execution where robots deviate from the wait-free strategy.*

5.5 Mutex

In the asynchronous shared memory model, a mutex is a common synchronization primitive which can be easily used to design linearizable data structures, but because a critical section of code is not guaranteed to end in a finite number of steps, they offer little help in guaranteeing wait-freedom. In this section and the following, we will be assuming a ***fair scheduler*** which gives turns to all robots infinitely-often, allowing us to use this blocking technique to guarantee progress in larger constructions.

The motion planning analogy of a shared memory mutex is an augmentation around a "critical section" of the motion planning graph that guarantees mutual exclusion of robots throughout its locations. For a section with k ports, this can be achieved using a single k-toggle: each port can be guarded by one of the k tunnels, which point inwards in the initial unlocked state and all flip outwards on any first traversal (locking), blocking other robots from entering until the first leaves (unlocks).

In lieu of requiring arbitrarily-large base gadgets like the k-toggle, we consider a pair of fixed-size gadgets: the input side uses the gadget shown in Figure 5.10, which can be simulated by toggles as in Figure 5.11, and the output side uses the gadget in Figure 5.9. $O(k)$ of these gadgets suffice to implement the mutex augmentation design shown in Figure 5.12, with the analogous shared memory mutex algorithms in Algorithm 5.

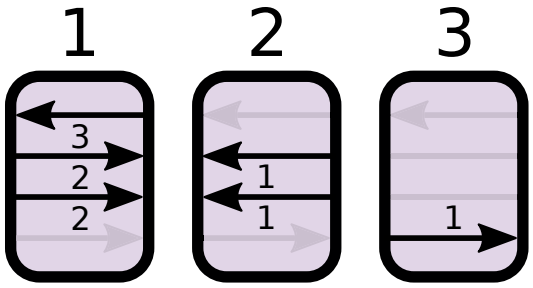


Figure 5.9: Mutex Output Gadget

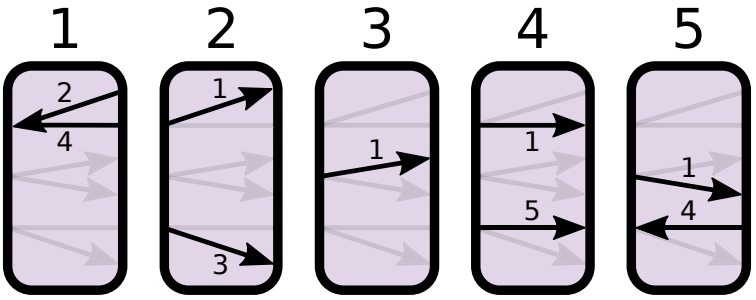


Figure 5.10: Mutex Input Gadget

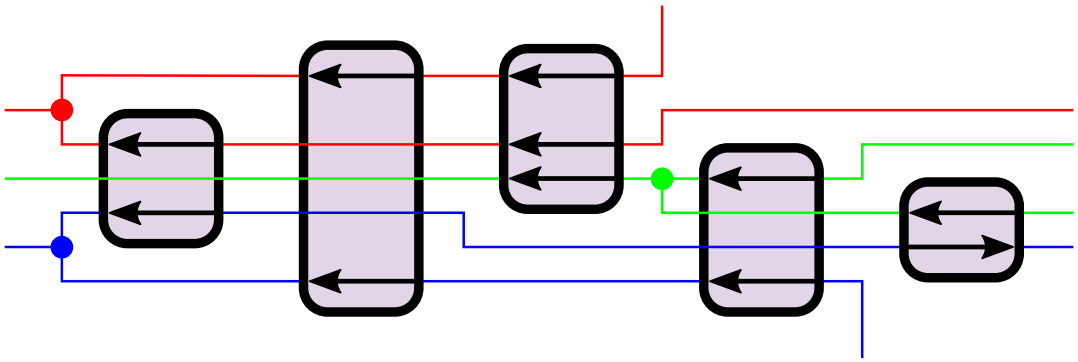


Figure 5.11: Construction of Mutex Input Gadget from a 3-toggle and 2-toggles.

We use an analogy of a LOCK() implementation where thread i repeatedly calls TRY-LOCK(i) until it succeeds. A robot outside the mutex at some port i performs TRY-LOCK(i) in order to attempt to enter the critical section, which follows the red paths starting at the top of Figure 5.12. First, the robot must enter through the input gadget $A_{in}[i]$ (state $1 \to 2$), which has the dual purpose of blocking any robot entering through higher-index input gadgets as well as opening the door to enter the critical section at the specific index i. If it gets in, to ensure no other robots can enter concurrently, the robot must then block all lower index ports by passing through each gadget $A_{in}[0:i]$ (state $1 \to 4$), which also completes the path from before $A_{in}[0]$ to the opening at $A_{in}[i]$ into the critical section.

Before that path can be traversed, the robot must check that no robot is currently performing UNLOCK, preventing interference in the future when performing UNLOCK itself. An invariant of an execution of UNLOCK(j) is that $A_{out}[j]$ is always in state 3, so a robot performing TRY-LOCK(i) which has prepared the input layer merely needs to visit each output layer gadget $A_{out}[j]$ and transition them from state $1 \to 2$ to guarantee that it is safe to enter the critical section.

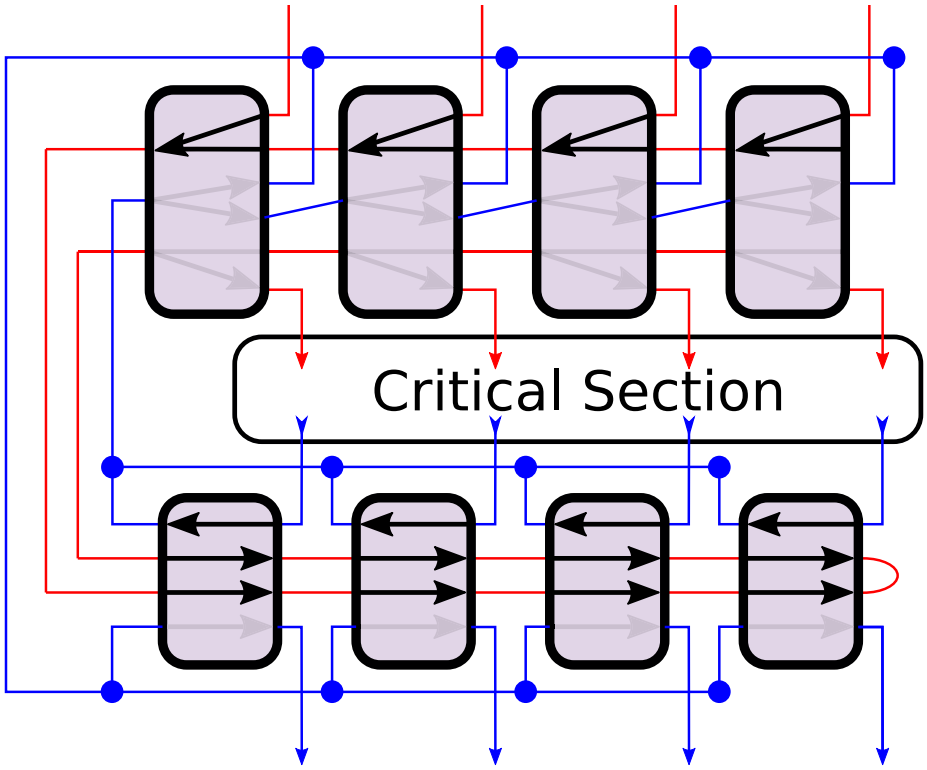


Figure 5.12: Mutex Augmentation protecting a Critical Section (4-in, 4-out), initially unlocked.

Up to this point, any of those checks may fail: $A_{in}[i]$ could be in state 2 or 4 due to another robot performing TRY-LOCK(j) for $i \leq j$, for some $j < i$ it could be that $A_{in}[j]$ is in state 2 because another robot is performing TRY-LOCK(j) (or in state 3 because they have finished performing it), and any $A_{out}[j]$ could be in state 3 because of a concurrent UNLOCK(j). If any of these gadgets fail to be in the expected "unlocked" state, then the strategy is to reverse direction and undo every previous transition until exiting from the original input port i, which will always be possible by design.

The final passes of a successful TRY-LOCK(i) will put each $A_{out}[j]$ back to state 1 then follow the path it opened which leads to $A_{in}[i]$ and transitions the gadgets $A_{in}[0:i]$ from states $4 \to 5$ and $A_{in}[i]$ from $2 \to 3$. Switching to states 5 and 3 still prevents other robots from entering but opens a reset path used in UNLOCK which can reset the input layer for future locking.

As such, a robot performs UNLOCK(j) by opening the output gadget $A_{out}[j]$ (state $1 \to 3$), following the aforementioned reset path from $A_{in}[0]$ to the original input gadget $A_{in}[i]$ to put them back in state 1, then returning to exit through $A_{out}[j]$ (state $3 \to 1$) to out-port j.

Both of these strategies only make $O(k)$ transitions. A successful TRY-LOCK(i) takes $2(i+1)$ transitions through input gadgets and $2k$ through output gadgets, and a failed execution makes up to $2(i+1)+2(k-1)$ transitions. An UNLOCK(j) will make $i+2$ transitions, following a successful TRY-LOCK(i). We can thus conclude that the mutex is deadlock-free, guaranteeing progress under our fair scheduler assumption.

This augmentation can be shown to guarantee mutual exclusion, even in the presence of robots following alternative, adversarial strategies, by considering two types of intervals in executions: first when $A_{in}[0]$ is not in state 1 and second when some $A_{out}[j]$ is in state 3. Because all paths from the outside must transition $A_{in}[0]$ out of state 1 to enter, and all exit paths must transition it back to

state 1 to leave, the first interval contains at most one robot with access to the inside of the design. As foreshadowed earlier, the second type of interval is the interval of a robot on an unlock path, which starts during the first type of interval for that robot and must end before any robot's can enter the inside again, ensuring that any unlocking is linearized before the next successful locking and thus mutual exclusion of the critical section.

Algorithm 5 Gadget Mutex's Analogous Shared Memory Algorithm

```
1: A_in  = [ i ↦ 1 | 0 ≤ i < k ]
2: A_out = [ i ↦ 1 | 0 ≤ i < k ]
3: function TRY-LOCK(i)
4:     if ¬ CAS(&A_in[i], 1, 2) then                          ▷ Reserve desired index
5:         return
6:     for i from i − 1 to 0 do                               ▷ Block lower indices
7:         if ¬ CAS(&A_in[i], 1, 4) then goto UNDO-BLOCK
8:     for i from 0 to k − 1 do                               ▷ Ensure no UNLOCK
9:         if ¬ CAS(&A_out[i], 1, 2) then goto UNDO-CHECK
10:    for i from k − 1 to 0 do
11:        A_out[i] = 1
12:    while A_in[i] = 4                                      ▷ Mark lower-indices for unlocking
13:        A_in[i] ← 5
14:        i ← i + 1
15:    A_in[i] ← 3                                            ▷ Mark desired index for unlocking
16:    return i
17:    label UNDO-CHECK:
18:    for i from i − 1 to 0 do                               ▷ Reset modified A_out
19:        A_out[i] ← 1
20:    label UNDO-BLOCK:
21:    for i ← i + 1; A_in[i] = 4; i ← i + 1                  ▷ Reset modified A_in
22:        A_in[i] ← 1
23:    A_in[i] ← 1                                            ▷ Exit from original index
24:    return i
25: function UNLOCK(i)
26:    A_out[i] ← 3                                           ▷ Mark desired exit
27:    for i ← 0; A_in[i] = 5; i ← i + 1                      ▷ Unblock lower indices
28:        A_in[i] ← 1
29:    A_in[i] ← 1                                            ▷ Unblock original index
30:    for i from 0 to k − 1 do                               ▷ Find desired exit
31:        if A_out[i] = 3 then
32:            A_out[i] ← 1
33:            return i
```

5.6 Demultiplexer

One of the major differences between the motion planning model and the shared memory model is that gadgets are not able to distinguish between robots at the same location. Many distributed and shared memory algorithms assume that threads have identifiers which can be used to uniquely

branch into different behaviors based on who is performing the action and to track the source of information. Models of anonymous processes have been studied, such as those operating on private inputs [AGM02] and those with access to randomization [FHS98], but the motion planning framework is unique because while robots with different knowledge, random bits, or team affiliations may choose different paths at the same location, they cannot be forced to make a choice based on those private differentiating factors as are threads follow a prescribed program.

As such, all of the constructions so far have ensured that robots never share locations, which lowers both the requirements on base gadgets and the complexity of the designs. In pursuit of building a self-contained simulation of a gadget, which may have robots share locations, we construct a lock-based demultiplexer augmentation. Analogous to the circuit component of the same name, a ***demultiplexer*** has k external locations that may contain a maximum of n robots, and has $m \geq n$ internal "critical sections" with guaranteed mutual exclusion. This is like giving a temporary "name" to the otherwise anonymous robot entering the gadget simulation, allowing gadgets to uniquely distinguish between robots, such as in Figure 5.13.

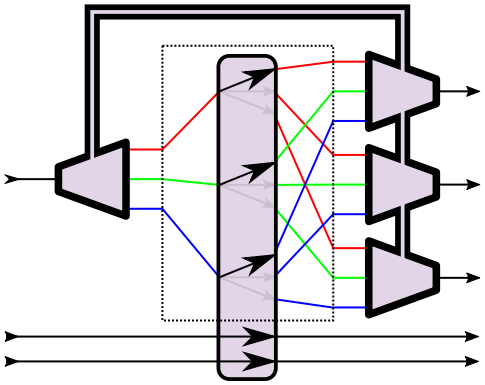


Figure 5.13: Example use of a Demultiplexer to combine the read lines of a MRSW Register.

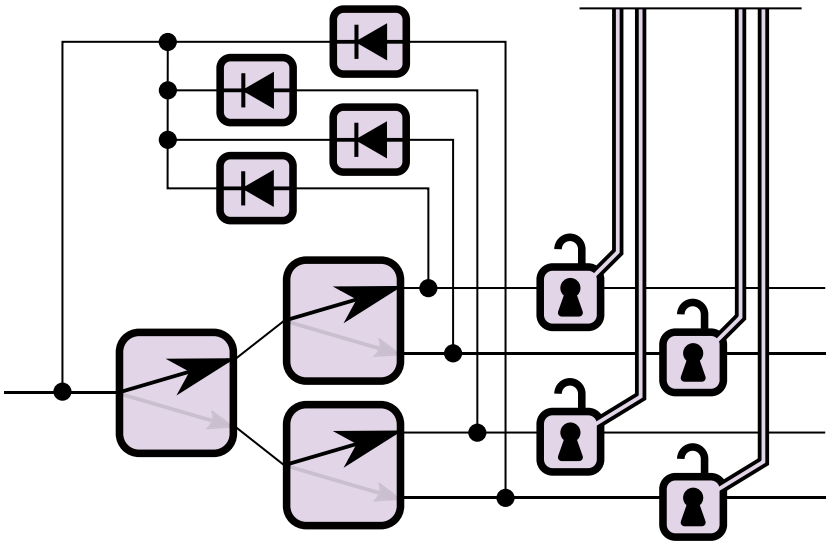


Figure 5.14: Demultiplexer access point (1-to-m, for $m = 4$), connected to others through mutexes.

What follows is a design for a demultiplexer based on toggle switch gadgets and a mutex with TRY-LOCK(i), as in Section 5.5. We will show it is linearizable and that it guarantees progress under

a ***round-robin scheduler***, a fair scheduler that allocates turns to every robot one after another in a cyclical order.

We design an access point of a demultiplexer in three parts, shown in Figure 5.14. A robot at access point i begins by entering a tree of directed toggle switches, which branches it to one of m locations, taking at least $\log m$ turns. A robot at branch j may now choose to enter in-port i of the mutex for critical section j, or choose to backtrack through a diode to leave or try another branch.

This construction strengthens the mutex, whose TRY-LOCK(i) does not guarantee success, by having a wait-free strategy DEMUX(i) which guarantees that a robot at access point i will pass through and enter one of the m critical sections. We prove that the strategy, which is simply to travel through the tree to some branch j, perform TRY-LOCK(i) on mutex j, and either succeed at entering or backtracking and repeating, will always terminate given sufficiently-large m as a function of n and the worst-case runtime $T(\text{TRY-LOCK})$ for a k-way mutex.

Suppose a robot performs $n-1$ iterations of this loop, so TRY-LOCK failed for $n-1$ different mutexes. Because TRY-LOCK is guaranteed to succeed if a robot is alone, each of those mutexes contained another robot at the time. Because a toggle switch at depth $d \leq \log m$ in the tree will only be visited once out of every 2^d traversals, the maximum rate robots can branch out of a particular leaf is once per about $(m/n)(\log m + 1)$ turns. Thus, on the n^{th} iteration of DEMUX(i), the robot is guaranteed to be alone in the mutex as long as:

$$\begin{aligned}(m/n)(\log m + 1) &\geq n(\log m + T(\text{TRY-LOCK}) + 1) \\ &> (n-1)(\log m + T_{fail}(\text{TRY-LOCK}) + 1) + \log m + T_{success}(\text{TRY-LOCK})\end{aligned}$$

which is satisfied by choosing $m = n^2 \times T(\text{TRY-LOCK})$.

5.7 Open Problems

In previous work such as [DGLR18, DHL20], gadget-to-gadget simulations are used in reductions to prove the hardness of motion planning through various gadgets, but they were not designed to be atomic. Additionally, we also showed a lower-bound for finite-length simulations of consensus gadgets, and we conjecture that the lower-bound should hold without the restriction. Given the power and apparent limitations in this model, often similar to those of shared memory, which of these simulation hardness results still hold?

Our register constructions were done with weakened requirements, promising that no base gadget will have a reader or writer area accessed by more than one robot at the same time. While this relaxation is sufficient for our universal simulation of bounded shared memory objects and generally allows for wider applicability of these results, our final constructed register gadget is only atomic under the same requirements. Our demultiplexer gadget was designed to allow for multiple robots to share ports while preserving exclusion of the inner gadget, with the trade-off that it is not wait-free. Is it possible to simulate fully-general atomic register gadgets, and what is the weakest base gadget requirements necessary?

For some applications, there are also other desirable properties for simulations that do not have traditional programming analogues, such as requiring a planar layout. Which existing planar results still hold, and which atomic planar simulations are impossible?

Chapter 6

Complexity of Reconfiguration in Surface Chemical Reaction Networks

This chapter presents results from the paper titled "Complexity of Reconfiguration in Surface Chemical Reaction Networks" that the thesis author coauthored with Robert M. Alaniz, Josh Brunner, Erik D. Demaine, Yevhenii Diomidov, Timothy Gomez, Elise Grizzell, Ryan Knobel, Jayson Lynch, Robert Schweller, and Tim Wylie. This paper is under submission at the time of the writing of this thesis [ABC+23].

Overview

We analyze the computational complexity of basic reconfiguration problems for the recently introduced surface Chemical Reaction Networks (sCRNs), where ordered pairs of adjacent species nondeterministically transform into a different ordered pair of species according to a predefined set of allowed transition rules (chemical reactions). In particular, two questions that are fundamental to the simulation of sCRNs are whether a given configuration of molecules can ever transform into another given configuration, and whether a given cell can ever contain a given species, given a set of transition rules. We show that these problems can be solved in polynomial time, are NP-complete, or are PSPACE-complete in a variety of different settings, including when adjacent species just swap instead of arbitrary transformation (swap sCRNs). Most problems turn out to be at least NP-hard except with very few distinct species (2 or 3).

6.1 Introduction

The ability to engineer molecules to perform complex tasks is an essential goal of molecular programming. A popular theoretical model for investigating molecular systems and distributed systems is Chemical Reaction Networks (CRNs) [CDS14, SCWB08]. The model abstracts chemical reactions to independent rule-based interactions that creates a mathematical framework equivalent [CSWB09] to other well-studied models such as Vector Addition Systems [KM69] and Petri nets [Pet62]. CRNs are also interesting for experimental molecular programmers, as examples have been built using DNA strand displacement (DSD) [SSW10].

Abstract Surface Chemical Reaction Networks (sCRNs) were introduced in [QW14] as a way to model chemical reactions that take place on a surface, where the geometry of the surface is used to assist with computation. In this work, the authors gave a possible implementation of the model similar to ideas of spatially organized DNA circuits [MSCS13]. This strategy involves DNA strands

being anchored to a DNA origami surface. These strands allow for "species" to be attached. Fuel complexes are pumped into the system, which perform the reactions. While these reactions are more complex than what has been implemented in current lab work, it shows a route to building these types of networks.

6.1.1 Motivation

Feed-Forward circuits using DNA hairpins anchored to a DNA origami surface were implemented in [CDM+17]. This experiment used a single type of fuel strand. The copies of the fuel strand attached to the hairpins and were able to drive forward the computation.

A similar model was proposed in [DKTT15], which modeled DNA walkers moving along tracks. These tracks have guards that can be opened or closed at the start of computation by including or omitting specific DNA species at the start. DNA walkers have provided interesting implementations such as robots that sort cargo on a surface [TLJ+17].

6.1.2 Previous Work

The initial paper on sCRNs [QW14] gave a 1D reversible Turing machine as an example of the computational power of the model. They also provided other interesting constructions such as building dynamic patterns, simulating continuously active Boolean logic circuits, and cellular automata. Later work in [CQW20] gave a simulator of the model, improved some results of [QW14], and gave many open problems- some of which we answer here.

In [BGYW19], the authors introduce the concept of swap reactions. These are reversible reactions that only "swap" the positions of the two species. The authors of [BGYW19] gave a way to build feed-forward circuits using only a constant number of species and reactions. These swap reactions may have a simpler implementation and also have the advantage of the reverse reaction being the same as the forward reaction, which makes it possible to reuse fuel species.

A similar idea for swap reactions on a surface that has been studied theoretically are friends-and-strangers graphs [DK21]. This model was originally introduced to generalize problems such as the 15 Puzzle and Token Swapping. In the model, there is a location graph containing uniquely labeled tokens and a friends graph with a vertex for every token, and an edge if they are allowed to swap locations when adjacent in the location graph. The token swapping problem can be represented with a complete friends graph, and the 15 puzzle has a grid graph as the location graph and a star as the friends graph (the 'empty square' can swap with any other square). Swap sCRNs can be described as multiplicities friends-and-strangers graph [Mil22], which relax the unique restriction, with the surface grid (in our case the square grid) as the location graph and the allowed reactions forming the edges of the friends graph.

6.1.3 Our Contributions

In this work, we focus on two main problems related to sCRNs. The first is the reconfiguration problem, which asks given two configurations and a set of reactions, can the first configuration be transformed to the second using the set of reactions. The second is the 1-reconfiguration problem, which asks whether a given cell can ever contain a given species. Our results are summarized in Table 6.1. The first row of the table comes from the Turing machine simulation in [QW14] although it is not explicitly stated. The size comes from the smallest known universal reversible Turing machine [MY07] (see [WN09] for a survey on small universal Turing machines.)

Problem	Type	Graph	Species	Rules	Result	Ref
Reconfiguration	sCRN	1D	17	67	PSPACE-complete	[QW14]
1-Reconfiguration	Swap sCRN	Grid	4	3	PSPACE-complete	Thm. 6.3.3
1-Reconfiguration	Swap sCRN	Any	≤ 3	Any	P	Thm. 6.3.6
1-Reconfiguration	Swap sCRN	Any	Any	≤ 2	P	Thm. 6.3.6
Reconfiguration	Swap sCRN	Grid	4	3	PSPACE-complete	Thm. 6.3.4
Reconfiguration	Swap sCRN	Any	≤ 3	Any	P	Thm. 6.3.5
Reconfiguration	Swap sCRN	Any	Any	≤ 2	P	Thm. 6.3.5
Reconfiguration	sCRN	Grid	3	1	NP-complete	Thm. 6.4.4
Reconfiguration	sCRN	Grid	≥ 3	1	NP-complete	Cor. 6.4.8
Reconfiguration	sCRN	Any	≤ 2	1	P	Thm. 6.4.3

Table 6.1: Summary of our and known complexity results for sCRN reconfiguration problems, depending on the type of sCRN, number of species, and number of rules. Note that all such problems are contained in PSPACE.

We first investigate swap reactions in Section 6.3. We prove both problems are PSPACE-complete using only four species and three swap reactions. For reconfiguration, we show this complexity is tight by showing with three or less species and only swap reactions the problem is in P.

In Section 6.4, we analyze reconfiguration for all sCRNs that have a reaction set of size one. For the case of only two species, we show for every possible reaction, the problem is solvable in polynomial time. With three species or greater, we show that reconfiguration is NP-complete.

Finally, in Section 6.5, we conclude the chapter by discussing the results as well as many open questions and other possible directions for future research related to surface CRNs.

In the original paper this chapter is based on [ABC$^+$23], we also studied a setting which is beyond the scope of this thesis, a restriction on surface CRNs called k-burnout where each species is guaranteed to only transition k times. This is similar to the freezing restriction from Cellular Automata [GMMRW21, GOT15, TO22] and Tile Automata [CLM$^+$18]. For 1-reconfiguration, we showed the problem is NP-complete in 1-burnout sCRNs, only using a constant number of species.

6.2 Surface CRN model

A ***chemical reaction network*** (CRN) is a pair $\Gamma = (S, R)$ where S is a set of species and R is a set of reactions, each of the form $A_1 + \cdots + A_j \to B_1 + \cdots + B_k$ where $A_i, B_i \in S$. (We do not define the dynamics of general CRNs, as we do not need them here.)

A ***surface*** for a CRN Γ is an (infinite) undirected graph G. The vertices of the surface are called ***cells***. A ***configuration*** is a mapping from each cell to a species from the set S. While our algorithmic results apply to general surfaces, our hardness constructions assume the practical case where G is a grid graph, i.e., an induced subgraph of the infinite square grid (where omitted vertices naturally correspond to cells without any species). When G is an infinite graph, we assume there is some periodic pattern of cells that is repeated on the edges of the surface. Figure 6.1 shows an example set of species and reactions and a configuration of a surface.

