Simpler hardness proofs via gadget frameworks

by

Della Hendrickson

B.S., Massachusetts Institute of Technology (2019)
M.S., Massachusetts Institute of Technology (2021)

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

May 2026

Authored by: Della Hendrickson
Department of Electrical Engineering and Computer Science
April 7, 2026

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

Simpler hardness proofs via gadget frameworks

by

Della Hendrickson

Submitted to the Department of Electrical Engineering and Computer Science
on April 7, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

ABSTRACT

In this thesis, I consider the general notion of a 'gadget framework', which is roughly a family of hard problems that are useful as sources of reductions, where such reductions consist of implementations of a set of 'gadgets', and especially when there is a notion of 'simulation' between gadgets. I describe five gadget frameworks that I have helped invent or develop.

First, I present the history of the motion-planning gadgets framework, which was designed for proving hardness of problems like video games. Next, I describe two gadget frameworks which provide zero-player analogs of motion-planning gadgets: input/output gadgets, which represent a train moving through a network of rails and switches, and a framework designed for proving PSPACE-hardness of reversible deterministic systems. Fourth, I describe a gadget framework developed for Minesweeper and similar limited-information puzzles, which is used to prove hardness of three natural decision problems about such puzzles. Finally, I describe T-metacells, a framework for proving NP- and ASP-hardness of logic puzzles involving drawing a loop.

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

Acknowledgments

I thank my advisor, Erik Demaine, for his support and endless patience throughout my PhD program. Much of my research life these past years has been in Erik's research-based classes, which some of us affectionately call "fun with Erik Demaine". This has greatly helped me connect to problems and collaborators, and provided an atmosphere and community that has been wonderful to do research in. I thank the hundreds of people who have helped create or been a part of this environment.

I thank Jayson Lynch for originally introducing me to motion-planning gadgets, and theoretical computer science research more broadly, and for their continued friendship. I thank my other closest collaborators, Hayashi Layers and Jenny Diomidova, for joining me in many of the explorations that led to this thesis. I thank my many other collaborators and coauthors: Adam Hesterberg, Adam Suhl, Andy Tockman, Jeffrey Bosboom, Josh Brunner, Lily Chung, Michael Coulombe, Zach Abel, and others too numerous to list.

Finally, I thank my partners, Andy Tockman and Jordan Hines, for supporting me over the years, both in my journey as a graduate student and in all other parts of my life.

Contents

List of Figures 11

List of Tables 15

1 Introduction **17**
- 1.1 Prior work . . . 18
- 1.2 Outline . . . 20

2 The Story of Motion-Planning Gadgets **23**
- 2.1 The preparadigmatic era . . . 23
- 2.2 The metatheorem era . . . 28
- 2.3 The state machine era . . . 30
- 2.4 The gizmo era . . . 36

3 Input/Output Gadgets **41**
- 3.1 Introduction . . . 41
  - 3.1.1 Gadget framework . . . 42
  - 3.1.2 Classifying Output-Disjoint Deterministic 2-State Gadgets . . . 43
  - 3.1.3 Our Results: Complexity . . . 45
  - 3.1.4 Our Results: Simulation . . . 47
- 3.2 Zero Players . . . 50
  - 3.2.1 Single Input . . . 50
  - 3.2.2 Bounded Gadgets . . . 51
  - 3.2.3 Unbounded gadgets . . . 54
- 3.3 Simulation . . . 61
  - 3.3.1 Universality . . . 61
  - 3.3.2 Closure . . . 66
- 3.4 One Player . . . 68
  - 3.4.1 Containment in NP . . . 69
  - 3.4.2 NP-hardness . . . 71
- 3.5 Two Players . . . 73
- 3.6 Applications . . . 74
  - 3.6.1 Trainyard . . . 75
  - 3.6.2 [the Sequence] . . . 77
  - 3.6.3 Factorio Trains . . . 80
  - 3.6.4 Factorio Transport Belts . . . 85

4 Reversible Deterministic Gadgets **91**
4.1 Introduction . . . 91
4.2 The framework . . . 93
4.2.1 Required gadgets . . . 94
4.2.2 PSPACE-hardness . . . 94
4.3 Deterministic Constraint Logic . . . 102
4.3.1 Issue with existing proof . . . 102
4.3.2 PSPACE-hardness . . . 103
4.4 Locking 2-toggles . . . 105
4.5 3-spinners . . . 106
4.6 Billiard balls . . . 107

5 Graph Orientation Gadgets and Minesweeper **113**
5.1 Introduction . . . 113
5.2 Prior work and definitions . . . 114
5.2.1 Consistency . . . 114
5.2.2 Inference . . . 115
5.2.3 Solvability . . . 117
5.3 The Minesweeper gadget framework . . . 118
5.3.1 Gates as gadgets . . . 120
5.3.2 Decision problems . . . 121
5.3.3 Hardness . . . 123
5.3.4 Simulation . . . 126
5.3.5 Simpler gadget sets . . . 129
5.3.6 Applying the framework . . . 131
5.4 Hardness proofs . . . 134
5.4.1 Vanilla clues . . . 137
5.4.2 Solvability from an empty board . . . 137
5.4.3 Cross clues . . . 138
5.4.4 Mini-cross clues . . . 139
5.4.5 Eyesight clues . . . 140
5.4.6 Wall and partition clues . . . 140

6 Grid Graphs and T-Metacells **143**
6.1 Introduction . . . 143
6.1.1 Our results . . . 144
6.2 Connections between problems . . . 146
6.3 ASP-completeness of TRVB . . . 152
6.4 Hamiltonian cycles in grid graphs . . . 153
6.4.1 Rectangular grid graphs . . . 154
6.4.2 Max-degree-3 spanning subgraphs of rectangular grid graphs . . . 156
6.4.3 Max-degree-3 grid graphs . . . 158
6.5 T-metacells . . . 160
6.6 Applications . . . 162
6.6.1 Loop-drawing paper-and-pencil logic puzzles . . . 163

6.6.2 Prior ASP-completeness results 164
6.6.3 Prior NP-hardness improved to ASP-completeness 164
6.6.4 New NP- and ASP-completeness Results 170
6.6.5 Open ASP-completeness Questions 179

References 180

List of Figures

2.1 An early simulation . . . 26
2.2 The door, abstractly . . . 30
2.3 The Set-Verify gadget . . . 31
2.4 Locking 2-toggle state diagram . . . 33
2.5 [Self-closing] door state diagrams . . . 34
2.6 [Doubly covered] 1-toggle state diagrams . . . 36

3.1 The switch/set-up line/set-down line . . . 42
3.2 Bounded multi-input input/output gadgets . . . 47
3.3 Unbounded multi-input input/output gadgets . . . 48
3.4 Compressing a switch . . . 49
3.5 A graph modification . . . 51
3.6 Reduction for NL-hardness of the toggle switch . . . 52
3.7 A NOR gate from switch/set-up lines . . . 53
3.8 Tunnel duplicator schematic . . . 55
3.9 Tunnel duplicator for switch/set-up line/set-down line . . . 55
3.10 Universal quantifier for switch/set-up line/set-down line . . . 56
3.11 CNF evaluation for switch/set-up line/set-down line . . . 57
3.12 Tunnel duplicator for toggle switch/toggle switch . . . 58
3.13 Simulating more toggle switches . . . 58
3.14 Simulating a switch/set-up line/set-down line with toggle switch/toggle switches 59
3.15 Simulating a toggle switch/toggle switch with toggle switch/toggle lines . . . 59
3.16 Tunnel duplicator for switch/toggle line . . . 59
3.17 Tunnel duplicator for set-up switch/toggle line . . . 60
3.18 Simulating a switch/toggle line with set-up switch/toggle lines . . . 60
3.19 Simulating a set-down switch/toggle line with set-up switch/set-down lines . 60
3.20 Simulating a set-up line/set-down switch with set-up line/toggle switches . . 61
3.21 Simulating a 4-switch with switch/set-up line/set-down lines . . . 62
3.22 Simulating more copies of the switch . . . 63
3.23 Infinite tunnel duplicator . . . 65
3.24 Simulating an ω-switch with switch/set-up line/set-down lines . . . 65
3.25 Simulating an ω^2-switch with ω-switches . . . 65
3.26 3SAT reduction for the set-down switch . . . 72
3.27 Simulating single-input gadgets . . . 73
3.28 Trainyard gadget state diagram . . . 76

3.29 Reverse branch state diagram . . . 77
3.30 Simulating a reverse branch with Trainyard gadgets . . . 78
3.31 Simulating a toggle switch/toggle line with Trainyard gadgets . . . 78
3.32 [the Sequence] modules . . . 79
3.33 [the Sequence] turn and fan-in . . . 80
3.34 [the Sequence] switch/set-up line/set-down line . . . 81
3.35 [the Sequence] win gadget . . . 82
3.36 Factorio train objects . . . 83
3.37 Factorio trains wire . . . 83
3.38 Factorio trains fan-in . . . 84
3.39 Factorio trains crossover . . . 84
3.40 Factorio trains win gadget . . . 85
3.41 Factorio trains switch/set-up line/set-down line . . . 86
3.42 Factorio transport belt example . . . 87
3.43 Factorio 0.15 transport belt toggle switch/toggle switch . . . 88
3.44 Factorio 0.16 transport belt toggle switch/toggle switch . . . 89
3.45 Factorio transport belt grouper . . . 89

4.1 Structure of reduction for reversible deterministic gadgets . . . 96
4.2 CNF evaluation for reversible deterministic gadgets . . . 98
4.3 Universal quantifier for reversible deterministic gadgets . . . 99
4.4 Order relation for edges in quantifier gadgets . . . 99
4.5 A/BA crossover for Deterministic Constraint Logic . . . 104
4.6 Switch for Deterministic Constraint Logic . . . 104
4.7 Locking 2-toggle state diagram . . . 105
4.8 Switch and Reversible Fan-in for locking 2-toggles . . . 106
4.9 3-spinner state diagram . . . 107
4.10 Reversible deterministic gadgets for 3-spinners . . . 107
4.11 Billiard ball model . . . 108
4.12 Wire for billiards . . . 109
4.13 Switch for billiards . . . 109
4.14 Switch for billiards, in action . . . 110
4.15 Reversible Fan-In for billiards . . . 111

5.1 A consistent Minesweeper puzzle . . . 115
5.2 An inconsistent Minesweeper puzzle . . . 115
5.3 A Minesweeper puzzle with an inference . . . 115
5.4 A Minesweeper puzzle without an inference . . . 116
5.5 Prior OR gate for Minesweeper . . . 117
5.6 A solvable Minesweeper puzzle . . . 118
5.7 Reduction coNP-hardness of PGO promise-inference . . . 124
5.8 Reduction for coNP-hardness of PGO uniqueness . . . 125
5.9 Summary of graph orientation simulations . . . 129
5.10 Simulating a NOR gate with base gadgets . . . 131
5.11 Connecting tile-based Minesweeper gadgets . . . 137

6.1 Relationships between Hamiltonicity and other problems 148
6.2 Bijections for proving equivalence of problems 149
6.3 Simulating a degree-4 unbreakable TRVB vertex 153
6.4 Simulating a degree-4 unbreakable TRVB vertex with degree-6 153
6.5 Example of reduction from TRVB to Hamiltonian cycle 154
6.6 TRVB gadgets for directed rectangular grid graphs 155
6.7 Filling holes in directed grid graphs 156
6.8 Example of filled holes 156
6.9 Example for max-degree-3 157
6.10 Example for undirected 158
6.11 TRVB gadget for undirected max-degree-3 spanning subgraphs 159
6.12 Vertex gadget for ASP-hardness in grid graphs 159
6.13 TRVB gadget for undirected max-degree-3 grid graphs 160
6.14 TRVB gadget for directed max-degree-3 grid graphs 161

List of Tables

3.1 Subunits of input/output gadgets 44
3.2 Results for input/output gadgets with multiple inputs 46
3.3 Results for input/output gadgets with one input 46

4.1 Time-reversal symmetric gadgets for PSPACE-hardness 95

5.1 Graph orientation gadgets 119

6.1 Complexity of Hamiltonian cycle in various restricted grid graphs 145
6.2 Hardness results for pencil-and-paper puzzles 146

Chapter 1

Introduction

One of the first tools any student of complexity theory learns to use is the gadget [Tut54, GJ02]. Gadgets conceptually simplify a hardness proof by providing an abstraction for portions of the reduction. In a reduction from 3SAT, you will likely describe 'variable gadgets' and 'clause gadgets', and in a reduction from Hamiltonian Path you will likely describe 'vertex gadgets' and 'edge gadgets'.

In a *gadget framework*, there are various gadgets you might be able to construct, and certain combinations of them that are sufficient for hardness. For example, when reducing from 3SAT to a planar problem, it suffices to build a variable gadget, a clause gadget, and a crossover gadget, and explain how to connect them (possibly using a fanout gadget).

There are several advantages to approaching hardness proofs in this way. First, we have a clear list of subtasks. This allows us to consider each gadget in isolation instead of holding the entire reduction in our minds, provided we ensure gadgets communicate with each other appropriately.

Next, there are often several sufficient combinations of gadgets. If, in an attempt to build a clause gadget for 3SAT, I instead stumble upon a gadget which forces *exactly* one of its inputs to be true, that's okay! We can just reduce from 1-in-3SAT instead [Sch78]. In practice, we often choose only a 'genre' of problem to try reducing from—e.g. "SAT variant" or "Hamiltonicity variant"—and play with gadgets, hoping to find enough to assemble a reduction from an appropriate problem.

Finally, the gadget abstraction layer, which sits between the source problem and the target problem, provides space for insight and simplification. The reduction from 3SAT to 1-in-3SAT can be elegantly presented as a 'simulation' of a 3SAT clause using variables and 1-in-3 clauses [Sch78]. It follows that a hardness proof that proceeds by constructing variable gadgets and 1-in-3 clause gadgets can be thought of as a reduction from 3SAT: if you can build these two gadgets in your target problem, the simulation tells you how to use them to build a 3SAT clause gadget. But we can abstract this away, by noting that the simulation gives us a new gadget set that suffices for hardness. This is particularly useful when we have simulations between gadgets which don't on their own suffice for hardness, since we can replace one gadget in a sufficient set without thinking about the others.

In the same way, we can build a crossover out of variables and 3SAT clauses [Lic82]. Since we're already building variable gadgets and clause gadgets anyway, this lets us reduce the set of gadgets necessary to prove hardness of a planar problem: we don't need a crossover

anymore. By working purely with abstract gadgets, we have made a discovery that will simplify all future hardness proofs of this form. This amounts to proving that Planar 3SAT is NP-complete, providing us with a new source of reductions.

Together, the simulation of the crossover and the simulation of the 1-in-3 clause (which is planar) prove hardness for Planar 1-in-3SAT [MR08]. At this point, we can prove hardness for a planar problem by constructing a 1-in-3 clause gadget and sufficient variable/negation/fanout gadgets. But we can simplify the latter half as well: a degree-3 fanout 'clause' (which forces three variables to be equal) and a degree-2 negation 'clause' (which forces two variables to be different) suffice. So to design a reduction from Planar 1-in-3SAT, we need only design three gadgets (1-in-3 clause, degree-3 fanout, and degree-2 negation) and 'wires' that connect them.

In this thesis, I consider the general concept of gadget frameworks, of which these SAT-related gadgets are an example. More generally, a gadget framework provides a strategy for constructing a reduction, usually for the purpose of proving hardness for a complexity class (most often NP or PSPACE). The term eludes a precise definition, but a good gadget framework will typically

- offer some sets of gadgets which are sufficient for hardness,
- be abstract or simple enough to be applicable to a variety of target problems, but specific enough to be a good fit for those problems, and
- have a notion of "simulation" that is related to reductions in the natural way.

From one perspective, every decision problem is a gadget framework, where constructing the gadget amounts to reducing from the problem. But some decision problems, and some families of similar decision problems, are more useful than others. Perhaps a better definition is that a gadget framework is a decision problem (or family of problems) whose primary purpose is to serve as an intermediate for reductions.

1.1 Prior work

A significant portion of the results of computational complexity theory can be viewed from the lens of gadget frameworks. I now give an overview of some of these frameworks.

Degree constrained subgraphs. The first application of the idea of gadgets, as described by Szabó [Sza09], was to the problem of finding an *f-factor* of a graph G, which is a subgraph such that each vertex v has degree $f(v)$. Tutte [Tut54] described what we now understand as a reduction from finding f-factors to finding 1-factors (which are perfect matchings), by showing how to replace each vertex of G with a gadget such that an f-factor of G corresponds to a 1-factor of the new graph. This result is not directly about hardness or complexity theory; its purpose was to adapt a simple characterization of graphs with 1-factors to an analogous characterization for f-factors.

In the General Factor Problem (GFP), each vertex v has a set of allowed degrees B_v, and we are interested in a subgraph in which the degree of v is in B_v. It is natural to ask when

it's possible to build a gadget using vertices with a limited family of sets that simulates a vertex with some other set; for example, Tutte's gadget [Tut54] shows that $\{1\}$ can simulate $\{k\}$. A well understood class of simulations is those of GFP vertices from Edge and Triangle Partitioning Problem (ETPP), a related problem which asks us to partition the vertices of a graph into edges and specified triangles. Loebl [Loe93] gives a full characterization of GFP vertices that can be simulated by ETPP. Each such simulation induces a reduction from GFP with a specific vertex type to ETPP; since there is a polynomial-time algorithm for ETPP [CHP82], this proves that the corresponding special case of GFP is also in P.

SAT variants. The primary example thus far, SAT variants, has a sprawling body of results going back to the beginning of complexity theory [Kar72, Sch78]. As a small sample:

- We understand which clause types (e.g. 1-in-3 or Not All Equal) yield NP-hardness, with a general characterization [Sch78]. Moreover, this can be seen as a universal simulation result: Schaefer proves hardness by showing that clause types can simulate every relation.
- Planar variations, from Planar 3SAT [Lic82] to Linked Monotone 3SAT, Sided Linked 3SAT [Pil19], and Monotone Rectilinear 3SAT [dBK10, BCC+23], are hard, including with different clause types [MR08, Pil19] and a dichotomy analogous to Schaefer's [KKR18].
- There are complexity results in many cases when the number of instances of each variable or literal is bounded [DDD18].
- Many of these results and characterizations have analogs for various two-player games, giving completeness for complexity classes including PSPACE and Σ_2^{P} [Sch78, Pil19, Lic82].

Each of these families of results can be thought of as providing different sets of gadgets sufficient for hardness: alternative clause gadgets, simpler crossovers or fewer planar arrangements, simpler fanouts, and two-player variables. Many of these results are proved by considering simulations between gadgets, especially clause types. This is particularly convenient when there are multiple decision problems with similar notions of simulation, such as one-player SAT and two-player quantified boolean formulas (QBF). In this case, a simulation induces multiple reductions, which can give disparate hardness results, such as Schaefer's dichotomies for SAT and for QBF [Sch78].

For a thorough survey of results in this space, see Filho's thesis [TFAF19].

Hamiltonicity. Finding Hamiltonian cycles is another classic NP-complete problem, going all the way back to Karp [Kar72]. Depending on the exact problem you want to reduce to, you may find a specific version of Hamiltonicity most convenient. Fortunately, a wide variety are NP-complete, including

- finding Hamiltonian paths instead of cycles, including when one or both endpoints is fixed [GJ02];

- directed or undirected graphs [Kar72];
- restricted graph types, such as planar graphs [GJT76] and even max-degree-3 grid graphs (which are planar and bipartite) [PV84].

The 'gadgets' here are vertices: in a problem with an appropriate global connectivity constraint, to prove hardness it suffices to construct an appropriate collection of vertex gadgets which each must be visited exactly once, and which can be connected into an appropriate graph. It is reasonable to consider this framework to encompass related problems that aren't quite Hamiltonicity, such as the Travelling Salesman Problem [GJ02], Minimum Steiner Tree [PV84], and Tree Residue Vertex Breaking (TRVB) [DR18]—though one could argue that at least TRVB deserves to be considered a gadget framework in its own right. Hamiltonicity and its friends have been used many times to prove other problems NP-hard [GJ02, Tan22, Yat00], and I'll discuss below an extension of this framework designed specifically for pencil-and-paper logic puzzles involving drawing a loop.

Constraint Logic. Hearn and Demaine's *Constraint Logic* [HD09] is concerned with directed graphs with weighted edges, where the total weight of edges directed towards a vertex must always be at least some bound. There are natural games (which beget natural decision problems) to play with such a graph, parameterized by allowed vertex and edge types, player count, and whether edges can be flipped multiple times.

The 'gadgets' are types of vertices, which have a natural notion of simulation. Hearn and Demaine primarily consider one particularly useful set of two vertex types, find some other vertices they can simulate (including, for some games, a crossover) and prove completeness for the appropriate complexity class of each game with those two vertices.

Constraint Logic has been used countless times as an intermediate problem for hardness proofs [DHH02, BB12, DHL20, BCD+20], and more recent work has explored other aspects of the framework, including hardness on constant-width graphs [vdZ15] and the complexity of counting the number of legal configurations [GBD+24].

1.2 Outline

In this thesis, I present several gadget frameworks that I have had a hand in developing.

Motion-planning. The first is the motion-planning gadget framework, which has been my largest research area [DHL20, BDD+20, DDH+23, Hen21, CDD+22, CDD+23, GBC+25]. This framework was inspired by hardness proofs of video games [Vig14, ADGV15, DVW16], and has been developed over the past decade into a powerful set of tools for studying the complexity of situations with an agent navigating and interacting with their environment [Lyn20, Lay23, BBC+23]. In Chapter 2, I explain how motion-planning gadgets have transformed a corner of complexity theory, drastically simplifying prior results, accelerating new ones, and enabling techniques to prove hardness of games for which that was previously intractable.

Next, there are two gadget frameworks that can be described as special cases of the motion-planning gadget framework, but have a unique flavor and are typically applicable to different kinds of problems than general motion-planning gadgets. Both are designed to model zero-player (fully deterministic) systems, whereas motion-planning gadgets are primarily used for one-player games.

Input/output. The first of these, which I discuss in Chapter 3, is input/output gadgets [DHLL23]. This framework models situations where a signal arrives at an input of a gadget and is deterministically sent to an output, based on the gadget's state. I began this line of inquiry while wondering about the complexity of trains in the video game Factorio. Using mostly occupied tracks, with a single 'hole' where there's no train, I found a way to use the movement of the hole to control the position of another train, which in turn controls the behavior of the hole. We now understand this as the "set-up/set-down/switch" gadget, and it inspired us to consider a family of similar gadgets, which we have fully characterized: all of them are either 'obviously' easy, or suffice for PSPACE-hardness and can simulate every input/output gadget. This enabled a handful of additional hardness proofs, including for the conveyor belts from the same game.

Reversible deterministic. The second framework models reversible deterministic systems, and especially systems symmetric under time-reversal [DHHL23], and is the topic of Chapter 4. This work was inspired by two things:

1. A year earlier, I had caught a subtle error in Demaine and Hearn's [HD09] proof for zero-player Constraint Logic (which is reversible and deterministic): certain gadgets weren't sending quite the combinations of signals that other gadgets were expecting.

2. I stumbled upon a 2011 paper by Tsukiji and Hagiwara [TH11] proving PSPACE-hardness of another reversible deterministic system based on Langton's ant, and noticed that the framework they used might be applicable more generally.

I happened to be at a workshop with both Demaine (my advisor) and Hearn when I encountered this paper, so it was the perfect time to fix the error. I simplified the framework a bit, since we didn't care about the geometric constraints that Tsukiji and Hagiwara were working under. The main result is a set of three fairly simple gadgets which suffice for PSPACE-hardness. Time-reversal symmetry heavily restricts the ways gadgets can interact with each other, so it is particularly convenient to have the framework handle this so that you don't have to think about it while working on such a system.

We then used the new framework to prove PSPACE-hardness of zero-player Constraint Logic and a few problems of a similar flavor. An interesting aspect of this is the connection between gadget frameworks: the new framework provided another abstraction layer, sitting between QBF and Constraint Logic, which made it easier to prove hardness of the latter. Hence a proof based on deterministic Constraint Logic is, in some sense, actually using (at least) three distinct gadget frameworks. The reduction can ultimately be factored

$$\text{QBF} \to \text{reversible deterministic gadgets} \to \text{Constraint Logic} \to \text{your problem.}$$

Lastly, there are two gadget frameworks which are unrelated to motion-planning gadgets.