A ***surface Chemical Reaction Network*** (sCRN) consists of a surface and a CRN, where every ***reaction*** is of the form $A + B \to C + D$ denoting that, when A and B are in neighboring cells, they can be replaced with C and D. A is replaced with C and B with D.

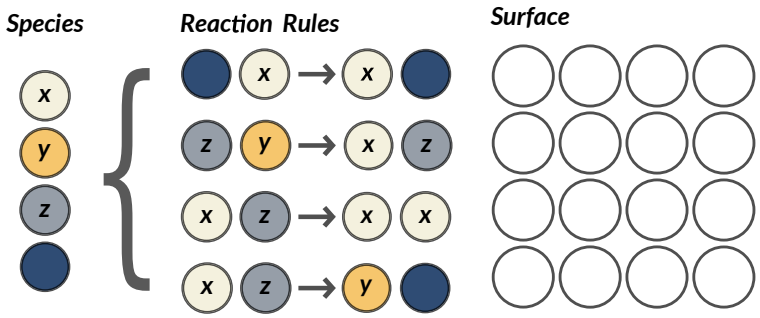


Figure 6.1: Example sCRN system.

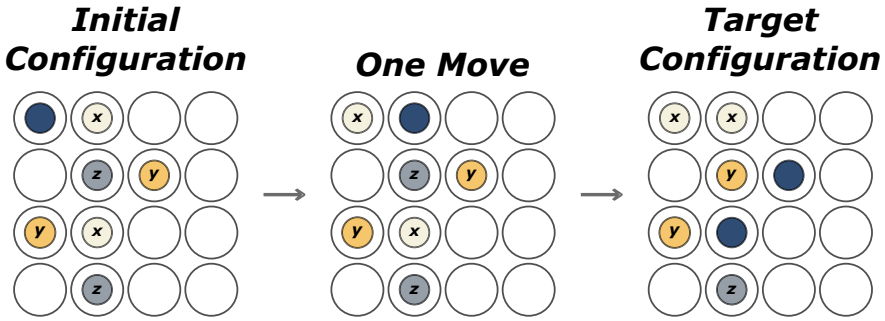


Figure 6.2: An initial, single step, and target configurations

For two configurations I, T, we write $I \to_\Gamma^1 T$ if there exists a $r \in R$ such that performing reaction r on a pair of species in I yields the configuration T. Let $I \to_\Gamma T$ be the transitive closure of $I \to_\Gamma^1 T$, including loops from each configuration to itself, meaning that T is ***reachable*** from I. Let $\Pi(\Gamma, I)$ be the set of all configurations T for which $I \to_\Gamma T$ is true.

6.2.1 Restrictions

A set of reactions R is ***reversible*** if, for every rule $A+B \to C+D$ in R, the reaction $C+D \to A+B$ is also in R. We may also denote this as a single reversible reaction $A + B \rightleftharpoons C + D$.

A reaction of the form $A + B \rightleftharpoons B + A$ is called a ***swap reaction***.

6.2.2 Problems

Reconfiguration Problem. Given a sCRN Γ and two configurations I and T, is $T \in \Pi(\Gamma, S)$?

1-Reconfiguration Problem. Given a sCRN Γ, a configuration I, a vertex v, and a species s, does there exist a $T \in \Pi(\Gamma, S)$ such that T has species s at vertex v?

6.3 Swap Reactions

In this section, we will show 1-reconfiguration and reconfiguration with swap reactions is PSPACE-complete with only 4 species and 3 swaps in Theorems 6.3.3 and 6.3.4. We continue by showing that this complexity is tight, that is, reconfiguration with 3 species and swap reactions is tractable in Theorems 6.3.5 and 6.3.6.

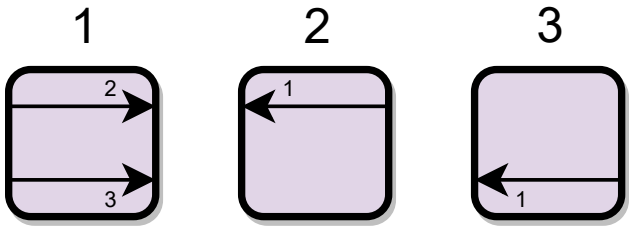


Figure 6.3: The Locking 2-Toggle (L2T) gadget and its states from the motion planning framework. The numbers above indicate the state and when a traversal happens across the arrows, the gadget changes to the indicated state.

6.3.1 Reconfiguration is PSPACE-complete

We prove PSPACE-completeness by reducing from the motion-planning-through-gadgets framework; see Chapters 1, 2, and 5 for details on this model. Important for this chapter is that the problem of changing the state of the entire system to a desired state has been shown to be PSPACE-complete [ADD$^+$22]. This reduction treats the model as a game where the player must perform reactions moving a robot species through the surface.

There are many sets of motion planning models and gadgets to build our reduction. We select 1-player over 0-player since in the sCRN model there are many reactions that may occur and we are asking whether there exists a sequence of reactions which reaches some target configuration; in the same way 1-player motion planning asks if there exists a sequence of moves which takes the robot to the target location. The existential query of possible moves/swaps remains the same regardless of whether a player is making decisions vs them occurring by natural processes. The complexity of the gadgets used here are considered in the 0-player setting in [DHHL22].

As a reminder, the ***Locking 2-toggle*** (L2T) is a 4 location, 3 state gadget. The states of the gadget are shown in Figure 6.3. The L2T has advantages because it universal for reversible deterministic gadgets. Reversibility was important to picking a gadget since swap reactions are naturally reversible.

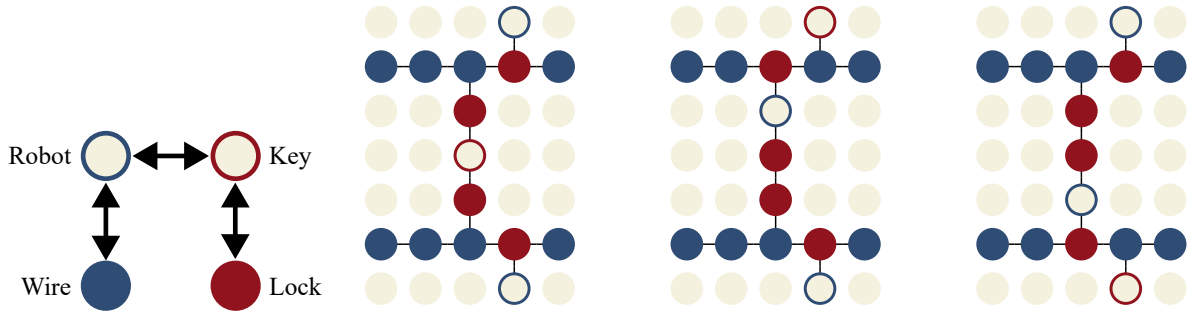
Constructing the L2T

We will show how to simulate the L2T in a swap sCRN system. Planar 1-player motion planning with the L2T was shown to be PSPACE-complete [DGLR18]. We now describe this construction.

Species. We utilize 4 species types in this reduction and we name each of them according to their role. First we have the ***wire***. The wire is used to create the connection graph between gadgets and can only swap with the robot species. The ***robot*** species is what moves between gadgets by swapping with the wire and represents the robot in the framework. Each gadget initially contains 2 robot species, and there is one species that starts at the initial location of the robot in the system. The robot can also swap with the key species. Each gadget has exactly 1 ***key*** species. The key species is what performs the traversal of the gadget by swapping with the lock species. The ***lock*** species can only swap with the key. There are 4 locks in each gadget. The locks ensure that only legal traversals are possible by the robot species.

These species are arranged into gadgets consisting of two length-5 horizontal tunnels. The two tunnels are connected by a length-3 central vertical tunnel at their 3rd cell. At the 4th cell of both tunnels there is an additional degree 1 cell connected we will call the holding cell.

States and Traversals. The states of the gadget we build are represented by the location of the key species in each gadget. If the key is in the central tunnel of the gadget then we are in



(a) Swap rules/species (b) State 1 (c) State 2 (d) State 3

Figure 6.4: Locking 2-toggle implemented by swap rules. (a) The swap rules and species names. (b-d) The three states of the locking 2-toggle.

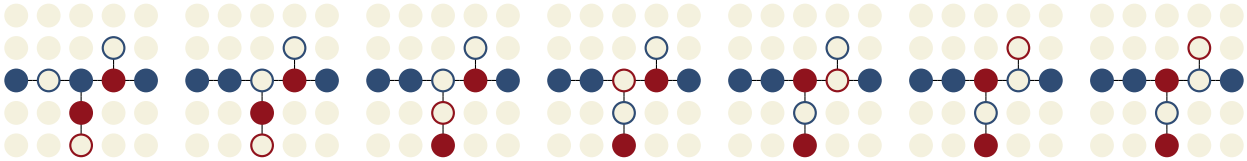


Figure 6.5: Traversal of the robot species.

state 1 as shown in Figure 6.4b. Note that in this state the key may swap with the adjacent locks, however we consider these configurations to also be in state 1 and take advantage of this later. The horizontal tunnels of the gadget in this state contain a single lock with an adjacent robot species.

States 2 and 3 are reflections of each other (Figures 6.4c and 6.4d). This state has a robot in the central tunnel and the key in the respective holding cell. The gadget in this state can only be traversed from right to left in one of the tunnels.

Figure 6.5 shows the process of a robot species traversing through the gadget. Notice when a robot species "traverse" a gadget, it actually traps itself to free another robot at the exit. We prove two lemmas to help verify the correctness of our construction. The lemmas prove the gadgets we design correctly implement the allowed traversals of a locking 2-toggle.

Lemma 6.3.1. *A robot may perform a rightward traversal of a gadget through the north/south tunnel if and only if the key is moved from the central tunnel to the north/south holding cell.*

Proof. The horizontal tunnels in state 1 allow for a rightward traversal. The robot swaps with wires until it reaches the third cell where it is adjacent to two locks. However the key in the central tunnel may swap with the locks to reach the robot. The key and robot then swap. The key is then in the horizontal tunnel and can swap to the right with the lock there. It may then swap with the robot in the holding cell. This robot then may continue forward to the right and the key is stuck in the holding cell.

Notice when entering from the left the robot will always reach a cell adjacent to lock species. The robot may not swap with locks so it cannot traverse unless the key is in the central tunnel. □

Lemma 6.3.2. *A robot may perform a leftward traversal of a gadget through the north/south tunnel if and only if the key is moved from the north/south holding cell to the central tunnel.*

Proof. In state 2 the upper tunnel can be traversed and in state 3 the lower tunnel can be traversed. The swap sequence for a leftward traversal is the reverse of the rightward traversal, meaning we are

undoing the swaps to return to state 1. The robot enters the gadget and swaps with the key, which swaps with the locks to move adjacent to the central tunnel. The key then returns to the central tunnel by swapping with the robot. The robot species can then leave the gadget to the left.

A robot entering from the right will not be able to swap to the position adjacent to the holding cell if it contains a lock. This is true in both tunnels in state 1 and in the non-traversable tunnels in states 2 and 3. □

We use these lemmas to first prove PSPACE-completeness of 1-reconfiguration. We reduce from the planar 1-player motion planning reachability problem.

Theorem 6.3.3. 1*-reconfiguration is PSPACE-complete with* 4 *species and* 3 *swap reactions or greater even when the surface is a subset of the grid graph.*

Proof. Given a system of gadgets create a surface encoding the connection graph between the locations. Each gadget is built as described above in a state representing the initial state of the system. Ports are connected using multiple cells containing wire species. When more than two ports are connected we use degree-3 cells with wire species. The target cell for 1-reconfiguration is a cell containing a wire located at the target location in the system of gadgets.

If there exists a solution to the robot reachability problem then we can convert the sequence of gadget traversals to a sequence of swaps. The swaps relocate a robot species to the location as in the system of gadgets.

If there exists a swap sequence to place a robot species in the target cell there exists a solution to the robot reachability problem. Any swap sequence either moves an robot along a wire, or traverses it through a gadget. From Lemmas 6.3.1 and 6.3.2 we know the only way to traverse a gadget is to change its state (the location of its key) and a gadget can only be traversed in the correct state. □

Now we show Reconfiguration in sCRNs is hard with the same set of swaps is PSPACE-complete as well. We do so by reducing from the Targeted Reconfiguration problem which asks, given an initial and target configuration of a system of gadgets, does there exist sequence of gadget traversals to change the state of the system from the initial to the target and has the robot reach a target location. Note prior work only shows reconfiguration (without specifying the robot location) is PSPACE-complete [ADD$^+$22] however a quick inspection of the proof of Theorem 4.1 shows the robot ends up at the initial location so requiring a target location does not change the computational complexity for the locking 2-toggle. One may also find it useful to note that the technique used in [ADD$^+$22] for gadgets and in [HD09] for Nondeterministic Constraint Logic can be applied to reversible deterministic systems more generally. This means the method described in those could be used to give an alternate reduction directly from 1-reconfiguration of swap sCRNs to reconfiguration of swap sCRNs.

Theorem 6.3.4. *Reconfiguration is PSPACE-complete with* 4 *species and* 3 *swap reactions or greater.*

Proof. Our initial and target configurations of the surface are built with the robot species at the robots location in the system of gadget, and each key is placed according to the starting configuration of the gadget.

Again as in the previous theorem we know from Lemmas 6.3.1 and 6.3.2 the robot species traversal corresponds to the traversals of the robot in the system of gadgets. The target surface can be reached if and only the target configuration in the system of gadgets is reachable. □

6.3.2 Polynomial-Time Algorithm

Here we show that the previous two hardness results are tight: when restricting to a smaller cases, both problems become solvable in polynomial time. We prove this by utilizing previously known algorithms for ***pebble games***, where labeled pebbles are placed on a subset of nodes of a graph (with at most one pebble per node). A ***move*** consists of moving a pebble from its current node to an adjacent empty node. These pebble games are again a type of multiplicity friends-and-strangers graph.

Theorem 6.3.5. *Reconfiguration is in P with* 3 *or fewer species and only swap reactions. Reconfiguration is also in P with* 2 *or fewer swap reactions and any number of species.*

Proof. First we will cover the case of only two swap reactions. There are two possibilities: the two reactions share a common species or they do not. If they do not, we can partition the problem into two disjoint problems, one with only the species involved in the first reaction and the other with only the species from the second reaction. Each of these subproblems has only one reaction, and is solvable if and only if each connected component of the surface has the same number of each species in the initial and target configurations.

The only other case is where we have three species, A, B, and C, where A and C can swap, B and C can swap, but A and B cannot swap. In this case, we can model it as a pebble motion problem on a graph. Consider the graph of the surface where we put a white pebble on each A species vertex, a black pebble on each B species vertex, and leave each C species vertex empty. A legal swap in the surface CRN corresponds to sliding a pebble to an adjacent empty vertex. Goraly et al. [GH10] gives a linear-time algorithm for determining whether there is a feasible solution to this pebble motion problem. Since the pebble motion problem is exactly equivalent to the surface CRN reconfiguration problem, the solution given by their algorithm directly says whether our surface CRN problem is feasible. □

Theorem 6.3.6. 1*-reconfiguration is in P with* 3 *or fewer species and only swap reactions.* 1*-reconfiguration is also in P with* 2 *or fewer swap reactions.*

Proof. If there are only two swap reactions, we again have two cases depending on whether they share a common species. If they do not share a common species, then we only need to consider the rule involving the target species. The problem is solvable if and only if the connected component of the surface of species involved in this reaction containing the target cell also has at least one copy of the target species. Equivalently, if the target species is A, and A and B can swap, then there must either be A at the target location or a path of B species from the target location to the initial location of an A species.

The remaining case is when we again have three species, A, B, and C, where A and C can swap, B and C can swap, but A and B cannot swap. If C is the target species, then the problem is always solvable as long as there is any C in the initial configuration. Otherwise, suppose without loss of generality that the target species is A. Some initial A must reach the target location. For each initial A, consider the modified problem which has only that single A and replaces all of the other copies of A with B. A sequence of swaps is legal in this modified problem if and only if it was legal in the original problem. The original problem has a solution if and only if any of the modified ones do. We then convert each of these problems to a robot motion planning problem on a graph: place the robot at the vertex with a single copy of A, and place a moveable obstacle at each vertex with a B. A legal move is either sliding the robot to an adjacent empty vertex or sliding an obstacle to an adjacent empty vertex. Papadimitriou et al. [PRST94] give a simple polynomial time algorithm for determining whether it is possible to get the robot to a given target location. By applying

their algorithm to each of these modified problems (one for each cell that has an initial A), we can determine whether any of them have a solution in polynomial time (since there are only linearly many such problems), and thus determine whether the original 1-reconfiguration problem has a solution in polynomial time.

□

6.4 Single Reaction

When limited to a single reaction, we show a complete characterization of the reconfiguration problem. There exists a reaction using 3 species for which the problem is NP-complete. For all other cases of 1 reaction, the problem is solvable in polynomial time.

6.4.1 2 Species

We start with proving reconfiguration is in P when we only have 2 species and a single reaction.

Lemma 6.4.1. *Reconfiguration with species $\{A, B\}$ and reaction $A+A \to A+B$ OR $A+B \to A+A$ is solvable in polynomial time on any surface.*

Proof. The reaction $A + B \to A + A$ is the reverse of the first case. By flipping the target and initial configurations, we can reduce from reconfiguration with $A + B \to A + A$ to reconfiguration $A + A \to A + B$.

We now solve the case where we have the reaction $A + A \to A + B$.

All cells that start and end with species B can be ignored as they do not need to be changed, and can not participate in any reactions. If there is a cell that contains B in the initial configuration but A in the target, the instance is 'no' as B may never become A.

Let any cell that starts in species A but ends in species B be called a ***flip*** cell, and any species that starts in A and stays in A a ***catalyst*** cell.

An instance of reconfiguration with these reactions is solvable if and only if there exists a set of spanning trees, each rooted at a catalyst cell, that contain all the flip cells. Using these trees, we can construct a reaction sequence from post-order traversals of each spanning tree, where we have each non-root node react with its parent to change itself to a B. In the other direction, given a reaction sequence, we can construct the spanning trees by pointing each flip cell to the neighbor it reacts with. □

Lemma 6.4.2. *Reconfiguration with species $\{A, B\}$ and reaction $A + A \to B + B$ is solvable in polynomial time on any surface.*

Proof. Reconfiguration in this case can be reduced to perfect matching. Create a graph M including a node for each cell in S containing the A species initially and containing B in the target, with edges between nodes of neighboring cells. If M has a perfect matching, then each edge in the matching corresponds to a reaction that changes A to B. If the target configuration is reachable, then the reactions form a perfect matching since they include each cell exactly once. □

Theorem 6.4.3. *Reconfiguration with 2 species and 1 reaction is in P on any surface.*

Proof. As we only have two species and a single reaction, we can analyze each of the four cases to show membership in P. We divide into two cases:

***A + A*:** When a species reacts with itself, it can either change both species, which is shown to be in P by Lemma 6.4.2; or it changes only one of the species, which is in P by Lemma 6.4.1.

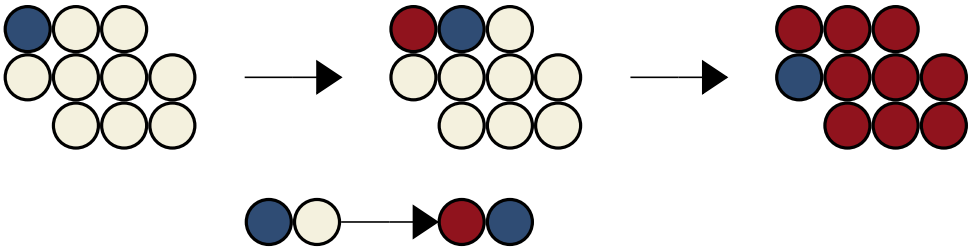


Figure 6.6: An example reduction from Hamiltonian Path. We are considering graphs on a grid, so any two adjacent locations are connected in the graph. Left: an initial board with the starting location in blue. Middle: One step of the reaction. Right: The target configuration with the ending location in blue. Bottom: the single reaction rule.

***A* + *B*:** When two different species react, they can either change to the same species, which is in P by Lemma 6.4.1; or they can both change, which is a swap and thus is in P by Theorem 6.3.5. □

6.4.2 3 or more Species

Moving up to 3 species and 1 reaction, we first show that there exists a reaction for which reconfiguration is NP-complete. We then give reactions for which reconfiguration between 3 species is in P, and in Corollary 6.4.8 we prove that all remaining reactions are isomorphic to one of the reactions we've analyzed.

Theorem 6.4.4. *Reconfiguration in sCRNs with species (A, B, C) and reaction $A + B \to C + A$ is NP-complete even when the surface is a subset of the grid graph.*

Proof. Let $\Gamma = \{(A, B, C), (A + B \to C + A)\}$. Given an instance of the Hamiltonian path problem on a grid graph H with a specified start and target vertex v_s and v_t, respectively, create a surface G where each cell in G is a node from H. Each cell contains the species B except for the cell representing v_s which contains species A. The target surface has species C in every cell except for the final node containing A, v_t.

The species A can be thought of as an agent moving through the graph. The species B represents a vertex that hasn't been visited yet, while the species C represents one that has been. Each reaction moves the agent along the graph, marking the previous vertex as visited.

($\Rightarrow$) If there exists a Hamiltonian path, then the target configuration is reachable. The sequence of edges in the path can be used as a reaction sequence moving the agent through the graph, changing each cell to species C finishing at the cell representing v_t.

($\Leftarrow$) If the target configuration is reachable, there exists a Hamiltonian path. The sequence of reactions can be used to construct the path that visits each of the vertices exactly once, ending at v_t.

We also see that this CRN has the property of 2-burnout, meaning that each cell can switch states at most 2 times before being stuck in its final state: the only possible individual cell state changes are $B \to A$ and $A \to C$. This bounds the maximum sequence length for reaching any reachable surface, putting the reconfiguration problem in NP. □

Lemma 6.4.5. *Reconfiguration with species (A, B, C) and reaction $A + B \to C + C$ is solvable in polynomial time on any surface.*

Proof. At a high level, we create a new graph of all the cells that must change to species C, and add an edge when the two cells can react with each other. Since a reaction changes both cells to

C we can think of the reaction as "covering" the two reacting cells. Finding a perfect matching in this new graph will give a set of edges along which to perform the reactions to reach the target configuration.

Consider a surface G and a subgraph $G' \subseteq G$ where we include a vertex v' in G' for each cell that contain A or B in the initial configuration and C in the target configuration. We include an edge (u', v') between any vertices in G' that contain different initial species, i.e. any pair of cell which one initially contains A and the other initially B.

Reconfiguration is possible if and only if there is a perfect matching in G'. If there is a perfect matching then there exists a set of edges which cover each cell once. Since G' represents the cells that must change states, and the edges between them are reactions, the covering can be used as a sequence of pairs of cells to react. If there is a sequence of reactions then there exists a perfect matching in G': each cell only reacts once so the matching must be perfect, and the cells that react have edges between them in G'. □

Lemma 6.4.6. *Reconfiguration with species (A, B, C) and reaction $A + B \to A + C$ is solvable in polynomial time on any surface.*

Proof. The instance of reconfiguration is solvable if and only if any cell that ends with species C either contained C in the initial configuration, or started with species B and have an A adjacent to perform the reaction. Additionally, since a reaction cannot cause a cell to change to A or B, each cell with an A or B in the target configuration must contain the same species in the initial configuration. □

The final case we study is 4 species 1 reaction. Any sCRN with 5 or more species and 1 reaction has a species which is not included in the reaction.

Lemma 6.4.7. *Reconfiguration with species (A, B, C, D) and the reaction $A + B \to C + D$ is in P on any surface.*

Proof. We can reduce Reconfiguration with $A + B \to C + D$ to perfect matching similar to Lemma 6.4.5. Create a new graph with each vertex representing a cell in the surface that must change species. Add an edge between each pair of neighboring cells that can react (between one containing A and the other B). A perfect matching then corresponds to a sequence of reactions that changes each of the species in each cell to C or D. □

Corollary 6.4.8. *Reconfiguration with 3 or greater species and 1 reaction is NP-complete on any surface.*

Proof. First, Theorem 6.4.4 shows that there exists a case of reconfiguration with 3 species that is NP-hard.

For membership in NP, we analyze each possible reaction. We note that we only need to consider two cases for the left hand side of the rule, $A + A$ and $A + B$. Any other reaction is isomorphic to one of this form as we can relabel the species. For example, rule $B + C \to A + A$ can be relabeled as $A + B \to C + C$. Also, we know that C must appear somewhere in the right hand side of the rule. If it does not then the reaction only takes place between two species, which is always polynomial time as shown above, or it involves a species we can relabel as C.

Here are the cases for $A + B$ and our analysis results:

When we have $A + A$ on the left side of the rule, the only case we must consider is $A + A \to B + C$ (since all 3 species must be included in the rule). We have already solved this reaction: first swap the labels of A and C giving rule $C + C \to B + A$, then reverse the rule to $B + A \to C + C$ and

$A + B \to A + C$	P in Lemma 6.4.6
$A + B \to C + B$	P in Lemma 6.4.6 under isomorphism
$A + B \to C + A$	NP in Theorem 6.4.4
$A + B \to B + C$	NP in Theorem 6.4.4 under isomorphism
$A + B \to C + C$	P in Lemma 6.4.5
$A + B \to C + D$	P in Lemma 6.4.7

swap the initial and target configuration. Finally since rules do not care about orientation this is equivalent to the rule $A + B \to C + C$ in Lemma 6.4.5.

Finally, for 4 species and greater, the only new case is $A + B \to C + D$, which is proven to be in P in Lemma 6.4.7. Any other case would have species that are not used since a rule can only have 4 different species in it.

Thus, all cases are either in NP, or in P which is a subset of NP, therefore, the problem is in NP. Also, since our results for each case apply for any surface, the same is true in general. □

6.5 Conclusion

In this chapter, we explored the complexity of the configuration problem within natural variations of the surface CRN model. While general reconfiguration is known to be PSPACE-complete, we showed that it is still PSPACE-complete even with several extreme constraints. We first considered the case where only swap reactions are allowed, and showed reconfiguration is PSPACE-complete with only four species and three distinct reaction types. We further showed that this is the smallest possible number of species for which the problem is hard by providing a polynomial-time solution for three or fewer species when only using swap reactions.

We next considered surface CRNs with rules other than just swap reactions. We showed reconfiguration is NP-complete for three species and one reaction type, leading to a general proof that that three species, one reaction type is NP-complete while showing that dropping the species count down to two yields a polynomial-time solution.

This work introduced new concepts that leaves open a number of directions for future work. While we have fully characterized the complexity of reconfiguration for the swap-only version of the model, the complexity of reconfiguration with general rule types for three species systems remains open if the system uses more than one rule. All of hardness results also use a square grid graph, while our algorithms work on general surfaces. We would like to know if the threshold for hardness can be lowered on more general graphs.

Chapter 7

Granular External Memory Model

This chapter presents results from an unpublished paper titled "Granular External Memory Model: Breaking the Shackles of Contiguity for Faster Algorithms" that the thesis author coauthored with Josh Brunner, Lily Chung, Erik D. Demaine, Yevhenii Diomidov, Markus Hecher, Siddhartha Jayanti, and Jayson Lynch [BCC+23].

Overview

We study a variation on the well-studied external-memory model: reads from and writes to external memory are still done in bulk (B words at once), but are no longer required to be in a contiguous block, refining the "granularity" of memory transfers. This granular model is motivated by RAM and SSD hardware being theoretically capable of bulk random access roughly as fast as bulk sequential access, if we use protocols to allow specifying multiple addresses in one operation (which many SSDs already provide). Like the standard external-memory model, this granular model is a special case of the I/O model of Aggarwal and Vitter from 1988, but the granular parameter settings have barely been studied beyond that original paper. We develop efficient data structures for dictionaries and union-find, and efficient algorithms for integer sorting, list ranking, dominating set, connected components, and minimum spanning forest, most of which beat the best known (or even the best possible) results for the external-memory model with analogous setting of parameters.

7.1 Introduction

Real-world computer systems are fundamentally limited by ***memory access latency***: as the memory system gets bigger, it necessarily gets farther away from the CPU, requiring a longer round-trip time for each memory transfer. To amortize away this latency cost, modern computer architectures increase the size of each memory transfer, transferring a ***block*** of B contiguous words containing the desired word instead of just a single desired word. The idea that a block transfer can cost only slightly more than a single word. More precisely, in a simple but reasonable model, transferring B words in a system with latency ℓ and bandwidth b takes $\ell + B/b$ time. For example, if we set $B = \ell \cdot b$, then a block transfer takes only twice as much time as a single word transfer — B words for roughly the price of one. The amortized perspective is that each word costs only a $1/B$ fraction of the transfer time, for a cost of $(\ell + B/b)/B = \ell/B + 1/b$ per word; when $B = \ell \cdot b$, this amortized cost is $2/b$ per word, which is independent of the latency ℓ. By increasing the architecture's bandwidth b (and the corresponding block size B), we can reduce the amortized

memory transfer time, effectively eliminating the latency — if the algorithm running on the CPU can actually use all B words of a memory transfer.

This setup has a counterintuitive asymmetry though: for a memory read, the request from the CPU still consists of a single word, specifying the memory address of the block to read, while the response from memory consists of B words. (For a memory write, the request is long while the response is short/empty.) What if we balanced the request and response to roughly the same length? The transfer time would remain roughly the same (up to constant factors). What could we do with a memory read where the request consists of B words instead of just one?

In this chapter, we explore a natural extension to the standard memory protocols: every memory transfer can specify B different memory addresses instead of just one. Specifically, a memory read fetches B different words potentially dispersed throughout memory, instead of being required to form a contiguous block. Similarly, a memory write specifies B words of content as well as B different addresses to write to, and executes those B writes in parallel. In other words, our extension refines the ***granularity*** of memory transfers, from the coarse grain of contiguous blocks to the fine grain of individual words, while preserving the number of words in each memory transfer. In a latency-limited system, these ***bulk*** memory transfers should be roughly as fast as the standard ***block*** memory transfers.

Indeed, we argue that our granular model should be practical in hardware architectures that are already fundamentally random-access. SDRAM [Wik22a] — the primary memory in most computers and mobile devices since the 1990s — already decomposes a block memory transfer into a pipelined sequence of individual column transfers (with column size varying from 4 to 16 bits), so the memory controller and bus protocol alone could be modified to support noncontiguous bulk transfers while using the same pipelining architecture.[1] SSD (Solid-State Drive) [Wik22b] — an increasingly popular secondary storage used exclusively on mobile devices and which for computers became the most purchased in 2020 [Kni21] — already "stripes" logically contiguous data across physical chips to parallelize block transfers, so the same approach should be equally efficient for any (well-distributed) random pattern of accesses.[2] Indeed, the modern NVMe protocol for SSDs [NVM21] (since version 1.1) provides a protocol called Scatter Gather List (SGL) for specifying noncontiguous portions of memory for parallel reads and writes, though currently they limit to unions of contiguous regions each at least 512 bytes long. Practical experiments show negligible if any difference in performance between logical blocks of 512 bytes and 4,096 bytes [Fel22], so we expect similar behavior for even smaller logical blocks. Spinning hard disk drives are the obvious exception where granular bulk accesses are slower than contiguous block accesses, because the time required to move the seek head is a fundamental bottleneck to random access.

7.1.1 Models

The original ***I/O model*** of Aggarwal and Vitter [AV88] actually defined memory transfers with both block access (required to be contiguous) and bulk access (with no contiguity requirement). It defines a system with a cache and an external memory (larger cache, RAM, or disk) by three parameters: the number M of words in the cache, the number B of words in a block (essentially, ***granularity***), and the number P of blocks that can be transferred in parallel at once. Their

[1] SDRAM decomposes memory into banks, rows, and columns, and can more quickly access multiple columns within a single row of a bank, as switching rows within a bank incurs an extra "recharge" cost. But the row switching cost can be mitigated by ensuring that the memory addresses are well distributed across the banks.

[2] Writing a word usually requires flashing the entire containing physical block, so random patterns of writes reduce the longevity of the disk, but this issue does not impact speed, and in practice disk controllers delay writes to combine them as much as possible.

motivation at the time was the (still popular) system of P spinning hard disk drives, which (via data striping) enables P parallel accesses roughly as fast as a single access, but where each drive still needs to be accessed in a large contiguous block in order to amortize away the seek latency. To evaluate an algorithm in this model, we count the number of parallel block memory transfers or ***I/Os*** (each reading or writing $B \cdot P$ words at once); we ignore the usually insignificant time spent computing on data in cache.

This model is extremely well-studied in theoretical computer science, with hundreds of algorithms and data structures; see the surveys [Arg01, Vit01, Dem02, AM09] and more recent papers [BBF+10, CFCS18, JL19, MN19]. It has also been shown to be extremely useful in practice [APSV02, CM99, DSSS04]. But essentially all work beyond the original paper [AV88] simplifies the model to just consist of two parameters, cache size M and block size B, assuming for simplicity that $P = 1$. We call this simplified model the ***external-memory (EM) model*** (following terminology from many papers that use this model). While the $P = 1$ assumption makes sense for a single spinning disk drive and with current SDRAM protocols, we have just argued that it is not consistent with the hardware capabilities of SDRAM and it already does not make sense for modern SSD protocols.

In this chapter, we make a different simplifying assumption of the I/O model, that $B = 1$. We call this model the ***granular external-memory (GEM) model***, as it is very similar to the external-memory model, with the key difference that bulk memory transfer has been refined in granularity from a single contiguous block of B words to an arbitrary pattern of P words. To ease comparison between granular and standard external-memory models with the same product $B \cdot P$, we relabel parameter P to B in the granular model. Thus both external-memory models transfer B words in each memory transfer, and all that differs is whether these words must be contiguous.

7.1.2 Results

The granular external-memory model is at least as powerful as the external-memory model, as it can in particular choose to arrange each bulk memory transfer of B words contiguously into a block. Our results show that the granular model is in fact strictly more powerful, by giving better GEM algorithms and data structures that beat the known lower bounds or best known EM results. Table 7.1 gives precise bounds for these GEM and EM results, which we summarize now:

1. We prove that the classic PRAM model for parallel algorithms can be efficiently simulated by the GEM model. This result improves the known EM simulation of the PRAM by a logarithmic factor, essentially because GEM enables random access while EM needs to sort. We use this PRAM technique to design our algorithms, which effectively provides parallel-for loops.

2. While dynamic arrays are easy to build in the EM model, in the GEM model arrays support random access in the sense that a ***batch*** of B READ(s) or B WRITE(s) (all known upfront) can be done in one I/O. We call these operations ***B-batch*** READ/WRITE. By contrast, EM needs one I/O per READ/WRITE, and thus $\Theta(B)$ I/Os for B-batch READ/WRITE, unless the B accessed elements happen to be contiguous.

   Furthermore, we show that (nonconsecutive) arrays of varying lengths can be concatenated with optimal factor-B speedup in the GEM, even if many of them are shorter than B elements. By contrast, EM needs to pay at least one I/O for every array.

3. We give several faster GEM sorting algorithms that beat the EM lower bound for sorting indivisible elements [AV88] or sorting arbitrary elements assuming a well-known network coding conjecture [FHLS20]. For indivisible elements, [AV88] already showed that comparison

Problem	**GEM Result** (number of I/Os)	**Best EM Result** (number of I/Os)
Basics (§7.2)		
p-processor PRAM Simulation (per round)	$\Theta\left(\frac{p}{B}\right)$ [Thm 7.2.1]	$O(\text{sort}(p))$ if $O(p)$ space [CGG+95] [Sorting results below define $\text{sort}(n)$]
Dynamic Array	$\Theta(1)$ for B-batch READ/WRITE, $\Theta\left(\frac{1}{B}\right)$ am. PUSH/POP [Thm 7.2.2]	$\Theta(1)$ for single READ/WRITE, $\Theta\left(\frac{1}{B}\right)$ am. PUSH/POP
Joining n Arrays of m Total Items	$\Theta\left(\frac{n+m}{B}\right)$ [Thm 7.2.3]	$\Theta\left(n+\frac{m}{B}\right)$
Sorting (§7.3)		
Sorting Indivisible Items	$O\left(\frac{n}{B}\log_M n\right)$ [AV88]	$\text{sort}(n)=\Theta\left(\frac{n}{B}\log_{M/B}\frac{n}{B}\right)$ [AV88]
Permutation	$\Theta\left(\frac{n}{B}\right)$ [Thm 7.3.2]	$\text{perm}(n)=\Theta(\min\{n,\text{sort}(n)\})$ [AV88]
Counting Sort	$O\left(\frac{n+u}{B}\right)$ [Thm 7.3.3]	$O(\text{sort}(n))$ [AV88]
Radix Sort	$O\left(\frac{n}{B}\log_n u\right)$ [Thm 7.3.4]	$O(\text{sort}(n))$ [AV88]
Union-Find (§7.4)		
Offline Union-Find	$O\left(\frac{n}{B}+\frac{m}{B}\log\frac{m}{M}\right)$ [Thm 7.4.2]	$\Omega(\text{perm}(m))$, $O\left(\text{sort}(m)\log\min\{\log B,\frac{m}{M}\}\right)$ [AAY10]
Online Union-Find	$O(\log n)$ B-batch UNION/FIND [Thm 7.4.3], $O(\alpha(n)/B)$ am. at least $(B\log(n)/\alpha(n))$-batch FIND [Thm 7.4.4]	$O(\alpha(n))$ am. ($O(\log n)$ worst case) single UNION/FIND [TvL84]
Hashing (§7.5)		
Open Addressing	$1+\Theta\left(\frac{\ln k}{W\left(\frac{B}{kx}\ln k\right)}\right)$ exp. k-batch FIND, given load factor $1-1/\Theta(x)$ [Thm 7.5.9]	$1+O\left(\frac{x}{B}\right)$ exp. single FIND, given the same assumption [BKK21]
Dynamic Perfect Hashing	$\Theta(1)$ B-batch FIND, $\Theta(1)$ exp. am. B-batch INSERT/DELETE [Thm 7.5.11]	$\Omega(\log_\lambda n)$ exp. single FIND if $O\left(\frac{\lambda}{B}\right)$ am. single INSERT [IP12]; matching upper bounds for $\lambda=\Omega(\log\log M+\log_M N)$ [IP12, CFCS18]
List Ranking (§7.6)		
List Ranking	$\Theta\left(\frac{n}{B}\right)$ [Thm 7.6.1]	$\Omega(\text{perm}(n))$, $O(\text{sort}(n))$ [CGG+95]
Graph Algorithms (§7.7)		
Maximal Independent Set	$\Theta\left(\frac{n+m}{B}\right)$ [Thm 7.7.1]	Open
Connected Components	$O\left(\frac{n+m}{B}\log B\right)$ [Thm 7.7.7]	$O\left(\min\{\text{sort}(n^2),\text{sort}(m)\log\frac{n}{M}\}\right)$ [CGG+95]; $\Theta(\text{sort}(n))$ if sparse [CGG+95]
Connected Components on Forest	$\Theta\left(\frac{n}{B}\right)$ [Thm 7.7.8]	$\Theta(\text{sort}(n))$ [CGG+95]
Minimum Spanning Forest	$O\left(\text{sort}(m)+\frac{m}{B}\log\frac{m}{M}\right)$ [Thm 7.7.10]	$O(\text{sort}(m)\log\log B)$ [ABT04]; $O\left(\min\{\text{sort}(n^2),\text{sort}(m)\log\frac{n}{M}\}\right)$ [CGG+95]

Table 7.1: Summary of our results in the Granular External Memory (GEM) model, and comparison to best known results in the External Memory (EM) model. Here O, Ω, and Θ denote upper bounds, lower bounds, and matching upper and lower bounds on the problems; and k denotes any batch size satisfying $k \leq B$.

sorting can be improved on the GEM by roughly a factor of $\Theta(\log B)$. Furthermore, we show that the GEM can execute a given array permutation as fast as scanning an array, beating the known EM lower bound. Leaving the indivisible model, we prove that GEM enables integer sorting faster than comparison sorting, specifically linear-time radix sort.

4. We develop GEM algorithms for offline union-find that improve the best EM algorithm by logarithmic factors. We develop GEM data structures for online union-find that improve the best EM data structure by a nearby factor of B in the worst case, assuming again that operations come in batches of B.

5. We develop a dynamic dictionary data structure supporting B-batch INSERT, DELETE, and FIND operations in $O(1)$ expected amortized time, beating known EM lower bounds [IP12]. While this data structure is based on dynamic perfect hashing, we also analyze simpler open-addressing hash tables, which turn out to be quite intricate (and slightly suboptimal).

6. We develop an optimal GEM algorithm for list ranking, which runs as fast as scanning an array, and is roughly a logarithmic factor faster than the best possible EM algorithm.

7. We develop efficient GEM graph algorithms for finding a maximal independent set, finding a dominating set of at most half the vertices, decomposing a graph or forest into connected components, and computing a minimum spanning forest. All of these results improve upon the best known EM algorithms for these problems.

We give detailed pseudocode for many of our algorithms to aid understanding and precision for our proofs. We borrow notation for pointers and types from C/C++ style languages, along with built-in functions and loop notation similar to Python.

7.2 Granular External Memory Model

We define the ***Granular External Memory (GEM) model*** to consist of a ***cache*** of M words connected to an external memory or ***disk*** of arbitrarily many words, where both cache and disk are random-access memories. A ***batch I/O operation*** is defined by a sequence $\mathfrak{d}$ of B disk addresses and a sequence $\mathfrak{c}$ of B corresponding cache addresses; a batch read copies data from $\mathfrak{d}$ to $\mathfrak{c}$, while a batch write copies from $\mathfrak{c}$ to $\mathfrak{d}$, in order.[3] The algorithm can do arbitrary computation on the data in cache, but cannot directly manipulate data on disk. The goal is to solve an algorithmic problem whose input is on disk, and whose output should generally be on disk, using the fewest possible batch I/O operations.

The GEM model is equivalent to the ***I/O model*** of [AV88] with the same M parameter, the GEM's B parameter plugged into the EM's P parameter, and the EM's $B = 1$. On the other hand, we define the ***External Memory (EM) model*** to be the I/O model with $P = 1$.

We concisely express batch I/O operations in GEM pseudocode as vector operations. For example, suppose A is an array stored on disk and $\mathfrak{i}$ and $\mathfrak{a}$ are arrays of length B in cache. The semantics of "$\mathfrak{a} \leftarrow A[\mathfrak{i}]$" is to locally add $A[0]$ to each index in $\mathfrak{i}$ into a temporary array $\mathfrak{i}'$ in cache then perform a batch read from $\mathfrak{i}'$ to $\mathfrak{a}$. If $\mathfrak{i}$ were length $n > B$, then this process could be performed for each contiguous block of $\mathfrak{i}$ and $\mathfrak{a}$ of length B, thus taking $\Theta(n/B)$ I/O operations and $O(B)$ extra cache space.

For breaking up more complex operations into explicit block-by-block loops, we use the notation "**for batch** $\mathfrak{i}$, $\mathfrak{b}$ **in** A" for code operating on each contiguous block $\mathfrak{b}$ (of length B, except possibly the final block) of elements at increasing indices $\mathfrak{i}$ of an array A. Algorithm 6 shows the desugaring. We sometimes omit the index vector $\mathfrak{i}$, and just write "**for** $\mathfrak{b}$ **in** A".

[3]Throughout this chapter, we use fraktur font to denote vector variables.

Algorithm 6 The "**foreach batch**" loop syntactic sugar.

```
for batch i, b in A
    ALG(i, b)
```

```
for i in 0 ... ⌊length(A)/B⌋
    i ← [ j | B · i ≤ j < min(B · (i + 1), length(A)) ]
    b ← A[i]
    ALG(i, b)
```

7.2.1 Simulating PRAM

The PRAM model [FW78] extends the RAM model with parallelism in the form of synchronized threads. A PRAM algorithm runs p threads in lockstep in rounds of the form: (1) read from one location in memory into a register, (2) perform one RAM operation in registers, then (3) write a register value to a location in memory. We show that GEM can achieve optimal speedup of PRAM simulation over sequential RAM simulation.

Theorem 7.2.1. *If a problem is solved by a PRAM algorithm that runs in $T(p)$ rounds given p threads, then it is solved by a GEM algorithm that uses $O\left(\frac{p}{B} \cdot T(p)\right)$ I/Os given a cache of size $M = \Omega(B)$.*

Proof. An explicit simulation of PRAM in GEM is shown in Algorithm 8. In each round, if all accumulator register values are stored in cache (if $M = \Omega(p)$) or saved in external memory (if $M = \Omega(B)$), we avoid problems where a simulated thread writes to a location before we had a chance to simulate another thread reading from that location. This can be adapted to work for any flavor of PRAM from EREW (exclusive-reader, exclusive writer) to CRCW (concurrent read concurrent write) with semantics for write conflicts that can be computed in $O(1)$ I/Os, which includes priority, arbitrary, and common submodels.

A high-level translation from PRAM to GEM is shown in Algorithm 7. □

The "**parallel for** e **in** $\mathfrak{v}$" statement takes a contiguous block $\mathfrak{v}$ and runs a $|\mathfrak{v}|$-thread PRAM simulation of the body of the loop, where the values e in $\mathfrak{v}$ are distributed among the threads.

The "**parallel for batch** i, e **in** v" statement is syntactic sugar for a "**for batch**" loop that performs a "**parallel for**" on each block.

Algorithm 8 Pseudocode for an Explicit GEM simulation of PRAM

```
function RUN-PRAM(reg: Reg*[], stop: int*): void
    while ¬ *stop                                  ▷ Loop until told to stop
        ips ← ip(*reg)                             ▷ Read thread instruction pointers
        op-codes ← *ips                            ▷ Read current op codes
        vals ← *read-addr(op-codes)                ▷ (1) Read values from specified locations
        acc(*reg) ← vals                           ▷ Put values into accumulator registers
        ops ← op(op-codes)                         ▷ Read RAM operations to perform
        for f in ops
            DO-RAM-INSTRUCTION(f, reg)             ▷ (2) Locally execute ops
        write ← write-addr(op-codes)               ▷ Get addresses to write to
        vals ← acc(*reg)                           ▷ Get values to write
        *write ← vals                              ▷ (3) Write values to memory
```

Algorithm 7 Recursive reduction of parallel loop syntactic sugar (left) to GEM code (right).

1: **parallel for** i, x **in** $\mathfrak{b}$ 2: ⌊ $\mathfrak{v}[i] \leftarrow A[x]$	1: $\mathfrak{v} \leftarrow A[\mathfrak{b}]$	▷ *Read I/O operation*
1: **parallel for** i, x **in** $\mathfrak{b}$ 2: ⌊ $A[x] \leftarrow \mathfrak{v}[i]$	1: $A[\mathfrak{b}] \leftarrow \mathfrak{v}$	▷ *Write I/O operation*
1: **parallel for** i, x **in** $\mathfrak{b}$ 2: ⌊ $\textsc{g}(i, x)$ 3: ⌊ $\textsc{h}(i, x)$	1: **parallel for** i, x **in** $\mathfrak{b}$ 2: ⌊ $\textsc{g}(i, x)$ 3: **parallel for** i, x **in** $\mathfrak{b}$ 4: ⌊ $\textsc{h}(i, x)$	▷ *Sequence*
1: **parallel for** i, x **in** $\mathfrak{b}$ 2: ⌊ **if** $\mathfrak{p}[i]$ **then** 3: ⌊ ⌊ $\textsc{g}(i, x)$ 4: ⌊ **else** 5: ⌊ ⌊ $\textsc{h}(i, x)$	1: $\mathfrak{t} \leftarrow \{\ i \mapsto \mathfrak{b}[i] \mid \mathfrak{p}[i]\ \}$ 2: **parallel for** j, y **in** $\mathfrak{t}$ 3: ⌊ $\textsc{g}(j, y)$ 4: $\mathfrak{f} \leftarrow \{\ i \mapsto \mathfrak{b}[i] \mid \neg\mathfrak{p}[i]\ \}$ 5: **parallel for** i, x **in** $\mathfrak{f}$ 6: ⌊ $\textsc{h}(i, x)$	▷ *Conditional*
1: **parallel for** i, x **in** $\mathfrak{b}$ 2: ⌊ **while** $\mathfrak{p}[i]$ 3: ⌊ ⌊ $\textsc{g}(i, x)$	1: $\mathfrak{j} \leftarrow \{\ i \mapsto \mathfrak{b}[i] \mid \mathfrak{p}[i]\ \}$ 2: **while** length($\mathfrak{j}$) > 0 3: ⌊ **parallel for** i, x **in** $\mathfrak{j}$ 4: ⌊ ⌊ $\textsc{g}(i, x)$ 5: ⌊ $\mathfrak{j} \leftarrow \mathfrak{j} \setminus \{\ i \mid \neg\mathfrak{p}[i]\ \}$	▷ *Iteration*

7.2.2 Maps and Arrays

Sets, maps, and dynamic arrays are useful basic building blocks for the algorithms presented in this chapter, so in this section we briefly explain the interface and implementations of sets, maps, and dynamic arrays that support efficient batch operations in the GEM model.

Given a universe of integer keys from $\{0, 1, \ldots, u\}$, the BATCH-MAP stores an array of $u + 1$ items. After the initial $O(u/B)$ I/Os to initialize an empty map, this map can support BATCH-FIND to search for up to B items by key, BATCH-INSERT to insert up to B items by key (reporting which were successful), and BATCH-REMOVE to remove up to B items (and report which were present) in $O(1)$ I/Os using parallel for each loops, as described in the previous section.

The BATCH-SET stores a set of keys from the universe as a BATCH-MAP with boolean values, supporting similar operations without item parameters to give a set interface with the same performance characteristics.

We also implement an efficient dynamic array, BATCH-ARRAY, that can provide PUSH and POP in amortized $O(1/B)$ I/Os. To do this, a BATCH-ARRAY contains a standard dynamic array (in external memory) plus up to B items in a separate buffer, logically at the end of the array. Usually, PUSH and POP can act only upon the buffer, costing zero I/Os as long as the it is cached in internal memory, until it becomes full or empty. At that point, the buffer is flushed to or refilled from the external array to contain $B/2$ items using $O(1)$ I/Os, whose cost amortizes to $O(1/B)$ I/Os per operation. This can also trigger a resize of the external array every $\Omega(m)$ operations, where m is its capacity, which takes $\Theta(m/B)$ I/Os to reallocate and move items, thus also amortizing to $O(1/B)$ I/Os per operation. For convenience, we also provide BATCH-PUSH to push multiple items at once. Full details can be found in Algorithm 9.