Minesweeper. The first is specialized for Minesweeper, which is particularly interesting from a complexity perspective because there are multiple natural decision problems due to hidden information. The simplest, *consistency*, has been known NP-complete since 2000 [Kay00]. The next, *inference*, had two claims of coNP-completeness in the literature [SSvR11, TB22], but both proofs have a subtle error that occasionally allows an unintended inference. The third, *solvability*, had not been studied before, but I believe it best captures the game of Minesweeper.

In our quest to resolve the complexity of inference and solvability, we developed a gadget framework [GHT24] that went through at least three significant renovations, each one refining the relationship between Minesweeper and the gadgets and simplifying both the requirements for proving hardness and the proofs that the reductions are correct. The result is a framework of graph orientation gadgets on a grid, where constructing 'silent' 'parsimonious' implementations of five simple gadgets is sufficient to prove hardness of all three decision problems. This allows us to easily prove hardness for Minesweeper, as well as a long list of Minesweeper variants with the same gradual-reveal mechanics but different rules. I discuss this framework in Chapter 5.

T-metacells. The final gadget framework builds upon the Hamiltonicity framework, and is covered in Chapter 6. It was first developed by Tang [Tan22] to enable extremely simple NP-hardness proofs for Nikoli-style logic puzzles involving drawing a loop, requiring only a single gadget, called a T-metacell. Copies of the T-metacell are rotated and placed in a grid to construct an instance of an NP-hard special case of finding Hamiltonian cycles.

In the process of proving more logic puzzles hard, we found this framework rather useful, and extended it in two ways [GBC+24]: first, using a recent result that finding Hamiltonian cycles in directed cubic planar graphs is ASP-complete [BCC+20], we obtained ASP-completeness for finding Hamiltonian cycles in grid graphs, allowing the framework to prove ASP-completeness. This involved a back-and-forth between the Hamiltonian cycle gadget framework and the Tree Residue Vertex Breaking [DR18] gadget framework, and we discovered an especially tight equivalence between these frameworks on 3-regular planar graphs.

Second, we found versions of the T-metacell framework that work for directed loops or use a restricted kind of T-metacell, providing more options when applying the framework. We used all of this to improve many of Tang's [Tan22] results to ASP-completeness and establish NP- and ASP-completeness of several new logic puzzles.

Chapter 2

The Story of Motion-Planning Gadgets

In this chapter, I present the history of the motion-planning gadget framework, roughly divided into 'eras' based on the level of formalization and systemization of the framework. In my experience, this story is representative of a common pattern: a gadget framework begins as a collection of unrelated hardness reductions which share some structure. Somebody notices the shared structure, and perhaps states an informal 'metatheorem', which provides a template for reductions when you have access to certain types of behavior. Eventually, the metatheorem becomes more precise, typically in the form of an intermediate problem that serves as a reduction source. Finally, this problem is generalized, other instances of it are studied, and a notion of 'simulation' is introduced to simplify reductions, resulting in a fully fledged gadget framework.

Motion-planning gadgets serve as an illustrative example for two reasons. First, they remained in each era for a significant amount of time, allowing us to see plenty of examples of papers produced at each stage. The other gadget frameworks in this thesis progressed much faster (sometimes in the course of writing a single paper), in part due to the experience I and my collaborators gained from motion-planning gadgets and in part because some portions of the frameworks could be directly lifted from motion-planning gadgets. Second, the motion-planning gadget framework has gone through multiple levels of formalization—first in terms of state machines, and later as 'gizmos'—so we are able to see some dynamics play out multiple times.

2.1 The preparadigmatic era

To the best of my knowledge, the story of motion-planning gadgets begins in 1988,[1] when Wilfong [Wil88] proved that determining whether an agent in a polygonal environment with movable polygonal obstacles can reach a goal position is NP-hard. To do so, he showed how to build four basic 'components' called the 'switch', 'blocker', 'orderer', and 'intersection', and then used these components to build an instance corresponding to any 3SAT formula. He

[1] There were earlier hardness proofs for similar problems such as the Warehouseman's Problem [HSS84], but the proofs don't seem related to motion-planning gadgets.

doesn't use the word 'gadget', but Wilfong's components are motion-planning gadgets in the modern sense,[2] and he places the abstraction layer exactly where we do today: separating the construction of simple gadgets within the particular game from the simulation of a reduction source like 3SAT using the gadgets.

Block-pushing terminology. The next several results under discussion are for games involving an agent on a square grid with some movable blocks. There are many variations on this idea, and there is some standardization in naming them (see e.g. [DHH02]),[3] which I describe (some of) here. A typical name is *Push-3F*. Here *Push-3* means the agent can push a row (or column) of up to 3 movable blocks, and *F* means the game also contains **f**ixed blocks, which can't be moved. Similar to F, *W* means the game can have thin **w**alls, blocking movement between two squares. *Push-*$*$ means there is no limit to the number of blocks the agent can push at a time. *Pull-k* is analogous to Push-*k*, but it describes the number of blocks the agent can pull at a time. *PushPush* (based on the Macintosh game *Push-Push*) is similar to Push, but pushed blocks slide until they hit a (movable or fixed) block or the edge of the grid. Usually, the goal is for the agent to reach a goal location (so the decision problem is whether this is possible), but in the *storage* version, the goal is to place a movable block on every given storage location. The suffix *X* means that the agent is not allowed to visit a tile more than once.

In the 1990s, inspired by Wilfong's work and by block-pushing mechanics in video games, there were more hardness results of the same flavor. Inspired by the Macintosh game Beast, Dhagat and O'Rourke [DO92] proved NP-hardness[4] of Push-$*$F (we now know this problem is PSPACE-complete [CDD+22]), again by constructing some simple gadgets[5] and reducing from 3SAT. Dor and Zwick [DZ99] considered the game Sokoban, which is storage Push-1F. They proved Sokoban NP-hard, though this proof involves blocks being pushed between gadgets and thus doesn't cleanly fit into the modern framework. More importantly for us, Dor and Zwick also consider Push-$*$Pull-1F, where the agent can push any number of movable blocks and pull only one. In this game, they construct a very similar set of gadgets to the ones Wilfong [Wil88] used,[6] and combine them to reduce from 3SAT in exactly the same way.

The same paper by Dor and Zwick [DZ99] has another proof of interest to this story: they prove PSPACE-hardness for a version of Push-$*$Pull-1F where movable blocks are 1×2 rectangles. This was the first PSPACE-hardness result using motion-planning gadgets, but it followed the same basic strategy: construct some simple gadgets, then use them to build a reduction. The two main gadgets used are what we now call the *door* and the *self-closing*

[2]They don't all have standard names, but they are (respectively) some unprotected locking 2-toggle, the directed crumbler, a distant opening gadget, and a particular weak crossover. With modern tools, it is not hard to show that the switch alone is sufficient for PSPACE-hardness by simulating a locking 2-toggle [DHL20].

[3]Of course, the earliest authors didn't use this notation, and the notation has evolved over time, including the use of 'F'.

[4]The authors assert NP-*completeness*, but only prove hardness.

[5]The directed crumbler, a distant closing gadget, and an elaborate weak crossover.

[6]They build a diode instead of a directed crumbler, and the other gadgets differ subtly in ways not relevant to the reduction.

door (see Fig. 2.5), and will be major players shortly. This paper, published in 1999, was the first to incorporate several key aspects of the gadget framework we know today:

- It has the first constructions at the level of the gadget abstraction: using (motion-planning) gadgets to build other gadgets (see Fig. 2.1 for an example). This became the idea of *simulation*.
- Dor and Zwick complete the PSPACE-hardness proof by simulating a Turing machine directly, which today would feel like showing off—more recent similar proofs, including ones with the same gadgets [Vig14, ADGV15], generally use a more approachable problem like QBF. In this case, a direct Turing machine was practical because the authors found a way to simulate any gadget they wanted using self-closing doors, and so the gadget that represents one cell of a Turing machine was as easy as any other. Unfortunately, they didn't have the terminology to express this result about simulations, and it wouldn't appear explicitly for twenty years [BDD+20].
- Finally, this was the first paper to use the word 'gadget' for a component of a reduction to a motion-planning problem. Complexity theorists had long used this word in a similar way more generally, but prior authors in the more narrow space had used a more generic word like 'component'.

Dor and Zwick's work was quickly adapted by Culberson [Cul97], who made the PSPACE-hardness proof work for Sokoban (storage Push-1F) itself. Structurally, it's the same story: he found a way to build the self-closing door, and used it to simulate a Turing machine. Hartline and Libeskind-Hadas [HLH03] applied the same technique to Lunar Lockout. The construction of the gadgets looks quite different, but after building them the rest of the proof is essentially identical.

Much of the power of a gadget framework comes from its ability to simplify hardness proofs, and proofs from before the framework is crystallized are a good demonstration. As a simple example, the sequence of proofs just described can be simplified by merely removing the construction of a Turing machine from the self-closing door, which they all repeat. Of course, this reduction must be carried out once, but a well-designed gadget framework allows it to be reused efficiently.

For a more spectacular example of this simplification, we move away from door gadgets and PSPACE-hardness. The study of block-pushing complexity initiated by Dhagat and O'Rourke [DO92] and Dor and Zwick [DZ99] had branched, and we now consider the parallel line of research proving NP-hardness of ever more block-pushing games. Demaine et al. [DDO00] demonstrates the sorts of gadgets this work tended to use: *one-way* tunnels, which must first be used in a given direction; *forks*, which allow the player to reach only one of two exits; *NAND* gadgets, which allow the player to use only one of two parallel tunnels, and so on. Unlike doors, these gadgets only 'do anything interesting' a finite number of times before becoming impassible or simple tunnels—this property was later formalized as 'LDAG' [Lyn20] or 'weakly bounded' [Hen21]. This particular paper builds six such gadgets, combines them to make a weak crossover gadget, and then implements a 3SAT formula. This paper is also the first to use directed crossovers to simulate an undirected crossover, a trick that has been used many times as crossovers are frequently one of the hardest basic gadgets to construct.

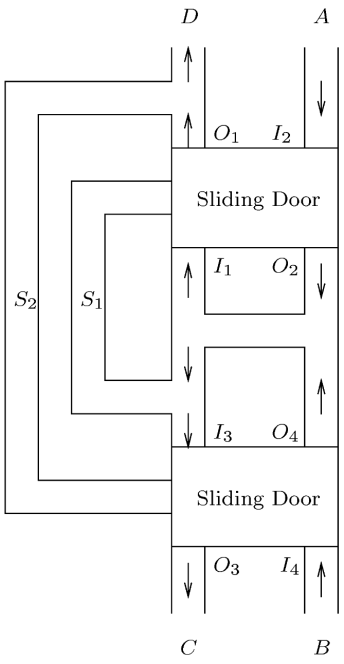


Figure 2.1: One of the earliest simulations between motion-planning gadgets, from [DZ99]. Doors (called 'sliding doors') and diodes (arrows) are used to build a directed crossover. The agent can traverse the left tunnel of each door gadget only when the door is *open*, and traversing the right tunnel *closes* it. A door can be opened by visiting the rightmost entrance. Initially both doors are open.

The capstone of this line of work was published in 2003 by Demaine et al. [DDHO03]. The main attraction is an NP-hardness proof for Push-1 which is designed to work for a menagerie of variations, including Push-k, Push-kX, and PushPush-k. The proof uses four gadgets: the one-way, a three-exit fork, the NAND gadget, and the *XOR crossover*, a weak crossover that is only guaranteed to work if only one of the paths is used. Really there are two different NAND gadgets, one with the tunnels in the same direction (*parallel*) and one with them in opposite directions (*antiparallel*). The construction of each gadget is quite simple, with only one or two blocks that can ever move, but they require a precise arrangement of tunnels. The proof proceeds by an elaborate reduction from planar 3-coloring. The agent chooses a color for each vertex at a fork gadget, and edges have an arrangement of NAND gadgets and XOR crossovers that enforce consistency.

We now know that the antiparallel NAND, without any other gadgets, suffices for NP-hardness [Lyn20]. So the other four gadgets are redundant, and if the authors were aware of this result, they could have saved a lot of work.

This is obviously unfair. Not only is this result twenty years more recent; it's also directly based on the proof by Demaine et al. [DDHO03]. Specifically, Bosboom et al. [BDD+20] simply shows how to simulate all of the other necessary gadgets using the antiparallel NAND.[7] But the power of the framework is made clear: now that these authors have done the hard work, a single simple gadget is enough to prove any other motion-planning problem NP-hard.

I want to mention one more paper from this era. Demaine et al. [DHH02] prove that Push-kF is PSPACE-complete for $k \geq 2$. To do this, they build a gadget similar to the door gadget, use it (and two simpler base gadgets) to simulate a directed crossover and then a full crossover, and finally reduce from Nondeterministic Constraint Logic. I believe this paper has the first explicit mention of the modularity of motion-planning gadgets, observing that they "prove that any model of pushing-block puzzles is PSPACE-complete if one can build a set of three gadgets with a certain functionality". Indeed, the main gadget used for $k \geq 3$ doesn't work for $k = 2$, so the authors build a much more complex but equivalent version for this case.

Of course, not all work on these kinds of problems became part of the motion-planning gadget framework. Some proofs, such as NP-hardness of Sokoban [DZ99] mentioned above and PSPACE-hardness of Atomix [HS04] and PushPush [DHH04], involve blocks being pushed between gadgets. The primary gadget framework established today assumes gadgets are self-contained and doesn't allow this, though some recent work has considered a generalization that does [DL25]. Other proofs, such as NP-hardness of Push-$*$ [DDHO03], don't fit the framework at all. Some of them lead to other gadget frameworks, notably including Constraint Logic [HD09].

After 2004, the motion-planning gadget conversation went quiet. Other gadget frameworks (again including Constraint Logic) continued to be developed, but I am not aware of any papers relevant to motion-planning gadgets until 2010, when Ritt [Rit10] proved NP-hardness of Pull-1F by implementing the same gadgets used by Demaine et al. [DDHO03]. The door of Dorit Dor and Uri Zwick [DZ99] seems to have been forgotten, and would not resurface until 2015 [Vig15, ADGV15].

[7] However, there is a more genuinely simpler approach based on another gadget framework that has been developed since [Pil19], which I discuss below.

Throughout this era, we have seen a variety of ad hoc hardness proofs gradually converging on the idea of motion-planning gadgets. Some of these proofs were able to reuse components of earlier proofs, but for the most part nobody has attempted to state general results about certain gadgets implying hardness.

2.2 The metatheorem era

That changed in 2010 when Forišek [For10] published four 'metatheorems' targeted at proving NP- or PSPACE-hardness of platforming games. Each states that if such a game contains some set of features, then the game is hard—for instance, the third says that 'long fall' and 'opening doors' suffice for NP-hardness. These features are not defined precisely[8] and the metatheorems are not formal statements; they are better thought of as recipes for hardness proofs. But, by design, they reference common tropes in the platforming genre, and so are frequently applicable. Forišek demonstrates this by quickly proving NP-hardness of nine games.

He also proves (by simulating a Turing machine) PSPACE-hardness of one game, and extracts an abstract metatheorem that 'long fall', 'opening doors', and 'closing doors' suffice for PSPACE-hardness. While metatheorems along these lines will eventually evolve into motion-planning gadgets, they cannot yet be expressed as such: the 'opening' and 'closing' used in the proof take the form of *pressure plates* that, when stepped on, affect a given door. In particular, any number of pressure plates can affect the same door, and a pressure plate can be far from the door it affects (which is relevant for planarity).

Soon after, Viglietta [Vig14] proved five more similar metatheorems, and used them to establish over a dozen hardness results for video games. For our purposes, the most interesting metatheorems are *doors and pressure plates* and *doors and buttons*. Each pressure plate or button is labelled with a set of instructions like 'open door A; close door B', which are carried out when the pressure plate or button is activated. There are two differences between the two actuators:

- A pressure plate is required: it is triggered whenever the agent walks across it. But a button is optional: the agent can choose whether to press it.
- Viglietta requires that each pressure plate controls only one door, but buttons can affect multiple doors.

Note that a single door can be controlled by several actuators.

When each actuator controls only one door (as with pressure plates), the combination of a door and all of its actuators can be modelled as a self-contained gadget. It has two states ('open' and 'closed'), with the door passable only when open, and walking through an actuator tunnel changes the state (optionally, in the case of buttons). Indeed, with one pressure plate that opens the door and one that closes it, this is exactly the door gadget we saw before [DZ99], except that actuators are not expected to be near the door they control.

[8] For example, 'opening doors' means that the game has "a variable number of doors and suitable mechanisms to open them".

Viglietta [Vig14] proved that doors and pressure plates suffice for NP-hardness even if each door is controlled by only one pressure plate, and for PSPACE-hardness even if each door is controlled by only two pressure plates. Although he did not describe it as such, the latter result is equivalent to PSPACE-hardness of reachability with the door gadget, except that the requirement for planarity is somewhat relaxed.

He also proved that doors and buttons suffice for NP-hardness even if each button controls at most two doors, and for PSPACE-hardness even if each button controls at most three doors. This can't be translated to the motion-planning gadget framework, but set the stage for improved results that can.

Forišek [For10] and Viglietta [Vig14] used their metatheorems to prove hardness of various two-dimensional video games, and they were also adapted to and used for three-dimensional games, including various mechanics in Portal [DLL18].

In 2015, while proving PSPACE-hardness of the game *Bloxorz*, van der Zanden and Bodlaender [VDZB15] considered a third type of actuator. The *switch* is optional, is labelled with some doors, and *toggles* the state of all of them when activated. Van der Zanden and Bodlaender don't state this as a metatheorem, but they prove PSPACE-hardness of doors and switches, even when each switch controls two doors and each door is controlled by one switch. This is another self-contained gadget, though it doesn't have a standard modern name. The authors note that, when each door is controlled by one switch, the switch can be replaced by two buttons, one for each direction of toggling. Thus doors and buttons are PSPACE-hard even when each door is controlled by two buttons and each button controls two doors,[9] fully resolving the complexity of doors and buttons as these two parameters are varied.

In the same year, we see the reemergence of the door gadget. Viglietta [Vig15] reformulates the 'doors and pressure plates' metatheorem into a new one—(self-contained) doors and crossovers suffice for PSPACE-hardness—and applies this to *Lemmings*. This appears to be an entirely independent discovery of the same gadget used nearly two decades earlier [DZ99, Cul97, HLH03]. Soon after, Aloupis et al. [ADGV15] used this gadget to prove PSPACE-hardness of several Nintendo games. To my knowledge, this was the first time we see a motion-planning gadget more complicated than the diode represented *fully* abstractly, without any relation to how it is constructed or laid out in an application (see Fig. 2.2). This version of the door gadget metatheorem was also used to prove Super Mario Bros. PSPACE-hard [DVW16], and was extended into a two-player game to prove EXPTIME-hardness of Xiangqi and Janggi (Chinese and Korean Chess) [Zha19].

Aloupis et al. [ADGV15] also introduced a framework for proving NP-hardness of platforming games, expressed as a template for a reduction from 3SAT if you can construct a list of gadgets: a crossover, a 'variable' which lets the player choose one of two exits (essentially the fork we saw earlier [DDO00]), and a 'clause' that has a door that can be opened by visiting the gadget at any of three entrances. As is common in this space, the crossover gadget is frequently quite hard to build, and frameworks that don't require them are easier to use. Reducing from planar 3SAT [Lic82] often helps, but in this case it isn't enough: the reduction works by having the player visit each variable gadget in sequence, and then visit

[9]Moreover, we can require that pairs of doors are controlled by pairs of buttons in a consistent way, so that the combination of two buttons and two doors is a self-contained gadget.

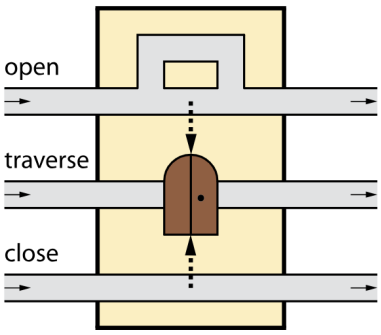


Figure 2.2: The first fully abstract representation of the door gadget, from [ADGV15].

each clause gadget in sequence, and the paths necessary to do so may cross variable-clause edges.

Pilz [Pil19] helped address this situation by improving the planar 3SAT gadget framework. He introduced, and proved NP-hardness of, several versions of planar 3SAT with stricter planarity requirements. For our purposes, the most relevant one is *sided linked planar 3SAT*. In this problem, we are given a 3CNF formula and a planar embedding of the graph that has

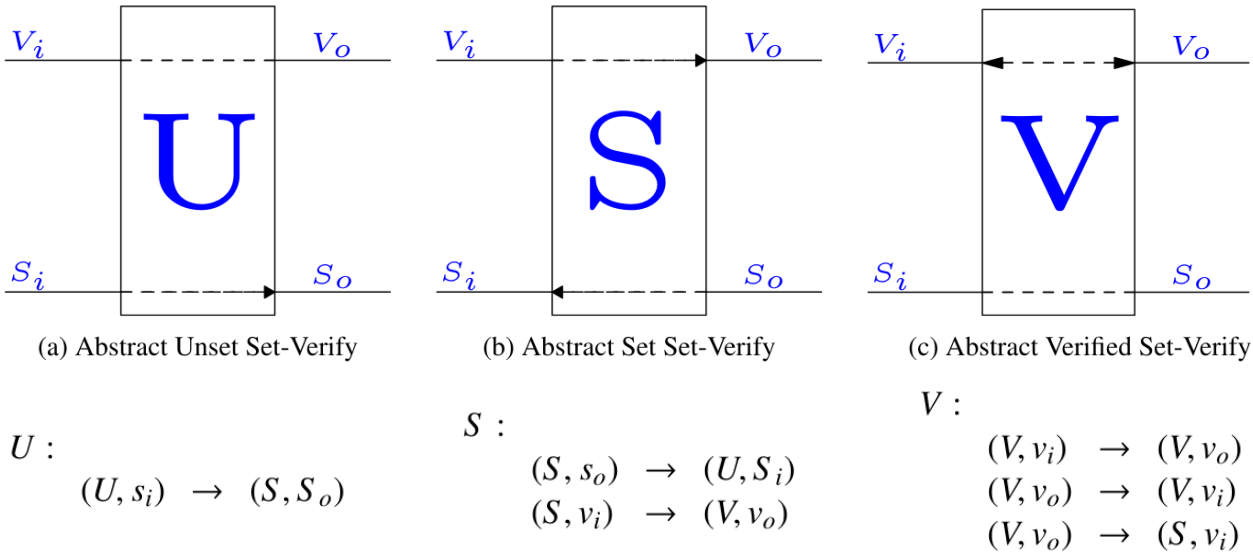
- a vertex for each variable and each clause,
- edges connecting each variable to each clause it appears in, and
- additional edges forming a Hamiltonian cycle that visits every variable vertex and then every clause vertex.

Furthermore, we require that edges corresponding to positive occurrences of variables are inside the cycle, and edges corresponding to negative occurrences are outside the cycle. The question, as always, is whether the formula is satisfiable. The additional cycle allows us to embed a 3SAT instance in a game so that the player can visit each gadget in the required order without crossings. For example, there is a reduction from sided linked planar 3SAT to reachability with NAND crossovers much simpler than the NP-hardness proof for this problem that goes through 3-coloring [DDHO03, BDD+20].

2.3 The state machine era

In 2017, Demaine et al. [DGL17] described their gadgets in a new abstract way: by listing states and transitions (see Fig. 2.3). Their application was Push-1 Pull-1W, which is *reversible*: every move can immediately be undone.[10] As a result, most gadgets studied by prior work (including the door, diode, and NAND crossover) can't be made, so this paper introduced a new family of gadgets. The first, used to prove NP-hardness, is called the

[10] I have elided this, but there is a distinction required pulling ('Pull!') and optional pulling ('Pull?') [ADD+20]. In this case, pulling is optional.