Theorem 7.2.2. *Dynamic arrays can support* PUSH *and* POP *in* $\Theta\left(\frac{1}{B}\right)$ *amortized I/Os alongside* $\Theta(1)$ *batch read and write operations on up to* B *locations.*

Algorithm 9 The BATCH-ARRAY Buffered Dynamic Array Data Structure

```
1: struct BATCH-ARRAY
2:     data : int[]                                   ▷ Items to stay in external memory
3:     buffer : int[]                                 ▷ Items to keep in cache
4: function NEW-BATCH-ARRAY(): BATCH-ARRAY
5:     return BATCH-ARRAY([], EMPTY-ARRAY-WITH-FIXED-CAPACITY(B))
6: function LENGTH(a: BATCH-ARRAY): int
7:     return length(data(a)) + length(buffer(a))
8: function BUFFER-RESET(a: BATCH-ARRAY): ()
9:     m ← ⌊B/2⌋                                      ▷ Ideal number of items in the buffer for amortization
10:    if length(buffer(a)) > m then
11:        d ← length(buffer(a)) − m
12:        EXTEND(data(a), TAKE(buffer(a), d))        ▷ Push d items out of buffer into data
13:        POP-FRONT-N(buffer(a), d)                  ▷ Remove those d items from buffer
14:    else
15:        d ← min(m − length(buffer(a)), length(data(a)))
16:        EXTEND-FRONT(buffer(a), TAKE-LAST(data(a), d)) ▷ Pull d items from data into buffer
17:        POP-BACK-N(data(a), d)                     ▷ Remove those d items from data
18: function PUSH(a: BATCH-ARRAY, x: int): ()
19:    if length(buffer(a)) = B then BUFFER-RESET(a)  ▷ Make room in buffer
20:    PUSH(buffer(a), x)                             ▷ Add x to the end of buffer
21: function BATCH-PUSH(a: BATCH-ARRAY, xs: int[]): ()
22:    for x in xs
23:        PUSH(a, x)                                 ▷ Push each item, mostly will only touch buffer
24: function POP(a: BATCH-ARRAY): int
25:    if length(buffer(a)) = 0 then BUFFER-RESET(a)  ▷ Bring items into buffer
26:    return POP(buffer(a))                          ▷ Add x to the end of buffer
27: function POP(a: BATCH-ARRAY): int
28:    if length(buffer(a)) = 0 then BUFFER-RESET(a)  ▷ Bring items into buffer
29:    return POP(buffer(a))                          ▷ Add x to the end of buffer
30: function DATA(a: BATCH-ARRAY): int[]
31:    EXTEND(data(a), buffer(a))                     ▷ Flush whole buffer into data
32:    length(buffer(a)) ← 0                          ▷ Clear buffer
33:    return data(a)
```

7.2.3 Joining Arrays

In GEM, we show how to scan a jagged 2D array, n arrays with m items total, in $O((n+m)/B)$ I/Os. Implementations are provided in Algorithm 10 and 11 of BATCH-JOINER, which produces batches of B items at a time as if each subarray were concatenated in order, as well as BATCH-ENUMERATE-JOINER, which concurrently outputs which subarray each item originated from. We assume that subarrays are given as "fat pointers": a slice of some underlying data specified by a front pointer and a length.

This improves upon the EM model, which has a worst-case where the subarrays are non-empty and not contiguous in external memory, thus necessarily requiring at least n I/Os to scan.

The straightforward EM algorithm of scanning each subarray in order thus achieves the optimal $\Theta(n + m/B)$ I/Os total to scan, which is worse than our result when $m = o(n \cdot B)$. This is a fundamental bottleneck in many EM algorithms, including graph algorithms on sparse graphs represented as adjacency lists.

BATCH-JOINER overcomes this bottleneck by exploiting parallelism to read multiple subarrays in one I/O. The algorithm scans the primary array in order, maintaining a working set in cache of at most $2B$ non-empty subarray slices of not-yet-output items. While this working set contains less than B slices, one I/O is used to read the next $B/2$ subarrays from the primary array and add the non-empty ones to the working set. Only when this working set collectively references at least B items (or the end of the primary array has been reached) will the items in those subarrays actually be read, and exactly which items to read next can be computed in internal memory using the front pointers and lengths of the slices in the working set. This strategy results in BATCH-JOINER taking a total of $O(n/B)$ I/Os to scan the primary array and $O(m/B)$ I/Os to scan the subarrays, giving the optimal speedup of $\Theta((n + m)/B)$ I/Os total.

Theorem 7.2.3. *We can join together n arrays into an array of m total items in $O((n + m)/B)$ I/Os.*

Algorithm 10 Scans an array of n arrays containing m items in $O((n + m)/B)$ I/Os

```
1:  function BATCH-JOINER(array: int[][]): ()
2:      curr ← []                                   ▷ Cached set of ≤ 2B non-empty subarrays
3:      curr-items ← 0                              ▷ Total number of items in curr
4:      while curr-items > 0 ∨ length(array) > 0
5:          if curr-items < B ∧ length(array) > 0 then
6:              next ← TAKE(array, B/2)             ▷ Read next block of slices
7:              POP-FRONT-N(array, B/2)             ▷ Advance front pointer into array by B/2
8:              for a in next
9:                  if length(a) = 0 then continue
10:                 PUSH(curr, a)                   ▷ Add up to B/2 non-empty subarrays
11:                 curr-items ← curr-items + length(a)   ▷ Each contributes at least 1 item
12:         else
13:             output ← PRODUCE-BLOCK-FROM(curr, min(B, curr-items))
14:             curr-items ← curr-items − length(output)
15:             yield output                        ▷ Generate a block of items for caller

16: function PRODUCE-BLOCK-FROM(curr: ref int[][], N: int): int[]
17:     k ← COUNT-UNTIL(s ⇒ s > N, CUMULATIVE-SUM([ length(a) | a ∈ cur ]))   ▷ Slices to take
18:     𝔭 ← JOINER([ [ ptr(a) + i | 0 ≤ i < length(a) ] | a ∈ TAKE(curr, k) ])   ▷ Compute pointers
19:     output ← *𝔭                                 ▷ Can read all in a single I/O
20:     POP-FRONT-N(curr, k)                        ▷ Advance front pointer by k
21:     if length(output) < N then                  ▷ Remaining items are at the start of the next slice
22:         r ← N − length(output)                  ▷ curr[0] must hold > r items
23:         EXTEND(output, TAKE(curr[0], r))
24:         POP-FRONT-N(curr[0], r)
25:     return output
```

Algorithm 11 Scan with BATCH-JOINER also including the indices of the items.

```
1: function BATCH-ENUMERATE-JOINER(array: int[][]): ()      ▷ Adds subarray indices of items
2:     joined ← BATCH-JOINER(array)
3:     indices ← ARRAY(EXPAND-FREQUENCIES([ length(a) | a ∈ array ]))
4:     for i, b in ZIP(CHUNKS(indices, B), joined)
5:         yield (i, b)                                  ▷ Generate pair of blocks (indices, items) for caller
```

7.3 Sorting

7.3.1 Indivisible Sorting

The classic result of [AV88] proved that sorting n indivisible items in the I/O model takes $\text{sort}(n) = \Theta\left(\frac{n \log(1+n/B)}{PB \log(1+M/B)}\right)$ I/Os, achieved using a variant of merge sort. The standard algorithm is improved to take advantage of parallelism while managing the contiguity of blocks; in particular, we observe that fixing the total bandwidth $P \cdot B$, the performance decreases when using fewer but larger contiguous blocks (larger B and smaller P). Consider the following trade-off:

Lemma 7.3.1. *If an algorithm in the I/O model runs in $f(P, B)$ I/Os, then for any integer $k \in [1, P]$ it can also be run in $f(\lfloor P/k \rfloor, B \cdot k)$ I/Os.*

Proof. We can simulate running the algorithm on an I/O machine capable of $P' = \lfloor P/k \rfloor$ parallel I/Os of blocks with size $B' = B \cdot k$. To do this, we read or write a simulated block of size B' by reading or writing a corresponding contiguous set of k actual blocks. Since our bandwidth is P actual blocks per actual I/O, we can thus read or write P' simulated blocks in parallel, thus perform one simulated I/O in one actual I/O, totaling in $f(P', B')$ I/Os. □

Assuming that $B < M < n$, so $0 < \log(1 + n/B) = \Theta(\log(n/B))$ and $0 < \log(1 + M/B) = \Theta(\log(M/B))$, Lemma 7.3.1 gives us a sorting algorithm that takes $O\left(\frac{n \log(n/(Bk))}{PB \log(M/(Bk))}\right)$ I/Os for any $k \in [1, P]$, which we show is an increasing function of k:

$$\begin{aligned}
M < n \text{ and } k' < k,& \\
\log(M/B)\log(k/k') &< \log(n/B)\log(k/k'), \\
-\log(n/B)\log k - \log(M/B)\log k' &< -\log(M/B)\log k - \log(n/B)\log k', \\
(\log(n/B) - \log k') \cdot (\log(M/B) - \log k) &< (\log(M/B) - \log k') \cdot (\log(n/B) - \log k), \\
\log(n/(Bk')) \cdot \log(M/(Bk)) &< \log(M/(Bk')) \cdot \log(n/(Bk)) \\
\frac{n \log(n/(Bk'))}{PB \log(M/(Bk'))} &< \frac{n \log(n/(Bk))}{PB \log(M/(Bk))}.
\end{aligned}$$

We see that the GEM regime of block size $B = 1$ gives the best trade-off by maximizing parallelism. In the GEM model, this sorting algorithm can be run in $O\left(\frac{n \log(1+n)}{B \log(1+M)}\right) = O\left(\frac{n}{B} \log_M n\right)$ I/Os, avoiding the cost of contiguity to get a base-M logarithmic factor.

7.3.2 Permutation

In the permutation problem, we are given n indivisible items with keys a permutation of $\{1, 2, \ldots, n\}$, and we must rearrange the items into sorted order. In GEM, this task is as easy as in the RAM model: we can create an output array of length n, and then scan the input in batches of B items

and writing them to their keyed output locations. Using a permutation to rearrange a second array of items is similarly easy; see Algorithm 12 for example implementations.

This gives a bound of $\Theta(n/B)$ I/Os to permute, which may appear to violate the bound expected from the I/O model, which is $\text{perm}(n) = \Omega(\min\{n/P, \text{sort}(n)\})$ by [AV88]. The difference comes from the fact that EM considers both P and B as asymptotically growing parameters, so the case that [AV88] shows requires $\Omega(\text{sort}(n))$ I/Os, when $\log n < B \log(M/B)$, can occur even if $n \geq M$. If we fix the granularity so $B = 1$, the GEM regime, then this case implies that $n < M$, thus we could achieve $O(n/P)$ I/Os here by simply permuting the input in internal memory.

Theorem 7.3.2. *In GEM, the permutation problem is solvable in $\Theta(n/B)$ I/Os.*

Algorithm 12 Permute an array of n items in $\Theta(n/B)$ I/Os

```
1: function PERMUTE(array: T[]): T[]
2:     output ← [ ⊥ | 0 ≤ i < length(array) ]
3:     for batch 𝔟 in array
4:         𝔦 ← [ key(x) | x ∈ 𝔟 ]
5:         output[𝔦] ← 𝔟
6:     return output
```

```
7: function INVERSE-PERMUTE(array: T[], π: int[]): T[]
8:     output ← [ ⊥ | 0 ≤ i < length(array) ]
9:     for batch 𝔦, 𝔧 in ENUMERATE(π)
10:        output[𝔦] ← array[𝔧]
11:    return output
```

7.3.3 Integer Sorting

In the case of integer sorting, the merge sort algorithm from the previous section is the best known, and when $P = 1$ it has been shown that, conditional on a well-known network coding conjecture, it remains optimal [FHLS20].

However, in GEM, these integer sorting lower bounds can be broken. First, we implement Counting Sort using only $O((n+u)/B)$ I/Os, where u is the maximum value in the input, presented in Algorithm 13. COUNTING-SORT is a standard sorting algorithm, whereas COUNTING-SORT-PERMUTATION produces an index permutation for key-based sorting. The latter gives us an implementation of Radix Sort, Algorithm 14, that uses $\Theta((n/B) \log_n u)$ I/Os, achieving optimal speedup over the RAM model.

Theorem 7.3.3. *In GEM, counting sort can be performed in $O((n + u)/B)$ I/Os.*

Theorem 7.3.4. *In GEM, radix sort can be performed in optimal $\Theta((n/B) \log_n u)$ I/Os.*

Algorithm 13 Sorts an array of n elements in the range $[0, u]$ using $O((n+u)/B)$ I/Os

```
function COUNTING-SORT(array: int[]): int[]
    COMPUTE-FREQUENCIES(array)
    return ARRAY(EXPAND-FREQUENCIES(freq))
function COUNTING-SORT-PERMUTATION(array: int[]): int[]
    COMPUTE-FREQUENCIES(array)
    return FREQUENCIES-TO-PERMUTATION(freq)
```

```
function COMPUTE-FREQUENCIES(array: int[]): int[]
    (n, u) ← (length(array), max(array))
    freq ← [ 0 | 0 ≤ i ≤ u ]
    for batch 𝔟 in array                                  ▷ Loop using O(n/B) I/Os
        𝔳 ← UNIQUE-SORT(𝔟)                                ▷ Locally gather unique items from block
        𝔣 ← [ COUNT(𝔟, x) | x ∈ 𝔳 ]                        ▷ Compute their frequencies within the block
        freq[𝔳] ← freq[𝔳] + 𝔣                              ▷ Add each to freq in bulk
    return freq
```

```
function EXPAND-FREQUENCIES(range: R): ()                 ▷ Loop using O(u/B) I/Os
    for batch 𝔦, 𝔣 in range
        for i, f in ZIP(𝔦, 𝔣)
            for _ in 0 ... f
                yield i
```

```
function FREQUENCIES-TO-PERMUTATION(array: int[], freq: int[]): int[]
    start ← CUMULATIVE-SUM(CHAIN([0],freq))               ▷ Start indices for each value's range
    π ← [ 0 | 0 ≤ i < length(array) ]                     ▷ π[i] = j will designate SORT(array)[i] ← array[j]
    for batch 𝔦, 𝔟 in array                               ▷ Loop using O(u/B) I/Os
        (𝔳, 𝔰, 𝔭) ← (UNIQUE-SORT(𝔟), start[𝔳], [])          ▷ Get unique items and current start indices
        for x in 𝔟
            j ← COUNT-UNTIL(𝔳, x)                         ▷ Find x's index in 𝔳
            PUSH(𝔭, 𝔰[j])                                 ▷ Assign this occurence x to its current start index
            𝔰[j] ← 𝔰[j] + 1                               ▷ Advance x's start index
        π[𝔭] ← 𝔦                                          ▷ Map sorted indices to input value indices
        start[𝔳] ← 𝔰                                      ▷ Write back the advanced start indices
    return π
```

Algorithm 14 Sorts an array of n elements in the range $[0, u]$ using $O((n/B)\log_n u)$ I/Os

```
1: function RADIX-SORT(array: int[]): int[]
2:     if length(array) = 0 then return
3:     (n, u) ← (length(array), max(array))
4:     for (m ← 1 ; m < u ; m ← m · n)
5:         column ← [ ⌊x/m⌋ mod n | x ∈ array ]
6:         π ← COUNTING-SORT-PERMUTATION(column)
7:         array ← INVERSE-PERMUTE(array, π)

8: function RADIX-SORT-PRESPLIT(array: int[d][]): int[d][]
9:     for i in 0, ..., d
10:        column ← [ x[i] | x ∈ array ]
11:        π ← COUNTING-SORT-PERMUTATION(column)
12:        array ← INVERSE-PERMUTE(array, π)
13:    return array
```

7.3.4 Duplicate Removal

An application of integer sorting is the ability to quickly remove duplicate elements from an array. Given a sorted array of n elements, we can copy its elements into a new duplicate-free array by pushing each element only if it differs from the last element. Given an array of n integers in the range $[0, u]$, we can thus use radix sort to remove duplicates in $O((n/B)\log_n u)$ I/Os. If we need to preserve the original order of items, we can attach the original index to each item, and at the end, counting sort the duplicate-free array by that index in $O(n)$ I/Os. This algorithm is called DUPLICATE-REMOVAL, detailed in Algorithm 15.

Corollary 7.3.5. *We can remove duplicates from an array as quickly as we can sort them, in particular in $O((n/B)\log_n u)$ for n integers in the range $[0, u]$.*

Another form of duplicate removal we will need in the context of graph algorithms is to remove the duplicates from each of n different arrays with a total of m elements. An efficient algorithm for this problem follows from combining BATCH-JOINER from Algorithm 10 with the single-array duplicate removal algorithm above. We label each element with the array it came from, join the n arrays together in $O((n+m)/B)$ I/Os by Theorem 7.2.3, and remove the duplicates from the joined array, where we view two elements as equal if their values are equal and they came from the same array. The elements will already be ordered by which array they came from; if we furthermore want to maintain the order within each array, we can use that variant of the single-array duplicate removal algorithm. We can then split up the resulting array into n subarrays if desired. This algorithm is called DUPLICATE-REMOVAL-MANY, detailed in Algorithm 16.

Corollary 7.3.6. *We can remove duplicates from n arrays with a total of m elements in $O(n/B)$ time plus the time to sort m elements, in particular in $O(\frac{n+m}{B}\log_n u)$ for integers in the range $[0, u]$.*

7.4 Union-Find

In GEM, we give two solutions to the union-find problem, where we must maintain a set of disjoint sets among n items and provide two operations: UNION(u, v), which unions the two sets that u and v are members of, and FIND(u), which returns a representative member of u's set. These

Algorithm 15 Duplicate removal on a non-negative integer array in $O((n+u)/B)$ I/Os

```
1: function BATCH-UNIQUE(array: int[]): int[]
2:     if length(array) = 0 then return array
3:     seen ← BATCH-SET(max(array) + 1)
4:     unique ← new BATCH-ARRAY
5:     for batch 𝔟 in array
6:         𝔟 ← UNIQUE-SORT(𝔟)                ▷ Remove duplicates in the block in cache
7:         𝔰 ← BATCH-INSERT(seen, 𝔟)          ▷ Try to insert each value into the seen set
8:         for i, v in 𝔟
9:             if 𝔰[i] then                   ▷ v was seen for the first time
10:                PUSH(unique, v)
11:    return DATA(unique)
```

solutions tackle the online data structure problem as well as the offline algorithmic problem where the sequence of operations is known in advance. In RAM, both problems were solved by [TvL84], giving a lower bound of $\Omega(n + m\alpha(m+n, n))$ time to perform m operations and matching data structures that support each operation in $O(\alpha(n))$ amortized time, with worst-case $O(\log n)$ time each, where α is the inverse Ackermann function defined as follows:

$$\begin{aligned} A(1,j) &= 2^j, \\ A(i,1) &= A(i-1,2), \\ A(i,j) &= A(i-1, A(i,j-1)), \end{aligned} \qquad \begin{aligned} \alpha(n,m) &= \min\{i \geq 1 \mid A(i, \lfloor m/n \rfloor) \geq \log_2 n\}, \\ \alpha(n) &= \alpha(n,n). \end{aligned}$$

7.4.1 Offline

In the offline union-find problem, we are given the sequence of m UNION(i,j) operations to perform, where all elements $\{1,\ldots,n\}$ are initially in singleton sets, and must output:

- for each operation UNION(i,j), whether or not it was redundant, i.e. whether i and j were already merged into the same set by previous operations; and
- for each element i, the unique representative $r =$ FIND(i) after all operations.

In EM, this problem was studied by [AAY10], who showed an $\Omega(\text{perm}(m))$ I/O lower bound, a $O(\text{sort}(m))$ I/O algorithm for a restricted variant (where the input contains no "redundant" UNION operations between two members of the same set), a general $O(\text{sort}(m) \log\log B)$ I/O algorithm, and a simple $O(\text{sort}(m) \log(m/M))$ I/O algorithm that performed well in practice.

We show how to implement their simpler algorithm in GEM taking $O(n/B + (m/B)\log(m/M))$ I/Os total, described in Algorithm 17. The strategy is divide-and-conquer on the sequence of operations, where each subproblem is a slice of the sequence and the goal is to output the representatives for each item referenced in that slice. To solve a subproblem that doesn't fit in internal memory, we first recurse on the first half of the slice, update the arguments in the second half of the slice so every item is a representative, recurse on the second half, then combine the two outputs into one map of items to their representatives; in EM, this takes $O(\text{sort}(m))$ I/Os per subproblem due to the limitations of contiguity.

In GEM, we can rename items before recursing to ensure that the item labels are always linearly bounded by the subproblem size m, making it space- and I/O-efficient to use arrays for storing the representative map and performing updates in batches, resulting in $O(m/B)$ I/Os per subproblem.

Algorithm 16 Duplicate removal of n arrays with m items total in $O((n+m+u)/B)$ I/Os

```
1: function DUPLICATE-REMOVAL-MANY(array: int[]): int[]
2:     if ∀a ∈ array. length(a) = 0 then return array          ▷ Trivial if entirely empty input
3:     u ← max([ max(a) | a ∈ BATCH-JOINER(array) ])
4:     seen ← new BATCH-SET(u + 1)      ▷ Remember items already seen in unfinished subarray
5:     rem ← 0                          ▷ Tracks number of items remaining in unfinished subarray
6:     (output-lengths, output-values) ← ([ 0 | 0 ≤ i < n ], new BATCH-ARRAY)
7:     for i, b in BATCH-ENUMERATE-JOINER(array)
8:         orig-gs ← GROUP-BY-KEY(ZIP(i, b))                   ▷ Group items by origin subarray
9:         uniq-gs ← [ UNIQUE-SORT(x) | (_, x) ∈ orig-gs ]     ▷ Uniq each group
10:        insert-0 ← BATCH-INSERT(seen, uniq-gs[0])           ▷ Mark first group (may be partial)
11:        uniq-gs[0] ← [ x | (x, s) ∈ ZIP(uniq-gs[0], insert-0), s = true ]   ▷ Remove already seen
12:        for uc in uniq-cs
13:            BATCH-PUSH(output-values, uc)                   ▷ Output new unique items
14:        (j, l) ← ([ FRONT(g)[0] | g ∈ orig-gs], [ length(x) | x ∈ uniq-gs ])
15:        output-lengths[j] ← output-lengths[j] + l           ▷ Count number of unique items seen
16:        if length(uniq-gs) > 1 then                         ▷ Block overlaps multiple subarrays
17:            CLEAR(seen)                                     ▷ Finished the first subarray
18:            rem ← length(array[BACK(j)])                    ▷ Get length of last subarray
19:            rem ← rem − length(BACK(orig-gs))               ▷ Subtract items seen in this block
20:            if rem > 0 then                                 ▷ Last subarray isn't finished
21:                BATCH-INSERT(seen, BACK(uniq-gs))           ▷ Remember items for next iteration
22:        else                                                ▷ Block is within one subarray
23:            if rem = 0 then rem ← length(array[BACK(j)])    ▷ If new subarray, get its length
24:            rem ← rem − length(BACK(org-gs))                ▷ Subtract items seen in this block
25:            if rem = 0 then CLEAR(seen)                     ▷ Clear if finished the subarray
26:    return CONVERT-DATA-LENGTH-TO-SLICES(DATA(output-values), output-lengths)

27: function CONVERT-DATA-LENGTH-TO-SLICES(data: int[], lengths: int[]): int[][]
28:     array ← new BATCH-ARRAY
29:     pos ← 0
30:     for batch l in lengths
31:         for ℓ in l
32:             PUSH(array, data[pos ... pos + ℓ])          ▷ Create "fat pointer" without copying
33:             pos ← pos + ℓ
34:     return DATA(array)
```

At the top-level, we also incur $O(n/B)$ I/Os to perform an initial renaming in case $n > m$. Both of these renamings are done using UNION-OPS-COMPRESS, which efficiently scans its input in batches and numbers the unique items it finds in the order they appear.

Lemma 7.4.1. *Offline Union-Find can be solved in $O((m/B)\log(m/M))$ I/Os given that $n = O(m)$.*

Theorem 7.4.2. *Offline Union-Find can be solved in $O(n/B + (m/B)\log(m/M))$ I/Os.*

Algorithm 17 Offline Union-Find in $O((m/B)\log(m/M))$ I/Os, given that $n = O(m)$.

```
function OFFLINE-UNION-FIND(operations: int[2][], size: int): (int[],bool[])
    redundant ← new BATCH-ARRAY
    function REC(ops: int[2][], n: int): int[]
        if problem fits into internal memory, i.e. n/2 ≤ LENGTH(ops) = O(M) then
            ds ← DISJOINT-SET(n)                                  ▷ Create a disjoint set in cache
            for (i, j) in ops
                success ← UNION(ds, i, j)
                PUSH(redundant, ¬ success)
            return [ FIND(ds, i) | 0 ≤ i < n ]
        else
            (left-ops, right-ops) ← (TAKE(ops, LENGTH(ops)/2), DROP(ops, LENGTH(ops)/2))
            (loc-ops, loc-index, loc-orig, loc-size) ← UNION-OPS-COMPRESS(left-ops, n)
            loc-next ← REC(loc-ops, loc-size)
            for batch 𝔯 in right-ops
                parallel for ref i in JOINER(𝔯)
                    if m_i ← loc-index[i] is not ⊥ then i ← loc-orig[loc-next[m_i]]
            (roc-ops, roc-index, roc-orig, roc-size) ← UNION-OPS-COMPRESS(right-ops, n)
            roc-next ← REC(roc-ops, roc-size)
            next ← [ i | 0 ≤ i < n ]
            parallel for (m_i, i) in roc-orig
                next[i] ← roc-orig[roc-next[m_i]]
            parallel for (m_i, i) in loc-orig
                if roc-index[i] is not ⊥ then continue
                r ← loc-orig[loc-next[m_i]]
                next[i] ← next[r]
            return next
    reps ← REC(operations, size)
    return (reps, DATA(redundant))

function UNION-OPS-COMPRESS(operations: int[2][], n: int): (int[2][],int[],int[],int)
    (flat-ops, index, orig) ← (new BATCH-ARRAY, [ ⊥ | 0 ≤ i < n ], new BATCH-ARRAY)
    for batch 𝔳 in JOINER(operations)                  ▷ Iterate over batches of flattened pairs
        𝔪 ← index[𝔳]
        𝔥 ← {i ↦ m_i | (i, m_i) ∈ ZIP(𝔳, 𝔪), m_i ≠ ⊥}                ▷ Quick lookup of mappings
        for (i, ref m_i) in ZIP(𝔳, 𝔪)                  ▷ Iterate over both batches in lockstep
            if i ∈ 𝔥 then
                m_i ← 𝔥[i]
            else
                m_i ← LENGTH(orig)
                PUSH(orig, i)
                𝔥[x] ← m_i
        index[𝔳] ← 𝔪
        BATCH-PUSH(flat-ops, 𝔪)
    return (CHUNKS(flat-ops, 2), index, orig, LENGTH(orig)) ▷ Return unflattened edge list and
    decompression information
```

7.4.2 Online Batched

For the online union-find problem in the EM model, there is no known improvement over the RAM data structures of [TvL84]. However, in PRAM, [STTW16] studied this problem in the "discretized streams" input model, where we replace individual operations with BATCH-UNION or BATCH-FIND, and give a work-efficient algorithm with total work $O((m+q)\alpha(m+q,n))$, where m is the total number of UNION calls, q is the total number of FIND calls, and each call takes worst-case $O(\text{polylog}(m,n))$ rounds.