(a) Abstract Unset Set-Verify (b) Abstract Set Set-Verify (c) Abstract Verified Set-Verify

Figure 2.3: The Set-Verify gadget, from [DGL17].

Set-Verify gadget, and is shown in Fig. 2.3. It is very similar to the gadget we now call the locking 2-toggle [DHL20], and in fact both basic crossovers the authors construct using it ('directed destructive crossover' and 'in-order directed crossover') are exactly crossing locking 2-toggles, starting in different states. For proving PSPACE-hardness in 3D, Demaine et al. introduce the 2-toggle and some of its friends, ultimately proving PSPACE-hardness of reachability with 4-toggles and crossovers. This proof is notable for its heavy use of simulations, and in particular a style of 'bootstrapping' that became a common tool for working with simple gadgets: use them to gradually build more and more powerful gadgets until you have everything you need. In this case, the 4-toggle is used to simulate a 2-toggle-lock, which is used to simulate a 2-toggle with more locks, which is finally used to simulate the specific gadgets that comprise a reduction from QBF.

The states-and-transitions model was made explicit a year later by Demaine et al. [DGLR18], in a deeper investigation into this style of reversible gadget. The community finally had a definition of the objects they had been working with for twenty years:

Definition 2.1. A *gadget* consists of some *states*, some *locations*, and some *transitions* of the form $(s, \ell) \to (s', \ell')$, where s and s' are states and ℓ and ℓ' are locations. This transition means that when the gadget is in state s, the agent can enter at ℓ, change the state to s', and exit at ℓ'. We say *traversal* for a movement $\ell \to \ell'$ through a gadget when we don't want to specify the states.

A *system of gadgets* consists of some gadgets with given initial state and connections between their locations. Usually, we require systems to be planar, in which case gadgets are equipped with a cyclic order of locations which must be respected by the planar embedding.

For a set of gadgets S, in the decision problem *[planar] reachability with* S, we are given a [planar] system of copies of gadgets in S, a location the agent starts at, and a goal location. The question is whether the agent can navigate the system, moving through

gadgets according to their transitions, to reach the goal location.

While the core of the definition has been stable, some details (and notation) change from paper to paper: for instance, do we allow many locations to be connected together, or only two at a time? Demaine et al. take the latter stance, but allow a special gadget called the *branching hallway* which can be used to connect more locations.

This paper also introduces the first concrete definition of simulation:

Definition 2.2 ([DGLR18])**.** Two systems of gadgets are *equivalent* if there is a bijective correspondence between their locations and a correspondence between their states such that the allowed transitions for all (locations, state) pairs are the same under these two correspondences. We will say that a gadget or set of gadgets *simulates* a target gadget if it is possible to combine gadgets from the set (possibly using duplicates) such that the resulting system is equivalent to the target gadget. We will always implicitly allow the use of the branching hallway gadget in these constructions.

We will see later that this definition is too strict for certain cases. But it makes it possible to start proving results about simulation itself, which Demaine et al. [DGLR18] do. Specifically, they define *reversible* gadgets (ones where every transition $(s, l) \to (s', l')$ has an inverse $(s', l') \to (s'l)$) and *deterministic* gadgets (ones where there is always at most one transition from (s, l)), and prove that both classes are closed under simulation. For deterministic gadgets, it is crucial that their definition only allows locations to be connected in pairs. The branching hallway, which is nondeterministic, is then used to introduce agent choice.

This paper considers two-state reversible deterministic gadgets on tunnels, and defines a naming scheme for them. Such a gadget contains some tunnels, which come in three types. The *lock* is freely passable in both directions in one state, and impassable in the other state. The *tripwire* is always passable, and changes the state of the gadget whenever the agent walks across it. The *toggle* can be traversed in a different direction in each state, and changes the state when used. For a gadget with multiple tunnels, we connect them with hyphens, except that we say '2-toggle' instead of 'toggle-toggle'. Gadgets generally have multiple planar layouts, which must be disambiguated. For instance, the 2-toggle has three: *crossing*, where the tunnels cross; *parallel*, where the tunnels don't cross and point in the same direction; and *antiparallel*, where the tunnels don't cross and point in opposite directions.

The primary goal of this paper [DGLR18] was to improve upon the previous one [DGL17], by bootstrapping all the way from the antiparallel 2-toggle. Demaine et al. use this gadget to simulate ever more powerful gadgets including crossing and parallel 2-toggles, toggle-locks, 2-toggle-locks, and ultimately a full crossover. In general, simulations between gadgets induce reductions between the corresponding reachability problems—in this case, from planar reachability with 2-toggle-locks and crossovers to planar reachability with antiparallel 2-toggles, proving the latter PSPACE-hard. This was the first proof to have most of the work performed at the level of motion-planning gadget simulations, rather than gadget implementations or in a reduction to the motion-planning framework. Having an explicit definition of the framework certainly helped: it allowed the authors to think clearly at the relevant level of abstraction, using simulations to devise reductions between abstract reachability problems.

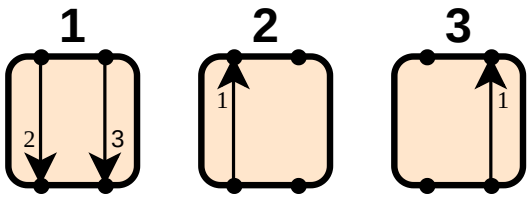


Figure 2.4: The state diagram of the locking 2-toggle, first introduced by [DHL20]. This gadget is similar to a 2-toggle, except that after using either tunnel, the other tunnel is 'locked' until the agent reverses the transition.

This was then applied to prove hardness of a Zelda game, and later of a problem called 'Full Tilt' [BMLC+19] and games with 'turnstiles' [GHH+21b].

Demaine et al. went further, and proved that every nontrivial 2-state 2-tunnel reversible deterministic gadget (there are four, each with two or three planar layouts) simulates every other, and characterizing the complexity of reachability with each 2-state reversible deterministic gadget, with any number of tunnels. In addition to its utility for applications, these results were a precursor for later universality results, and in my opinion are quite interesting on their own.

This is where I finally enter the story. I joined Erik Demaine (who would become my advisor) and Jayson Lynch to try to extend these results to three states [DHL20]. While many 3-state 2-tunnel reversible deterministic gadgets fell quickly to simulations, a few—including one that became known as the locking 2-toggle, whose state diagram[11] is shown in Fig. 2.4—were resistant. (We now know that the locking 2-toggle is 'balanced', and thus can't simulate the 2-toggle [Hen21].) So instead I found a new reduction, from Nondeterministic Constraint Logic [HD09], to prove PSPACE-hardness. This result has since been used to prove hardness of various problems including reconfiguring pivoting robots [ADG+21], Pull-kF with gravity [ADD+20], and Block Dude [CDD+22].

Inspired by the simulation results for 2-state gadgets, I found a way to simulate a locking 2-toggle with *any* k-tunnel reversible deterministic gadget, provided it has 'any interesting behavior' (*interacting tunnels*) at all. As a consequence, reachability with any such gadget with interacting tunnels is PSPACE-complete, and without interacting tunnels it's in NL. This containment result was the first example of a versatile technique called *shortcutting*. We argue that if a path to the goal visits a location multiple times, we can take a shortcut, cutting out the loop and obtaining a shorter valid path. This implies that if there is any valid path, there is one (namely the shortest) that never visits a location twice, and it follows that the problem can be solved as a directed reachability problem.

Mirroring the structure of Constraint Logic [HD09], we also adapted the reduction to work for a two-player game (proving EXPTIME-completeness) and for a team game with hidden information (proving RE-completeness), for all of the same gadgets. In addition, we considered a bounded case, with *DAG* gadgets (later also called *strictly bounded* [Hen21]):

[11] Throughout this thesis, I draw state diagrams with orange gadgets and simulations with purple gadgets, to distinguish between these two kinds of picture.

Figure 2.5: State diagrams for the door and self-closing door.

ones which can be used a limited number of times before becoming impassable. For reachability, we similarly found that any DAG gadget is NP-hard if it has interacting tunnels, and in NL otherwise. For the multiplayer games, we found something surprising: every DAG gadget that allows any transition has a single-use transition, and a single-use transition suffices for PSPACE-hardness of the two-player game and NEXPTIME-hardness of the team game. Today, we call the gadget with a single-use transition the *crumbler* if it is bidirectional, and the *directed crumbler* or *dicrumbler* if it is only in one direction.

In the next big motion-planning gadgets paper, my close collaborators and I thoroughly explored the door, the self-closing door, and similar gadgets [BDD+20]. See Fig. 2.5 for state diagrams of those two gadgets. The door gadget has three directed tunnels, called *open*, *close*, and *traverse*. The first two tunnels affect the state of the door, and the traverse tunnel is only passable if the door is open. The self-closing door has a directed tunnel which closes when traversed, and a button that reopens it. Previously, it was known that nonplanar reachability with either gadget is PSPACE-complete, and that one planar layout of the door can simulate a crossover [DZ99, Vig15]. We discovered that each planar layout of the door can simulate a crossover, and this has PSPACE-complete planar reachability, except for one ('OTtocC') which has now been solved in unpublished work. Along the way, we also proved that self-closing doors can simulate crossovers and doors, and introduced another variant called the *symmetric self-closing door* (or *mismatched dicrumblers*), which has two directed tunnels, each of which closes itself and opens the other. Doors are often a relatively easy gadget to construct, so these results have been used for a large number of applications, including Sokobond and eight 3D Mario games in this paper, and later (another version of) Pull-kF with gravity [ADD+20], several Zelda games [BBC+23], Celeste [CD23], and thirteen 2D Mario games [GDH+24].

We also proved that each version of the door gadget can simulate *every* gadget [BDD+20]. Dor and Zwick [DZ99] had gestured at this, but we now finally had the terminology to understand what 'every gadget' even means. This was the first true universality result for motion-planning gadgets. I found it surprising that such a thing was possible: the door is the 'hardest possible' gadget, in that you can replace any system of any gadgets with a network of doors. There is a strong parallel to complexity classes and completeness. In both cases, the objects (decision problems or gadgets) have a natural preorder (reductions or simulations). We consider classes downwards-closed under the preorder (e.g. NP or reversible

gadgets), and sometimes such a class has a maximum element. Simulations are of course much stricter than reductions—a simulation of G' using G immediately gives a reduction from reachability with G' to reachability with G, but the reverse is far from true. So one perspective is that simulations provide a more refined notion of how 'difficult' a gadget is, whereas complexity classes are very coarse.

Throughout this era, we also explored other directions for the framework. With many of the same authors, I investigated the complexity of questions about systems of gadgets other than reachability [DDH+23], and just how much computational power is needed if we allow gadgets to create an unlimited number of robots [CDD+23]. Others have used gadgets with infinitely many states to simulate counter machines for RE-hardness [GDH+25] and studied which hardness results are maintained when restricted to systems of gadgets with small bandwidth [GDD+25]. I think of these as offshoots of the 'core' motion-planning gadgets framework—certainly interesting in their own right, but not the focus of this story. I discuss two other offshoots at length in Chapters 3 and 4.

One of our papers, which proves PSPACE-hardness of Push-1F [CDD+22], straddles the boundary between this era and the next. It was written after much of the development of gizmos, and uses some of the ideas underlying gizmos, including adapting their definition of simulation. But everything is expressed in terms of the state machine formulation of motion-planning gadgets, so I think it does belong here. In it, we introduced a notion of generalizing simulations to allow some additional infrastructure:

Definition 2.3. A finite set of gadgets $\mathcal{G}$ *nonlocally simulates* a gadget H if, for every finite set of gadgets $\mathcal{G}'$, there is a polynomial-time reduction from reachability with $\{H\} \cup \mathcal{G}'$ to $\mathcal{G} \cup \mathcal{G}'$.

For an example of what this can look like, suppose $\mathcal{G}$ contains a single gadget, which is H with an extra tunnel. Then one nonlocal simulation is to replace each copy of H with this gadget, and then add a path from the original goal location that goes through each extra tunnel and then arrives at the new goal location. This is the idea behind the nonlocal simulations we constructed, except that the 'H with an extra tunnel' (called a *simply checkable* H) only behaves correctly when we require the extra tunnel to be passable at the end.

Nonlocal simulation provides a relationship broader than true simulation but stricter (and so a more useful tool) than arbitrary reductions. The presence of $\mathcal{G}'$ forces a nonlocal simulation to work at the gadget level, and allows us to compose nonlocal simulations, including with true simulations. To take advantage of this, we define the *postselection* of a gadget G by a *checking traversal sequence* C, which behaves like G, except that if a transition would make it impossible to then follow each traversal of C in G, that transition is not allowed in the postselection. With some auxiliary gadgets, we can require the agent to walk through each checking traversal sequence after reaching the goal, forcing each copy of G to be used as though it were the postselected gadget. This is a nonlocal simulation of the postselection.

We applied these ideas to Push-1F, which had long been known NP-hard [DZ99] but had thus far evaded PSPACE-hardness. We built the handful of auxiliary gadgets necessary, most of which were based on earlier gadgets for this game. Then we built a gadget we helpfully named the *checkable proto-precursor*, which on its own is quite lax and difficult to work with,

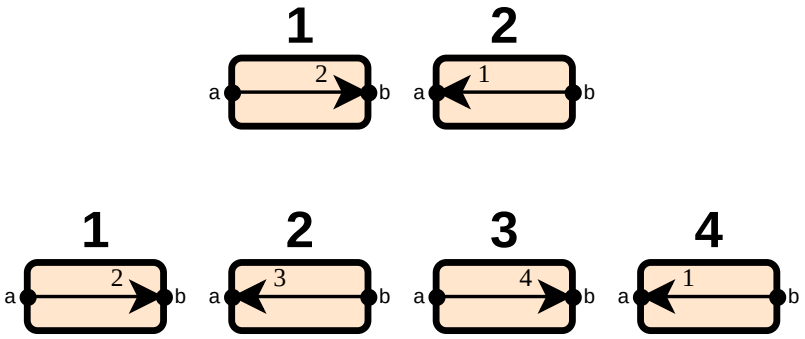


Figure 2.6: The state diagrams for the 1-toggle (top) and the doubly covered 1-toggle (bottom). We would like to treat these gadgets as equivalent.

but after multiple rounds of postselection and simulation, creates the self-closing door. This was a real test of the power of the framework—without it, a proof of this complexity would be entirely unreasonable. But with the right abstractions in hand, we can construct (or nonlocally simulate) gadgets of increasing usefulness, without needing to concern ourselves with implementation details below the current level of abstraction.

2.4 The gizmo era

With the attention we had placed on simulation, we noticed it didn't always behave quite how we wanted. There were occasionally constructions that morally should be simulations of some gadget, but failed to meet the definition we had been using [DGLR18]. Maybe the simulation has multiple states that correspond to a single state of the gadget, or maybe the simulation allows the agent to do something the gadget doesn't but which is never beneficial. The toy example I like to use is the *doubly covered 1-toggle*. This gadget is like the 1-toggle, except that it has more states, and takes four traversals to return to the initial state. State diagrams for both gadgets are shown in Fig. 2.6. We want to be able to say that these gadgets simulate each other. In particular, replacing a 1-toggle with a doubly covered 1-toggle in a system of gadgets can never change the answer to a reachability problem—this is what simulation (in the context of reachability) should really mean, and what our definition should aim to capture. However, under the definition of simulation requiring a bijection between states, this does not qualify. Even worse, the doubly covered 1-toggle isn't reversible according to the original definition, and so provably can't be simulated by 1-toggles.

With help from my collaborators, particularly Jenny Diomidova and Hayashi Layers, I developed a new version of the motion-planning gadgets framework based on objects called *gizmos* to address these issues at the cost of an additional layer of abstraction; this became my Master's thesis [Hen21]. The central idea is that instead of considering state machines, we should consider the languages (over the alphabet of traversals) they recognize. For instance, the 1-toggle and the doubly covered 1-toggle recognize precisely the same language:

it contains strings like $[a \to b, b \to a, a \to b]$ but not $[a \to b, a \to b]$.

Definition 2.4. A *gizmo* on a set of locations L is a language over the alphabet of traversals on L (i.e. a set of sequences of traversals of form $l \to l'$) that is closed under

- insertion of a *self-loop* $l \to l$, and
- *transitive contraction*, meaning replacing $a \to b, b \to c$ with $a \to c$.

These conditions address common ways gadgets could be practically equivalent but not equal—allowing the agent to enter and immediately leave from the same location without changing the state has no effect on reachability. This definition actually applies more generally than reachability, to a decision problem called *targeted set reconfiguration*, where the agent must reach the goal location while leaving each gadget in an accepting state. For reachability, we consider *prefix-closed* gizmos G, meaning ones where whenever $XY \in G$, the prefix $X \in G$. This essentially says that all states of the gadget are accepting.

I also defined simulation for gizmos, in three stages:

- *Tensor product*: Put some gizmos next to each other to make a new gizmo. The new gizmo has all the locations on the component gizmos, and the component gizmos act independently.
- *Quotient*: Connect locations of a gizmo according to an equivalence relation.
- *Subgizmo*: Choose some set of locations, called *ports*, to be accessible to the outside world. Forget about all other locations.

Any simulation is a combination of these operations, in this order. After studying more gadget frameworks, I have noticed that this form of definition applies quite generally, though the most natural form is a bit different—if I devised these definitions today, I would make different choices, though not in ways that have a large impact on the resulting structure. Specifically, I now believe the best approach is to replace quotient and subgizmo with a single operation which connects two locations and removes them from the gizmo, so locations can only be connected two at a time (as in [DGLR18]). For motion-planning gadgets, this means we need to include branching hallway gadgets everywhere, but so many other gadget frameworks, including all of the others discussed in this thesis, inherently have this requirement.

Having a more flexible definition of simulation opened the door to proving many more closure and universality results. In my thesis [Hen21], I translated and proved closure for reversible, weakly bounded (LDAG), and strictly bounded (DAG) gadgets. I defined the *balanced* gadgets, which allowed us to prove that the locking 2-toggle can't simulate a 2-toggle. The crucial property is that in any traversal sequence on a locking 2-toggle, the agent can't repeatedly enter at a location unless they exit at that location a similar number of times. Most excitingly, I introduced an infinite family of closed gizmo classes defined by 'implication properties', which algebraically represent a condition on gizmos (e.g. prefix-closed is "$XY \implies X$"). I also proved several universality results, including doors for all gizmos (translating the corresponding result for gadgets [BDD+20]), the 2-toggle for reversible gizmos, and certain door-like gizmos for weakly and strictly bounded gizmos.

In as-yet-unpublished work, my collaborators have further developed these ideas: Jenny Diomidova found a way to simulate every balanced reversible gadget with locking 2-toggles, and Ritam Nag characterized reversible gizmos in terms of implication properties. In her Master's thesis [Lay23], Layers investigated simulability between parameterized gizmos similar to the directed crumbler, and found fairly weak conditions that imply a gizmo can simulate the directed crumbler.

The purpose of gizmos is to understand the theory of simulation, rather than to directly provide value for hardness proofs. As a result, and also because the state machine formulation was good enough for most purposes, there are few applications of the ideas of this era. But gizmos were instrumental to proving PSPACE-hardness of Push-1 [GBC+25], in a sequel to the PSPACE-hardness of Push-1F [CDD+22]. In this paper, we adapted the ideas of checkable gadgets and postselection to gizmos. The definition is very simple in the language of gizmos: the *postselection* G^C of a gizmo G by a traversal sequence C is the gizmo containing all sequences X such that $XC \in G$. In other words, it behaves like G, but only if after you're done interacting with it, C would be legal.

Crucially, even if G is prefix-closed, G^C can be not prefix-closed. As a simple example, take G to be the gizmo representing the 1-toggle (in state 1, with both states accepting) and take C to be $[a \to b]$. Then G^C is the 1-toggle, but is required to end in state 1. In this way, we can use postselection to create non-prefix-closed gizmos, and thus a targeted set reconfiguration (as opposed to reachability) problem. For Push-1, reconfiguration is much easier to work with: for instance, the reconfiguration problem storage Push-1F, or Sokoban, was proved PSPACE-hard [Cul97] over twenty years before the corresponding reachability problem Push-1F [CDD+22].

As with Push-1F, we can nonlocally simulate postselections by adding a path the agent must walk along after reaching the goal, and thus reduce from a reconfiguration problem to a reachability problem. The reconfiguration problem we chose is Nondeterministic Constraint Logic [HD09], and along the way we also described a general way to encode graph orientation vertices as motion-planning gadgets.

Interestingly, another preprint proving Push-1 PSPACE-complete was released only five days later [DL25]. In it, DeStefano and Liang use the simpler checkable gadgets framework we developed for Push-1F [CDD+22], as well as a very different approach: they extend the gadget framework to allow the agent to blocks between gadgets. In the extension, the agent has a state, representing whether it is bringing a block with it. This version of the framework can describe some preexisting proofs that the usual framework cannot [DZ99, HS04, DHH04]. DeStefano and Liang's [DL25] primary gadget that takes advantage of agent state is the *stackable ejector*, which can be used to force the agent to push several blocks into a gadget, blocking it, in order to access a button. This is similar to an idea we used in our proof of the same result [GBC+25], where the checking sequence for our gadgets involves pushing multiple blocks all the way into the 'main' part of the gadget to confirm that there's enough space for them—though in our case this happens within a single gadget instead of being split across multiple.

Motion-planning gadgets began with the study of block-pushing games [Wil88, DO92, DZ99], and the most advanced versions of the framework are finally powerful enough to prove PSPACE-hardness of Push-1, one of the simplest block-pushing games and known NP-hard since 2000 [DDO00]. Along the way, we have created a set of tools that makes it vastly easier

to prove hardness of certain types of problems.

Looking to the future, it is not obvious that there are any further refinements of the motion-planning gadget framework that would increase its usefulness even more. But there are plenty of interesting related research directions left to explore. For a small sample:

- There is no shortage of video games and other problems to prove hard.
- In my opinion, gizmo simulability is an interesting area on its own, even without applications to complexity. There is a lot that we don't understand about closed classes and universality.
- The motion-planning gadget framework, and especially the gizmo formulation, is narrowly targeted at reachability or targeted set reconfiguration problems. We could aim to develop analogs for problems like reconfiguration [DDH+23], multiplayer games [DHL20], and more.
- We could move up a level of abstraction: many gadget frameworks exhibit a similar structure, where we draw lines connecting certain points on blobs and call it a simulation. Perhaps there is a unifying description of such gadget frameworks which would allow us to make general claims about simulations, likely using tools from category theory.

Chapter 3

Input/Output Gadgets

In this chapter, I describe an offshoot of the motion-planning gadget framework called the *input/output gadget* framework. It is designed for problems with a single agent moving deterministically, though it also addresses some one-player and two-player games.

This chapter represents joint work with Erik Demaine, Hayashi Layers, and Jayson Lynch, assisted by valuable conversations with Jeffrey Bosboom and Mikhail Rudoy, and one construction by Twan van Laarhoven. These results first appeared in [DHLL23], with the exceptions of Theorem 3.15 and Section 3.3.2. They have since been used to prove PSPACE-hardness of the game Celeste, even without any player input [CD23].

3.1 Introduction

Imagine a train proceeding along a track within a railroad network. Tracks are connected together by "switches": upon reaching one, the switch chooses the train's next track deterministically based on the state of the switch and where the train entered the switch; furthermore, the traversal changes the switch's state, affecting the next traversal. ARRIVAL [DGK+17] is one game of this type, where every switch has a single input and two outputs, and alternates between sending the train along the two outputs; the goal is to determine whether the train ever reaches a specified destination. Even this seemingly simple game has unknown complexity, but is known to be in NP ∩ coNP [DGK+17], so cannot be NP-hard unless NP = coNP. More recent work includes stronger containment results in UP ∩ coUP, CLS [GHH+18], PLS [Kar17], and UEOPL [FGMS20], and a subexponential time algorithm [GHH21a]. But what about other types of switches?

In this chapter, we introduce a very general notion of "input/output gadgets" that models the possible behaviors of a switch, and analyze the resulting complexity of motion planning/prediction (does the train reach a desired destination?) while navigating a network of switches/gadgets. This framework gives us an expressive set of problems for various complexity classes to use as starting points for hardness reductions to other problems of interest. For example, it is related to the "reachability switching games" of [FGMS21], which in turn generalize "switching systems" known as Propp machines. In addition to ARRIVAL, our framework captures other toy-train models, including those in the video games Factorio and Trainyard. In many cases, we obtain PSPACE-hardness, enabling building of a (polynomial-

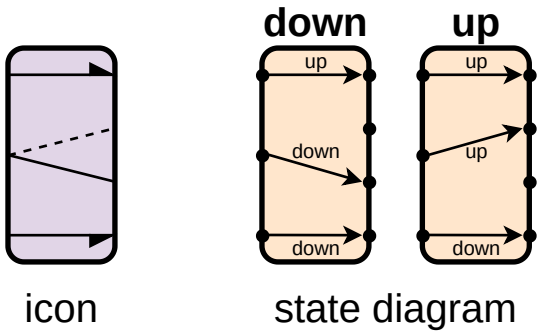


Figure 3.1: An example gadget—the switch/set-up line/set-down line of Fig. 3.3b—which is a 3-input 4-output 2-state input/output gadget. The agent can enter at any of the three inputs on the left, and exit at a corresponding outputs on the right. Traversing the top or bottom line sets the gadget's state to 'up' or 'down', respectively, which controls the output of the middle traversal to be the top or bottom, respectively, of its two options. The middle traversal does not change the state.

space) computer out of a deterministic railway system with a single train. Intuitively, our model is similar to a circuit model of computation, but where the state is stored in the gates (gadgets) instead of the wires, and gates update only according to visits by a single deterministically controlled agent (the train).

This work builds off of prior work on the computational complexity of agent-based motion planning [DGLR18, DHL20], extending it to zero-player situations. An analogous generalization of computational problems based on the number of players and boundedness of moves can be found in Constraint Logic [HD09], which has served as a framework for a large number of hardness proofs for reconfiguration problems as well as games and puzzles.

Our framework is also related to Chalcraft and Greene's work on model train sets [CG94, Ste94]. Specifically, their 'lazy point' is equivalent to our 'set-up line/set-down line/switch' with some locations merged, their C is equivalent to our 'toggle line/switch', and their 'distributor' is equivalent to our 'toggle switch'.

3.1.1 Gadget framework

Our model is a natural zero-player adaptation of the motion-planning gadget framework discussed in Chapter 2. An *input/output gadget* consists of a set of states, and set of *inputs*, a set of *outputs*, and a set of *transitions* of the form $(s, a) \to (t, b)$ for states s and t, input a, and output b. This transition says that when the gadget is in state s, if the agent enters at a, it can move to b while changing the state to t. We require that, for each state s and input a, there is at least one transition from (s, a). Fig. 3.1 shows an example of an input/output gadget. As with motion-planning gadgets, we describe specific input/output gadgets using state diagrams.

A *system of (input/output) gadgets* consists of some gadgets, their initial states, and, for each output on each gadget, a specification of which input (on any gadget) the agent will move to from that output. Note that each output leads to exactly one input, but multiple

outputs may lead to the same input. While we don't allow input/output gadgets to have inherent dead-ends, we can easily implement them as needed with a loop through a single gadget.

Input/output gadgets can be thought of as a special kind of motion-planning gadget, where each location is designated an input or an output, and transitions always go from inputs to outputs. A system of input/output gadgets has the additional restriction that each output is connected to exactly one input.

An input/output gadget is *deterministic* if for each state s and input a, there is exactly one transition $(s, a) \to (t, b)$. This is consistent with the definition of 'deterministic' for motion-planning gadgets [DGLR18].

An input/output gadget is *output-disjoint* if, for each output location, all of the transitions to it (including those from different states) are from the same input location. This condition is analogous to k-tunnel gadgets, but allows a one-to-many relation from a single input to multiple outputs.

Let $\mathcal{G}$ be an input/output gadget. In *reachability with* $\mathcal{G}$, we are given start and goal locations of a single agent in a system of gadgets from $\mathcal{G}$, and we are asked whether there is a path through the system that brings the agent from the start to the goal.

With deterministic input/output gadgets, which are the focus of most of this chapter, reachability is a natural zero-player game. At each point, the agent has only one possible action: at an input, take the unique available transition through the gadget, and at an output, move to the connected input. So reachability asks whether, when running this deterministic process, the agent will reach the goal location before getting stuck in a loop.

A simple nondeterministic input/output gadget is the *fanout* gadget, which has one input, two outputs, and one state; the player may choose which output location to take. *One-player reachability with* $\mathcal{G}$ is reachability with $\mathcal{G}$ and the fanout.

In *two-player reachability with* $\mathcal{G}$, we are given a system of gadgets from $\mathcal{G}$ and fanouts, with each gadget *owned* by one of the two players, Black and White. The agent begins at a designated start location and moves through the system. When it arrives at a fanout, the owner of that fanout chooses which output it takes. White's goal is for the agent to reach a designated goal location, and the decision problem is whether White is able to force victory. The owner of a deterministic gadget is irrelevant.

3.1.2 Classifying Output-Disjoint Deterministic 2-State Gadgets

In this chapter, we are primarily interested in output-disjoint deterministic 2-state input/output gadgets. In this section, we omit the adjectives and refer to them simply as "gadgets", and we categorize these gadgets as "trivial", "bounded", or "unbounded".

The behavior of an input location to a gadget is described by how it changes the state and which output location it sends the agent to in each state. If the input location does not change the state and always uses the same output location, it can be ignored (the agent's path can be "shortcut" to skip that transition); we call this a *trivial line*. Otherwise, the input location corresponds to one of the five nontrivial subunits shown in Table 3.1. A gadget is then a disjoint union of some of these subunits, which interact by sharing state; Figs. 3.2 and 3.3 show some different ways these subunits can be assembled into different gadgets.

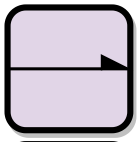
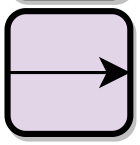
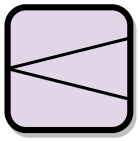
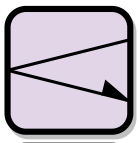
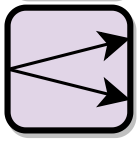
	**Set-Up Line**	A tunnel that can always be traversed in one direction and sets the state of the gadget to a specific state (‘up’).
	**Toggle Line**	A tunnel that can always be traversed in one direction and toggles the state with each crossing.
	**Switch**	A three-location gadget with one input which transitions to one of two outputs (‘top’ or ‘bottom’) depending on the state (‘up’ or ‘down’ respectively), without changing the state.
	**Set-Up Switch**	A switch that also sets the state of the gadget to a specific state (‘up’).
	**Toggle Switch**	A switch that also toggles the state of the gadget with each crossing.

Table 3.1: The five possible subunits (modulo up/down symmetry) for output-disjoint deterministic 2-state input/output gadgets, whose states are named ‘up’ and ‘down’. In general, unannotated lines denote transitions that do not change the state, full arrowheads denote transitions that always toggle the state, and half arrowheads denote transitions that always set the state to a specific value (‘up’ or ‘down’ according to the half arrowhead).

We call the states of a two-state gadget *up* and *down*, and assume that each switch transitions to the top output in the up state and the bottom output in the down state; because we are not concerned with planarity, this assumption is fully general by possible reflection of each subunit. There are two versions of the set line and set switch: one that sets the gadget to each state, up or down. For example, a gadget with a set-up switch and set-up line (Fig. 3.2b) is meaningfully different from a gadget with a set-up switch and set-down line (Fig. 3.3d). We draw the set-down line and switch as the reflections of the set-up version in Table 3.1. To represent the current state of a gadget, we draw one of the lines in each switch dashed, so that the next transition would be made along a solid line.

The ARRIVAL problem [DGK+17] is equivalent to reachability with just the toggle switch from Table 3.1: each vertex in their switch graph corresponds to a toggle switch in a system of gadgets. We will use their terminology when referring to switch graphs in ARRIVAL [DGK+17]; however, when referring to gadgets in our model, a switch is a gadget (or part of a gadget) which does not change state when traversed (as in Table 3.1). More generally, reachability with an arbitrary set of deterministic single-input input/output gadgets (with gadgets specified as part of the instance) is equivalent to explicit zero-player reachability switching games, as defined in [FGMS21].

We categorize gadgets into three families:

1. *Trivial* gadgets have either no state change or no state-dependent behavior; they are composed entirely of switches or entirely of toggle and set lines. Trivial gadgets are equivalent to (collections of) trivial lines, or equivalently always-open tunnels. Reachability with trivial gadgets is in L by straightforwardly simulating the agent for a number of steps equal to the number of locations.

2. *Bounded* gadgets have state-dependent behavior (i.e., some kind of switch) and have only one-way state change, either only to the up state or only to the down state. A bounded gadget can change its state at most once, so such gadgets naturally give rise to bounded games in which the maximum number of moves is polynomially bounded.

3. *Unbounded* gadgets have state-dependent behavior (switches) and have transitions that change state in both directions. For example, the toggle switch of ARRIVAL is unbounded. Unbounded gadgets naturally give rise to unbounded games in which the number of moves can be exponential.

We will find that the complexity of reachability with a given gadget also depends on whether the gadget is *single-input* or *multi-input*, where we count only "nontrivial" input locations. A *nontrivial input* must have a transition from that input that either changes the state of the gadget or does not exist in all states of the gadget. The only nontrivial single-input gadgets are the set switch and toggle switch, which are bounded and unbounded, respectively.

3.1.3 Our Results: Complexity

Table 3.2 summarizes our main complexity results for reachability with output-disjoint deterministic 2-state input/output gadgets. While our main motivation was to analyze reachability with deterministic gadgets, we also characterize the complexity of one-player reachability

	Trivial (always-open tunnels)	Bounded & multiple nontrivial inputs	Unbounded & multiple nontrivial inputs
Zero-player (fully deterministic) [§3.2]	L	P-complete	PSPACE-complete
One-player [§3.4]	NL-complete	NP-complete	PSPACE-complete

Table 3.2: Complexity of zero- and one-player reachability for arbitrary output-disjoint deterministic 2-state input/output gadget(s), with multiple nontrivial inputs in nontrivial gadgets.

	Contained in	Hard for
Zero-player (fully deterministic) [§3.2]	UP ∩ coUP [GHH+18]	NL for toggle switch [§3.2.1] (cf. [FGMS21])
One-player [§3.4]	NP [§3.4.1] (cf. [FGMS21])	NP [§3.4.2] (cf. [FGMS21])
Two-player [§3.5]	EXPTIME [§3.5] (cf. [FGMS21])	PSPACE [§3.5] (cf. [FGMS21])

Table 3.3: Complexity results for zero-, one-, and two-player reachability with any nontrivial single-input input/output gadget(s) (the toggle switch and/or the set switch).

problems for contrast. These complexity results apply to *any* gadget in the family specified in each column, and more generally to any nonempty set of gadgets in the family (optionally with gadgets from simpler families in leftward columns). In particular, we prove that reachability with *any* multi-input bounded gadget(s) (and optionally with trivial gadgets) is P-complete for zero-player and NP-complete for one-player; while reachability with *any* multi-input unbounded gadget(s) (and optionally with trivial or bounded gadgets) is PSPACE-complete for both zero- and one-player.

Table 3.3 summarizes our results for *single-input* nontrivial input/output gadgets. This case is a more immediate generalization of ARRIVAL [DGK+17], and is equivalent to the reachability switching games studied in [FGMS21]. We strengthen the results of [FGMS21] in two ways. First, we show that the containments in NP and EXPTIME still hold when we allow nondeterministic gadgets. Second, we show hardness for specific constant-size gadgets—the toggle switch for zero-player, and each of the toggle switch and set switch for one- and two-player—instead of having unbounded-size gadgets specified as part of the instance. In particular, these hardness results apply to all (two) nontrivial single-input gadgets for one- and two-player; the complexity of the set switch for zero-player remains open.

Our complexity results for zero-player, one-player, and two-player reachability are presented in Sections 3.2, 3.4, and 3.5, respectively.

In Section 3.6, we apply our input/output gadget framework to prove PSPACE-complete-

(a) Switch/set-up line. (b) Set-up switch/set-up line.

Figure 3.2: A basis for bounded multi-input gadgets: all such gadgets can simulate one of these two.

ness of mechanics in several video games: one-train colorless Trainyard, the game [the Sequence], trains in Factorio, and transport belts in Factorio are all PSPACE-complete. The first result improves a previous PSPACE-completeness result for two-color Trainyard [ALP18a] by using a strict subset of game features. Factorio in general is trivially PSPACE-complete, as players have explicitly built computers using the circuit network; here we prove hardness for the restricted problems with only train-related objects and only transport-belt-related objects.

3.1.4 Our Results: Simulation

Our approach to proving that reachability *any* gadget in an family is P- or PSPACE-complete is to reduce the infinite families to finitely many cases through the concept of "simulation".

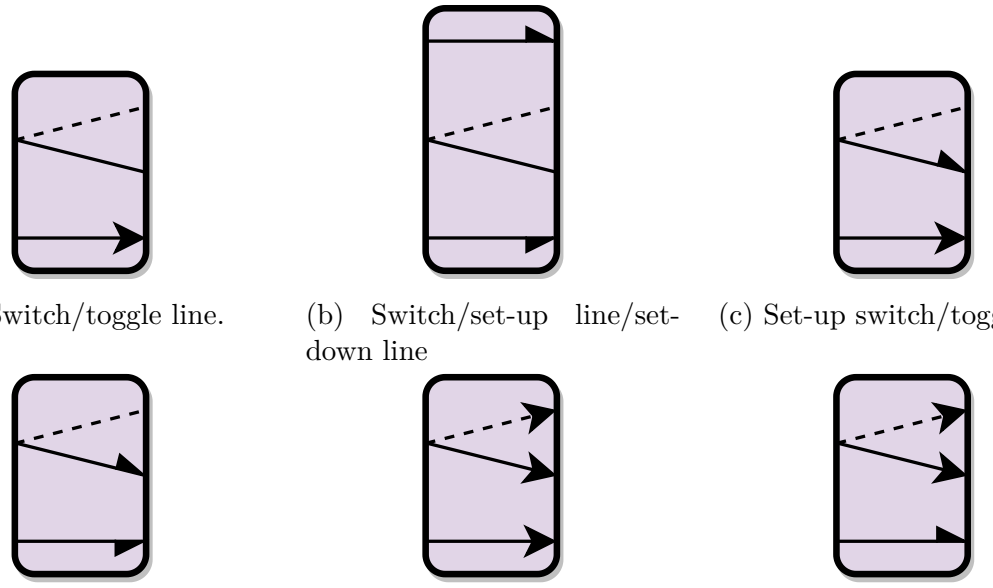
A *simulation* is a system of gadgets that is allowed to have 'dangling' outputs not connected to an input, with some inputs designated as *ports*. We also consider dangling outputs to be ports. A *simulation of* G is a simulation together with bijections from input ports to inputs of G and from output ports to outputs of G, that has the same behavior as G in the natural sense: a sequence of traversals from input ports to output ports is possible if and only if the corresponding sequence of traversals through G is possible. We say that G' *simulates* G if there is a simulation of G' containing only copies of G.

For reachability with deterministic gadgets, an equivalent condition is that, for each sequence of input ports, if the agent enters the simulation at each port in the sequence in order, the simulation sends the agent to the sequence of output ports corresponding to what would happen in G.

This definition is based on the definition of simulation for motion-planning gadgets [Hen21, CDD+22]. A simulation between input/output gadgets can be interpreted as a simulation between motion-planning gadgets, but the reverse is not true due to the stricter notion of 'system' for input/output gadgets.

Crucially, simulations yield logarithmic-space polynomial-time reductions: simply replace each copy of G with a copy of the system simulating it. In particular, simulations preserve hardness of zero-player and one-player reachability for NL, P, NP, and PSPACE.

To characterize all multi-input input/output gadgets, we show that they all simulate at least one of the eight gadgets listed in Lemma 3.1 and shown in Fig. 3.2 (bounded) and Fig. 3.3 (unbounded), and thus it will suffice to show hardness for these eight cases.



(a) Switch/toggle line.

(b) Switch/set-up line/set-down line

(c) Set-up switch/toggle line.

(d) Set-up switch/set-down line.

(e) Toggle switch/toggle line.

(f) Toggle switch/set-up line.

Figure 3.3: A basis for the unbounded multi-input gadgets: all such gadgets can simulate one of these six. We later show that Fig. 3.3b alone forms a one-gadget basis.

Lemma 3.1. *Let G be an output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs.*

- *If G is bounded, then it simulates either a switch/set-up line or a set-up switch/set-up line (Fig. 3.2).*
- *If G is unbounded, then it simulates one of the following gadgets (Fig. 3.3):*
  - *(a) switch/toggle line,*
  - *(b) switch/set-up line/set-down line,*
  - *(c) set-up switch/toggle line,*
  - *(d) set-up switch/set-down line,*
  - *(e) toggle switch/toggle line, or*
  - *(f) toggle switch/set-up line.*

Proof. First we *compress* every switch, set switch, and toggle switch, except for one, by merging (connecting) its two outputs. This operation transforms set switches into set lines, toggle switches into toggle lines, and ordinary switches into trivial lines. Fig. 3.4 shows an example. If the gadget has any ordinary switches, we use one of them as the switch that does not get compressed. The resulting gadget has the same boundedness as the original gadget, has a single switch of some type, and still has multiple nontrivial inputs: if it had only one nontrivial input, then the other inputs must have all been ordinary switches which

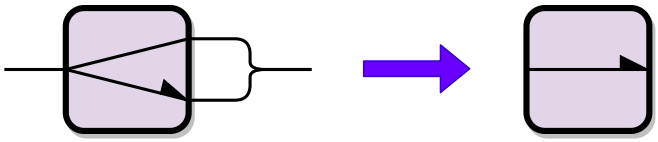


Figure 3.4: Compressing a set-up switch by merging its outputs yields a set-up line.

got compressed, so the remaining uncompressed input is also an ordinary switch, and thus the original gadget contained only ordinary switches and was trivial.

For multi-input bounded gadgets, we now have either a switch or a set switch (any sort of toggle would make the gadget unbounded), and at least one set line. Each set switch and line must set the gadget to the same state (which we can assume by symmetry is the up state), and we can ignore all but one set line. In particular, without loss of generality, the resulting gadget contains exactly a set-up line and either a switch (3.2a) or a set-up switch (3.2b).

For multi-input unbounded gadgets, there are multiple cases to consider based on the type of the single switch which was not compressed. First, if the switch is an ordinary switch, then there must be lines that can set the state in both directions, which must include either a toggle line (3.3a) or two set lines in different directions (3.3b). If the switch is a set switch, then there must be a line that can set the state in the opposite direction, which can be either a toggle line (3.3c) or a set line opposite the set switch (3.3d). Finally, if the switch is a toggle switch, then there must be some nontrivial line: either a toggle line (3.3e) or a set line (3.3f). We have made arbitrary choices for the directions of set lines and set switches; these are without loss of generality because we can reflect the gadget (or rename the up and down states). □

These simulation results are of independent interest. They show that there is a two-gadget *basis* for multi-input bounded input/output gadgets, and a six-gadget basis for multi-input unbounded input/output gadgets, where every gadget in each family can simulate at least one gadget in the basis. In fact, Section 3.2.3 shows the stronger result that multi-input unbounded input/output gadgets have a one-gadget basis, namely, the switch/set-up line/set-down line of Fig. 3.1 or 3.3b. Past work on motion-planning gadgets [DHL20] established a one-gadget basis for a particular gadget family: every reversible deterministic interacting-k-tunnel gadget can simulate a locking 2-toggle.

At the other extreme from a basis, we can ask for *universality*. For example, each door gadget from [BDD+20] simulates *every* motion-planning gadget, and there are several universality results for classes of gizmos [Hen21]. In Section 3.3.1, we prove universality results for input/output gadgets: the same switch/set-up line/set-down line of Figs. 3.1 and 3.3b simulates every deterministic input/output gadget (not just those that are output-disjoint and 2-state). Thus the switch/set-up line/set-down line both simulates and can be simulated by every unbounded multi-input output-disjoint deterministic 2-state input/output gadget, and thus every such gadget also has this property. For one-player reachability, we prove that the switch/set-up line/set-down line and fanout together simulate every input/output gadget, and, when viewed as motion-planning gadgets, simulate every motion-planning gadget. This again also applies to every unbounded multi-input output-disjoint deterministic

2-state input/output gadget.

We also take universality a step further, considering infinitely large simulations. In this case, we prove that the switch/set-up line/set-down line simulates every countable deterministic input/output gadget, with an analogous collection of corollaries.

Finally, in Section 3.3.2 we prove that certain gadgets *can't* simulate other gadgets, by proving that 'nonexpanding' and 'boundedly expanding' gadgets are closed under simulation.

3.2 Zero Players

In this section, we study the complexity of reachability with deterministic input/output gadgets from several classes. In Section 3.2.1, we consider such gadgets with a single input. In Section 3.2.2, we consider bounded gadgets with multiple inputs, which are naturally P-complete. Finally, in Section 3.2.3 we consider unbounded gadgets with multiple inputs, which are naturally PSPACE-complete.

Lemma 3.2. *Reachability with deterministic input/output gadgets is in PSPACE.*

Proof. In polynomial space, we can keep track of the current configuration of a system of gadgets and current location of the agent. Thus we can simply simulate the deterministic agent until either it reaches the goal location, or it makes more transitions than there are configurations, and thus is stuck in a cycle. □

3.2.1 Single Input

In this section, we consider reachability with deterministic single-input input/output gadgets. If the gadgets are described (for concreteness, using transition graphs) as part of the instance, this is equivalent to the explicit zero-player reachability switching games of [FGMS21]. In our language, [FGMS21] shows that reachability with instance-specified deterministic single-input input/output gadgets is NL-hard. As pointed out in [FGMS21], the proofs in [GHH+18], which only considered ARRIVAL, also apply to explicit zero-player reachability switching games. In our language, they show that reachability with instance-specified deterministic single-input input/output gadgets is in UP ∩ coUP (which is contained in NP ∩ coNP).

We strengthen the NL-hardness result of [FGMS21] by showing that reachability with just the toggle switch is NL-hard. This is a straightforward modification of the proof of NL-hardness in [FGMS21]; we present the full argument for completeness and to translate it to our terminology. There is still a large gap between the lower bound of NL-hard and the upper bound of UP ∩ coUP.

Theorem 3.3. *Reachability with the toggle switch is NL-hard.*

Proof. We reduce from reachability in directed graphs, which is NL-complete [Wig92].

First we modify the graph to have out-degree 0 or 2 at every vertex without changing reachability; refer to Fig. 3.5. We replace every vertex v with out-degree $k > 2$ with a sequence of k vertices each with out-degree at most 2: if v has edges to $w_1, \ldots, w_k$, we replace v with $v_1, \ldots, v_k$ with edges $v_i \to v_{i+1}$ and $v_i \to w_i$, and edges to v now go to v_1.

Then we remove any vertices with out-degree 1 by setting their incoming edges to instead go to the target of their unique outgoing edge. This reduction to where every vertex has out-degree exactly 2 can be done in logarithmic space and does not affect reachability.

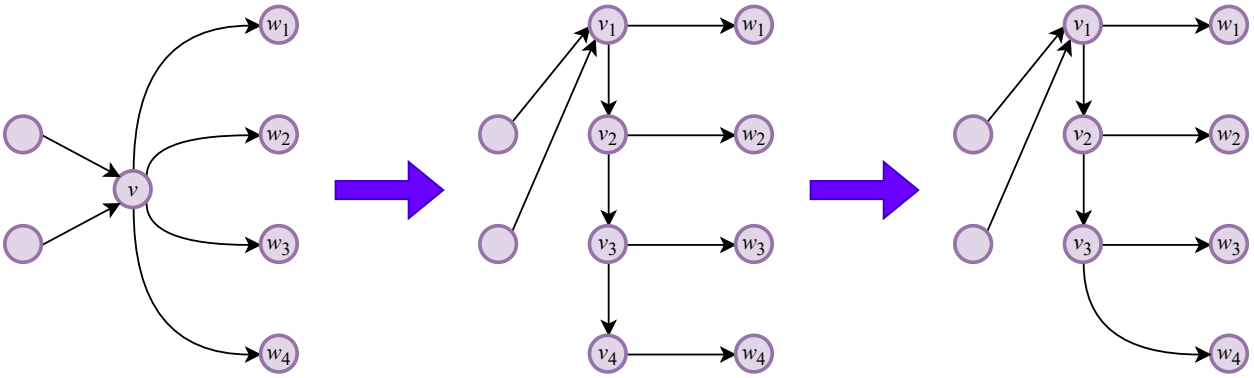


Figure 3.5: Modifying a directed graph to have out-degree 0 or 2 at every vertex: splitting vertices of high out-degree, and removing vertices of out-degree 1.