In GEM, we adopt the model of [STTW16] and present BATCH-DISJOINT-SET in Algorithms 18 and 19, which supports batches of up to B operations in $O(\log n)$ I/Os. This data structure optimizes the RAM strategy of maintaining a tree over items, each with a single next pointer and subtree size, to take advantage of parallelism and internal memory. BATCH-FIND can simply run each individual FIND in lockstep, where we can have the additional capability to jump ahead between I/Os whenever our current working set of items in internal memory contains both an item and its next item, using the BATCH-FIND-READ-COMPRESSED function.

On the other hand, BATCH-UNION cannot just independently perform UNION in parallel since these operations mutate the tree and may not take effect in an order consistent with a one-at-a-time execution. To overcome this, after using BATCH-FIND to identify the at-most-B pairs of representatives to be physically linked, we can solve the problem of what unions to perform in internal memory then actually write the new next pointers and size values using just 2 I/Os before returning which UNION calls were successful and which were redundant.

By using the union-by-size strategy, it is known that the longest path in such a tree is $O(\log n)$

Algorithm 18 BATCH-DISJOINT-SET Data Structure: NEW and FIND methods

```
1: struct BATCH-DISJOINT-SET
2:     next, size : int[]

3: function NEW-BATCH-DISJOINT-SET(n: int): BATCH-DISJOINT-SET
4:     return BATCH-DISJOINT-SET([ i | 0 ≤ i < n ], [ 1 | 0 ≤ i < n ])

5: function BATCH-FIND((next, _): BATCH-DISJOINT-SET, 𝔟: int[]): int[]
6:     𝔫 ← BATCH-FIND-READ-COMPRESSED(next, 𝔟)
7:     while 𝔟 ≠ 𝔫
8:         𝔫₂ ← BATCH-FIND-READ-COMPRESSED(next, 𝔫)
9:         next[𝔟] ← 𝔫₂                ▷ Path compression via splitting (plus the local compression)
10:        (𝔟, 𝔫) ← (𝔫, 𝔫₂)
11:    return 𝔟

12: function BATCH-FIND-READ-COMPRESSED(next: int[], 𝔟: int[]): int[]
13:    𝔫 ← next[𝔟]                    ▷ Read 𝔟's next pointers (the only I/O)
14:    𝔥 ← {x ↦ n | next[x] = n}      ▷ Convert pointer data into a local map
15:    for b in KEYS(𝔥)                ▷ For each unique b ∈ 𝔟
16:        (x, n) ← (b, 𝔥[b])          ▷ Sequentially run FIND(b) as far as possible within 𝔥
17:        while x ≠ n ∧ n₂ ← 𝔥[n]
18:            (x, n) ← (n, n₂)
19:        𝔥[b] ← n                    ▷ Save result (an actual root or item outside 𝔥)
20:    return 𝔥[𝔟]                     ▷ Return compressed next pointers as an array
```

steps, which gives us a worst-case bound of $O(\log n)$ I/Os for both BATCH-FIND and BATCH-UNION for batch sizes up to B operations. While BATCH-DISJOINT-SET uses path compression, the techniques of [TvL84] which showed $O(\alpha(n))$ amortized time in the RAM model do not directly apply to give a tighter upper bound. If $\ell(v)$ is the length of the path from v to its representative FIND(v), a BATCH-FIND$(\mathfrak{b})$ operation will take $O(\max\{\ell(v) \mid v \in \mathfrak{b}\})$ I/Os, so when path lengths are not proportional, the throughput is penalized compared to the optimal speedup of B over to the total work. It remains open how this penalty affects the amortized performance of BATCH-DISJOINT-SET.

Theorem 7.4.3. *On a* BATCH-DISJOINT-SET *with n items,* BATCH-FIND *or* BATCH-UNION *of batch size at most B takes $O(\log(n))$ I/Os.*

Algorithm 19 BATCH-DISJOINT-SET Data Structure: UNION methods

```
1:  function BATCH-UNION(ds: BATCH-DISJOINT-SET, 𝔲: int[], 𝔳: int[]): bool[]
2:      (𝔲, 𝔳) ← (BATCH-FIND(ds, 𝔲), BATCH-FIND(ds, 𝔳))        ▷ Advance items to representatives
3:      𝔯 ← UNIQUE-SORT(CHAIN(𝔲, 𝔳))          ▷ Create a local disjoint set over all representatives
4:      𝔫₂ ← {u ↦ u | u ∈ 𝔯}                          ▷ Initial next pointers: each rep is a singleton
5:      𝔰₂ ← {u ↦ s_u | (u, s_u) ∈ ZIP(𝔯, size(ds)[𝔯])}          ▷ Initial sizes are of the original sets
6:      𝔟 ← []
7:      for u, v in ZIP(𝔲, 𝔳)              ▷ Iterate in lockstep over pairs to run union-find in cache
8:          PUSH(𝔟, DISJOINT-SET-UNION(𝔫₂, 𝔰₂, u, v))
9:      next[𝔯] ← [ DISJOINT-SET-FIND(𝔫₂, x) | x ∈ 𝔯 ]      ▷ Write new nexts to external memory
10:     size[𝔯] ← 𝔰₂[𝔯]                                    ▷ Write new sizes to external memory
11:     return 𝔟

12: function DISJOINT-SET-UNION(next: int[int], size: int[int], u: int, v: int): bool
13:     (u, v) ← (DISJOINT-SET-FIND(next, u), DISJOINT-SET-FIND(next, v))  ▷ Find representatives
14:     if u = v then return false                          ▷ Fail if already in the same set
15:     if size[u] < size[v] then (u, v) ← (v, u)           ▷ Let u represent the larger set
16:     next[v] ← u                                                        ▷ Point u to u
17:     size[u] ← size[u] + size[v]                         ▷ Add v's set's size to u's
18:     return true

19: function DISJOINT-SET-FIND(next: int[int], x: int): int[]
20:     n ← next[x]
21:     while x ≠ n
22:         n₂ ← next[n]
23:         next[x] ← n₂                                    ▷ Path compression via splitting
24:         (x, n) ← (n, n₂)
25:     return x
```

7.4.3 Online with Supersized Batches

In this section, we consider how a union-find data structure can handle batch sizes larger than B. We show that we can improve upon the $O(\log n)$ I/O bound from the previous section.

Theorem 7.4.4. *On a* BATCH-DISJOINT-SET *with n items,* FIND *operations in batches of size $k \geq B(\log n)/\alpha(n)$ can be served in $O(k\alpha(n)/B)$ I/Os, thus in amortized $O(\alpha(n)/B)$ I/Os per item.*

Proof. We simulate a PRAM-like algorithm, imagining B processors that each take a FIND operation out of the batch to perform, including compression. However, if multiple processors ever simultaneously reach the same item, which we can detect in internal memory while simulating, then to avoid duplicating the work going forward, all but one of them is stopped and given the next FIND operation in the batch to start instead. Once a processor reaches a representative, it also moves onto the next operation in the batch. While processors are traversing the tree, they add the items they visit in order to a log (a BATCH-ARRAY, giving $O(1/B)$ I/Os per append).

At the point where all searches in the batch have been completed, the algorithm will play the log backwards to propagate the representative identities down from each item's parent, eventually reaching the queried items in the batch. This is done in batches: we pop the last B logged items, read their parents from memory and their next pointers, internally propagate the next pointer values down the tree among the items just popped, and then write the updated next pointers of the items we popped back to memory. Assuming by induction that each item in earlier popped batches have their next pointer as their representative, since we appended to the log in topological order, this will correctly propagate in reverse topological order, and thus at the end, every item in the query batch will now point to their representative, which we can now read and return in $O(k/B)$ I/Os.

Next, we prove the claimed I/O bound. During the first phase of the algorithm, as long as the query batch has not been exhausted, the B processes will take steps of $O(1)$ I/Os to in parallel either traverse an edge or skip from a representative or collision partway up the tree to the next item in the query batch. Each of these units of work is done perfectly in parallel, and consists of either work done by the sequential algorithm, so $O(k\alpha(n))$ work, or work that happens once per query item, so $O(k)$ additional work. Once the query batch is exhausted but until the remaining at-most-B searches are completed, the worst case number of I/Os is $O(\log n)$, the maximum height of a tree. The second phase of the algorithm takes $O(1)$ I/Os per B items popped from the log, so the first phase dominates. Overall, we see that the perfectly parallel work dominates when $k = \Omega(B(\log n)/\alpha(n))$, as we can bound the total number of I/Os by:

$$O\left(\frac{k\alpha(n)+k}{B} + \log n\right) = O\left(\frac{k\alpha(n)}{B}\right)$$

□

Theorem 7.4.5. *On a* BATCH-DISJOINT-SET *with n items, a mix of* UNION *and* FIND *operations in batches of size $k \geq B \log n$ can be served in $O(k\alpha(n)/B)$ I/Os, thus in amortized $O(\alpha(n)/B)$ I/Os per item, for an internal memory size M which is $\Omega(k)$.*

Proof. First, we treat each UNION(i, j) as the pair FIND(i) and FIND(j) and run the algorithm of the previous Theorem 7.4.4 to find the representatives of every item present in the query, which takes $O(k\alpha(n)/B)$ I/Os since there are at most $2k$ items.

Next, we can take the query items plus their representatives, up to $4k$ unique items, and their sizes, and simulate the batch of operations within internal memory like in BATCH-UNION. Once all the responses are collected, we can write back the changes to next pointers and sizes back to the

external memory data structure and return the responses. Given that M is large enough to store and operate on the internal data structure over up to $4k$ items, this step will only require $O(k)$ I/Os, thus the first step dominates. □

7.5 Hashing

In this section, we analyze GEM hash tables as a ubiquitous and performant class of data structures for implementing unordered dictionaries with INSERT, DELETE, and FIND operations. In EM, there are strong lower bounds [IP12]: for any EM dictionary that supports INSERT in $O(\lambda/B)$ amortized I/Os, FIND requires $O(\Omega(\log_\lambda n))$ expected I/Os. There are also practical EM upper bounds that match these lower bounds for $\lambda = \Omega(\log\log M + \log_M N)$, even in the cache-oblivious model [CFCS18].

7.5.1 Open Addressing

Open addressing is a widely used hash-table strategy in which items are stored directly in a table without indirection. To find an item, we hash its key to determine the beginning of a probe sequence, indices of the table to scan in order until successfully finding the item or reaching a "terminator" slot that is empty or otherwise signals that the item is not present. Recent EM results give a linear-probing open addressed dictionary that supports individual operations in $1 + O(x/B)$ amortized expected I/Os when the load factor is maintained at $1 - 1/\Theta(x)$ for some $x \leq B$ [BKK21].

In the GEM model, we consider the potential benefits of batched operations, such as BATCH-FIND to query up to B items at once. If we exploring each individual probe in parallel, we would have the same penalty as in the previous section on union-find where the performance is bounded by the longest probe sequence rather than the total number of slots to scan. However, unlike when chasing pointers, in one I/O we can both read from multiple probe sequences in parallel and read multiple slots in the same probe sequence. Without any knowledge of the length of each probe sequence up to its terminator, a natural strategy arises:

Definition 7.5.1. Given $k \leq B$ probe sequences of unknown length to scan, the ***Balanced Parallel Search*** strategy does the following:

1. Read $\lfloor B/k \rfloor$ words from each probe sequence, using one I/O.
2. Identify the k' probe sequences where the terminator was not read.
3. If $k' > 0$, recursively continue Balanced Parallel Search on the rest of those k' probe sequences.

Unfortunately, in the pursuit of implementing an efficient GEM open-addressed hash table, we will prove that this strategy does not give optimal speedup over the work of performing the equivalent sequential searches in the RAM model. Specifically, we will do this by measuring the worst-case ***throughput***: the number of ***required*** words (words in a probe sequence up to and including the terminator) read per I/O during the search as a function of the ***distribution*** of words (the multiset of the lengths of each probe sequence).

First, we relate the throughput with the following function $R_{\mathbb{N}} : \mathbb{N} \times \mathbb{N}^+ \to \mathbb{N}$.

Definition 7.5.2. For integers $k \geq 1$, $j \geq 0$, let

$$R_{\mathbb{N}}(j,k) = \begin{cases} \min\limits_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{N}}(j-1,k') \right) & \text{if } j > 0, \\ 0 & \text{if } j = 0. \end{cases}$$

Lemma 7.5.1. *Given $k \in [1, B]$ probe sequences, a minimum throughput distribution where Balanced Parallel Search take h I/Os must scan L required words such that*

$$\frac{L-k}{B} = \Theta\left(R_{\mathbb{N}}(h-1,k)\right).$$

Proof. For an execution of Balanced Parallel Search, let k_i be the number of probe sequences remaining in iteration $i \in [1, h+1]$, meaning $B \geq k = k_1 \geq k_2 \geq \cdots \geq k_h > k_{h+1} = 0$.

In any iteration i where a probe sequence ends, the throughput is minimized when its terminator was the only required word read from that probe sequence, thus the remaining $\lfloor B/k_i \rfloor - 1$ words were wasted reads past the end. This tell us a minimum throughput distribution must satisfy

$$L = \sum_{i=1}^{h} \left(k_i + k_{i+1} \cdot (\lfloor B/k_i \rfloor - 1)\right) = k_1 + \sum_{i=1}^{h} \left(k_{i+1} \lfloor B/k_i \rfloor\right) = k + B \cdot \Theta\left(\sum_{i=1}^{h} \frac{k_{i+1}}{k_i}\right).$$

For any fixed h, we can thus characterize the minimum throughput distribution that takes h I/Os satisfying these constraints by minimizing L over all values of k_i. By a rearrangement of the above formula for L and a simple inductive argument, we get the desired result:

$$\frac{L-k}{B} = \Theta\left(\min_{k=k_1 \geq k_2 \geq \cdots \geq k_h > 0} \sum_{i=1}^{h} \frac{k_{i+1}}{k_i}\right) = \Theta\left(R_{\mathbb{N}}(h-1,k)\right).$$

□

To understand $R_{\mathbb{N}}(j,k)$, we define the following function $R_{\mathbb{R}} : \mathbb{N} \times \mathbb{R}^{\geq 1} \to \mathbb{N}$ that generalizes the k parameter to the reals. We will then derive an exact formula for $R_{\mathbb{R}}(j,k)$ and show that $R_{\mathbb{N}}(j,k) = \Theta(R_{\mathbb{R}}(j,k))$.

Definition 7.5.3. For real $k \geq 1$ and integer $j \geq 0$, let

$$R_{\mathbb{R}}(j,k) = \begin{cases} \min\limits_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{R}}(j-1,k')\right) & \text{if } j > 0, \\ 0 & \text{if } j = 0. \end{cases}$$

Lemma 7.5.2. *For all integers $k \geq 1$, $j \geq 0$, we have $R_{\mathbb{R}}(j,k) \leq R_{\mathbb{N}}(j,k)$.*

Lemma 7.5.3. *$R_{\mathbb{R}}(j,k) = j/k^{1/j}$ for $j, k \geq 1$.*

Proof. By induction on j:

- $R_{\mathbb{R}}(1,k) = \min\limits_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{R}}(0,k')\right) = \min\limits_{k \geq k' \geq 1} \frac{k'}{k} = \frac{1}{k}$.
- $R_{\mathbb{R}}(2,k) = \min\limits_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{R}}(1,k')\right) = \min\limits_{k \geq k' \geq 1} \left(\frac{k'}{k} + \frac{1}{k'}\right)$.

  This is minimized when $0 = \frac{d}{dk'}\left(\frac{k'}{k} + \frac{1}{k'}\right) = \frac{1}{k} - \frac{1}{k'^2}$, so when $k' = \sqrt{k}$, thus we have $R_{\mathbb{R}}(2,k) = \frac{\sqrt{k}}{k} + \frac{1}{\sqrt{k}} = \frac{2}{\sqrt{k}}$.
- $R_{\mathbb{R}}(3,k) = \min\limits_{k > k' \geq 1} \left(\frac{k'}{k} + R_{\mathbb{R}}(2,k')\right) = \min\limits_{k > k' \geq 1} \left(\frac{k'}{k} + \frac{2}{\sqrt{k'}}\right)$.

  This is minimized when $0 = \frac{d}{dk'}\left(\frac{k'}{k} + \frac{2}{\sqrt{k}}\right) = \frac{1}{k} - \frac{1}{k'^{3/2}}$ so when $k' = k^{2/3}$, thus we see that $R_{\mathbb{R}}(3,k) = \frac{k^{2/3}}{k} + \frac{2}{\sqrt{k^{2/3}}} = \frac{3}{k^{1/3}}$.

- Assume for induction that $R_{\mathbb{R}}(j,k') = j/(k')^{1/j}$ for some $j \geq 3$ and all $k' \geq 1$. Then

$$R_{\mathbb{R}}(j+1,k) = \min_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{R}}(j,k') \right) = \min_{k \geq k' \geq 1} \left(\frac{k'}{k} + \frac{j}{(k')^{1/j}} \right)$$

which is minimized when

$$0 = \frac{d}{dk'} \left(\frac{k'}{k} + \frac{j}{(k')^{1/j}} \right) = \frac{d}{dk'} \frac{k'}{k} + \frac{d}{dk'} \frac{j}{(k')^{1/j}} = \frac{1}{k} - \frac{1}{(k')^{1+1/j}} \implies k' = k^{j/(j+1)}.$$

Thus we get

$$R_{\mathbb{R}}(j+1,k) = \frac{k^{j/(j+1)}}{k} + \frac{j}{\left(k^{j/(j+1)}\right)^{1/j}} = \frac{1}{k^{1/(j+1)}} + \frac{j}{k^{1/(j+1)}} = \frac{j+1}{k^{1/(j+1)}}.$$

□

Lemma 7.5.4. *$R_{\mathbb{N}}(j,k) \leq 2 \cdot j/k^{1/j}$ for integers $k, j \geq 1$.*

Proof. For $k = 1$, we have $R_{\mathbb{N}}(j,1) = 1 + R_{\mathbb{N}}(j-1,1)$, thus by a simple inductive argument this collapses to $R_{\mathbb{N}}(j,1) = j \leq 2 \cdot j/1^{1/j}$.

For $k = 2$, we now have two possibilities, $k' = 1$ (the previous case) and $k' = 2$, so we get $R_{\mathbb{N}}(j,2) = \min\left(\frac{1}{2} + R_{\mathbb{N}}(j-1,1), 1 + R_{\mathbb{N}}(j-1,2)\right)$. Again, by induction we see that the terms produced in the recursion will be $j-1$ ones and one $1/2$, thus $R_{\mathbb{N}}(j,2) = j - 1/2 < j \leq 2 \cdot j/2^{1/j}$.

For $k \geq 3$, we proceed by induction on j:

- $R_{\mathbb{N}}(1,k) = \min_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{N}}(0,k') \right) = \min_{k \geq k' \geq 1} \frac{k'}{k} = \frac{1}{k} \leq \frac{2}{k}$.

- $R_{\mathbb{N}}(2,k) = \min_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{N}}(1,k') \right) = \min_{k \geq k' \geq 1} \left(\frac{k'}{k} + \frac{1}{k'} \right)$.

  If $k = 3 > 2$, then the minimum is achieved when $k' = 2$ since $\frac{7}{6} = \frac{2}{3} + \frac{1}{2} < \frac{1}{3} + \frac{1}{1} = \frac{4}{3}$, so we get that $R_{\mathbb{N}}(2,3) = \frac{7}{6} \leq 2 \cdot 2/3^{1/2}$.

  If $k \geq 4$, it is minimized when $\frac{d}{dk'} \left(\frac{k'}{k} + \frac{1}{k'} \right) = \frac{1}{k} - \frac{1}{k'^2}$ is closest to zero, so when $k' = \lceil \sqrt{k} \rceil$ or $k' = \lfloor \sqrt{k} \rfloor$. If we express $k' = \varepsilon + \sqrt{k}$ for some offset $\varepsilon \in (-1,1)$ such that $k' \in \mathbb{N}$, we find the following constraints: $k' \geq 2$, $k' < 1 + \sqrt{k} < 2\sqrt{k}$, and $\varepsilon/\sqrt{k} \leq \varepsilon/2 < 1/2$.

  Substituting, we get $R_{\mathbb{N}}(2,k) = \frac{\varepsilon + \sqrt{k}}{k} + \frac{1}{\varepsilon + \sqrt{k}} = \frac{(\varepsilon+\sqrt{k})^2 + k}{k(\varepsilon+\sqrt{k})} = \frac{\varepsilon^2 + 2\varepsilon\sqrt{k} + 2k}{\varepsilon\sqrt{k} + k} \cdot \frac{1}{\sqrt{k}}$. and we can bound the left factor using our constraints as follows:

$$\frac{\varepsilon^2 + 2\varepsilon\sqrt{k} + 2k}{\varepsilon\sqrt{k} + k} = \frac{\varepsilon^2}{\varepsilon\sqrt{k} + k} + 2 \leq \frac{1}{k - \sqrt{k}} + 2 \leq \frac{1}{2} + 2 \leq 2 \cdot 2.$$

  Therefore $R_{\mathbb{N}}(2,k) \leq 2 \cdot 2/k^{1/2}$.

- Assume for induction that $R_{\mathbb{N}}(j,k') = 2 \cdot j/(k')^{1/j}$ for some $j \geq 2$ and all $k' \geq 1$. Then

$$R_{\mathbb{N}}(j+1,k) = \min_{k \geq k'} \left(\frac{k'}{k} + R_{\mathbb{N}}(j,k') \right) = \min_{k \geq k' \geq 1} \left(\frac{k'}{k} + \frac{2 \cdot j}{(k')^{1/j}} \right)$$

We will show that this upper-bound is minimized where the derivative of the expression is nearest zero, thus when

$$0 \approx \frac{d}{dk'}\left(\frac{k'}{k} + \frac{2 \cdot j}{(k')^{1/j}}\right) = \frac{d}{dk'}\frac{k'}{k} + \frac{d}{dk'}\frac{2 \cdot j}{(k')^{1/j}} = \frac{1}{k} - \frac{2}{(k')^{1+1/j}} \implies k' \approx (2k)^{j/(j+1)}.$$

Since $k \geq 3$ and $j \geq 2$, we have $k' \geq \left\lfloor (2k)^{j/(j+1)} \right\rfloor \geq \left\lfloor (2 \cdot 3)^{2/3} \right\rfloor \geq 1$, as required.

Next, like before, we express integer $k' = \varepsilon + (2k)^{j/(j+1)}$ for some offset $\varepsilon \in (-1, 1)$ and substitute it into our upper bound:

$$\begin{aligned} R_{\mathbb{N}}(j+1,k) &\leq \frac{\varepsilon + (2k)^{j/(j+1)}}{k} + \frac{2 \cdot j}{(\varepsilon + (2k)^{j/(j+1)})^{1/j}} \\ &= \frac{2}{k^{1/(j+1)}}\left(\frac{k^{1/(j+1)}}{2} \cdot \frac{\varepsilon + (2k)^{j/(j+1)}}{k} + \frac{k^{1/(j+1)}}{2} \cdot \frac{2 \cdot j}{(\varepsilon + (2k)^{j/(j+1)})^{1/j}}\right) \\ &= \frac{2}{k^{1/(j+1)}}\left(\frac{\varepsilon}{2 \cdot k^{j/(j+1)}} + \frac{1}{2^{1/(j+1)}} + \frac{k^{1/(j+1)}}{(\varepsilon + (2k)^{j/(j+1)})^{1/j}} \cdot j\right). \end{aligned}$$

Again using $k \geq 3$ and $j \geq 2$, we can bound each term in the parentheses as follows:

$$\frac{\varepsilon}{2 \cdot k^{j/(j+1)}} \leq \frac{1}{2 \cdot 3^{2/3}} < \frac{1}{4} \qquad 1 \leq -1/3 + 2^{2/3} \leq -1/k^{j/(j+1)} + 2^{j/(j+1)}$$

$$k^{j/(j+1)} \leq -1 + (2k)^{j/(j+1)} \leq \varepsilon + (2k)^{j/(j+1)}$$

$$\frac{1}{2^{1/(j+1)}} < \frac{1}{2} \qquad \frac{k^{1/(j+1)}}{(\varepsilon + (2k)^{j/(j+1)})^{1/j}} \leq 1$$

Therefore we can successfully conclude that

$$R_{\mathbb{N}}(j+1,k) \leq \frac{2}{k^{1/(j+1)}}\left(\frac{1}{4} + \frac{1}{2} + 1 \cdot j\right) \leq 2 \cdot (j+1)/k^{1/(j+1)}.$$

□

Combining Lemma 7.5.2 and Lemma 7.5.4, we get a tight bound:

Corollary 7.5.5. $R_{\mathbb{N}}(j,k) = \Theta(j/k^{1/j})$.

For the final result of this section, we will need the Lambert W function, whose definition and useful asymptotic properties are presented below.