Now we use a construction based on that in [FGMS21]; refer to Fig. 3.6. Let V be the set of vertices in the modified graph G, where we are interested in a path from s to t. Our system of gadgets has $|V|^2$ toggle switches, named (v, i) for $v \in V$ and $1 \leq i \leq |V|$. For a vertex $v \neq t$ with edges to w_1 and w_2 and $i < |V|$, the outputs of (v, i) are connected to the inputs of $(w_1, i+1)$ and $(w_2, i+1)$. For a vertex $v \neq t$ with out-degree 0, both outputs of (v, i) are connected to the input of $(s, 1)$. For $v \neq t$, both outputs of $(v, |V|)$ are connected to the input of $(s, 1)$. Finally, for each i, both outputs of (t, i) are connected to the goal location, which then leads back to $(s, 1)$. The start location is the input of $(s, 1)$.

When the agent moves through this system, it follows paths in G starting from s and counts the number of steps taken, resetting after $|V|$ steps or when it reaches a vertex with out-degree 0. By construction, a toggle switch (v, i) is reachable from $(s, 1)$ exactly when there is a path of length $i - 1$ from s to v. If the agent reaches the goal location, it must have entered (t, i) for some i, and thus there is a path (of length $i - 1$) from s to t.

Because all paths eventually return to $(s, 1)$, the agent must enter $(s, 1)$ infinitely many times, so it must use each output of $(s, 1)$ infinitely many times. By induction, it uses every toggle switch reachable from $(s, 1)$ infinitely many times. If there is a (simple) path from s to t, it has some length $i < |V|$, so $(t, i+1)$ is reachable from $(s, 1)$. Then $(t, i+1)$ is visited infinitely many times, so the agent reaches the goal location. □

3.2.2 Bounded Gadgets

In this section, we consider the complexity of reachability with a bounded output-disjoint deterministic 2-state input/output gadget which has multiple nontrivial inputs. We will find that this problem is always P-complete.

A gadget is *bounded* if the number of times it can change states is bounded; this generalizes the definition in Section 3.1.2.

Theorem 3.4. *Reachability with bounded deterministic input/output gadgets is in P.*

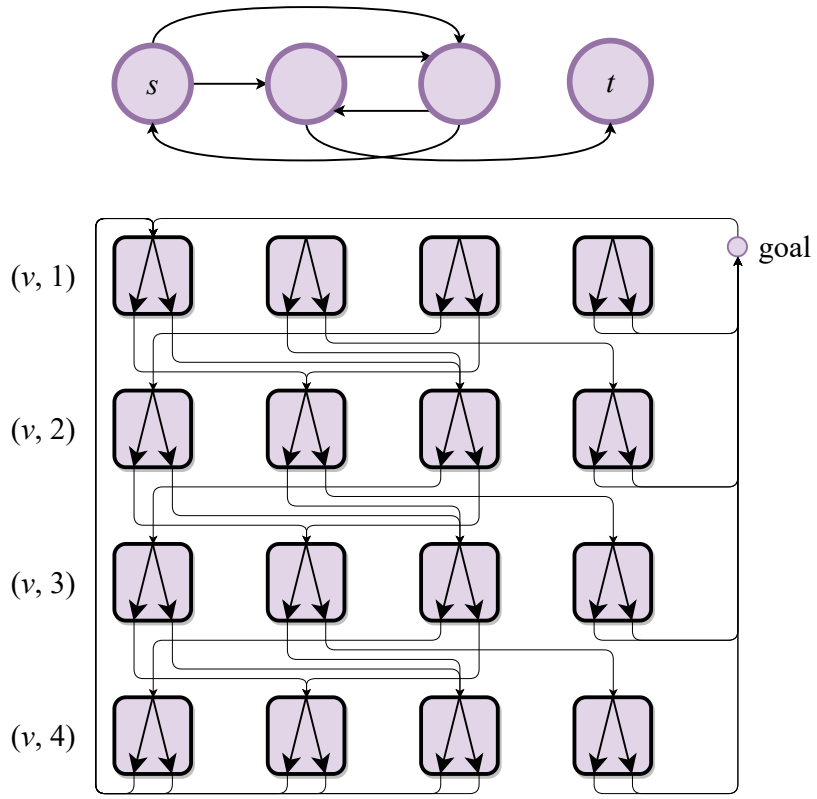


Figure 3.6: Reduction from reachability in directed graphs of out-degree 0 or 2 to reachability with the toggle switch.

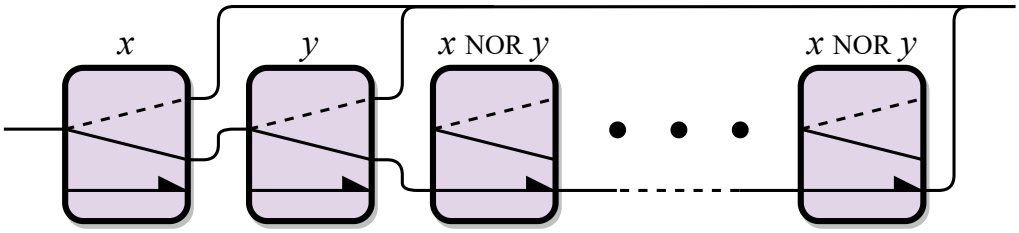


Figure 3.7: A NOR gate for P-hardness of reachability with the switch/set-up line. If neither x nor y is set to true (up), the agent sets each x NOR y gadget to true.

Proof. Suppose we have a system with n copies of the gadget. Let k be the maximum number of state changes a gadget in the system can make, and let i be the maximum number of input locations a gadget in the system has. Then gadget states can change at most kn times. Between consecutive state changes, the agent can visit each input of each gadget at most once (otherwise it is stuck in a cycle), so consecutive state changes are separated by at most in traversals. Hence after ikn^2 traversals, the agent must be in a cycle, which involves no state changes of length at most in. So we can solve the problem in polynomial time by simulating the agent for $in(kn+1)$ steps and seeing whether it reaches the goal location by then. □

Lemma 3.1 tells us that every output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs simulates either the switch/set-up line or the set-up switch/set-up line. Thus to prove that reachability with any such gadget is P-hard, it suffices to show P-hardness for these particular two gadgets. This is what we do for the remainder of this section.

Theorem 3.5. *Reachability with the switch/set-up line or the set-up switch/set-up line is P-hard (under logarithmic space reductions).*

Proof. We provide a reduction to each of these problems from the problem of evaluating a circuit containing only NOR gates and fan-out, with the gates listed in a topological order. This restricted version of circuit evaluation is known to be P-complete [GHR95]. The two reductions are nearly identical: we present the reduction for the switch/set-up line, and the reduction for the set-up switch/set-up line is the same with each gadget replaced. We shall see that the agent never goes over a switch multiple times, so these two systems of gadgets behave the same.

Our reduction builds a system of switch/set-up lines which has one gadget for each input of a NOR gate; this gadget indicates whether the input is true or false, and is initially set to false. The agent will evaluate each NOR gate in the order they are listed in the input, setting the gadgets for outputs of that gate to true if appropriate. This is accomplished with the gadget in Fig. 3.7. For each NOR gate, we build one of these gadgets, where x and y are the inputs, and the gadgets labelled x NOR y are the outputs (and inputs of other NOR gates). There are as many output gadgets as the fan-out of this NOR gate. The entrance and exit to the NOR gate gadgets are connected in series, in the given order of the gates.

To complete the construction, we place the start location at the entrance to the first NOR gate. The exit of the last NOR gate enters a switch which holds the output of the final NOR

gate, and the goal location is the top output of that switch. Every switch/set-up line starts in the down state except for those that correspond to true inputs to the circuit.

When the agent moves through this system of gadgets, it goes through each NOR gate in order. Because the input circuit was given with gates in topological order, the agent goes through both gates that provide the inputs x and y to a gate that computes x NOR y before going through that gate itself. If either x or y is set to true (i.e., in the up state), the agent leaves x NOR y false, but if x and y are both false, it goes through the set-up lines to set x NOR y true. This correctly computes x NOR y, and by induction it computes the value of the circuit. At the end, the agent reaches the goal location if the value is true and gets stuck in a nearby dead-end (actually a small inescapable loop) if the value is false. □

By the basis simulation result of Lemma 3.1, these two cases establish hardness for all multi-input output-disjoint deterministic 2-state input/output gadgets:

Corollary 3.6. *Reachability with any bounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs is P-complete.*

3.2.3 Unbounded gadgets

In this section, we consider reachability with an unbounded output-disjoint deterministic 2-state input/output gadget which has multiple nontrivial inputs. We show that this problem is PSPACE-complete for every such gadget through a reduction from Quantified Boolean Formula (QBF), which is PSPACE-complete, to reachability with the switch/set-up line/set-down line, and by showing that every such gadget simulates the switch/set-up line/set-down line. We also show that the switch/set-up line/set-down line (and thus every unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs) can simulate every deterministic input/output gadget in reachability.

Tunnel duplicators

Many of our simulations involve building an *tunnel duplicator*, shown in Fig. 3.8. A tunnel duplicator is a construction which allows us to effectively make a copy of a line from X to X' in a gadget. This is achieved by routing two inputs A and B to X, and then sending the agent from X' to one of two outputs A' or B' corresponding to the input used. The details of the construction of a tunnel duplicator depend on the gadget used; see Fig. 3.9 for an example.

If we have access to a tunnel duplicator, we can duplicate tunnels in gadgets. Note that this is not enough to duplicate switches, since we would have to account for both outputs getting duplicated.

PSPACE-Hardness of the Switch/Set-Up Line/Set-Down Line

In this section, we show that reachability with the switch/set-up line/set-down line is PSPACE-hard through a reduction from QBF. Recall from Fig. 3.1 or 3.3b that the switch/set-up line/set-down line is a 2-state input/output gadget with three inputs: one sets the state to up, one sets it to down, and one sends the agent to one of two outputs based on the current state.

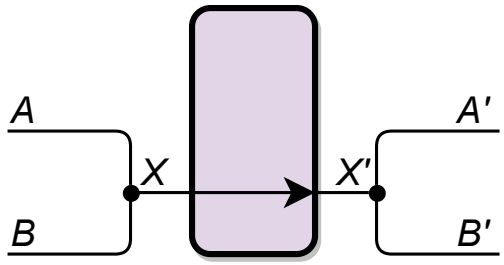


Figure 3.8: The schematic of a tunnel duplicator. An agent entering at A or B exits at A' or B', respectively, having gone over the central path. This duplicates the tunnel in the center.

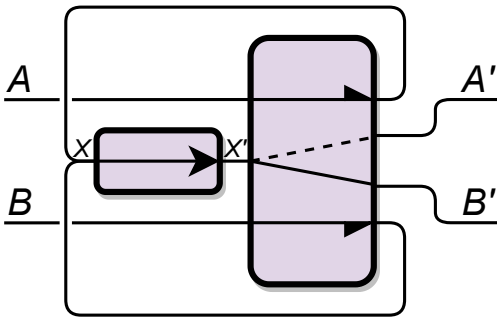


Figure 3.9: An tunnel duplicator for the switch/set-up line/set-down line. A agent entering on the left sets the state of the switch, goes across the duplicated tunnel (which is one subunit of some bigger gadget), and exits based on the state it set the switch to.

Theorem 3.7. *Reachability with the switch/set-up line/set-down line is PSPACE-hard.*

Proof. First we build an tunnel duplicator, shown in Fig. 3.9. This allows us to use gadgets with multiple set-up or set-down lines.

Next we present a reduction from QBF. Given a quantified Boolean formula where the unquantified formula is 3-CNF, we construct a system of gadgets which evaluates the formula, ultimately sending the agent to one of two locations based on its truth value. The system consists of a sequence of *quantifier gadgets*, which set the values of variables, followed by the *CNF evaluation*, which checks whether the formula is satisfied by a particular assignment and reports this to the quantifier gadgets.

Each quantifier gadget has three inputs, called In, True-In, and False-In, and three outputs, called Out, True-Out, and False-Out. The agent will always first arrive at In. This sets the variable controlled by that quantifier to true, and the agent leaves at Out, which sends it to the next quantifier gadget. Eventually the agent will return to either True-In or False-In, depending on the truth value of the rest of the quantified formula with the variable set to true. Depending on the result, the quantifier gadget either sends the agent to True-Out or False-Out to pass this message to the previous quantifier gadget, or the quantifier gadget sets its variable to false, and again sends the agent to the next quantifier. When it gets a truth value in response the second time, it sends the appropriate truth value to the previous quantifier. The last quantifier communicates with the CNF evaluation instead of with another quantifier.

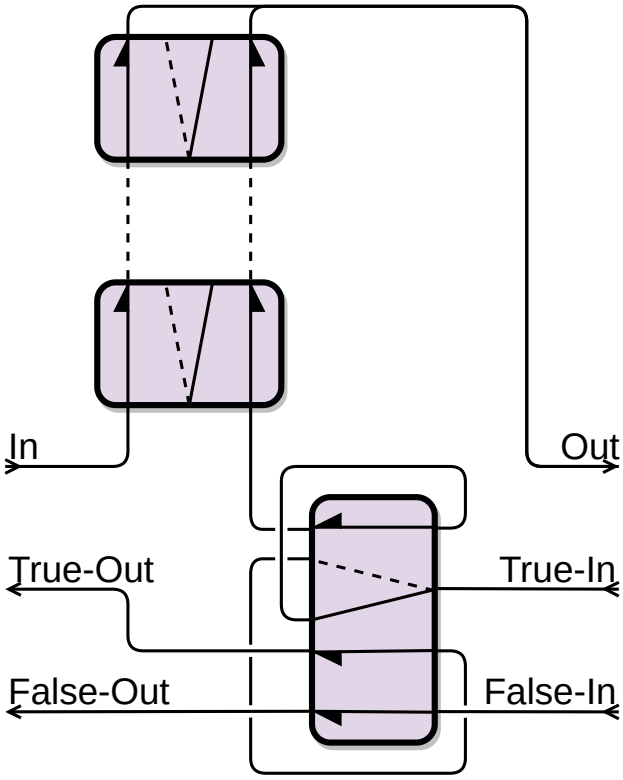


Figure 3.10: The universal quantifier for the switch/set-up line/set-down line. An tunnel duplicator (Fig. 3.9) is used to give the bottom gadget two set-down lines.

The universal quantifier gadget is shown in Fig. 3.10. The chain of gadgets at the top encodes the state of the variable controlled by this quantifier, and has as many gadgets as there are instances of the variable in the formula. The variable is true when they are set to the 'left' state and false when they are set to the 'right' state, where the direction refers to the position, in the figure as drawn, of the output which would be taken if the agent enters the switch.

When the agent enters In, it sets the variable to true and exits Out. If it then returns to True-In, the first time it takes the bottom branch of the switch, sets that gadget to the up state, sets the variable to false, and exits Out again. If it returns to True-In a second time, that means the rest of the formula was true for both settings of the universally quantified variable: it takes the top branch, resets that gadget to down, and exits True-Out. If after either trial the agent enters at False-In, it resets the bottom gadget to the down state and exits False-Out. This is the intended behavior of the universal quantifier: it reports true if the result was true for both settings of the variable, and false otherwise.

The existential quantifier is identical except that True-Out and False-Out are swapped, and True-In and False-In are swapped. It reports false if the result was false for both settings, and true otherwise.

For CNF evaluation, we use the switches controlled by each quantifier to read the value of a variable. For each clause, the agent passes through a switch corresponding to each of the literals in the clause. If all three literals are false, it exits False-Out. Otherwise, it moves

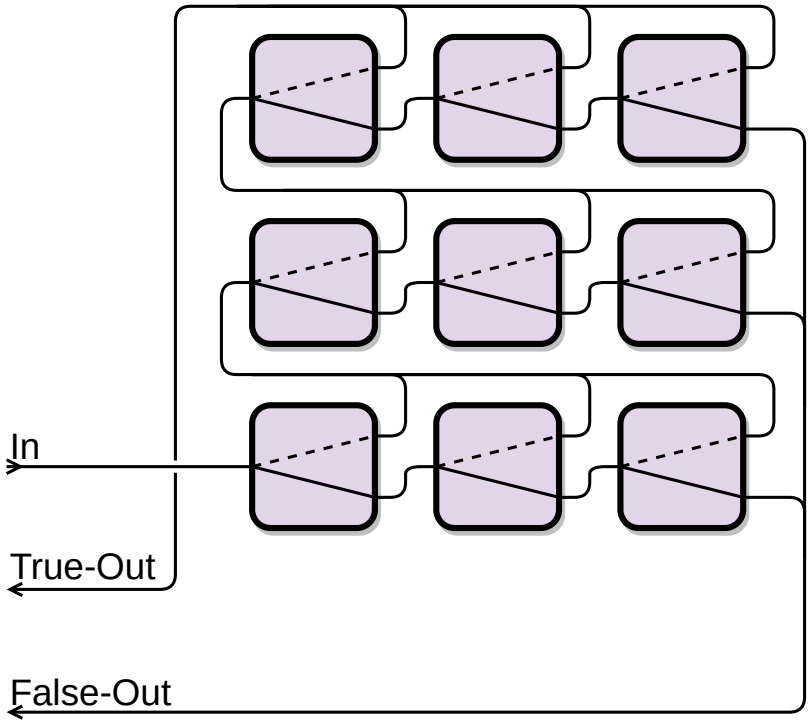


Figure 3.11: Three clauses of CNF evaluation for the switch/set-up line/set-down line; each clause is a row of three switches. The switches are part of gadgets in the quantifiers. We assume the top output of each switch corresponds to that literal being true; all literals are set to false in this image.

on to the next clause, eventually exiting True-Out if all clauses are satisfied. This is shown, for 3 clauses, in Fig. 3.11. Ultimately, the agent exits True-Out or False-Out depending on whether the formula is satisfied by the current assignment.

It follows by induction that, for each quantifier, when the agent arrives at In, it will eventually leave either True-Out or False-Out depending on the truth value of the suffix of the formula beginning with that quantifier under the assignment of the earlier quantifiers. Thus, if the agent starts in the first quantifier at In, it reaches True-Out on the first quantifier if and only if the formula is true. We place the goal location at True-Out on the first quantifier, and have both True-Out and False-Out lead to dead-ends, implemented as infinite loops through a single gadget. □

Other Gadgets Simulate the Switch/Set-Up Line/Set-Down line

In this section, we show that every unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs simulates the switch/set-up line/set-down line. In other words, the switch/set-up line/set-down line forms a one-gadget basis for unbounded output-disjoint deterministic 2-state input/output gadgets. We only need to show that the five other gadgets from Lemma 3.1 simulate the switch/set-up/set-down. It follows that reachability with any such gadget is PSPACE-complete, since we can replace each gadget in a system of switch/set-up/set-down with a simulation of it.

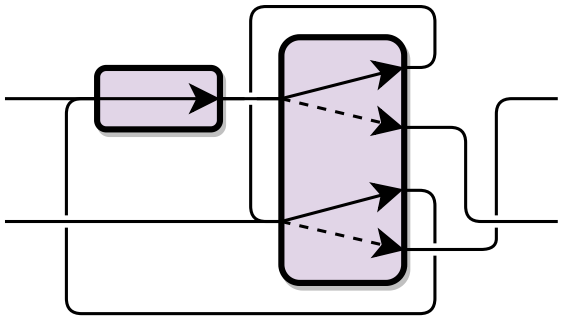


Figure 3.12: A tunnel duplicator for the toggle switch/toggle switch. The tunnel on the left is duplicated.

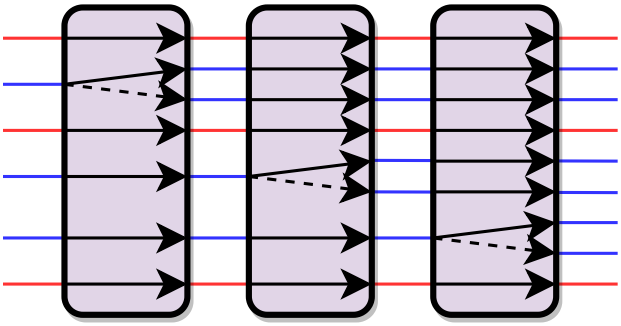


Figure 3.13: A simulation of three toggle lines and three toggle switches from gadgets with one toggle switch and 5, 6, and 7 toggle lines. The red tunnels are toggle lines and the blue tunnels are toggle switches.

Toggle Switch/Toggle Switch. We begin with the toggle switch/toggle switch, which is not part of our basis of gadgets from Lemma 3.1, but will be a useful intermediate gadget. It simulates a tunnel duplicator, as shown in Fig. 3.12. We can merge the two outputs of one of the toggle switches to simulate a toggle switch/toggle line, and then duplicate the toggle line to make a gadget with one toggle switch and any number of toggle lines.

By putting such gadgets in series, we can simulate a gadget with any number of toggle lines and any number of toggle switches. Fig. 3.13 shows this for three toggle lines and three toggle switches, which is as large as we need. This simulated gadget can finally simulate the switch/set-up line/set-down line, as shown in Fig. 3.14.

Toggle Switch/Toggle Line. We simulate the toggle switch/toggle switch using toggle switch/toggle lines, as shown in Fig. 3.15.

Switch/Toggle Line. First we build a tunnel duplicator, shown in Fig. 3.16. Then we can duplicate the toggle line and put one copy in series with the switch, constructing a toggle switch/toggle line.

Set-Up Switch/Toggle Line. First we build a tunnel duplicator, shown in Fig. 3.17. Then we simulate the switch/toggle line, shown in Fig. 3.18.

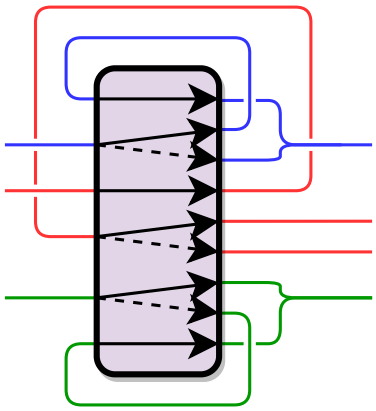


Figure 3.14: A simulation of a switch/set-up line/set-down line, currently in the down state, from the gadget built in Fig. 3.13. Each component of the switch/set-up line/set-down line is made from one toggle line and one toggle switch; the switch, set-up line, and set-down line are red, green, and blue, respectively.

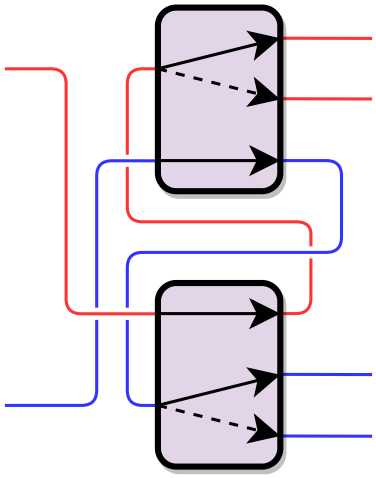


Figure 3.15: A simulation of a toggle switch/toggle switch from the toggle switch/toggle line. Each color corresponds to one of the toggle switches.