Definition 7.5.4. Let $W : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ be the principle branch of the Lambert W function, satisfying

$$W(x) = w \iff we^w = x.$$

Lemma 7.5.6. *For $a \geq 1$, $W(x) \leq W(ax) \leq aW(x)$ and thus $W(x)/a \leq W(x/a) \leq W(x)$.*

Proof. $W(x)$ is a monotonically increasing function, so $W(x/a) \leq W(x) \leq W(ax)$. Define x' such that

$$W(ax) = w \implies we^w = ax \implies x = \frac{w}{a}e^w \geq \frac{w}{a}e^{w/a} = x'.$$

Plugging x' into W, we have

$$W(x') = w/a \implies W(ax) = w = aW(x') \leq aW(x),$$

from which the remaining inequalities follow. □

Lemma 7.5.7. *If $f(x) = \Theta(g(x))$ and $h(x) = \Theta(W(f(x))$ then $h(x) = \Theta(W(g(x)))$.*

Proof. Take any x such that $a \cdot g(x) \leq f(x) \leq b \cdot g(x)$ and $c \cdot W(f(x)) \leq h(x) \leq d \cdot W(f(x))$ for constants $a, b, c, d > 0$. We analyze the following cases using Lemma 7.5.6 and monotonicity:

- If $a \geq 1$ then $h(x) \geq c \cdot W(f(x)) \geq c \cdot W(f(x)/a) \geq c \cdot W(g(x))$.
- If $a < 1$ then $h(x) \geq c \cdot W(f(x)) \geq c \cdot W(a \cdot g(x)) \geq (ac) \cdot W(g(c))$.
- If $b \geq 1$ then $h(x) \leq d \cdot W(f(x)) \leq d \cdot W(b \cdot g(x)) \leq (bd) \cdot W(g(x))$.
- If $b < 1$ then $h(x) \leq d \cdot W(f(x)) \leq d \cdot W(f(x)/b) \leq d \cdot W(g(x))$.

Thus, with constant factors $[c \cdot \min(1, a), d \cdot \max(1, b)]$, we get that $h(x) = \Theta(W(g(x)))$. □

We now have everything we need to bound the number of I/Os that Balanced Parallel Search takes to scan more than one probe sequence, as a function of the number of required words to scan.

Lemma 7.5.8. *Given $k \in [2, B]$ probe sequences containing L required words to scan, including the k terminators, the maximum number of I/Os that Balanced Parallel Search will perform is*

$$1 + \Theta\left(\frac{\ln k}{W\left(\frac{B \ln k}{L-k}\right)}\right).$$

Proof. By Lemmas 7.5.3 and 7.5.4, we find that $R_{\mathbb{N}}(h-1, k) = \Theta((h-1)/k^{1/(h-1)})$, thus by the result of Lemma 7.5.1, we have

$$\frac{L-k_1}{B} = \Theta\left(\frac{h-1}{k^{1/(h-1)}}\right) = \Theta\left(\frac{1}{\lambda k^\lambda}\right) \text{ where } \lambda = \frac{1}{h-1}, \quad \text{so } \lambda k^\lambda = \Theta\left(\frac{B}{L-k}\right).$$

Since $k > 1$, we can use the following base-changing identity about W to solve for λ:

$$\frac{W(\lambda k^\lambda \ln k)}{\ln k} = w \iff (w \ln k)e^{(w \ln k)} = \lambda k^\lambda \ln k \iff wk^w = \lambda k^\lambda \iff w = \lambda.$$

Therefore, by Lemma 7.5.7, we have

$$\frac{1}{h-1} = \lambda = \Theta\left(\frac{W\left(\frac{B \ln k}{L-k}\right)}{\ln k}\right).$$

By re-arranging this bound to solve for h, we obtain the desired constraint. □

This worst case upper bound directly leads to a matching upper bound on the expected number of I/Os for open addressed hash table operations:

Theorem 7.5.9. *On an open addressing hash table with an expected probe sequence length of $\mathbb{L}$ items plus a terminator, performing a batch* FIND *of $k \in [2, B]$ items using Balanced Parallel Search takes $O\left(1 + \frac{\ln k}{W\left(\frac{B}{k \cdot \mathbb{L}} \ln k\right)}\right)$ expected I/Os. Similarly, batch* INSERT *and* REMOVE *take $O\left(1 + \frac{\ln k}{W\left(\frac{B}{k \cdot \mathbb{L}} \ln k\right)}\right)$ expected amortized I/Os.*

Proof. By Lemma 7.5.8, k probe sequences take up to $1 + \Theta\left(\frac{\ln k}{W\left(\frac{B \ln k}{L-k}\right)}\right)$ I/Os to search, which is an increasing, concave-down function of L. By linearity of expectation, these sequences are expected to contain $\mathbb{E}[L - k] = k \cdot \mathbb{L}$ items, so by Jensen's Inequality we get

$$\frac{\ln k}{W\left(\frac{B \ln k}{k \cdot \mathbb{L}}\right)} = \frac{\ln k}{W\left(\frac{B \ln k}{\mathbb{E}[L-k]}\right)} \geq \mathbb{E}\left[\frac{\ln k}{W\left(\frac{B \ln k}{L-k}\right)}\right]$$

Therefore using Balanced Parallel Search performs $O\left(1 + \frac{\ln k}{W\left(\frac{B}{k \cdot \mathbb{L}} \ln k\right)}\right)$ I/Os in expectation, including the work of finding, inserting, or removing items scanned during the search in each operation, respectively. □

As a corollary, we see that an open addressing hash table can perform k-batch FIND more efficiently using Balanced Parallel Search compared to doing k individual operations of $\Theta(1 + \mathbb{L}/B)$ I/Os, breaking below the $\Omega(1)$ I/O lower bound per query. However, because $\frac{\ln k}{W\left(\frac{B \ln k}{L-k}\right)} = \omega((L-k)/B)$, our analysis suggests it is not sufficient to obtain the ideal of $O(1)$ expected I/Os for implementing k-batched FIND in GEM.

7.5.2 Chaining

Due to the current suboptimal upper-bound of Balanced Parallel Search, we now consider another common hash table strategy: chaining, a form of closed addressing. Here, we use a chain data structure, such as a dynamic array that supports random access and knows its length, to store each item that hashes to a given slot. In this way, finding a set of items can be done in parallel without ever reading non-required words beyond a terminator using a more-sophisticated strategy:

Definition 7.5.5. Given $k \leq B$ chains of known lengths to scan, the ***Length-Aware Parallel Search*** strategy does the following:

1. If all chains fit into one block (i.e. $\sum_{\ell \in \text{lengths}} \ell \leq B$), then scan them completely in one I/O, otherwise we calculate the minimum $t \in [1, B]$ such that $B \leq \sum_{\ell \in \text{lengths}} \min(t, \ell) < 2B$ then scan at most t remaining words from each chain in at most two I/Os.
2. Identify the k' chains that were not been fully scanned yet.
3. If $k' > 0$, recursively do Length-Aware Parallel Search on the remaining words of those k' chains.

We show that this strategy gives optimal speedup on the problem of scanning every word of every given chain.

Lemma 7.5.10. *The Length-Aware Parallel Search strategy takes $\Theta(1 + L/B)$ I/Os, where L is the sum of lengths of each chain.*

Proof. To start the scan, one I/O is necessary to read the lengths of each probe run. Each iteration, if $L = \sum_{\ell \in \text{lengths}} \ell > B$, then at most two I/Os are performed to read the next between B and $2B$ words, and once $L \in [1, B]$, then exactly one last I/O is performed to finish the scan. □

Unfortunately again, in the pursuit of implementing an efficient GEM chaining hash table, Lemma 7.5.10 does not imply optimal speedup if a problem requires terminating an individual search immediately once it is found, such as performing a BATCH-FIND where every item is present. Since the termination occurs at an unknown position within each chain, the search is no longer "length-aware" and thus is equivalent to Balanced Parallel Search.

7.5.3 Dynamic Perfect Hashing

To overcome the apparent limitations of Balanced Parallel Search, we turn to dynamic perfect hashing. The Dynamic FKS approach uses a table of collision-free hash tables of items, giving worst-case $\Theta(1)$ time for find and $O(1)$ expected amortized time for insert and remove in the RAM model [DKM+94]. We can take this approach into the GEM model to achieve optimal speedup on batch operations with the BATCH-FKS-MAP data structure.

The primary change needed to support $O(1)$ expected amortized I/O batch insertions and removals is the memory allocation strategy. As originally described, inserting an item that collides with an existing item in an inner hash table results in immediately rebuilding that table with a new size and hash function. During a batched insert, if many small inner tables need to be rebuilt, then for I/O efficiency we must allocate their backing memory in bulk, assuming that allocating x contiguous words takes $\Theta(\lceil x/B \rceil)$ I/Os. Bulk allocations must also be done when resizing the entire table, at least for inner tables requiring $o(B)$ space. Because resizing the table occurs every $O(n)$ insertions or removals, fragmentation does not cause more than a constant factor overhead in space usage.

Theorem 7.5.11. *The* BATCH-FKS-MAP *performs* BATCH-FIND *of $k \leq B$ items in $\Theta(1)$ I/Os, and* BATCH-INSERT *and* BATCH-REMOVE *of $k \leq B$ items in $O(1)$ expected amortized I/Os.*

7.6 List Ranking

In the list ranking problem, studied in the EM model in [CGG+95], we are given an n-node linked list, represented as an unordered array of nodes each with the index of their successor, and must label each node with its distance from the head of the linked list (its "rank").

Theorem 7.6.1. *Ranking a list of n nodes is solvable in $\Theta(\frac{n}{B})$ I/Os.*

Proof. We give an $O(\frac{n}{B})$ I/O algorithm called LIST-RANK in Algorithm 20 based on the approach of [CGG+95]. Specifically, LIST-RANK is a recursive algorithm for the more general problem of weighted list ranking, where each edge is given a weight and the rank of a node is the cumulative weight along the path to it from the head. To solve unweighted list ranking, we can simply set all edge weights to 1.

Assuming the problem does not fit into internal memory, given an array of n successor indices and weights, we first compute a predecessor-indices array. Then we construct a maximal independent set (MIS) of nodes, which can be done in $O(1 + n/B)$ I/Os by scanning the nodes in batches and

maintaining a table of which items have been added to the MIS for $O(1)$ I/O batch lookup and updates. Since the nodes form a linked list, the size of this MIS is $n' \in [n/3, \lceil n/2 \rceil]$, since there can be at most two nodes in a row not present in the MIS but never two in a row that are.

Second, we form a compressed list that only contains the nodes in the MIS by contracting all non-MIS vertices, obtaining new successor and weight arrays of size n'. We do this by in parallel walking at most two edges down the list from every node in the MIS to find the next MIS node, adding the weights of traversed edge to determine the compressed edge's weight, and relabeling the nodes to fit in the range $[0, n' - 1]$. Batching this parallel work can all be done in $O(1 + n/B)$ I/Os.

This gives us a size-n' subproblem that we can solve recursively to get the ranks of every MIS node in the compressed list, which by construction are the ranks in the original list. Therefore, we can lastly compute the ranks of the non-MIS nodes in the original list by in parallel walking at most two edges down the list from every node in the MIS and propagating the path weight, again taking only $O(1 + n/B)$ I/Os.

The I/O cost $T(n)$ of LIST-RANK satisfies, for $n \geq M/3$, the recurrence $T(n) = T(n') + O(n/B) \leq T(\lceil n/2 \rceil) + O(1 + n/B)$ (where the inequality is because $n' \leq \lceil n/2 \rceil$ and $T(n)$ is monotonic increasing). Once $n < M/3$, the problem fits in cache so $T(n) = O(n/B)$ as a base case. The recurrence thus expands to a sum of $n/M + O(1)$ terms, each of which consists of $O(1)$ plus a geometrically decreasing term. Therefore $T(n) = O(\max(\frac{n}{B}, \log \frac{n}{M}))$. The n/B dominates because $B \leq M$, so $T(n) = \Theta(\frac{n}{B})$, meeting the trivial lower bound. □

Algorithm 20 Computes the rank of every node in a linked list in $O(n/B)$ I/Os

```
1: function LIST-RANK(succ: int[], w: int[]): int[]
2:     if length(succ) < M/3 then return LIST-RANK-IN-CACHE(succ, w)
3:     predecessors ← INVERT-LIST(succ)
4:     (mis, mis-set) ← LIST-MIS(succ, pred)
5:     mis-inv ← INVERT-MIS(mis-set)
6:     (succ₂, w₂) ← COMPRESS-LIST(mis, mis-set, mis-inv, succ, w)
7:     d₂ ← LIST-RANK-REC(succ₂, w₂)
8:     return DECOMPRESS-RANKS(mis, succ, w, succ₂, d₂)
```

Algorithm 21 Compute inverse list or inverse map in $O(n/B)$ I/Os

```
1: function INVERT-LIST(succ: int[]): int[]
2:     pred ← [ i | 0 ≤ i < length(succ) ]          ▷ Initialized so head node will have no inverse
3:     for batch i, 𝔰 in succ
4:         parallel for i, s in ZIP(i, 𝔰)            ▷ Iterate over nodes and their successors in lockstep
5:             if i = s then continue
6:             pred[s] ← i                           ▷ Invert map for all but tail node
7:     return pred

8: function INVERT-MIS(mis: int[]): BATCH-MAP
9:     mis-inv ← BATCH-MAP(1+max(mis))
10:    for batch i, 𝔪 in mis
11:        BATCH-INSERT(mis-inv, 𝔪, i)
12:    return mis-inv
```

Algorithm 22 Computes a maximal independent set of a list in $O(n/B)$ I/Os

```
1: function LIST-MIS(succ: int[], pred: int[]): (int[], BATCH-SET)
2:     mis ← [FIXED-POINT(pred), FIXED-POINT(succ)]      ▷ Start with beginning and end of list
3:     mis-set ← BATCH-SET(length(succ))
4:     BATCH-INSERT(mis-set, mis)                                          ▷ Initialize set
5:     for batch 𝔦, 𝔰, 𝔭 in ZIP(succ, pred)
6:         𝔢𝔩𝔦𝔪 ← (𝔰 ∪ 𝔭) ∩ mis-set          ▷ Lookup neighbors that are already in mis
7:         𝔟𝔲𝔣 ← []                                                ▷ Buffer of to-be-inserted nodes
8:         for i, s, p in ZIP(𝔦, 𝔰, 𝔭)
9:             if i ∈ 𝔢𝔩𝔦𝔪 ∨ p ∈ 𝔟𝔲𝔣 ∨ s ∈ 𝔟𝔲𝔣 then continue      ▷ Check whether i can be added
10:            PUSH(𝔟𝔲𝔣, i)                                  ▷ Note i as being inserted into mis
11:        BATCH-PUSH(mis, 𝔟𝔲𝔣)                                ▷ Flush buffer of inserted nodes
12:        BATCH-INSERT(mis-set, 𝔟𝔲𝔣)
13:    return (mis, mis-set)

14: function FIXED-POINT(array: int[]): int[]      ▷ Returns an index i such that array[i] = i
15:    for batch 𝔦, 𝔟 in array
16:        for i, x in ZIP(𝔦, 𝔟)                          ▷ Iterate over index and value in lockstep
17:            if i = x then
18:                return i
19:    return ⊥
```

Algorithm 23 Builds a list connecting the nodes in an MIS in $O(n/B)$ I/Os

```
1: function COMPRESS-LIST( mis: int[], mis-set: BATCH-SET,
                           mis-inv: BATCH-MAP, succ: int[], w: int[] ): (int[], int[])
2:     n_2 ← length(mis)
3:     (succ_2, w_2) ← ([ ⊥ | 0 ≤ i < n_2 ], [ 0 | 0 ≤ i < n_2 ])
4:     parallel for batch i, m in mis
5:         next ← m
6:         repeat
7:             w_2[i] ← w_2[i] + w[next]
8:             next ← succ[next]
9:         until next ∉ mis-inv
10:        succ_2[i] ← mis-inv[next]
11:    return (succ_2, w_2)
```

Algorithm 24 Computes ranks of all nodes from ranks of MIS nodes in $O(n/B)$ I/Os

```
1: function DECOMPRESS-RANKS( mis: int[], succ: int[],
                              w: int[], succ_2: int[], d_2: int[] ): (int[], int[])
2:     d ← [ ⊥ | 0 ≤ i < length(mis) ]
3:     parallel for batch i, m in mis
4:         d[m] ← d_2[i]
5:         last ← mis[succ_2[i]]
6:         s ← succ[m]
7:         while s ≠ last
8:             d[s] ← d[m] + w[m]
9:             (m, s) ← (succ[s], m)
10:    return d
```

Algorithm 25 Compute list ranks fully within cache (requires n no more than $M/3$)

```
1: function LIST-RANK-IN-CACHE(succ: int[], w: int[]): int[]
2:     d ← [ 0 | 0 ≤ i < length(mis) ]
3:     Read succ, w, and d into cache
4:     for (cur ← FIXED-POINT(succ) ; cur ≠ succ[cur] ; cur ← succ[cur])
5:         d[succ[cur]] ← d[cur] + w[cur]
6:     return d
```

7.7 Graph Algorithms

Graph algorithms are well-studied in EM, but many important problems lack known optimal algorithms [AM09]. We begin exploring the advantages of GEM in this field by giving improved algorithms for four problems: computing maximal independent sets and small dominating sets, graph clustering, connected components, and minimum spanning forests.

For our results in this section, we assume that every graph $G = (V, E)$ with n vertices and m edges is given in adjacency list representation, where $V = \{0, 1, \ldots, n-1\}$ and adj[u] is an array of the neighbors of $u \in V$, and that there are no self-loops or parallel edges.

7.7.1 Maximal Independent Set and Small Dominating Sets

First, we will define our terms and make some observations relating maximal independent sets and dominating sets of undirected graphs.

Definition 7.7.1. An ***independent set*** of an undirected graph $G = (V, E)$ is a set $I \subseteq V$ such that no pair $u, v \in I$ is adjacent. An independent set is ***maximal*** if, for all $x \in V \setminus I$, $I \cup \{x\}$ is not independent.

Theorem 7.7.1. *A maximal independent set of an undirected graph can be computed in* $\Theta((n+m)/B)$ *I/Os.*

Proof. Our algorithm giving optimal speedup is MAXIMAL-INDEPENDENT-SET, described in Algorithms 26 and 27. The algorithm processes the vertices in batches, maintaining a table specifying which nodes have been added to the MIS so far (and which have been rejected) and an output list. For each batch of B vertices, we can ignore those already marked as rejected and then use

BATCH-ENUMERATE-JOINER to efficiently scan each of their neighbors for other nodes already added to the MIS to determine which of them can be added. This takes only $O((n+m)/B)$ I/Os, matching the trivial lower bound. □

Definition 7.7.2. A ***dominating set*** of an undirected graph $G = (V, E)$ is a subset $D \subseteq V$ such that for all $u \in V \setminus D$, u is adjacent to some $v \in D$ (we say that v ***dominates*** u).

Lemma 7.7.2. *An independent set is maximal if and only if it is also a dominating set.*

Lemma 7.7.3. *If I is an independent set of $G = (V, E)$, and G has no isolated vertices, then its complement $V \setminus I$ is a dominating set of G.*

Proof. Consider any $u \in I = V \setminus (V \setminus I)$. Because u is not isolated, it is adjacent to some other vertex v. Because I is an independent set, we must have $v \in V \setminus I$. Thus u is dominated by v. □

Lemma 7.7.4. *Any undirected graph $G = (V, E)$ with no isolated vertices has a dominating set D of size at most $n/2$.*

Proof. Consider any maximal independent set I of G. Both I and $V \setminus I$ are dominating sets (by Lemmas 7.7.2 and 7.7.3). Let D be the smaller of these two dominating sets, which must have at most $(|I| + |V \setminus I|)/2 = n/2$ vertices. □

Thus, by Theorem 7.7.1 we get the following:

Theorem 7.7.5. *A dominating set with size at most $n/2$ exists in any undirected graph with no isolated vertices and can be computed in $\Theta((n+m)/B)$ I/Os.*

Algorithm 26 Computes a maximal independent set of a graph using $O((n+m)/B)$ I/Os

```
1:  function MAXIMAL-INDEPENDENT-SET(adj: int[][]): int[]
2:      mis ← new BATCH-ARRAY
3:      mis-map ← BATCH-MAP(n)                    ▷ Stores whether u ∈ V is in the mis
4:      for batch 𝔟 in 0...n
5:          inserted ← BATCH-FIND(mis-map, 𝔟)
6:          proposed ← [ 𝔟[i] | inserted[i] == false ]
7:          to-add ← MAXIMAL-INDEPENDENT-SET-BLOCK(adj, proposed)
8:          BATCH-INSERT(mis-map, to-add, [ true | 0 ≤ i < length(to-add) ])
9:          BATCH-PUSH(mis, to-add)
10:         to-add-adj ← [ adj[u] | u ∈ to-add ]
11:         for batch 𝔫 in BATCH-JOINER(to-add-adj)
12:             BATCH-INSERT(mis-map, 𝔫, [ false | 0 ≤ i < length(to-add) ])
13:     return DATA(mis)
```

Algorithm 27 Find MIS of $n' \leq B$ nodes with m' edges in $O((n'+m')/B)$ I/Os

```
1: function MAXIMAL-INDEPENDENT-SET-BLOCK(adj: int[][], proposed: int[]): int[]
2:     (mis, elim) ← (∅, ∅)                          ▷ Block-sized sets of vertices in/out of MIS
3:     prev ← proposed[0]
4:     p-adj ← [ adj[u] | u ∈ proposed ]
5:     for batch i, v in BATCH-ENUMERATE-JOINER(p-adj)
6:         for i, v in ZIP(i, v)                      ▷ Iterate over edges in lockstep
7:             u ← proposed[i]
8:             if prev ≠ u ∧ prev ∉ elim then
9:                 mis ← mis ∪ {prev}
10:                prev ← u
11:            if v ∈ mis then
12:                elim ← elim ∪ {u}
13:    if prev ∉ elim then
14:        mis ← mis ∪ {prev}
15:    return ARRAY(mis)
```

7.7.2 Dominating Set Clustering

Our algorithms for connected components and minimum spanning forest in the following sections are based on what we call a ***dominating set clustering*** of a graph.

Definition 7.7.3. Given an undirected graph $G = (V, E)$ and a dominating set D of G, let a ***dominating set clustering*** of G about D be a graph $G' = (V', E')$ such that

1. V' is a partition of V into $|D|$ disjoint subsets,
2. each $X \in V'$ consists of exactly one $v \in D$ and a subset of its neighbors, and
3. there is an edge $(X, Y) \in E'$ if and only if $(x, y) \in E$ for some $x \in X$ and $y \in Y$.

We provide DOMINATING-SET-CLUSTER in Algorithm 28 to find for a graph $G = (V, E)$ a dominating set D with at most $|V|/2$ vertices (as described in Lemma 7.7.5) and compute a clustering about D in $O((n+m)/B)$ I/Os. Like the maximal independent set algorithm, DOMINATING-SET-CLUSTER scans the graph and its dominating set in batches and greedily adds the neighbors of each root node $v \in D$ to v's subset in the clustering. As a subroutine in later algorithms, it also represents the output cluster graph as an adjacency list with integer labels along with a mapping from V to V' and V' to D.

Theorem 7.7.6. *A dominating set clustering of a graph can be computed in $\Theta((n+m)/B)$ I/Os.*

Algorithm 28 Cluster a graph around a Dominating Set using $O((n+m)/B)$ I/Os

```
1: struct DOM-SET-CLUSTER
2:     adj: int[][]
3:     roots: int[]
4:     parent, new-root-index: BATCH-MAP
5: function DOMINATING-SET-CLUSTER(adj: int[][]): DOM-SET-CLUSTER
6:     n ← length(adj)
7:     c ← new DOM-SET-CLUSTER(⊥, ⊥, new BATCH-MAP(n), new BATCH-MAP(n))
8:     roots(c) ← MAXIMAL-INDEPENDENT-SET(adj)                    ▷ Get a MIS
9:     if length(roots(c)) > n/2 then          ▷ Replace MIS with its complement if too big
10:        roots(c) ← SET-DIFFERENCE([ i | 0 ≤ i < n ], COUNTING-SORT(roots(c)))
11:    for batch i, u in roots(c)                                 ▷ A root is its own parent
12:        BATCH-INSERT(parent(c), u, u)
13:        BATCH-INSERT(new-root-index(c), u, i)
14:    for batch u in roots(c)                  ▷ A nonroot is a child of some root neighbor
15:        nbrs ← [ adj[u] | u ∈ u ]
16:        for i, v in BATCH-ENUMERATE-JOINER(nbrs)
17:            BATCH-INSERT(parent(c), v, u[i])
18:    adj(c) ← [ [] | 0 ≤ i < length(roots(c)) ]
19:    for u, v in BATCH-ENUMERATE-JOINER(adj)                    ▷ Add edges to contracted graph
20:        up ← BATCH-VALUE(parent(c), u)                         ▷ Look up parents of each endpoint
21:        vp ← BATCH-VALUE(parent(c), v)
22:        ur ← BATCH-VALUE(new-root-index(c), up)                ▷ Look up cluster indices
23:        vr ← BATCH-VALUE(new-root-index(c), vp)
24:        parallel for batch u′, v′ in ZIP(ur, vr)
25:            if u′ ≠ v′ then                                    ▷ Eliminate edges within cluster
26:                PUSH(adj(c)[u′], v′)
27:                PUSH(adj(c)[v′], u′)
28:    adj(c) ← DUPLICATE-REMOVAL-MANY(adj(c))                    ▷ Remove parallel edges
29:    return c
```

7.7.3 Connected Components

Definition 7.7.4. A ***connected component (CC)*** of an undirected graph $G = (V, E)$ is a subset $S \subseteq V$ such that, for every edge $\{u, v\} \in E$, $u \in S$ if and only if $v \in S$. A ***connected component labeling*** is an integer map $c : V \to [0, k)$ such that the k connected components of G are the sets $S_\ell = \{u \mid c(u) = \ell\}$ for $\ell \in \{0, 1, \ldots, k-1\}$.

We show how to compute a connected component labeling of an undirected graph using $O((n+m)/B)$ I/Os in Algorithm 30. CONNECTED-COMPONENTS is a recursive algorithm with two steps per call to reduce the size of the graph and two base cases. We initiate the recursion with some graph G_1 with n_1 vertices. If the current graph becomes empty, we return an empty labeling; otherwise we perform the following steps.

Step 1 is to reduce to a graph with no isolated vertices, which are trivial connected components. We start by using REMOVE-SINGLETONS from Algorithm 29 to create a copy of the graph without the isolated vertices (if any). After finding its connected components in Step 2, we use the stored vertex mapping to apply the labels back onto the original vertices and produce new labels for the removed isolated vertices.

In Step 2, CONNECTED-COMPONENTS-NO-SINGLETONS accepts a graph G with no isolated vertices and branches based on its density. If density measure $\Delta = m/n$ is at least B, or the number of vertices is less than n_1/B, then we can afford to spend one I/O per vertex. In this case, we use the PRAM depth-first search algorithm developed by Träff et al. [Trä13]: we can apply it directly find the labeling of G and end the recursion early using $O(n+m/B)$ I/Os by our PRAM simulation Theorem 7.2.1.

On the other hand, if the graph is sparse and large, then we use DOMINATING-SET-CLUSTER to reduce G to a cluster graph G_2 with at most half as many vertices. Since G_2 may contain isolated vertices, we recursively call CONNECTED-COMPONENTS to find connected component labels for the dominating set vertices, which we can apply back onto the original graph and propagate within each cluster.

Theorem 7.7.7. CONNECTED-COMPONENTS *on a graph $G = (V, E)$ with n vertices and m edges uses $O((n+m)/B \cdot \log B)$ I/Os*

Proof. Step 1 calls REMOVE-SINGLETONS and later applies the results of Step 2 to the original graph, both using $O((n+m)/B)$ I/Os. Step 2 calls TRÄFF-CONNECTED-COMPONENTS in the case that $m/B \geq n$, thus using $O(n+m/B) = O((n+m)/B)$ I/Os. While the graph stays sparse ($m/B < n$), it takes $O((n+m)/B)$ I/Os to reduce to a graph G_2 with at most half as many vertices and later apply the results to the original graph.

Consider the series of graphs $(G_1, G_2, G_3, \ldots, G_k)$ produced by the recursion and the corresponding size $s_i = n_i + m_i$ of the adjacency list representation of G_i, where $G_1 = G$, $n_1 = n$, $m_1 = m$, and G_k is a base case. Since for each graph G_i the algorithm uses $O(s_i/B)$ I/Os, it remains to upper bound the size progression s_i and recursion depth k.

In each level $i < k$, because we only recurse when $m_i < B \cdot n_i$, we can upper bound $s_i < (1+B) \cdot n_i = s_i^*$. Since clustering reduces the number of vertices by at least half, we have the relation $n_i \geq 2n_{i+1}$ and thus for $i+1 < k$ we get $s_i^* \geq 2s_{i+1}^*$. Lastly, by construction, every edge in G_{i+1} represents at least one edge in G_i and at least $n_i/2$ edges are removed, so we can see that $m_i - n_i/2 \geq m_i - (n_i - n_{i+1}) \geq m_{i+1}$.

The maximum possible size of G_{i+1} is thus when $n_i = 2n_{i+1}$ and $m_i - n_i/2 = m_{i+1}$. In terms of G_1, this can be expressed as $n_i = n_1/2^{i-1}$ and $m_i = m_1 - \sum_{j=1}^{i-1} n_j/2 = m_1 - n_1 \sum_{j=1}^{i-1} (\frac{1}{2})^j$, thus for $i > 1$ we get $s_i = n_i + m_i = m_1 - n_1 \sum_{j=1}^{i-2} (\frac{1}{2})^j \in [m_1 + n_1/2, m_1 - n_1]$.

Thus if $m_1 = \omega(n_1)$ then the worst shrinkage would result in $s_i = \Theta(s_1)$. Given density measure $\Delta_i = m_i/n_i$, we can solve for the maximum depth k that this minimum shrinkage can persist before crossing over the base case threshold of $\Delta_k = B$:

$$
\begin{aligned}
m_k &= m_1 - n_1 \sum_{j=1}^{k-1} (1/2)^j = B \cdot n_1/2^{k-1} = B \cdot n_k \\
2^{k-1} m_1 - n_1(2^{k-1} - 1) &= B \cdot n_1 \\
k &= 1 + \log_2 \frac{B \cdot n_1 - n_1}{m_1 - n_1} = 1 + \log_2 \frac{B-1}{\Delta_1 - 1} = O(\log B)
\end{aligned}
$$

Thus, assuming minimum shrinkage in the clustering step from G_i to G_{i+1}, after $k = O(\log B)$ recursive calls, G_k must be dense enough to run TRÄFF-CONNECTED-COMPONENTS, which uses $O(n_k + m_k/B) = O(s_k/B)$ I/Os. However, without minimum shrinkage, it is possible for G_k to be below the density threshold, but if we recall that the number of vertices always decreases by at least

half, we see that $n_k \leq n_1/2^{k-1} = n_1/B$, meaning the graph has been sufficiently contracted such that we can bound TRÄFF-CONNECTED-COMPONENTS to use $O(n_k + m_k/B) = O(n_1/B + m_k/B)$ I/Os, thus end the recursion in the same way.

In total, the base case use at most $O((n+m)/B)$ I/Os, which is dominated by the recursive cases, which use up to $O\left(\sum_{i=1}^{\log B} s_i/B\right) = O\left(s_1/B \cdot \log B\right)$ I/Os, as claimed. □

Algorithm 29 Removes singleton nodes from an undirected graph using $O((n+m)/B)$ I/Os

```
1: struct MAPPED-GRAPH
2:     adj: int[][]                          ▷ Adjacency list of mapped subgraph G_2 of G
3:     index-map: BATCH-MAP                  ▷ Maps u → u_2 for vertices u that aren't deleted from G
4:     inv-map: int[]                        ▷ Maps u_2 → u for all vertices u_2 in G_2
5: function REMOVE-SINGLETONS(adj: int[][]): MAPPED-GRAPH
6:     indices ← [ i | length(adj[i]) > 0 ]  ▷ Compute (sorted) list of non-singletons
7:     index-map ← new BATCH-MAP(length(adj))
8:     (output-values, output-lengths) ← (new BATCH-ARRAY, new BATCH-ARRAY)
9:     for batch i_2, i in indices
10:        BATCH-INSERT(index-map, i, i_2)   ▷ Assign new node indices
11:        s ← adj[i]
12:        l ← [ length(s) | s ∈ s ]
13:        BATCH-PUSH(output-lengths, l)     ▷ Assign node degrees
14:    for v in BATCH-JOINER(adj)
15:        v_2 ← BATCH-VALUE(index-map, v)   ▷ Get new indices of each edge endpoint
16:        BATCH-PUSH(output-values, v_2)    ▷ Store endpoint (flattened) for adj_2
17:    adj_2 ← CONVERT-DATA-LENGTH-TO-SLICES(DATA(output-values), DATA(output-lengths))
18:    return new MAPPED-GRAPH(adj_2, index-map, indices)
```

Algorithm 30 Compute connected components of a graph using $O((n+m)/B \cdot \log B)$ I/Os

```
1: function CONNECTED-COMPONENTS(adj: int[][], n_1: int): int[]
2:     if length(adj) = 0 then return []
3:     G_2 ← REMOVE-SINGLETONS(adj)
4:     cc_2 ← CONNECTED-COMPONENTS-NO-SINGLETONS(adj(G_2), n_1)
5:     return CC-ADD-BACK-SINGLETONS(G_2, cc_2)
```

```
6: function CONNECTED-COMPONENTS-NO-SINGLETONS(adj: int[][], n_1: int): int[]
7:     if length(adj) = 0 then return []
8:     (n, m) ← (length(adj), SUM([ length(x) | x ∈ adj ])) ▷ Count the number of nodes and edges
9:     if min(m, n_1) ≥ B · n then    ▷ Few enough vertices to use Träff's O(n + m/B) technique
10:        return TRÄFF-CONNECTED-COMPONENTS(adj)
11:    else                                                  ▷ Sparse enough to cluster and recurse
12:        cluster ← DOMINATING-SET-CLUSTER(adj)  ▷ Cluster to reduce number of nodes by half
13:        cc_2 ← CONNECTED-COMPONENTS(adj(cluster), n_1)  ▷ Find CC labels of clustered graph
14:        cc ← new BATCH-ARRAY
15:        for batch 𝔲 in 0 ... length(adj)
16:            𝔭 ← BATCH-VALUE(parent(cluster), 𝔲)              ▷ Get the parent node in cluster
17:            𝔭_2 ← BATCH-VALUE(new-root-index(cluster), 𝔭)    ▷ Get parents' cluster indices
18:            𝔠 ← cc_2[𝔭_2]                                    ▷ Gets parent's CC labels
19:            BATCH-PUSH(cc, 𝔠)                ▷ Output parent label as the child's label
20:        return DATA(cc)
```

```
21: function CC-ADD-BACK-SINGLETONS(G_2: MAPPED-GRAPH, cc_2: int[]): int[]
22:     cc ← new BATCH-ARRAY
23:     next ← max(cc_2) + 1                          ▷ Fresh CC label to give a singleton
24:     for batch 𝔲 in 0 ... length(index-map(G_2))
25:         𝔫𝔟𝔯 ← BATCH-FIND(index-map(G_2), 𝔲)      ▷ Identify singletons not present in G_2
26:         𝔲_2 ← BATCH-VALUE(index-map(G_2), 𝔲)     ▷ For the rest, lookup the node in G_2
27:         𝔠 ← cc_2[𝔲_2]                             ▷ Copy CC labels for non-singletons
28:         for i, has-neighbor, u in ZIP(𝔫𝔟𝔯, 𝔲)
29:             if ¬ has-neighbor then
30:                 𝔠[i] ← next                       ▷ Give each singleton a new CC label
31:                 next ← next + 1
32:         BATCH-PUSH(cc, 𝔠)
33:     return DATA(cc)
```

Theorem 7.7.8. CONNECTED-COMPONENTS *on a forest graph $G = (V, E)$ with n vertices uses $O(n/B)$ I/Os.*

Proof. In the special case that G is a forest, we can improve the analysis from Theorem 7.7.7. Notice that a dominating set clustering of an tree T must be a tree T_2: each edge of T_2 represents at least one edge of T, and T has no cycles, thus no cycle can exist in T_2 either. Given that each connected component of G is a tree, we can conclude that each graph G_i in recursive level i is a forest and thus we can bound $m_i < n_i \leq n_1/2^{i-1}$, which gives a geometric decrease bounding the total recursive cases as using $O\left(\sum_{i=1}^{k} s_i/B\right) = O\left(n_1/B \sum_{i=1}^{\infty} 1/2^{i-1}\right) = O(n_1/B)$ I/Os. □

7.7.4 Minimum Spanning Trees

Definition 7.7.5. A ***minimum spanning tree*** (MST) of an connected undirected weighted graph $G = (V, E, w)$ is a subgraph (V, E') whose edges $E' \subseteq E$ form a tree that minimizes $\sum_{e \in T} w(e)$. More generally, a ***minimum spanning forest*** (MSF) consists of an MST for each connected component of a graph.

Theorem 7.7.9. *An undirected MSF on a weighted graph $G = (V, E, w)$ with n vertices and m edges can be found using $O(\text{sort}(m) + (m/B)\log(m/M))$ I/Os.*

Proof. We implement Kruskal's MSF algorithm [Kru56] using the offline union-find algorithm of Lemma 7.4.1.

After using $O(\text{sort}(m))$ I/Os to sort the edges from lightest to heaviest weight, we apply the algorithm of Lemma 7.4.1 where the sequence of operations is UNION(u, v) for each edge (u, v) in the sorted order. This uses uses $O((m/B)\log(m/M))$ I/O's and tells us the non-redundant operations, which correspond to the edges in an MSF.

Because this only produces undirected edges, we also provide a recursive dominating set clustering algorithm to direct the edges to produce rooted MST's with Algorithm 31 using an additional $O(n + m)/B)$ I/Os. □

For the remainder of this section, we will restrict our attention to the case where the edges are given in sorted order.

We show how to compute a minimum spanning forest of an undirected weighted graph using $O(\text{sort}(m) + \alpha(n) \cdot (n + m)/B)$ I/Os, where $\alpha(n)$ is the slow-growing inverse Ackermann function. Our Algorithm 32 implements Kruskal's algorithm [Kru56] efficiently using the BATCH-DISJOINT-SET data structure.

Because this only produces undirected edges, we also provide a recursive dominating set clustering algorithm to direct the edges to produce rooted MST's with Algorithm 31 using an additional $O(n + m)/B)$ I/Os.

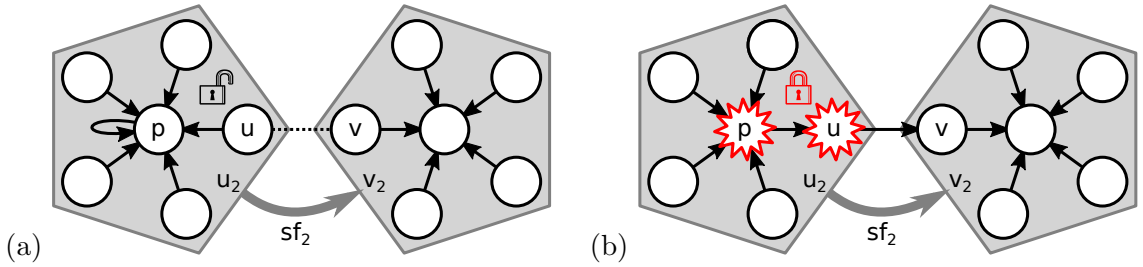


Figure 7.1: Illustration of SF-UNCLUSTER from Algorithm 31. (a) First edge (u, v) found to connect cluster u_2 to parent cluster $v_2 = \text{sf}_2[u_2]$. (b) Creating the path $p \to u \to v$ merges the two rooted trees into one, and u_2 is "locked" to ensure (u, v) will be the only outgoing edge from u_2 to v_2.

Theorem 7.7.10. MINIMUM-SPANNING-FOREST *on a weighted graph $G = (V, E, w)$ with n vertices and m edges uses $O(\text{sort}(m) + \log(n) \cdot (n + m)/B)$ I/Os.*

Proof. We first look at each line at the top level. REMOVE-SINGLETONS uses $O((n + m)/B)$ I/Os, and filtering w and SPANNING-FOREST-ADD-BACK-SINGLETONS uses $O(n/B)$ I/Os.

Inside MINIMUM-SPANNING-FOREST-NO-SINGLETONS, we perform Kruskal's algorithm by first using $O((n + m)/B)$ I/Os in ADJ-TO-WEIGHTED-EDGES to get a list of m edges, sort the list

using $O(\text{sort}(m))$ I/Os, and finish with UNION-FIND-EDGES using $O(n/B + (m/B)\log(m/M))$ I/Os by Theorem 7.4.2. Finally, we can direct the edges of the spanning forest using only $O((n+m)/B)$ I/Os with FOREST-EDGES-TO-PARENTS, which reconstructs an adjacency list and recursively constructs parent pointers with the dominating set clustering strategy of CONNECTED-COMPONENTS (Algorithm 30).

The primary difference with clustering here is SF-UNCLUSTER, which translates cluster parents to vertex parents using just $O((n+m)/B)$ I/Os. Figure 7.1 details this key operation in producing a full spanning tree given a spanning tree of the cluster graph, taking advantage of the property that each cluster is constructed to be a star graph to efficiently merge them into a single rooted tree. As we are operating on a forest, we appeal to the argument for Theorem 7.7.8 of Theorem 7.7.7

Algorithm 31 Computes a directed-edge spanning forest of a graph

```
1:  function SPANNING-FOREST(adj: int[][], n_1: int): int[]
2:      if length(adj) = 0 then return []
3:      G_2 ← REMOVE-SINGLETONS(adj)
4:      sf_2 ← SPANNING-FOREST-NO-SINGLETONS(adj(G_2), n_1)
5:      return SPANNING-FOREST-ADD-BACK-SINGLETONS(G_2, sf_2)
```

```
6:  function SPANNING-FOREST-NO-SINGLETONS(adj: int[][], n_1: int): int[]
7:      if length(adj) = 0 then return []
8:      (n, m) ← (length(adj), SUM([ length(a) | a ∈ adj ]))
9:      if min m, n_1 ≥ B · n then      ▷ Few enough vertices to use Träff's O(n + m/B) technique
10:         return TRÄFF-SPANNING-FOREST(adj)
11:     else                                          ▷ Sparse enough to cluster and recurse
12:         cluster ← DOMINATING-SET-CLUSTER(adj)  ▷ Cluster to reduce number of nodes by half
13:         sf_2 ← SPANNING-FOREST(adj(cluster), n_1)          ▷ Find parent pointers of clusters
14:         return SF-UNCLUSTER(adj, cluster, st_2)          ▷ Translate to parent pointers for G
```

```
15: function SF-UNCLUSTER(adj: int[][], cluster: DOM-SET-CLUSTER, sf_2: int[]): int[]
16:     sf ← BATCH-VALUE(parent(cluster), [ i | 0 ≤ i < length(adj) ]) ▷ Direct edges towards par-
        ents
17:     locked ← [ u_2 = p_2 | (u_2, p_2) ∈ ENUMERATE(sf_2) ]          ▷ Track processed clusters
18:     for batch 𝔲, 𝔳 in BATCH-ENUMERATE-JOINER(adj)
19:         𝔭_𝔲 ← BATCH-VALUE(parent(cluster), 𝔲)                    ▷ Get parent vertices in G
20:         𝔭_𝔳 ← BATCH-VALUE(parent(cluster), 𝔳)
21:         𝔲_2 ← BATCH-VALUE(new-root-index(cluster), 𝔭_𝔲)      ▷ Get clusters of each endpoint
22:         𝔳_2 ← BATCH-VALUE(new-root-index(cluster), 𝔭_𝔳)
23:         𝔭_2 ← sf_2[𝔲_2]                                          ▷ Get parents in cluster MSF
24:         𝔩 ← locked[𝔲_2]                                  ▷ Read which u_2's need to be processed
25:         𝔥 ← {}                                          ▷ Local map to cache ≤ 2B writes to sf
26:         for u, v, p_u, u_2, v_2, p_2, ref ℓ in ZIP(𝔲, 𝔳, 𝔭_𝔲, 𝔲_2, 𝔳_2, 𝔭_2, 𝔩)
27:             if ¬ℓ ∧ p_2 = v_2 then          ▷ If u → v will connect u_2 → p_2 (for the first time)
28:                 𝔥[u] ← v                                  ▷ Connect clusters by directing edge u → v
29:                 if u ≠ p_u then 𝔥[p_u] ← u      ▷ If u isn't the root of its cluster, direct p_u → u
30:                 ℓ ← true                ▷ Lock u_2 to mark it as processed, freeze parent pointers
31:         sf[KEYS(𝔥)] ← VALUES(𝔥)                    ▷ Write new parent pointers to external memory
32:     return sf
```

that this recursion uses $O((n+m)/B)$ I/Os in total as well.

Thus, the dominant steps of this algorithm are sorting the edges by weight and performing union-find on the edges, using a combined $O(\text{sort}(m)+n/B+(m/B)\log(m/M))$ I/Os as claimed. □

Algorithm 32 Computes MSF using $O(\text{sort}(m) + \alpha(n) \cdot (n+m)/B)$ I/Os

```
1: function MINIMUM-SPANNING-FOREST(adj: int[][], w: int[][]): int[]
2:     G_2 ← REMOVE-SINGLETONS(adj)
3:     w_2 ← [ a | a ∈ w, length(a) > 0 ]
4:     msf_2 ← MINIMUM-SPANNING-FOREST-NO-SINGLETONS(adj(G_2), w_2)
5:     return SPANNING-FOREST-ADD-BACK-SINGLETONS(G_2, mst_2)
```

```
6: struct EDGE
7:     weight, src, dest: int
8: function MINIMUM-SPANNING-FOREST-NO-SINGLETONS(adj: int[][], w: int[][]): int[]
9:     edges ← ADJ-TO-WEIGHTED-EDGES(adj, w)          ▷ Convert graph into a flat edge list
10:    SORT-BY(weight, edges)                          ▷ Sort edges by weight, from lightest to heaviest
11:    msf-edges ← UNION-FIND-EDGES(edges, length(adj))   ▷ Compute tree edges
12:    return FOREST-EDGES-TO-PARENTS(msf-edges, length(adj))   ▷ Direct tree edges
```

```
13: function ADJ-TO-WEIGHTED-EDGES(adj: int[][], w: int[][]): EDGE[]
14:    edges ← new BATCH-ARRAY
15:    for batch (𝔲, 𝔳), 𝔴 in ZIP(BATCH-ENUMERATE-JOINER(adj), BATCH-JOINER(w))
16:        for u, v, w in ZIP(𝔲, 𝔳, 𝔴)          ▷ Iterate over edges and their weights in lockstep
17:            PUSH(edges, new EDGE(w, u, v))
18:    return DATA(edges)
```

```
19: function UNION-FIND-EDGES(edges: EDGE[], n: int): int[2][]
20:    msf-edges ← new BATCH-ARRAY
21:    ds ← new BATCH-DISJOINT-SET(n)
22:    for batch 𝔟 in edges
23:        (𝔲, 𝔳) ← ([ src(e) | e ∈ 𝔟 ], [ dest(e) | e ∈ 𝔟 ])
24:        𝔩 ← BATCH-UNION(ds, 𝔲, 𝔳)          ▷ Union along every edge, prioritizing lighter edges
25:        for u, v, ℓ in ZIP(𝔲, 𝔳, 𝔩)          ▷ Iterate over edges and success flags in lockstep
26:            if ¬ℓ then continue                          ▷ Union failed
27:            BATCH-PUSH(msf-edges, [[u, v], [v, u]])   ▷ Add edges where union succeeded to MSF
28:    return DATA(msf-edges)
```

```
29: function FOREST-EDGES-TO-PARENTS(edges: int[2][], n: int): int[]
30:    sorted-edges ← RADIX-SORT-PRESPLIT(edges)          ▷ Sort edges (v, u) by v then by u
31:    data ← [ u | (u, _) ∈ sorted-edges ]                          ▷ Extract list of v's
32:    lengths ← [ length(g) | g ∈ GROUP(((_, v_1), (_, v_2)) ⇒ v_1 = v_2, sorted-edges) ]
33:    adj ← CONVERT-DATA-LENGTH-TO-SLICES(data, lengths)          ▷ Build adjacency list
34:    return SPANNING-FOREST(adj)          ▷ Use recursive algorithm to direct the edges
```

Algorithm 33 Adds singletons back to a spanning forest using $O(n/B)$ I/Os

```
1: function SPANNING-FOREST-ADD-BACK-SINGLETONS(G2: MAPPED-GRAPH, sf2: int[]): int[]
2:     n ← capacity(index-map(G2))
3:     if length(sf2) = 0 then return [ i | 0 ≤ i < n ]        ▷ All singletons, so all self-pointers
4:     x ← inv-map(G2)[0]      ▷ Pick some non-singleton for avoiding missing key errors below
5:     sf ← new BATCH-ARRAY
6:     for batch 𝔲 in 0...n                                    ▷ For each batch of original vertices
7:         𝔰 ← [ ¬b | b ∈ BATCH-FIND(index-map(G2), 𝔲) ]       ▷ Identify which are singletons
8:         𝔲′ ← [ if s then x else u | (s, u) ∈ ZIP(𝔰, 𝔲) ]    ▷ Eliminate singletons for next line
9:         𝔲2 ← BATCH-VALUES(inv-map(G2), 𝔲′)                 ▷ Get mapped vertices of non-singletons
10:        𝔳 ← index-map(G2)[sf2[𝔲2]]                         ▷ Get non-singletons' parents in original graph
11:        for u, v, s in ZIP(𝔲, 𝔳, 𝔰)                        ▷ Iterate over edges and singleton flag in lockstep
12:            PUSH(sf, if s then u else v)                   ▷ Point to self (singleton) or parent (non-singleton)
13:    return DATA(sf)
```

7.7.5 Shortest Paths

Theorem 7.7.11. *SSSP with positive integer weights can be found in $O(n + (m + D)/B)$ I/Os, where D is the maximum distance of any vertex.*

Proof. We show how to implement Dial's Algorithm, a bucketing approach to Dijkstra's Algorithm where the queue Q is an array where bucket $Q[i]$ is a list of vertices v with $d[v] = i$.

To back the memory of these lists, stored as unordered dynamic arrays, we first create a simple bump allocator A on top of an dynamic array. The maximum capacity of A is $O(m)$ since each vertex insertion into a bucket is caused by visiting an edge for the first time and each vertex will be reallocated $O(1)$ times on average. Next, we initialize Q in $\Theta(D/B)$ I/Os to an array of D empty buckets, which are just null slices into A. If D is unknown, we can use a dynamic array or even a circular buffer of size $\Theta(\max_e w(e))$ instead. Lastly, we also create a table T where we will maintain a mapping from each vertex to its position in the bucket structure (its current distance estimate and index into the bucket), and insert s into the distance zero bucket.

The algorithm proceeds by scanning Q from closest to furthest distance, and maintaining the most-recently-scanned block of B buckets in internal memory. When we scan a new block, we start with its first non-empty bucket $Q[i]$ and pop a vertex v to visit in $\Theta(1)$ I/Os. Once the visit is complete, we find the next non-empty bucket in the block of Q and repeat until all are empty, then continue the scan until all vertices have been visited.

To visit v, we scan adj$[v]$ and $w[v]$, using $\Theta(1 + \deg(v)/B)$ I/Os, and for each block of neighbors identify those that need their distance estimates decreased. We can delete each of these neighbors from their current buckets and push them onto the lower-distance bucket lists using $\Theta(1)$ amortized I/Os. Since neighbors may be anywhere in their old buckets, we fill the gaps with the last vertices when necessary. Since some of their new buckets may be at capacity, we resize those before inserting by performing one bulk bump allocation onto A then BATCH-JOINER to scan and copy the data, using $\Theta(1 + \sum_j |Q[j]|/B)$ I/Os that are subsumed in amortization.