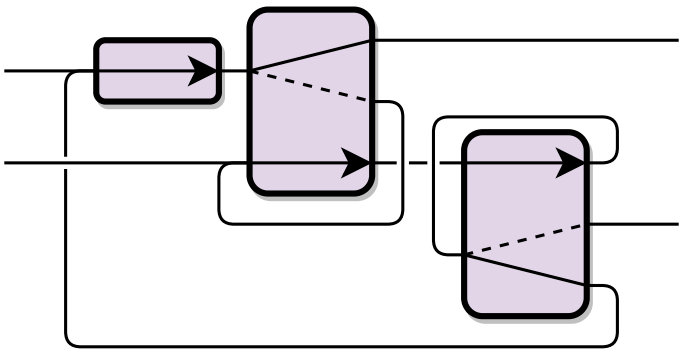


Figure 3.16: A tunnel duplicator for the switch/toggle line. The leftmost tunnel is duplicated.

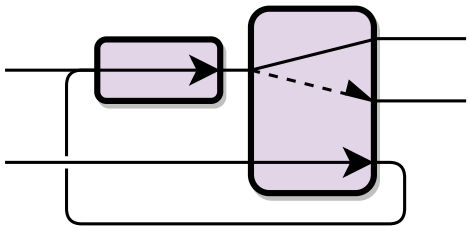


Figure 3.17: A tunnel duplicator for the set-up switch/toggle line. The leftmost tunnel is duplicated.

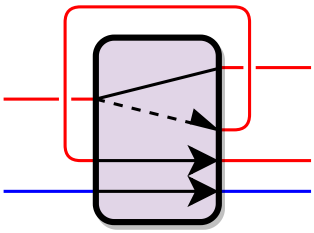


Figure 3.18: A simulation of a switch/toggle line using the set-up switch/toggle line. Red corresponds to the switch and blue corresponds to the toggle line.

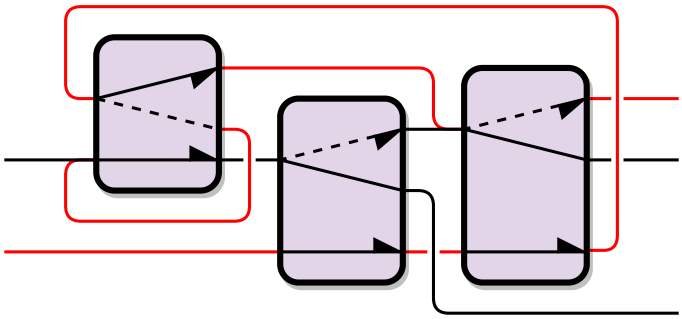


Figure 3.19: A simulation of a set-down switch/toggle line using the set-up switch/set-down line. When the agent is not inside the simulation, the rightmost gadget is in the down state and the other two gadgets are in opposite states and encode the state of the simulated gadget. Red lines indicate the toggle line: when the agent enters the bottom input, it takes one of the internal paths depending on the state and exits the top output, reversing the state of the left and middle gadgets. When it enters the top input, it exits one of the bottom two outputs and resets the state to down.

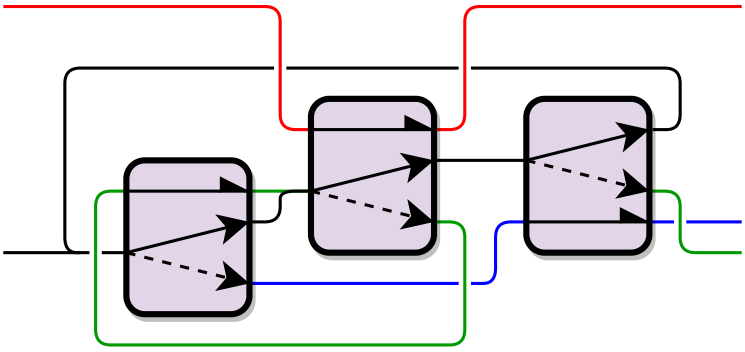


Figure 3.20: A simulation of a set-up line/set-down switch from the set-up line/toggle switch. The state of the simulated gadget is the same as the state of the center gadget. The red path corresponds to the set-up line. When it enters the set-down switch, the agent goes along the blue lines if the state is down, the green lines if the state is up, and the black lines in both cases.

Set-Up Switch/Set-Down Line. We simulate a set-down switch/toggle line (equivalent to a set-up switch/toggle line) using the set-up switch/set-down line, as shown in Fig. 3.19.

Toggle Switch/Set-Up Line. We simulate a set-up line/set-down switch using the toggle switch/set-up line, as shown in Fig. 3.20; this is equivalent to a set-up switch/set-down line.

These simulations, together with Lemma 3.1, give the following theorem:

Theorem 3.8. *Every unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs simulates the switch/set-up line/set-down line.*

Corollary 3.9. *Let G be an unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs. Then reachability with G is PSPACE-complete.*

Proof. Containment in PSPACE is given by Lemma 3.2. All of our simulations preserve PSPACE-hardness: we can reduce from reachability with the switch/set-up line/set-down line (shown PSPACE-hard in Section 3.2.3) to reachability with G by replacing each gadget in a system of switch/set-up line/set-down lines with a simulation built from G. The resulting system of G has the same behavior as the system of switch/set-up line/set-down lines. □

3.3 Simulation

In this section, we consider some theoretical questions about which input/output gadgets can simulate each other. These results have no direct bearing on questions of computational complexity.

3.3.1 Universality

In this section, we show how to simulate an arbitrary deterministic input/output gadget using the switch/set-up line/set-down line; i.e., that this gadget is universal for all deterministic

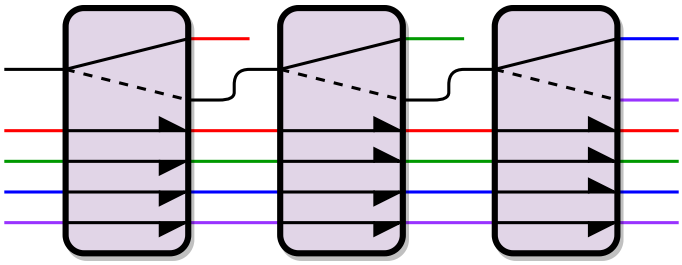


Figure 3.21: A simulation of a 4-switch using the switch/set-up line/set-down line. Colors indicate the outputs corresponding to set lines.

input/output gadgets. We also observe some interesting consequences of this result—of particular note is Corollary 3.13 that, as a motion-planning gadget, the switch/set-up line/set-down line simulates every gadget; i.e., is fully universal (just like the doors of [BDD+20]).

Theorem 3.10. *The switch/set-up line/set-down line simulates every deterministic input/output gadget.*

Proof. We present simulations of gradually more powerful gadgets. First, the tunnel duplicator (Fig. 3.9) lets us have any number of copies of the set-up and set-down lines.

Next, we simulate a generalization of the switch/set-up line/set-down line which we call the k-switch. This gadget has k states, k lines which each set the gadget to a particular state, and an input which does not change the state and sends the agent to one of k locations depending on the state. The switch/set-up line/set-down line is a 2-switch. The simulation for $k = 4$ is shown in Fig. 3.21, and generalizes easily to arbitrary k: we need $k-1$ gadgets connected in series, where the ith gadget has i set-up lines and $k-1-i$ set-down lines.

We now duplicate the large switch in a k-switch using the construction in Fig. 3.22. Thus the switch/set-up line/set-down line can simulate a gadget with any number of states, any number of lines which set it to a particular state, and any number of inputs which send the agent to different outputs depending on the state but do not change the state.

Finally, let G be an arbitrary deterministic input/output gadget. If G has k states and m input locations, we use a k-switch with m copies of the switch to simulate G. The m inputs lead directly to the m switches. For each transition $(s, a) \to (t, b)$ of G—meaning that when the agent enters at a in state s, it exits at b and changes the state to t—we connect the output taken in s of the switch corresponding to a to a line which sets the state to t, and connect the output of that line to b. This encodes the correct behavior for that transition. Because G is deterministic, there is only one such transition for each pair (s, a), so we only connect each output of a switch to one input location, as is required. □

Corollary 3.11. *Every unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs simulates every deterministic input/output gadget.*

Corollary 3.12. *The switch/set-up line/set-down line and the fanout together simulate every input/output gadget.*

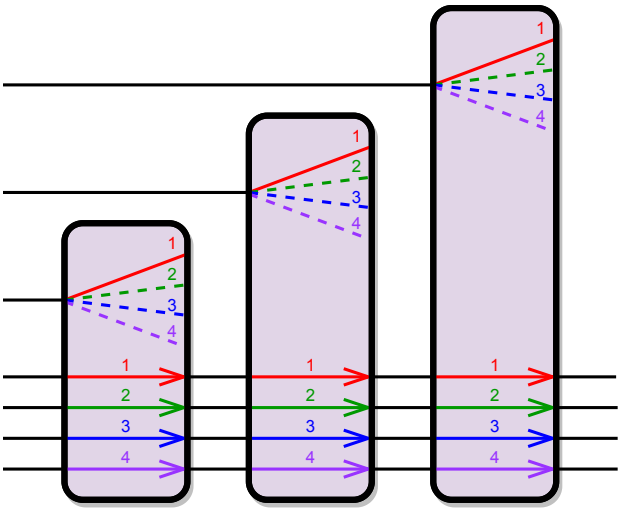


Figure 3.22: Simulating a 4-switch which has three copies of the switch.

Proof. We use the same construction as in the proof of Theorem 3.10. If G is nondeterministic—say it has multiple transitions when entering a in state s—then we place fanouts to connect the output taken in s of the switch corresponding to a to multiple input locations. □

Corollary 3.13. *As a motion-planning gadget, the switch/set-up line/set-down line simulates every gadget.*

Proof. Let G be an arbitrary gadget. We define an input/output gadget G' with the same states as G, an input a_{in} and an output a_{out} for each location a of G, and a transition $(s, a_{in}) \to (t, b_{out})$ for each transition $(s, a) \to (t, b)$ of G. To ensure G' always has at least one available transition, we include each transition $(s, a_{in}) \to (s, a_{out})$. By Corollary 3.12, the switch/set-up line/set-down line (as an input/output gadget) simulates G'. But, as a motion-planning gadget, G' simulates G simply by connecting both a_{in} and a_{out} to a. □

Corollary 3.14. *Every unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs, when considered as a motion-planning gadget, simulates every gadget.*

If we allow infinitely large simulations, we can even simulate gadgets with infinitely many states or locations. However, the proof of Theorem 3.10 fails, because simulating an ∞-switch would require the agent to go through infinitely many set lines, and it would never reach the end. This raises some definitional questions regarding infinite simulations, which we will avoid by ensuring that the path taken by the agent is always finite in our simulations. Note that this property is preserved by composition of simulations. This is consistent with the approach taken for 'arbitrary simulation' of gizmos [Hen21].

We say a gadget is *countable* if it has countably (or finitely) many locations. This implies that there are countably many finite sequences of traversals through the gadget, and thus countably many reachable states. We ignore any unreachable states.

Theorem 3.15. *With infinite simulations, the switch/set-up line/set-down line simulates every countable deterministic input/output gadget.*

Proof. We follow the structure of Theorem 3.10, but with new simulations. It will be convenient to use set-up switch/set-down lines in our construction; we know how to simulate these. First, we create an infinite tunnel duplicator, shown in Fig. 3.23. This is just a chain of infinitely many standard tunnel duplicators, but we must check that the agent's path is always finite. Suppose the agent enters at tunnel k. Initially each set-up switch/set-down line is in the up state, and the agent sets the kth to the down state. It then crosses the duplicated tunnel and goes through only the first k gadgets before exiting.

Next, we simulate the *ω-switch*, which has countably many states, a set line for each one, and a single switch that sends the agent to one of countably many outputs. This simulation is shown in Fig. 3.24. It is very similar to our infinite tunnel duplicator, but there are additional switch/set-up line/set-down lines for the infinitely large switch. When the agent enters at the kth set line, it sets the first $k-1$ of these gadgets to the down state and the kth to the up state, without affecting the others. When the agent enters the switch, its output depends only which switch/set-up line/set-down line is the *first* one in the up state. Note that this simulation has many (indeed, uncountably many) configurations corresponding to each state of the ω-switch, because the state of each switch/set-up line/set-down line after the first one that is up is irrelevant.

It is not difficult to extend this to simulate a version of the ω-switch with two infinite switches, or with any finite number of them. But we need the *ω^2-switch*, which is like the ω-switch but has countably many copies of the switch (and one copy of each set line, which can be duplicated as needed). This simulation is more involved, and is shown in Fig. 3.25. When the agent enters a set line, it simply sets the state of the leftmost ω-switch, 'master copy' of the state of the entire construction. When the agent enters a switch, it needs to copy this state to the relevant ω-switch. It proceeds down the chain of ω-switches, setting them all to the same state, until it reaches one of the set-up switches along the bottom that is in the down state. The agent then heads back to the top, where it goes through the set-down line/set-up switches to find the ω-switch next to where it first entered. If the state was successfully copied to that gadget, the agent exits. Otherwise, that ω-switch is still in the purple 'unknown' state, and we try again: the agent again goes through the chain of ω-switches, copying the state until it reaches a set-up switch in the down state. Because of the set-up switches, each state-copying operation will go one more step down the chain before returning. To use the kth switch, the agent performs up to k state-copying operations (possibly fewer, if switches have been used already), and eventually exits the ω^2-switch.

Finally, let G be an arbitrary countable deterministic input/output gadget. We simulate G as in Theorem 3.10, using a single ω^2-switch with each set line duplicated infinitely many times. Each input leads to one of the switches. For each transition $(s, a) \to (t, b)$, we connect the output in state s of the switch corresponding to a to a copy of the set line for state t, which then leads to b. □

Corollary 3.16. *With infinite simulations, every unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs simulates every countable deterministic input/output gadget.*

Corollary 3.17. *With infinite simulations, the switch/set-up line/set-down line and the fanout together simulate every countable input/output gadget.*

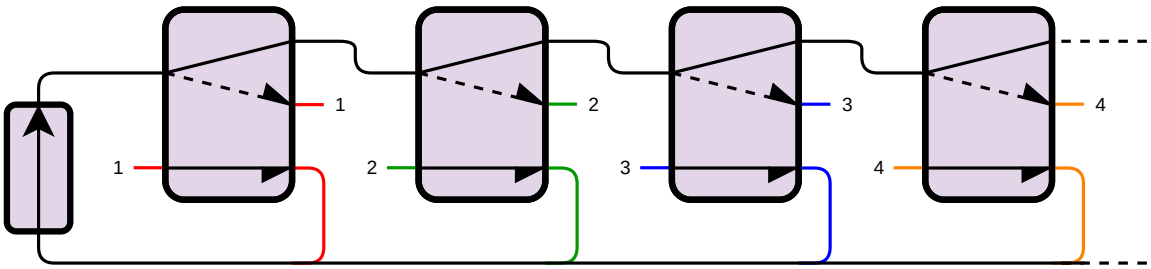


Figure 3.23: An infinite tunnel duplicator using set-up switch/set-down lines. The leftmost tunnel is duplicated. Colors indicate the edges used by only one path through the construction.

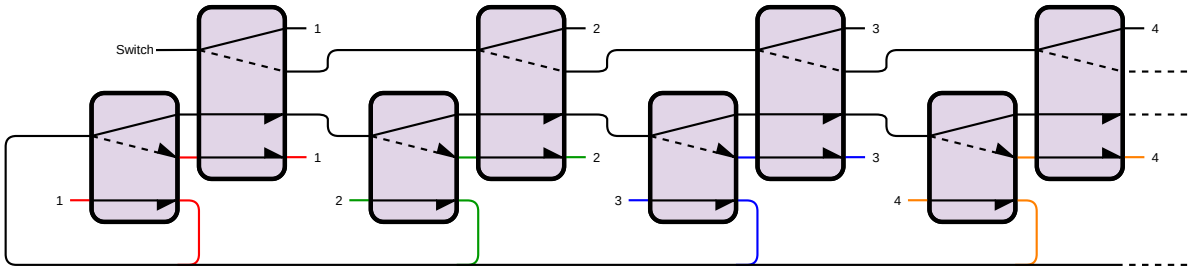


Figure 3.24: An infinite simulation of an ω-switch using the switch/set-up line/set-down line, shown in state 1. Colors indicate the path taken on each set line, and labels indicate the correspondence between set lines and outputs of the infinite switch.

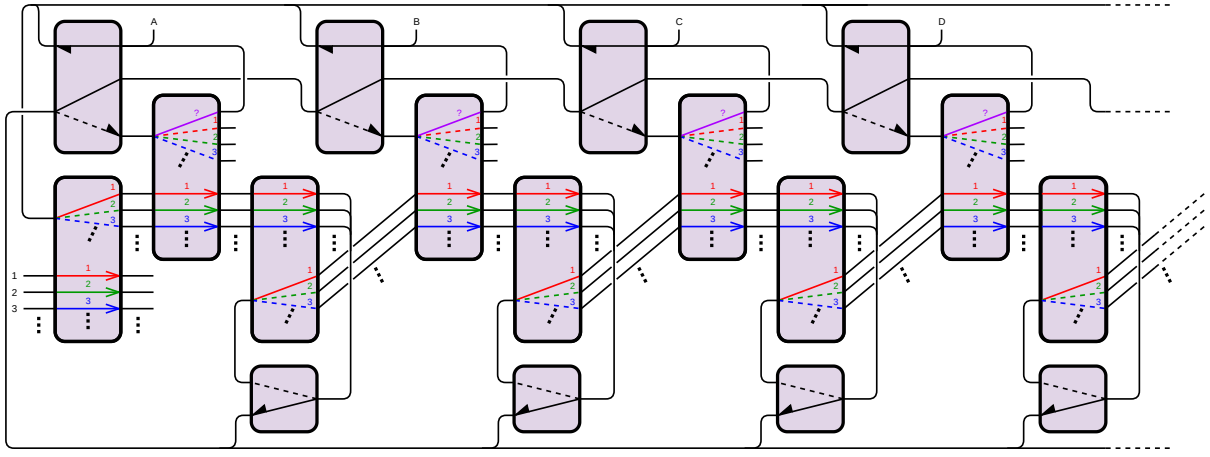


Figure 3.25: An infinite simulation of an ω^2-switch using the ω-switch and some simple finite gadgets. Numbers and colors correspond to states of the simulated gadget, except that purple is a special state indicating 'unknown'. The set lines use only the leftmost ω-switch. The input to each switch is labelled with a letter, and the corresponding outputs are on the nearest ω-switch.

Proof. We must address the situation where there are infinitely many available transitions when entering a in state s. Since the gadget is countable, there are only countably many such transitions. We can create a countable fanout using an infinite chain of ordinary fanouts, so that the top exit of the kth fanout leads to the kth available transition (similar to the top of Fig. 3.24). □

3.3.2 Closure

We now explore some situations in which we can prove that an input/output gadget doesn't simulate another input/output gadget. This comes in the form of defining a property of gadgets which is *closed under simulation*, meaning that in any simulation using gadgets with the property, the simulated gadget also has the property. For example, it is not hard to show that bounded gadgets can only simulate other bounded gadgets.

Our most interesting closure result is inspired by Chalcraft and Greene's proof that the 'distributor' (our toggle switch) can't be built from 'lazy points' (essentially our set-up line/set-down line/switch with the outputs of set lines merged) and 'sprung points' (essentially a fan-in) [CG94]. But we begin with a simpler similar class of gadgets.

Definition 3.18. A *subgadget* of an input/output gadget is induced by a subset of the input locations and a starting state, and contains the output locations, states, and transitions reachable from the starting state using the provided input locations.

Definition 3.19. An input/output gadget G is *nonexpanding* if every subgadget of G has at least as many inputs as outputs.

Most nonexpanding gadgets are uninteresting: they include the gadgets we called 'trivial' in Section 3.1.2 and little else.

Theorem 3.20. *Nonexpanding gadgets are closed under simulation.*

Proof. Let G be a nonexpanding gadget, and consider a simulation using G. We consider input ports to be 'outputs' and output ports to be 'inputs', so that the signal is always sent from an output to an input.

We clean up the simulation by deleting any obviously unreachable locations, meaning inputs (including output ports) with no output leading to them, and outputs not in the subgadget induced by the remaining input locations and starting state of each copy of G.

The connections between locations define a function from outputs to inputs. After cleaning up, this function is a surjection. In particular, there are at least as many outputs as inputs. Letting ℓ_{in} and ℓ_{out} be the numbers of and output locations on gadgets in the simulation, and p_{in} and p_{out} be the numbers of input and output ports, this means $\ell_{out} + p_{in} \geq \ell_{in} + p_{out}$. Since G is nonexpanding and the simulation now contains only subgadgets of G, also $\ell_{in} \geq \ell_{out}$. Thus $p_{in} \geq p_{out}$.

We can make the same argument for any subgadget of the simulation: from some initial configuration, remove a subset of input ports, and then clean up. Hence every subgadget of the simulated gadget has at least as many inputs as outputs, meaning the simulated gadget is nonexpanding. □

We now move to a more complicated closed class that uses many of the same ideas, generalizing Chalcraft and Greene's result [CG94].

Definition 3.21. A sequence of transitions of an input/output gadget is *periodic* if each transition leaves the gadget in the initial state of the next one, and the final transition returns the gadget to the initial state of the first one. A periodic sequence is *expanding* if in addition, it contains more distinct output locations than distinct input locations. An input/output gadget is *unboundedly expanding* if it has an expanding sequence, and *boundedly expanding* otherwise.

We can also use the idea of periodic sequences to generalize the definition of 'unbounded' from Section 3.1.2: a gadget is *unbounded* if it has a periodic sequence which contains two transitions that send the agent from the same input to different outputs. Note that every expanding sequence has this property, and thus every unboundedly expanding gadget is unbounded.

Every unbounded output-disjoint gadget is unboundedly expanding. Unbounded but boundedly expanding gadgets include the result of merging any two outputs in any of the first four gadgets in Fig. 3.3 (but not the final two, since the toggle switch alone is unboundedly expanding).

Every unboundedly expanding gadget is expanding: the subgadget induced by the input locations and initial state of an expanding sequence has more outputs than inputs.

Theorem 3.22. *Boundedly expanding gadgets are closed under simulation.*

Proof. Let G be a boundedly expanding gadget, and consider a simulation using G. Consider a periodic sequence of transitions through the simulated gadget. Although states of the simulated gadget may not correspond exactly to configurations of the simulation, we can recover a periodic sequence through the simulation: repeat the sequence a large number of times, until a configuration of the simulation at the end of a repetition occurs twice. The sequence of transitions between the two occurrences is periodic, and consists of some number of repetitions of the original sequence.

Now simplify the system by removing all locations and ports that aren't used by the path of this periodic sequence. Define ℓ_{in}, ℓ_{out}, p_{in}, and p_{out} as in Theorem 3.20 for this new system. Since every remaining input location and output port is reached from an output location or input port, as before we have $\ell_{out} + p_{in} \geq \ell_{in} + p_{out}$. Considering how the periodic sequence through the simulation uses a specific copy of G in the simulation, we observe that it defines a periodic sequence on that gadget. Since G is boundedly expanding, the path uses at least as many inputs as outputs on this gadget. Thus $\ell_{in} \geq \ell_{out}$, and again we have $p_{in} \geq p_{out}$. That is, the periodic sequence uses no more inputs than outputs of the simulation, or equivalently of the simulated gadget; hence the simulated gadget is boundedly expanding. □

As a consequence, any of the boundedly expanding gadgets mentioned above can't simulate any unboundedly expanding gadget, including any of the gadgets for which we know reachability to be PSPACE-complete.

This implies the result of Chalcraft and Greene [CG94], which is essentially that the set-up line/set-down line/set switch with the outputs of the two lines merged can't simulate the

toggle switch. Interestingly, Chalcraft and Greene also show how to simulate a toggle switch using infinitely many copies of this gadget. Our proof implicitly assumed that the simulation was finite: otherwise, a configuration might never repeat, so the periodic sequence on the simulated gadget doesn't necessarily provide one on the simulation.

3.4 One Player

In this section, we consider one-player reachability with input/output gadgets. Recall that this means we always allow the fanout gadget. We no longer require all gadgets to be deterministic, though our hardness results are still for deterministic gadgets.

We characterize the complexity of one-player reachability with an output-disjoint deterministic 2-state input/output gadget as follows; refer to the bottom row of Table 3.2 and the middle row of Table 3.3. If the gadget is trivial, then one-player reachability is just reachability in a directed graph, which is NL-complete [Wig92]. If the gadget is unbounded and multi-input, then one-player reachability is PSPACE-complete by Lemma 3.23 below and by Corollary 3.9 because it is a generalization of zero-player reachability. Otherwise, one-player reachability is in NP by Lemma 3.23 if it is bounded and by Theorem 3.24 if it is single-input. In either case, reachability is NP-hard by Corollary 3.30.