Overall, this algorithm uses $\Theta(D/B)$ I/Os to scan Q, $\Theta(n)$ I/Os to visit the vertices, and $\Theta(m/B)$ I/Os to scan all edges, thus a total of $\Theta(n + (m + D)/B)$ I/Os. □

Breadth-First Search (as well as Depth-First Search) can be performed in $O(n + m/B)$ I/Os by using the Arc Elimination techniques of [Trä13] from the PRAM model, which only gives optimal

speedup when $m/B = \Omega(n)$. However, when $D = O((n+m)/B)$, we can give a more direct implementation of the ordinary BFS algorithm that does gives optimal speedup.

Theorem 7.7.12. *A Breadth-First Search can be performed in $O((n+m)/B + D)$, where D is the maximum distance of any vertex.*

Proof. We tweak the ordinary BFS algorithm to take advantage of the GEM model during the processing of each frontier of vertices L_ℓ at distance ℓ from the source. Given array L_ℓ, we first use BATCH-JOINER to collect all neighbors of vertices in L_ℓ into an array N_ℓ, taking $\Theta(\frac{1}{B}\sum_j(1+\deg L_\ell[j]))$ I/Os. Next, we process N_ℓ in batches of size B to filter out vertices that have already been marked as seen, with the remainder becoming $L_{\ell+1}$, taking $O(|N_\ell|/B)$ I/Os. Lastly, we mark the vertices of $L_{\ell+1}$ as seen at level $\ell+1$ and set them as the next frontier, taking $O(|L_{\ell+1}|/B)$ I/Os.

Overall, each frontier L_ℓ is processed in $O(1 + |L_\ell|/B + |N_\ell|/B)$ I/Os. Since there are D non-empty frontiers, and $\sum_\ell |L_\ell| \le n$ and $\sum_\ell |N_\ell| \le 2m$, we get the claimed I/O bound. □

7.8 Conclusion

We have shown that refining the granularity of bulk accesses to external memory can result in significantly faster algorithms for classic problems. On the theoretical side, our results give a sense of where restricting to contiguous accesses holds back performance in the EM model. On the practical side, these algorithms motivate the development of hardware protocols to support granular access. In the case of SSD NVMe [NVM21], for example, it would suffice to simply remove the 512-byte lower bound on SGL regions.

In the meantime, real-world external-memory systems involving SDRAM and/or SSD already behave closer to the general I/O model than the extreme cases of EM or GEM. It would thus be interesting to combine the GEM algorithms from this chapter with the existing EM algorithms to obtain hybrid algorithms that optimize for both B and P (with better speedup from P).

GEM algorithms currently need to explicitly manage the movement of data between external and internal memory. While this is a common way to interact with secondary memory such as SSD, interaction between cache and primary memory such as SDRAM is normally handled implicitly by the machine architecture. This idea has been formalized for the EM model by the ***cache-oblivious model*** [FLPR99, Dem02], which defines how to automatically convert individual memory-word accesses into block memory transfers, in particular via competitive cache block replacement strategies. Is a similar programming abstraction possible for GEM algorithms? And is it possible to design GEM algorithms that are efficient despite not knowing the machine model parameters B and M, as there are for the cache-oblivious/EM model?

Some of our algorithms and data structures leave room for further improvement.

On Breadth-First Search, it is not obvious that one can do better than EM or PRAM algorithms when the maximum distance D is large. The structure of BFS as well as Depth-First Search do not seem to give rise to simple graph-contraction-based recursive formulations, unlike connected components and minimum spanning forest.

For the Online Union-Find problem, the simple worst-case $O(\log n)$ bound is sufficient for our algorithms, but we do not have a better amortized bound such as $O(\alpha(n)/B)$ for queries with $o(B \log n)$ query operations and internal memory available. The problem of performing amortized analysis of batched or parallel operations is worth exploring further in other data structures as well.

Bibliography

[AAY10] Pankaj K. Agarwal, Lars Arge, and Ke Yi. I/O-efficient batched union-find and its applications to terrain analysis. *ACM Transactions on Algorithms*, 7(1):1–21, November 2010.

[ABC+23] Robert M. Alaniz, Josh Brunner, Michael Coulombe, Erik D. Demaine, Yevhenii Diomidov, Ryan Knobel, Timothy Gomez, Elise Grizzell, Jayson Lynch, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Complexity of reconfiguration in surface chemical reaction networks, 2023.

[ABD+20] Joshua Ani, Jeffrey Bosboom, Erik D. Demaine, Yevhenii Diomidov, Dylan 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.

[ABT04] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004. Publisher: Elsevier.

[ACD+22] Joshua Ani, Lily Chung, Erik D. Demaine, Yevhenii Diomidov, Dylan 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*, pages 2:1–2:30, Island of Favignana, Sicily, Italy, May–June 2022.

[ACD+23] Joshua Ani, Michael Coulombe, Erik D. Demaine, Yevhenii Diomidov, Timothy Gomez, Dylan Hendrickson, and Jayson Lynch. Complexity of motion planning of arbitrarily many robots: Gadgets, Petri nets, and counter machines, 2023.

[ADD+22] Joshua Ani, Erik D. Demaine, Yevhenii Diomidov, Dylan H. Hendrickson, and Jayson Lynch. Traversability, reconfiguration, and reachability in the gadget framework. In Petra Mutzel, Md. Saidur Rahman, and Slamin, editors, *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation*, volume 13174 of *Lecture Notes in Computer Science*, pages 47–58, Jember, Indonesia, March 2022.

[ADG+21] Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Dyl an H. Hendrickson, Adam Hesterberg, Matias Korman, Oliver Korten, Ja yson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots. In Kevin Buchin and Éric Colin de Verdière, editors, *Proceedings of the 37th International Symposium on Computational Geometry*, LIPIcs, pages 10:1–10:20, 2021.

[ADGV15] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015. Originally at FUN 2014.

[ADHL22] Joshua Ani, Erik D. Demaine, Dylan Hendrickson, and Jayson Lynch. Trains, games, and complexity: 0/1/2-player motion planning through input/output gadgets. In Petra Mutzel, Md. Saidur Rahman, and Slamin, editors, *Proceedings of the 16th International Conference and Workshops on Algorithms and Computation*, volume 13174 of *Lecture Notes in Computer Science*, pages 187–198, Jember, Indonesia, March 2022.

[AFG+22] Robert M. Alaniz, Bin Fu, Timothy Gomez, Elise Grizzell, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Reachability in restricted chemical reaction networks. *arXiv preprint arXiv:2211.12603*, 2022.

[AGM02] Hagit Attiya, Alla Gorbach, and Shlomo Moran. Computing in totally anonymous asynchronous shared memory systems. *Information and Computation*, 173(2):162 – 183, 2002.

[AM09] Deepak Ajwani and Ulrich Meyer. Design and engineering of external memory traversal algorithms for general graphs. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*, pages 1–33. Springer, 2009.

[APSV02] Lars Arge, Octavian Procopiuc, and Jeffrey Scott Vitter. Implementing I/O-efficient data structures using TPIE. In Rolf Möhring and Rajeev Raman, editors, *Proceedings of the 10th Annual European Symposium on Algorithms*, pages 88–100, 2002.

[Arg01] Lars Arge. External memory data structures. In Friedhelm Meyer auf der Heide, editor, *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 1–29, 2001.

[AV88] Alok Aggarwal and Jeffrey Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[BBC+19] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[BBC+22] Jeffrey Bosboom, Josh Brunner, Michael Coulombe, Erik D. Demaine, Dylan H. Hendrickson, Jayson Lynch, and Elle Najt. The Legend of Zelda: The Complexity of Mechanics, March 2022. arXiv:2203.17167 [cs].

[BBF+10] Michael A. Bender, Gerth Stølting Brodal, Rolf Fagerberg, Riko Jacob, and Elias Vicari. Optimal sparse matrix dense vector multiplication in the I/O-model. *Theory of Computing Systems*, 47:934–962, 2010.

[BBM11] Eric Blais, Joshua Brody, and Kevin Matulef. Property testing lower bounds via communication complexity. In *2011 IEEE 26th Annual Conference on Computational Complexity*, pages 210–220, San Jose, CA, USA, June 2011. IEEE.

[BCC+23] Josh Brunner, Lily Chung, Michael Coulombe, Erik D. Demaine, Yevhenii Diomidov, Markus Hecher, Siddhartha Jayanti, and Jayson Lynch. Granular external memory model: Breaking the shackles of contiguity for faster algorithms, 2023.

[BCD+17] 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. *Theor. Comput. Sci.*, 675:43–55, 2017.

[BDH+15] Jeffrey Bosboom, Erik D Demaine, Adam Hesterberg, Jayson Lynch, and Erik Waingarten. Mario Kart is hard. In *Japanese Conference on Discrete and Computational Geometry and Graphs*, pages 49–59. Springer, 2015.

[BGYW19] Tatiana Brailovskaya, Gokul Gowri, Sean Yu, and Erik Winfree. Reversible computation using swap reactions on a surface. In *International Conference on DNA Computing and Molecular Programming*, pages 174–196. Springer, 2019.

[BHW+13] Aaron Becker, Golnaz Habibi, Justin Werfel, Michael Rubenstein, and James McLurkin. Massive uniform manipulation: Controlling large populations of simple robots with a common input signal. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 520–527. IEEE, 2013.

[BKK21] Michael A. Bender, Bradley C. Kuszmaul, and William Kuszmaul. Linear probing revisited: Tombstones mark the demise of primary clustering. In *Proceedings of the 62nd IEEE Annual Symposium on Foundations of Computer Science*, pages 1171–1182, 2021.

[BMLC+19] Jose Balanza-Martinez, Austin Luchsinger, David Caballero, Rene Reyes, Angel A Cantu, Robert Schweller, Luis Angel Garcia, and Tim Wylie. Full tilt: Universal constructors for general shapes with uniform external forces. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2689–2708. SIAM, 2019.

[Bow11] Jack R. Bowman. *Obstruction-free Snapshot, Obstruction-free Consensus, and Fetch-and-add Modulo k*. Undergraduate Thesis, Dartmouth College, June 2011.

[BWLB21] Anton Bakhtin, David J. Wu, Adam Lerer, and Noam Brown. No-press diplomacy from scratch. *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 18063–18074, 2021.

[CCBG19] Andrea Celli, Marco Ciccone, Raffaele Bongo, and Nicola Gatti. Coordination in adversarial sequential team games via multi-agent deep reinforcement learning. *arXiv preprint arXiv:1912.07712*, 2019.

[CCG+20] David Caballero, Angel A. Cantu, Timothy Gomez, Austin Luchsinger, Robert Schweller, and Tim Wylie. Relocating units in robot swarms with uniform control signals is PSPACE-complete. *CCCG 2020*, 2020.

[CDM+17] Gourab Chatterjee, Neil Dalchau, Richard A. Muscat, Andrew Phillips, and Georg Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature nanotechnology*, 12(9):920–927, 2017.

[CDS14] Ho-Lin Chen, David Doty, and David Soloveichik. Deterministic function computation with chemical reaction networks. *Natural computing*, 13:517–534, 2014.

[CFCS18] Alex Conway, Martín Farach-Colton, and Philip Shilane. Optimal hashing in external memory. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 39:1–39:14, 2018.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[CL18] Michael J. Coulombe and Jayson Lynch. Cooperating in Video Games? Impossible! Undecidability of Team Multiplayer Games. *9th International Conference on Fun with Algorithms (FUN 2018)*, 100, 2018.

[CL20] Michael J. Coulombe and Jayson Lynch. Cooperating in video games? impossible! undecidability of team multiplayer games. *Theoretical Computer Science*, 839:30–40, 2020.

[CL22] Michael Coulombe and Jayson Lynch. Characterizing the decidability of finite state automata team games with communication. In Pierre Ganty and Dario Della Monica, editors, Proceedings of the 13th International Symposium on *Games, Automata, Logics and Formal Verification,* Madrid, Spain, September 21-23, 2022, volume 370 of *Electronic Proceedings in Theoretical Computer Science*, pages 213–228. Open Publishing Association, 2022.

[CLM+18] Cameron Chalk, Austin Luchsinger, Eric Martinez, Robert Schweller, Andrew Winslow, and Tim Wylie. Freezing simulates non-freezing tile automata. In *DNA Computing and Molecular Programming: 24th International Conference, DNA 24, Jinan, China, October 8–12, 2018, Proceedings 24*, pages 155–172. Springer, 2018.

[CM99] Andreas Crauser and Kurt Mehlhorn. LEDA-SM: Extending LEDA to secondary memory. In Jeffrey S. Vitter and Christos D. Zaroliagis, editors, *Proceedings of the 3rd International Workshop on Algorithm Engineering*, pages 228–242, 1999.

[CO22] Wojciech Czerwiński and Łukasz Orlikowski. Reachability in vector addition systems is Ackermann-complete. In *Proceedings of the 62nd Annual IEEE Symposium on Foundations of Computer Science*, pages 1229–1240, 2022.

[CQW20] Samuel Clamons, Lulu Qian, and Erik Winfree. Programming and simulating chemical reaction networks on a surface. *Journal of the Royal Society Interface*, 17(166):20190790, 2020.

[CSWB09] Matthew Cook, David Soloveichik, Erik Winfree, and Jehoshua Bruck. Programmability of chemical reaction networks. In *Algorithmic bioprocesses*, pages 543–584. Springer, 2009.

[Dem02] Erik D. Demaine. Cache-oblivious algorithms and data structures. *Lecture Notes from the EEF Summer School on Massive Data Sets*, 8(4):1–249, 2002.

[DGLR18] Erik D. Demaine, Isaac Grosof, 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.

[DH08] Erik D. Demaine and Robert A. Hearn. Constraint Logic: A Uniform Framework for Modeling Computation as Games. In *2008 23rd Annual IEEE Conference on Computational Complexity*, pages 149–162, College Park, MD, USA, June 2008. IEEE.

[DHHL22] Erik D. Demaine, Robert A. Hearn, Dylan Hendrickson, and Jayson Lynch. PSPACE-completeness of reversible deterministic systems. In *Proceedings of the 9th Conference on Machines, Computations and Universality*, pages 91–108, Debrecen, Hungary, August–September 2022.

[DHL20] Erik D. Demaine, Dylan 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*, pages 62:1–62:42, Seattle, Washington, January 2020.

[DK21] Colin Defant and Noah Kravitz. Friends and strangers walking on graphs. *Combinatorial Theory*, 1, 2021.

[DKL20] Erik D. Demaine, Justin Kopinsky, and Jayson Lynch. Recursed is not recursive: A jarring result. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation (ISAAC 2020)*, volume 181 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 50:1–50:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[DKM+94] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, August 1994.

[DKTT15] Frits Dannenberg, Marta Kwiatkowska, Chris Thachuk, and Andrew J Turberfield. DNA walker circuits: computational potential, design, and verification. *Natural Computing*, 14(2):195–211, 2015.

[DLL18] 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 13–15 2018.

[DS97] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, March 1997.

[DSSS04] Roman Dementiev, Peter Sanders, Dominik Schultes, and Jop Sibeyn. Engineering an external memory minimum spanning tree algorithm. In Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell, editors, *Proceedings of the IFIP 18th World Computer Congress on Exploring New Frontiers of Theoretical Informatics*, pages 195–208, 2004.

[Esp05] Javier Esparza. Decidability and complexity of Petri net problems – an introduction. *Lectures on Petri Nets I: Basic Models: Advances in Petri Nets*, pages 374–428, 2005.

[FAdFW16] Jakob N. Foerster, Yannis M. Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2137–2145, 2016.

[Fel22] Carlos Felicio. 512E vs 4KN NVME performance. Technology Blog post, January 2022. https://carlosfelic.io/misc/512e-vs-4kn-nvme-performance/.

[FHLS20] Alireza Farhadi, MohammadTaghi Hajiaghayi, Kasper Green Larsen, and Elaine Shi. Lower bounds for external memory integer sorting via network coding. *SIAM Journal on Computing*, pages 97–105, October 2020.

[FHS98] Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, September 1998.

[FL81] Aviezri S. Fraenkel and David Lichtenstein. Computing a perfect strategy for n x n chess requires time exponential in n. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming*, pages 278–293, Berlin, Heidelberg, 1981. Springer Berlin Heidelberg.

[FLPR99] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, pages 285–297, 1999.

[For10] Michal Forišek. Computational complexity of two-dimensional platform games. In *Proceedings of the 5th International Conference on Fun with Algorithms (FUN 2010)*, pages 214–227, 2010.

[FW78] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[GH10] Gilad Goraly and Refael Hassin. Multi-color pebble motion on graphs. *Algorithmica*, 58:610–636, 2010.

[GKSW17] Rati Gelashvili, Idit Keidar, Alexander Spiegelman, and Roger Wattenhofer. Brief announcement: Towards reduced instruction sets for synchronization. In *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91, page 53. Schloss Dagstuhl, Leibniz-Zentrum für Informatik, 2017.

[GLN14] Luciano Guala, Stefano Leucci, and Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-) hard. In *IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[GMMRW21] Eric Goles, Diego Maldonado, Pedro Montealegre, and Martín Ríos-Wilson. On the complexity of asynchronous freezing cellular automata. *Information and Computation*, 281:104764, 2021.

[GOT15] Eric Goles, Nicolas Ollinger, and Guillaume Theyssier. Introducing freezing cellular automata. In *Cellular Automata and Discrete Complex Systems, 21st International Workshop (AUTOMATA 2015)*, volume 24, pages 65–73, 2015.

[Ham14] Linus Hamilton. Braid is undecidable. *arXiv preprint arXiv:1412.0784*, 2014.

[HD09] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* A. K. Peters, Ltd., Natick, MA, USA, 2009.

[Hen21] Dylan Hendrickson. Gadgets and gizmos: A formal model of simulation in the gadget framework for motion planning. Master's thesis, Massachusetts Institute of Technology, 2021.

[Her88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing - PODC '88*, pages 276–290, Toronto, Ontario, Canada, 1988. ACM Press.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13:26, January 1991.

[HS11] Maurice Herlihy and Nir Shavit. On the nature of progress. In *Principles of Distributed Systems*, pages 313–328. Springer, 2011.

[HSLS20] Maurice Herlihy, Nir Shavit, Victor Luchangco, and Michael Spear. *The art of multiprocessor programming.* Newnes, 2020.

[HV95] S. Haldar and K. Vidyasankar. Constructing 1-writer multireader multivalued atomic variables from regular variables. *Journal of the ACM*, 42(1):186–203, January 1995.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[IP12] John Iacono and Mihai Pătraşcu. Using hashing to solve the dictionary problem (in external memory). In *Proceedings of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 570–582, January 2012.

[IPS82] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982.

[JL19] Shunhua Jiang and Kasper Green Larsen. A faster external memory priority queue with DecreaseKeys. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1331–1343. Society for Industrial and Applied Mathematics, 2019.

[JLH+19] Natasha Jaques, Angeliki Lazaridou, Edward Hughes, Caglar Gulcehre, Pedro Ortega, DJ Strouse, Joel Z. Leibo, and Nando De Freitas. Social influence as intrinsic motivation for multi-agent deep reinforcement learning. *Proceedings of the 36th International Conference on Machine Learning*, 97:3040–3049, 09–15 Jun 2019.

[KM69] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *Journal of Computer and System Sciences*, 3(2):147–195, 1969.

[Kni21] Shawn Knight. SSD shipments outpaced HDDs in 2020, but capacity still favors mechanical drives. *TechSpot*, February 2021. https://www.techspot.com/news/88645-ssd-shipments-outpaced-hdds-2020-but-capacity-favors.html.

[Kru56] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[Ler22] Jérôme Leroux. The reachability problem for Petri nets is not primitive recursive. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1241–1252. IEEE, 2022.

[LS19] Jérôme Leroux and Sylvain Schmitz. Reachability in vector addition systems is primitive-recursive in fixed dimension. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–13. IEEE, 2019.

[LTE+20] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science*, 368(6495), June 2020.

[Lyn20] Jayson Lynch. *A framework for proving the computational intractability of motion planning problems*. PhD thesis, Massachusetts Institute of Technology, 2020.

[Mil22] Aleksa Milojevic. Connectivity of old and new models of friends-and-strangers graphs. *arXiv preprint arXiv:2210.03864*, 2022.

[Mis] MisterMike. https://www.spriters-resource.com/submitter/MisterMike/.

[MN19] J. Ian Munro and Yakov Nekrich. Dynamic planar point location in external memory. In Gill Barequet and Yusu Wang, editors, *Proceedings of the 35th International Symposium on Computational Geometry*, volume 129 of *Leibniz International Proceedings in Informatics*, pages 52:1–52:15, 2019.

[MSCS13] Richard A. Muscat, Karin Strauss, Luis Ceze, and Georg Seelig. DNA-based molecular architecture with spatially localized components. *ACM SIGARCH Computer Architecture News*, 41(3):177–188, 2013.

[MY07] Kenichi Morita and Yoshikazu Yamaguchi. A universal reversible turing machine. In *Machines, Computations, and Universality: 5th International Conference, MCU 2007, Orléans, France, September 10-13, 2007. Proceedings 5*, pages 90–98. Springer, 2007.

[NVM21] NVM Express, Inc. NVM Express base specification, versions 1.0–2.0a, 2011–2021. https://nvmexpress.org/specification/nvm-express-base-specification/.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Rheinisch-Westfälischen Institutes für Instrumentelle Mathematik an der Universität Bonn, 1962.

[PLB+19] Philip Paquette, Yuchen Lu, Steven Bocco, Max O. Smith, Satya Ortiz-Gagne, Jonathan K. Kummerfeld, Joelle Pineau, Satinder Singh, and Aaron C. Courville. No-press diplomacy: Modeling multi-agent gameplay. *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 4476–4487, 2019.

[PR79] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*, pages 348–363, San Juan, Puerto Rico, October 1979. IEEE.

[PRA01] Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer noncooperative games of incomplete information. *Computers & Mathematics with Applications*, 41(7-8):957–992, April 2001.

[PRST94] Christos H Papadimitriou, Prabhakar Raghavan, Madhu Sudan, and Hisao Tamaki. Motion planning on a graph. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 511–520. IEEE, 1994.

[Pup] PuppetMaster9. Flying machine. https://www.zeldaspeedruns.com/botw/techniques/flying-machine.

[PV84] 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.

[QW14] Lulu Qian and Erik Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In *DNA Computing and Molecular Programming: 20th International Conference, DNA 20, Kyoto, Japan, September 22-26, 2014. Proceedings*, volume 8727, page 114. Springer, 2014.

[Rac78] Charles Rackoff. The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6(2):223–231, 1978.

[Rei79] John H Reif. Universal games of incomplete information. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 288–308. ACM, 1979.

[Rei84] John H Reif. The complexity of two-player games of incomplete information. *Journal of computer and system sciences*, 29(2):274–301, 1984.

[Rei21] Frederick Reiber. The crew: The quest for planet nine is np-complete. *CoRR*, 2021.

[Rob83] J. M. Robson. The complexity of Go. *Proc. 9th World Computer Congress on Information Processing, 1983*, pages 413–417, 1983.

[Rob84] J. M. Robson. N by N Checkers is Exptime Complete. *SIAM Journal on Computing*, 13(2):252–267, May 1984.

[roc] rocktyt. Cave of flames. https://www.vgmaps.com/Atlas/GBA/index.htm#LegendOfZeldaMinishCap.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[Sch78] Thomas J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16(2):185–225, April 1978.

[SCWB08] David Soloveichik, Matthew Cook, Erik Winfree, and Jehoshua Bruck. Computation with finite stochastic chemical reaction networks. *natural computing*, 7:615–633, 2008.

[SRG17] Matthew Stephenson, Jochen Renz, and Xiaoyu Ge. The computational complexity of angry birds and similar physics-simulation games. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 13, pages 241–247, 2017.

[SSW10] David Soloveichik, Georg Seelig, and Erik Winfree. DNA as a universal substrate for chemical kinetics. *Proceedings of the National Academy of Sciences*, 107(12):5393–5398, 2010.

[STTW16] Natcha Simsiri, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Work-efficient parallel union-find with applications to incremental graph connectivity. In Pierre-François Dutot and Denis Trystram, editors, *Euro-Par 2016: Parallel Processing*, volume 9833 of *Lecture Notes in Computer Science*, pages 561–573. Springer International Publishing, Cham, 2016.

[TLJ+17] Anupama J. Thubagere, Wei Li, Robert F. Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, et al. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

[TO22] Guillaume Theyssier and Nicolas Ollinger. Freezing, bounded-change and convergent cellular automata. *Discrete Mathematics & Theoretical Computer Science*, 24, 2022.

[Trä13] Jesper Larsson Träff. A note on (parallel) Depth- and Breadth-First Search by arc elimination. arXiv:1305.1222, November 2013. http://arXiv.org/abs/1305.1222.

[TvL84] Robert E. Tarjan and Jan van Leeuwen. Worst-case Analysis of Set Union Algorithms. *Journal of the ACM*, 31(2):245–281, March 1984.

[VBC+19] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/, 2019.

[VDZB15] Tom C Van Der Zanden and Hans L Bodlaender. PSPACE-completeness of Bloxorz and of games with 2-buttons. In *International Conference on Algorithms and Complexity*, pages 403–415. Springer, 2015.

[Vid21] Video Game Sales Wiki. The Legend of Zelda. https://vgsales.fandom.com/wiki/The_Legend_of_Zelda, 2021.

[Vig14] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

[Vit01] Jeffrey Scott Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

[Wik22a] Wikipedia. Synchronous dynamic random-access memory, 2003–2022. https://en.wikipedia.org/wiki/Synchronous_dynamic_random-access_memory.

[Wik22b] Wikipedia. Solid-state drive, 2006–2022. https://en.wikipedia.org/wiki/Solid-state_drive.

[WN09] Damien Woods and Turlough Neary. The complexity of small universal turing machines: A survey. *Theoretical Computer Science*, 410(4-5):443–450, 2009.

[Zel21a] Zelda Dungeon. Category:items. https://www.zeldadungeon.net/wiki/Category:Items, 2021.

[Zel21b] Zelda Wiki. Category:items. https://zelda.fandom.com/wiki/Category:Items, 2021.