We begin with straightforward containment results:

Lemma 3.23. *One-player reachability with any input/output gadgets is in PSPACE, and one-player reachability with bounded input/output gadgets is in NP.*

Proof. One-player reachability can easily be simulated by a nondeterministic polynomial-space algorithm which guesses player choices, so it is in NPSPACE = PSPACE [Sav70].

For bounded input/output gadgets, define k, i, and n as in Theorem 3.4. As before, the number of state changes is bounded by kn. The shortest solution visits each input at most once between consecutive state changes, and thus has total length at most $in(kn+1)$. This is a polynomial, so we can use the list of transitions as a certificate for NP. □

For the remainder of this section, we focus on one-player reachability with single-input input/output gadgets.

One-player reachability switching games, studied in [FGMS21], are equivalent to one-player reachability with deterministic single-input input/output gadgets. Fearnley, Gairing, Mnich, and Savini [FGMS21] show that this problem is NP-complete when the gadgets are described as part of the instance.

In this section, we improve on this result in two ways. First, we show in Theorem 3.24 that the problem remains in NP even when we allow nondeterministic single-input input/output gadgets, which cannot all obviously be simulated by deterministic gadgets. Our proof is similar to the proof of containment in NP in [FGMS21].

Second, we show in Section 3.4.2 that the problem remains NP-hard with a specific gadget instead of instance-specified gadgets. In particular, we show that one-player reachability with the toggle switch or the set switch is NP-complete. Our reduction is simpler than the one in [FGMS21], and the technique can be used to prove NP-hardness for many other single-input input/output gadgets.

3.4.1 Containment in NP

First we show that reachability with any single-input input/output gadgets (including nondeterministic ones) is in NP, generalizing a result from [FGMS21]. Our proof is similar, but requires more care to account for nondeterministic gadgets.

Theorem 3.24. *Reachability with single-input input/output gadgets is in NP.*

The input can describe the gadgets in the system by listing their states and locations and specifying their transition graphs.

Proof. A single-input input/output gadget is described by a labelled directed graph, with states as vertices and transitions as edges, where each edge is labeled with an output location. An edge labeled b from s to t indicates that, when the agent enters the unique input location in state s, it can exit at b and change the state to t. If you prefer, this can be thought of as a nondeterministic finite automaton (NFA) whose alphabet is the locations of the gadgets in the motion-planning problem.

We will adapt the certificates used in [FGMS21], controlled switching flows, to work for nondeterministic gadgets. The number of times each output location (or edge in the equivalent reachability switching game) is used is no longer enough information, since it may in general be hard to determine whether a nondeterministic gadget has a legal sequence of transitions which uses each location a specified number of times.[1] Instead, we will have the certificate include the number of times each *traversal* in each gadget is used, which will be enough information to be checked quickly. We modify the definition of controlled switching flows as follows.

Definition 3.25. A *controlled switching flow* in a system of single-input input/output gadgets is a function f from the set of transitions gadgets in the system to the natural numbers (including zero) which is "locally consistent" in the following sense:

- For an input i, let H_i be the set of transitions whose input is i (namely all transitions of the single-input gadget containing i), and let H_o be the set of transitions whose output is connected to i. Let S be the set containing i and each output connected to i. Then

$$\sum_{t \in H_i} f(t) - \sum_{t \in H_o} f(t) = \begin{cases} 1 & S \text{ contains the start location but not the goal location} \\ -1 & S \text{ contains the goal location but not the start location} \\ 0 & \text{otherwise.} \end{cases}$$

- For each gadget, there is a legal sequence of transitions from its starting state s which uses each transition t in the gadget exactly $f(t)$ times.

[1] In fact, this is NP-hard by a reduction from the existence of a Hamiltonian path in a directed graph: given a graph with n vertices, construct a gadget with n states and n output locations whose transition graph is the input graph, and ask for a sequence of transitions which uses each output location exactly once. In terms of finite automata, determining whether a given NFA accepts any anagram of a given string is NP-complete.

That is, thinking of $f(t)$ as the number of times the agent uses the transition t, the agent enters and exits each connected component the same number of times, except that it exits the start location and enters the goal location once, and the agent uses the transitions of each gadget a consistent number of times.

To prove containment in NP, our certificate that it is possible to reach the goal location is a controlled switching flow. Note that each gadget has a polynomial number of transitions, so f has polynomially many entries. We need the following three lemmas:

Lemma 3.26. *If there is a controlled switching flow, then it is possible to reach the goal location.*

Lemma 3.27. *If it is possible to reach the goal location, then there is a polynomial-length controlled switching flow, i.e., one where $f(t)$ is at most exponential in the size of the system.*

Lemma 3.28. *There is a polynomial-time algorithm which determines whether a function f is a controlled switching flow.*

Together these imply that controlled switching flows can actually be used as certificates, and thus the reachability problem is in NP.

Proof of Lemma 3.26. Let f be a controlled switching flow. For each gadget G, pick a legal sequence of transitions of length $\ell_G = \sum_{t \in G} f(t)$ which uses each transition t exactly $f(t)$ times; this exists by the definition of a controlled switching flow. We will control the agent as it plays reachability in the system. Our strategy is based on the chosen sequences: whenever we arrive at a gadget, take the next transition in the sequence. We stop when we reach the connected component of the goal location, or when our strategy fails us because we are at the input of gadget G that we have already used ℓ_G times.

We claim this strategy must reach the goal location. If it does not, we must eventually get stuck with no moves (specifically, within $\sum_t f(t)$ steps), and we will show this cannot happen because f is a controlled switching flow. For the sake of contradiction, suppose we find ourselves stuck at an input i, and define H_i, H_o, and S as in the definition of controlled switching flow. To be stuck, we must have previously transitioned from i exactly $\sum_{t \in H_i} f(t)$ times. Assume for the moment that S contains neither the start nor the goal location. Then we must have transitioned to an output connected to i at least $\sum_{t \in H_o} f(t) + 1$ times, so $\sum_{t \in H_i} - \sum_{t \in H_o} \leq -1$, which violates the assumption that f is a controlled switching flow. If S contains the start location this difference is 1 greater, and if S contains the goal location it is 1 smaller; in any case we have a contradiction. □

Proof of Lemma 3.27. For some path that ends at the goal location, let $f(t)$ be the number of times the path uses the traversal t. Then f is clearly a controlled switching flow. The number of traversals in the shortest solution path is at most the number of configurations of the system of gadgets, which is at most nk^n if the system contains n gadgets which have at most k states. So the shortest solution path gives us a controlled switching flow f where $f(t) \leq nk^n$ and thus f has polynomial length. □

Proof of Lemma 3.28. The first condition for f to be a controlled switching flow can be easily checked in polynomial time by computing the relevant sums.

For the second condition, think of a gadget in the system as a directed multigraph with states as vertices and transitions as edges (labelled with their output locations). The second condition says that (for each gadget) there is a walk through this graph starting at the initial state s which uses each edge t a specified number $f(t)$ of times. This is equivalent to an Euler tour in the (possible exponentially large) graph with $f(t)$ copies of the edge t. To verify that such a walk exists, we only need to check that the total in- and out-degrees match at each vertex (except possibly off by one at s and one other vertex) and that the set of used transitions, i.e., those t where $f(t) > 0$, is connected. This can all be checked in polynomial time. □

This concludes the proof of Theorem 3.24. □

3.4.2 NP-hardness

In this section, we prove NP-hardness of one-player reachability with each of the nontrivial single-input 2-state deterministic gadgets: the set switch and toggle switch. Our proofs can be easily adapted to prove NP-hardness of the corresponding problem for many input/output gadgets, but we leave open the problem of providing a characterization.

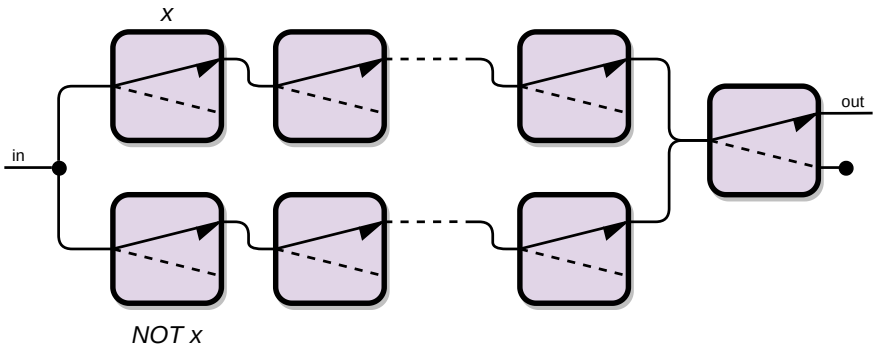
Our reduction is simpler than that in [FGMS21], and we show hardness for specific gadgets instead of general reachability switching games, which are equivalent to instance-specified gadgets.

Theorem 3.29. *One-player reachability with either the toggle switch or the set switch is NP-hard.*

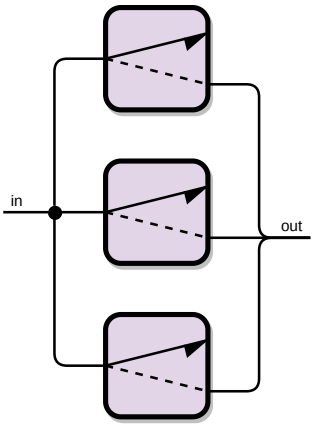
Proof. We provide essentially identical reductions from 3SAT to the two motion-planning problems. In the reduction, the player will never be able to traverse a gadget more than two times, so the difference between the toggle switch and the set switch is irrelevant. Each gadget will begin in the state which sends the agent to the 'top' output, and after a single traversal moves to the state which sends the agent to the 'bottom' output. We will describe the reduction in terms of the set-down switch, but it is equally applicable to the toggle switch.

For each variable in a 3SAT instance, there is a fork where the player may choose one of two paths. Each path passes through a series of set-down switches, exiting each from the top and setting them to the down state. The paths then merge and go through one more set-down switch, whose down output is a dead-end. The number of gadgets in each branch depends on the number of instances of each literal in the formula. These variables are connected in series beginning at the start location, so the player is forced to walk through each variable, picking a side to use for each one. This corresponds to picking an assignment of each variable. After setting the variables in this way, the last set-down switch at the end of each variable is in the down state.

For each clause, there is a 3-way fork, where the player must choose to go through one of the gadgets corresponding to a literal in the clause. If the chosen gadget was already traversed (during the variable-setting phase), the agent exits the bottom and can continue



(a) A variable. The player must pick one of the two branches to cross, and render the final set-down switch useless.



(b) A clause. The player can only cross if at least one of the literals is set to true; otherwise they get stuck in a dead-end a variable.

Figure 3.26: Our reduction from 3SAT for one-player reachability with the set-down switch.

to the exit of the clause. If the chosen gadget was not already traversed, the agent exits the top, and finds itself in a variable. The player now has no choice but to walk down the variable path until the agent goes through the set-down switch at the end of the variable, which is in the down state, so the agent is now stuck in a dead-end.

The clauses are connected in series, with the last variable leading to the first clause and the last clause leading to the goal location. In order to reach the goal location, the agent must pass through each variable and then each clause. In order to get through a clause without getting stuck, at least one gadget in the clause must have already been traversed; equivalently, at least one literal in the clause must be true under the assignment corresponding to the path taken during variable setting. Thus the agent can reach the goal location if and only if the formula has a satisfying assignment.

For the set-down switch, once a gadget is in the down state it remains there forever, so it does not matter what order the clauses or gadgets within each variable path are in. However, for the toggle switch, when the agent walks through a clause the gadget it uses returns to the up state, which could lead to the agent later escaping that variable using the same gadget, instead of getting stuck in a dead-end.

To prevent this, we order the gadgets on each variable path carefully. Specifically, we first choose an order for the clauses. For each literal ℓ, we have the path corresponding to ℓ go through the set-down switches representing ℓ in the same order they appear in clauses. So if the agent moves from a clause to a variable through a gadget corresponding to ℓ (because ℓ was false), the agent cannot have previously interacted with any of the gadgets further along the line corresponding to ℓ during the clause-checking phase. In particular, those gadgets are all in the up state, and so the agent is in fact forced to go to the dead-end at the end of the variable. □

(a) The switch/set-up line simulates the set-up switch.

(b) The switch/toggle line simulates the toggle switch.

Figure 3.27: Simulating nontrivial single-input gadgets from gadgets without a toggle or set switch.

Corollary 3.30. *One-player reachability with any nontrivial output-disjoint deterministic 2-state input/output gadget is NP-hard.*

Proof. It suffices to show that such a gadget can simulate either the toggle switch or the set switch, since Theorem 3.29 says one-player reachability with either of these gadgets is NP-hard.

The only single-input output-disjoint deterministic 2-state input/output gadgets are the toggle switch and set switch, which simulate themselves. If the gadget is multi-input, by Lemma 3.1 it simulates one of the eight basis gadgets shown in Figs. 3.2 and 3.3. All but three of these contain a toggle switch or set switch. The switch/set-up line/set-down line contains the switch/set-up line, so we need only consider the switch/toggle and switch/set-up line. These can simulate, respectively, the toggle switch and the set-up switch, as shown in Fig. 3.27. □

3.5 Two Players

In this section, we consider two-player player reachability with input/output gadgets. This game is analogous to the two-player reachability switching games of [FGMS21], and we improve upon their results. This game is different from two-player motion planning as defined in [DHL20], which has two agents each controlled by one player, whereas we have a single agent which is affected by choices of both players.

Two-player reachability switching games are equivalent to two-player reachability with deterministic single-input input/output gadgets specified as part of the instance. It is shown in [FGMS21] that this problem is in EXPTIME and PSPACE-hard.

In this section, we improve on this result in two ways. First, we show that two-player reachability with any input/output gadgets (with any number of inputs) is in EXPTIME. Second, we show that two-player reachability with just the toggle switch or the set switch is PSPACE-hard. We give a reduction from Geography which is simpler than the reduction in [FGMS21].

We do not show EXPTIME-hardness for two-player reachability with any gadget. We conjecture that two-player reachability is EXPTIME-hard with any multi-input unbounded

output-disjoint deterministic 2-state input/output gadget, and perhaps even with any single-input such gadget (the only one being the toggle switch).

Lemma 3.31. *Two-player reachability with input/output gadgets is in EXPTIME.*

Proof. The two-player game can be simulated on an alternating Turing machine using polynomial space, where White's decisions are made by existential states and Black's decisions are made by universal states. Thus the problem is in APSPACE = EXPTIME. □

Theorem 3.32. *Two-player reachability with either the toggle switch or the set switch is PSPACE-hard.*

Proof. We provide essentially identical reductions from a version of Geography to the two problems. In the reduction, the agent will never be able to traverse a gadget more than two times, so the difference between the toggle switch and the set switch is irrelevant. As in the proof of Theorem 3.29, we will describe the reduction in terms of the set-down switch, but it is equivalent for the toggle switch.

Vertex-Partizan Directed Vertex Geography is a game played on a directed graph with specified start vertex, where each vertex is assigned to a player. Two players each move a marker along an edge whenever it is at one of their vertices, with the rule that the marker cannot visit the same vertex multiple times. A player loses if they have no moves. In Vertex-Partizan Max-Degree-3 Directed Vertex Geography, we assume every vertex has degree at most three, with at most two incoming edges and at most two outgoing edges. This problem is introduced and shown PSPACE-complete in [BCC+20]; vertex-partizan is a slight variation on bipartite Geography. We will refer to Vertex-Partizan Max-Degree-3 Directed Vertex Geography as simply *Geography*.

We construct an instance of two-player reachability with the set-down switch from an instance of Geography as follows. Each Geography vertex will be a single gadget, connected according to geography edges. If a vertex has in-degree 1 and out-degree 2, we replace it with a fanout gadget labelled with the player who is assigned that vertex. If a vertex has in-degree 2 and out-degree 1, we replace it with a set-down switch initially set to 'up', with the 'up' output leading to the edge out of the vertex and the 'down' output leading to the goal location if the vertex is assigned to White and a dead-end otherwise. The agent starts at the location corresponding to the start vertex.

Play on this system of gadgets proceeds as follows. When the agent reaches a fanout gadget, the player assigned the corresponding vertex chooses which output to take. When the agent reaches a set-down switch for the first time, it continues along the outgoing edge to another vertex. When it reaches a set-down switch for the second time, the game ends and the player who is assigned the corresponding vertex wins. This is the same as the game of Geography: a player loses if the agent moves from one of their vertices to a set-down switch it visited before, which is equivalent to players not being allowed to move the marker to an already-visited vertex. □

3.6 Applications

In this section, we use the results of this chapter to prove PSPACE-completeness of the mechanics in several video games: one-train colorless Trainyard, [the Sequence], trains in

Factorio, and transport belts in Factorio.[2] For each of these problems, the decision problem is the long-term behavior of a deterministic system, e.g., whether a train ever reaches a specific location. Another interesting decision problem, which we do not consider here, is the design problem: given some set of constraints, is it possible to build a configuration with a desired behavior? This is perhaps more natural because, for example, it captures the question of deciding whether a level in Trainyard is solvable.

3.6.1 Trainyard

The study of the complexity of Trainyard began with [ALP18b], which showed that finding a solution to a Trainyard level is NP-hard. Later, [ALP18a] showed that checking a solution to a Trainyard level is PSPACE-complete—verifying solutions may be harder than finding them. We improve on this result by showing that checking a solution to a Trainyard level is PSPACE-hard even with only one train, and with no color changes.

Trainyard is a puzzle game in which the goal is to build a system of rails so that trains of the correct colors reach certain stations. We consider one-train colorless Trainyard, where solutions consist of only rails, crossings, and *switches*[3] There is a single train which moves forward along the rails; it succeeds if it reaches a designated location, and crashes and fails if the track ends. Rails can be traversed in both directions.

The only nontrivial behavior comes from switches, which have two states. A switch changes state every time the train moves through it. It has three locations: two of them always route the train to the third, and the third routes the train to one of the first two depending on the state. We can model this as a toggle line/toggle line/toggle switch with some locations identified; we call this the *Trainyard gadget*, which is shown in Fig. 3.28. Since tracks can bend and cross each other, the planarity of a system of Trainyard gadgets does not matter. One-train colorless Trainyard is equivalent to (zero-player) reachability with the Trainyard gadget—except that the Trainyard gadget is not input/output, so we have not defined reachability with it.

Definition 3.33. *Reachability with the Trainyard gadget* takes place in a system of Trainyard gadgets where locations are connected in a partial matching. That is, each location is either paired with one other location or a *dead-end*.

An agent moves through the system similarly to with input/output gadgets. When it enters a Trainyard gadget, it takes the unique available transition. When it exits a Trainyard gadget, it moves to the unique paired location, or stops and *crashes* if it is at a dead-end. The decision problem is whether the agent eventually reaches a designated goal location.

Theorem 3.34. *Reachability with the Trainyard gadget, or equivalently checking a solution to one-train colorless Trainyard, is PSPACE-hard.*

Proof. We will reduce from reachability with the toggle switch/toggle line. Formally, we actually reduce from a restricted version of this problem, where we are promised that the

[2]Factorio in general is already known to be PSPACE-complete, as players have explicitly built computers using the circuit network; for instance, see https://forums.factorio.com/viewtopic.php?t=42708 or https://redd.it/6imjhv. We consider the restricted problems with only train-related objects and only transport belt-related objects.

[3]This use of the word 'switch' is unrelated to the component of input-output gadgets.

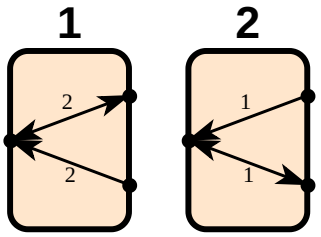


Figure 3.28: The state diagram of the Trainyard gadget. Note that the states are symmetric to each other.

agent does not enter an nontrivial infinite loop—it either reaches the goal location or a dead-end in the form of a single-gadget loop. The system constructed by Section 3.2.3 satisfies this property, and the simulations in Section 3.2.3 preserve it, so the restricted promise problem is still PSPACE-hard.

We cannot quite directly simulate a toggle switch/toggle line, for a few reasons:

- The Trainyard gadget, and thus any gadget simulated by it, can be entered at any location, not just input locations. To account for this, we will denote some vertices in the simulation as input and output, and the arrangement of gadgets will ensure that the agent always enters simulated gadgets at input-denoted locations and exits and output-denoted locations. In particular, paths emerging from output-denoted locations always lead to input-denoted locations.

- Reachability with the Trainyard gadget does not include fan-ins. However, we can easily simulate fan-in in the above sense by denoting two locations as input and one as output on the Trainyard gadget—the Trainyard gadget is a fan-in provided the agent never enters at the left location.

- Even with the above caveats, we have not been able to simulate the toggle switch/toggle line (or any unbounded output-disjoint deterministic 2-state input/output gadget with multiple nontrivial inputs) with the Trainyard gadget. Instead, we simulate a toggle switch/toggle line for *exponentially long*. Formally, we describe a system of Trainyard gadgets for each natural number k such that the kth system has the same behavior as the toggle switch/toggle line for at least 2^k transitions, and can be constructed in time polynomial in k. Consider a system of n toggle switch/toggle lines in which the agent never enters an infinite loop (such as the one used to prove PSPACE-hardness). The system has at most 2^n configurations and $5n$ locations for the agent; thus after at most $5n2^n$ transitions, the agent reaches either the goal location or an infinite loop dead-end. If we pick a polynomial-size k such that $2^k > 5n2^n$ (e.g., $k = 2n + 3$ suffices), then the system of Trainyard gadgets we obtain by replacing each toggle switch/toggle line with the kth simulation has the same behavior long enough for the agent to reach either the goal location or a dead-end. We replace each dead-end with a track that ends, so that the train stops when it reaches one. Hence these exponentially long simulations suffice for PSPACE-hardness.

Thus it suffices to find an exponentially long simulation of the toggle switch/toggle line. Before describing this simulation, we present an exponentially long simulation of an intermediate gadget called the *reverse branch*, shown in Fig. 3.29. This gadget has one state and three locations. We assume the top right location is only used as an input, and the bottom right location is only used as an output.

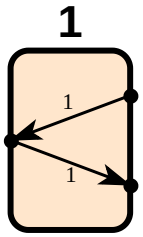


Figure 3.29: The reverse branch gadget. Our constructions ensure the agent never enters at the bottom right.

Fig. 3.30 shows our exponentially long simulation of a reverse branch. The k gadgets in the bottom row serve as fan-ins, since we assume the agent never enters at the bottom right. Consider the states of the top row of $k+1$ gadgets as describing a number in binary: up (state 1) is 0, down (state 2) is 1, and the bits are read right to left. When the agent enters at the left, it increments this number (mod 2^{k+1}) and exits at the bottom right, unless the states are all up so the number is 0, in which case it exits the top right. When the agent enters at the top right, it flips the state of every gadget in the top row and exits at the left; this changes the number by $x \mapsto -x-1$. In particular, the distance from 0 changes by at most 1 with each transition. By starting at 2^k as in Fig. 3.30, it takes at least 2^k transitions to reach 0, so the simulation is correct for 2^k transitions.

Now we simulate a toggle switch/toggle line using a Trainyard gadget and two reverse branches, as shown in Fig. 3.31. When the agent enters at a, it exits at b, flipping the state of the Trainyard gadget (in the middle); this is the toggle line. When the agent enters at c, it exits at d or e depending on the state of the Trainyard gadget, and flips the state; this is the toggle switch. Each transition through the simulated gadget makes at most one transition through each reverse branch, so if the reverse branches are correct for 2^k transitions, so is the toggle switch/toggle line. □

3.6.2 [the Sequence]

[the Sequence] is a puzzle game in which the player attempts to place *modules* to move *binary units* from a *source* to a *target*. There are seven different modules which have different effects; for instance, the *pusher* moves anything immediately in front of it one square away.[4] In this section, we prove that determining the correctness of a solution to a [the Sequence] puzzle is PSPACE-complete.

[4]Module names are not given in the game, so we use our own names.

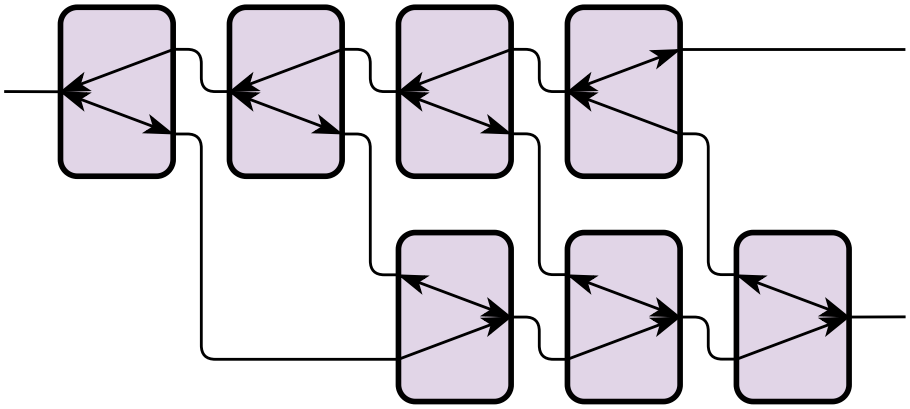


Figure 3.30: An exponentially long simulation of a reverse branch using Trainyard gadgets.

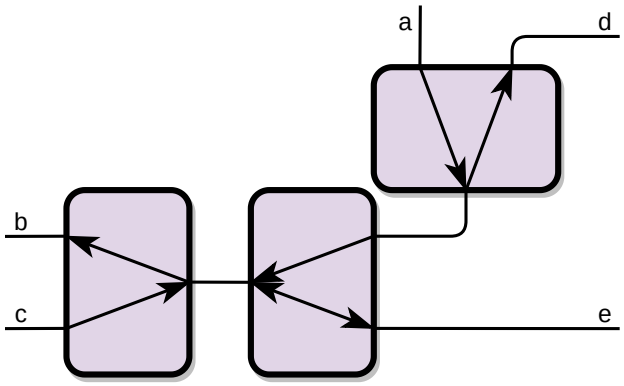


Figure 3.31: A simulation of a toggle switch/toggle line using a Trainyard gadget and reverse branches. The central gadget is a Trainyard gadget, and the other two are reverse branches. The simulated toggle line goes from a to b, and the simulated toggle switch goes from c to d and e.

(a) The mover module. (b) The turner module. (c) The puller module. (d) A fixed block.

Figure 3.32: [the Sequence] modules used in our proof, and the fixed block.

First we describe the mechanics of [the Sequence] that are necessary for our proof. The game takes place on a bounded square grid, containing the source and target, some fixed blocks, and some modules (which the player has placed, and which have an orientation). A deterministic simulation occurs in a series of rounds. Each round begins with the source creating a binary unit if it does not already have one. Then each module acts in a specified order; their actions are detailed below. Only modules and binary units can be moved. A binary unit disappears when it reaches the target. If objects (binary units, modules, or walls) ever collide, the simulation stops. The solution is correct if it moves an arbitrary number of binary units from the source to the target.[5]

The modules used in our proof are the following, shown in Fig. 3.32:

- The *mover* moves one square forward, bringing any module or binary unit immediately to its left (relative to the direction it's facing) with it.[6]
- The *turner* rotates any module immediately in front of it 90° counterclockwise.
- The *puller* moves any module or binary unit two squares in front of it to only one square in front of it.

Theorem 3.35. *Determining correctness of a solution to a [the Sequence] puzzle is PSPACE-complete.*

We prove hardness using a reduction from reachability with the switch/set-up line/set-down line. Our proof is robust to the definition of correctness, in the following sense: if the agent reaches the goal location, the solution moves arbitrarily many binary units to the target. If the agent does not reach the goal location, the solution runs forever without moving any binary units. A simple modification to the reduction makes the simulation crash if the agent does not reach the goal (though this requires the property of the proof of PSPACE-hardness of reachability that the agent reaches a specific location exactly when it does not reach the goal).

Proof. The game is a deterministic simulation with a polynomial amount of state (each square has at most one module or binary unit, which takes a constant amount of memory), so the simulation can be carried out in polynomial space. Determining whether arbitrarily many binary units will be delivered to the target is harder, but can still be done in PSPACE by detecting a cycle in the configuration, perhaps using a tortoise-and-hare algorithm.

[5]The game checks that four binary units are successfully moved, but an unlimited number is more natural.

[6]These modules can also be reflected, but we do not use that.

Figure 3.33: Bending paths and a fan-in for [the Sequence]. The movers represent where the mover might enter; they do not simultaneously exist. If the mover enters in either location, it is turned and possibly pulled, and then exits the right. The fixed blocks are only to help visualize the paths taken.

For PSPACE-hardness, we reduce from reachability with the switch/set-up line/set-down line. The agent is represented by a single mover. Turners rotate the mover to control its path. Fan-in is accomplished using a puller to merge to adjacent parallel paths, as shown in Fig. 3.33. Paths can easily cross each other since they only require modules at corners and fan-ins.

The switch/set-up line/set-down line, shown in Fig. 3.34, is built using three pullers. When the agent enters the set-up or set-down line, the mover moves the central puller to a particular side. When the agent enters the switch, the mover is pulled if the puller is on the appropriate side; the path it exits depends on the state of the gadget.

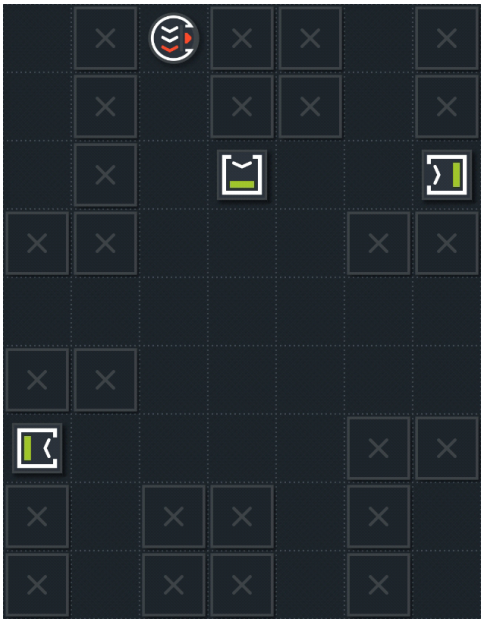
If the agent reaches the goal location, the mover reaches a cycle which has it deliver binary units to the target, shown in Fig. 3.35. Otherwise it gets stuck in the maze of modules forever, and never moves any binary units. □

3.6.3 Factorio Trains

Our first application for the factory-building video game Factorio is showing that trains in Factorio are PSPACE-complete. The decision problem we consider is whether a particular *target train* ever reaches its target station, in a world with only a train system and no player interaction. Other work on the computational power of Factorio train systems includes the simulation of the cellular automaton Rule 110 on a bounded tape [Min20]. The logical infrastructure used to implement Rule 110 is significantly more sophisticated than ours and is likely sufficient for PSPACE-hardness. We provide our own construction and prove PSPACE-completeness by reducing from reachability with the switch/set-up line/set-down line.

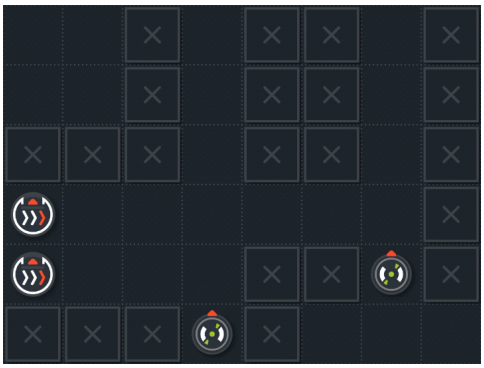
Next we describe the train components of Factorio, illustrated in Fig. 3.36. Trains in Factorio are constrained to *rails*, which can bend, fork, and cross each other. *Stations* are locations which trains will try to reach. Each train is provided with a *schedule*, which is a list of stations; the train will move to each station in the list in cyclic order. If there are

(a) The gadget in the down state, with a mover entering to set it to the up state.



(b) The gadget in the up state, with a mover entering to set it to the down state.

(c) The gadget in the down state, with a mover entering the switch.



(d) Separating the adjacent paths, analogous to Fig. 3.33.

Figure 3.34: A switch/set-up line/set-down line for [the Sequence]. The puller in the middle encodes the state. When the mover enters the bottom right (a) or top left (b), it moves the middle puller to the top or bottom and then gets pulled away, setting the state. When it enters the left, which row it exits on the right depends on whether the middle puller was at the bottom to pull the mover down. Finally, we separate these two paths with appropriate turners (d).

Figure 3.35: A cycle for the mover to deliver binary units. The two objects in the middle are the source and target. If the mover enters at the left as shown, it is pulled into the loop similarly to the fan-in. The turners then keep it moving in a rectangle, and it moves a binary unit from the source to the target on each loop.

multiple stations with the same name, the train will find the cheapest path to any of them, where the cost of a path depends on its length and also on properties including the number of other trains blocking the path and the amount of time the train has been waiting so far. Trains are prevented from crashing into each other using *rail signals* and *chain signals*. These partition the rail system into *blocks*, and (roughly) trains will not enter a block that is already occupied by another train.

Here are some caveats applying to our hardness proof:

- We assume that the only objects in the world are rails, locomotives (we make no use of cargo wagons), train stations, rail signals, and chain signals. In particular, there are no players, construction robots, circuit networks, or biters.
- We ignore fuel requirements of trains, assuming they have unlimited fuel. Without this assumption, and without allowing some mechanism to provide fuel, the problem would be in NP (since each train would move a bounded distance before running out of fuel).
- We do not use the complexity in train wait conditions: the only wait condition used is `0 seconds`. In fact, every train's schedule consists of just two stations `A` and `B`, which the train will alternate between.
- We do not know all of the details of the behavior of trains, but under the plausible assumptions that a single game tick is simulated in polynomial time and the amount of memory associated with each train (and other train-related component) is polynomially bounded, the decision problem is clearly in PSPACE.

Theorem 3.36. *In a Factorio world with only rails, locomotives, train stations, rail signals, and chain signals, and where each train's schedule alternates between the same two stations*

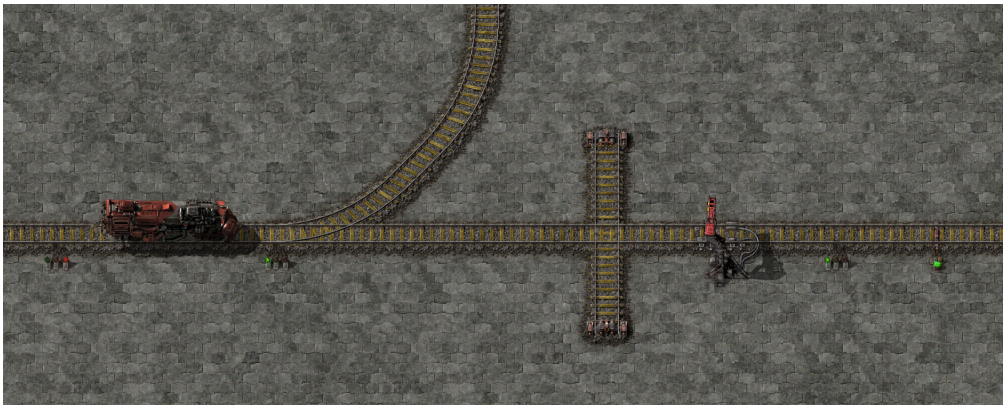


Figure 3.36: A demonstration of the train-related objects in Factorio. From left to right, we have a rail signal, locomotive, rail signal, rail fork, rail crossing, train station, rail signal, and chain signal. The leftmost rail signal is red, indicating the presence of a train in the block in front of it.

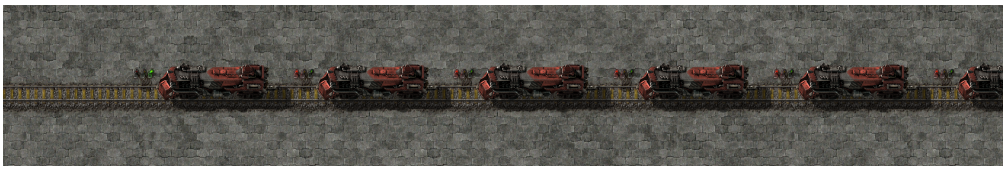


Figure 3.37: A wire of Factorio trains. The leftmost rail signal is green, so the leftmost train is free to move left. Then the train behind it can also move left, and so on, moving the gap right.

`A` and `B` with the trivial wait condition, it is PSPACE-hard to determine whether a specified target train ever reaches its next station.

Proof. We show PSPACE-hardness through a reduction from reachability with the switch/set-up line/set-down line. The rail network is mostly full of trains, one in each block, and the motion-planning agent is represented by a *gap* (not by a train), a block which does not contain a train and thus which a train can move into. A simple 'wire' is constructed by a line of blocks all occupied by trains, as shown in Fig. 3.37. When the gap reaches the front of the line, each train in turn is able to move forward one block, moving the gap to the end of the line. The movement of the gap is in the opposite direction of the movement of the trains.

Fan-ins are achieved using a fork in a track, as shown in Fig. 3.38. When the gap arrives at either branch of the fork, the train just entering the fork will move forward to fill the gap, since trains prefer paths with fewer other trains in the way. To ensure the path-finding works as expected, we place stations before and after the fork which make the paths the train at the fork needs to find short.

Since the system of gadgets may be nonplanar, we need a crossover. This is easy to build using two crossing rail lines with the appropriate configuration of rail and chain signals, shown in Fig. 3.39.

The initial location of the agent is represented by an empty block—in fact, the only

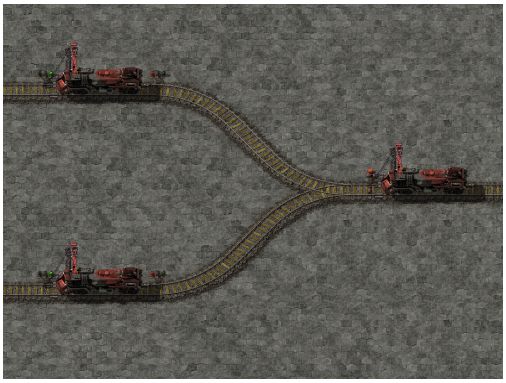


Figure 3.38: A fan-in for Factorio trains. Both stations on the left are named `B` and the station on the right is named `A`. When either train on the left leaves, the train on the right will fill its spot: it takes the cheapest path to `B`, and paths blocked by trains are considered more expensive. The chain signal immediately before the fork prevents the train from choosing a branch before one of them is empty.

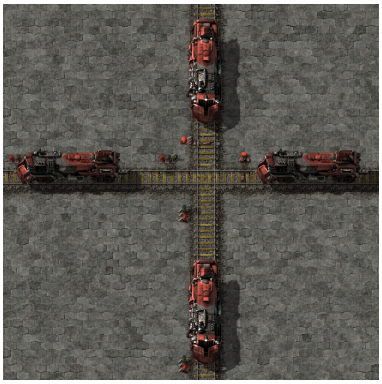


Figure 3.39: A crossover for Factorio trains. The two crossing wires can move independently. Chain signals prevent trains from blocking the intersection until the train in front moves out of the way.

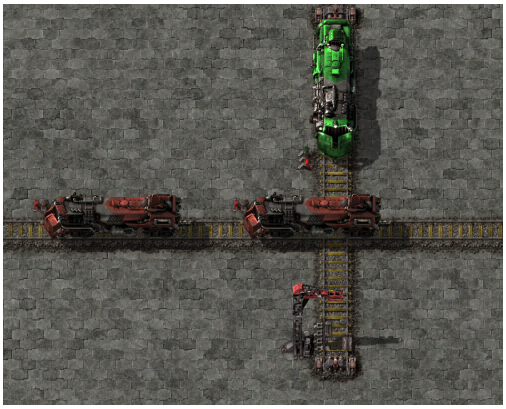


Figure 3.40: The win gadget for Factorio trains. Once the gap arrives on the left, the green train will be able to move forward and reach the station.

empty[7] block in the network. We place the target train on a short rail line blocked by the train in the block representing the goal location, as shown in Fig. 3.40: the target train will move forward and reach its target station if and only if the train blocking it moves, which happens exactly when the agent reaches the goal location.

Finally, we need to build the switch/set-up line/set-down line using trains. This gadget is shown in Fig. 3.41. We have a train trapped in a loop in the gadget, which encodes the state. When the gap traverses the set-up or set-down line, the trapped train is temporarily not blocked and moves forward. When the gap enters the switch, the output it takes depends on what the trapped train is currently blocking. □

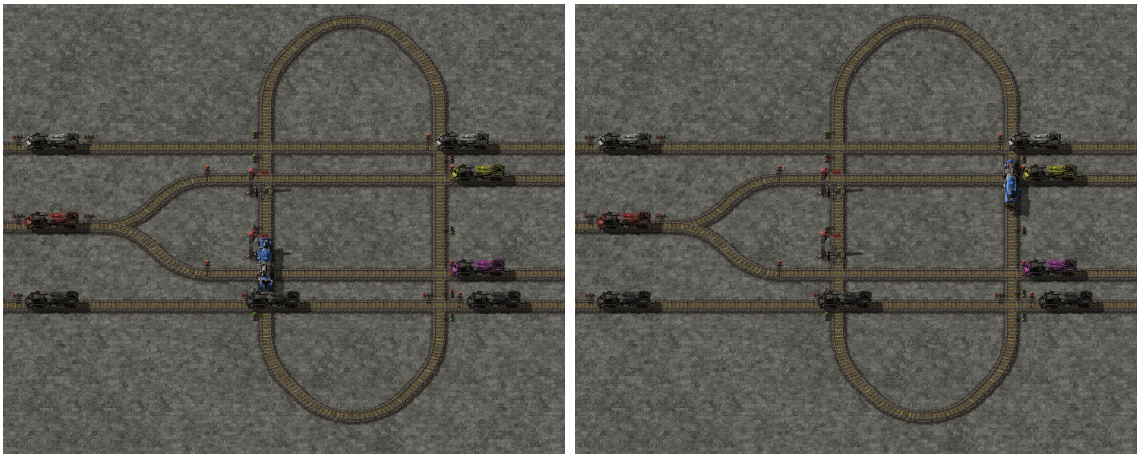
3.6.4 Factorio Transport Belts

In this section, we show that determining whether an item ever reaches a goal location in a Factorio world with only transport belts, underground belts, and splitters is PSPACE-complete. This result holds even with a small bounded number of mobile items: in Factorio 0.15 and earlier, we use a single mobile item (and a polynomial number of immobile items for a technical reason), and in Factorio 0.16 and later, we use two mobile items (and no immobile items).

Transport belts move any items on them in the direction the belt is facing. Items move smoothly between transport belts, but will stop if there is not another transport belt (or similar object) in front—transport belts will not dump items onto the ground. *Underground belts* can be used to have belts cross; two matching underground belts with at most four tiles between them will transfer items "underground".[8] *Splitters* can have up to two transport belts feeding in and two transport belts feeding out. Splitters alternate which output they send items to, regardless of the input they came from, except that if one output is blocked by items, all items will go to the other output. The details of the alternation changed slightly

[7]Technically, there is also an empty block at each intersection. However, these blocks have chain signals at their entrances, so trains will never stop in them, and the fact that they are empty does not allow any trains to move.

[8]The distance is longer for fast and express underground belts, but we do not need them.



(a) The gadget in the up state. (b) The gadget in the down state.

Figure 3.41: A switch/set-up line/set-down line made of Factorio trains. The blue train is trapped in the loop and encodes the state of the gadget; it alternates between the two train stations in the loop. From the down state, if the gap enters the top track of white trains, the blue train is briefly not blocked and moves to where it is blocked by a train in the bottom track. Similarly when the bottom track advances the blue train moves to be blocked by a train in the top track. When the gap enters the middle track and the red train leaves, whichever of the yellow or purple trains is not blocked by the blue train moves forward.

in Factorio version 0.16; we will explain and investigate the complexity of both versions of the mechanic.

Transport belts have two *lanes*, one on each side of the belt, which move items independently, are preserved going around corners and through splitters. A transport belt facing into the side of another transport belt will deliver items to the nearer lane; this is called *sideloading*. When sideloading onto an underground belt, only one lane of the incoming belt is able to move; the other lane is blocked.

As with trains, containment in PSPACE is trivial assuming each game tick is simulated in polynomial time.

Theorem 3.37. *In a Factorio world with only transport belts, underground belts, splitters and items, it is PSPACE-hard to determine whether an item ever reaches a goal location. In 0.15 and earlier, this remains PSPACE-hard when only one item can move and a polynomial number of items are stuck. In 0.16 and later, this remains PSPACE-hard when there are only two items.*

Proof. For both versions of splitter behavior, we show PSPACE-hardness through a reduction from reachability with the toggle switch/toggle switch. Wires are simply chains of transport belts, fan-in is accomplished by sideloading, and crossovers can be built using underground belts.

The toggle switch/toggle switch is different for the two versions, and depends on the details of splitter behavior.

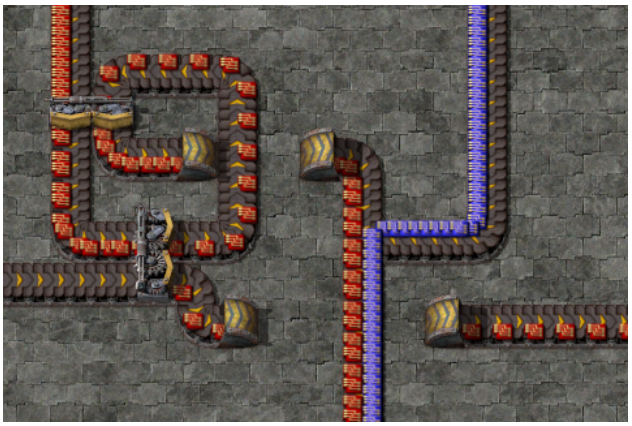


Figure 3.42: An example transport belt layout, demonstrating transport belts, underground belts, splitters, lanes, and sideloading.

Factorio 0.15 and earlier. Prior to 0.16, splitters alternate both lanes together and each item type separately. For each item type, all items of that type entering the splitter will alternate which output belt they take regardless of the lane they are on. The lane an item is on is preserved.[9] We view a splitter as having two lanes as inputs, and four outputs: two lanes on each of two belts. The splitter then behaves as a toggle switch/toggle switch—each lane is a toggle switch, and they share a state.[10]

We need to make a toggle switch/toggle switch which uses transport belts instead of lanes as inputs and outputs. This can be accomplished using sideloading onto the correct lane for the inputs and sideloading onto an underground belt for the outputs, shown in Fig. 3.43.

The initial state of each toggle switch/toggle switch is encoded by the state of a splitter. We place a single item at the start location, and it simulates the agent in reachability, reaching the goal location if and only if the agent does.

Factorio 0.16 and later. In 0.16, splitters were changed to alternate each lane separately and all item types together. A splitter now has only two bits of state, one for each lane, and all items of any type entering on the same lane will alternate output belts. We will always have items in the left lane.[11] Also in 0.16, splitters were given a setting to sort items: items of a specified type take one exit belt, and all others take the other exit belt.

Now our toggle switch/toggle switch for 0.15 and earlier is two independent toggle switches, and thus no longer suffices for PSPACE-hardness. Instead, we can take advantage of the sorting feature and the fact that item types alternate to construct a toggle switch/toggle switch, shown in Fig. 3.44. The agent will now be simulated by a pair of items of different types; we use an advanced circuit ("red circuit") and a processing unit ("blue

[9]Since each item type alternates independently, the splitter requires one bit of state for each item type. One can take advantage of this complexity for tasks including sorting items; we will not use it because there will be only one mobile item.

[10]Really it is a separate toggle switch/toggle switch for each item type, but we will have only one mobile item.

[11]To ensure this, we make fan-ins using sideloading of the right handedness, or just sideload onto the left lane immediately before entering each gadget.

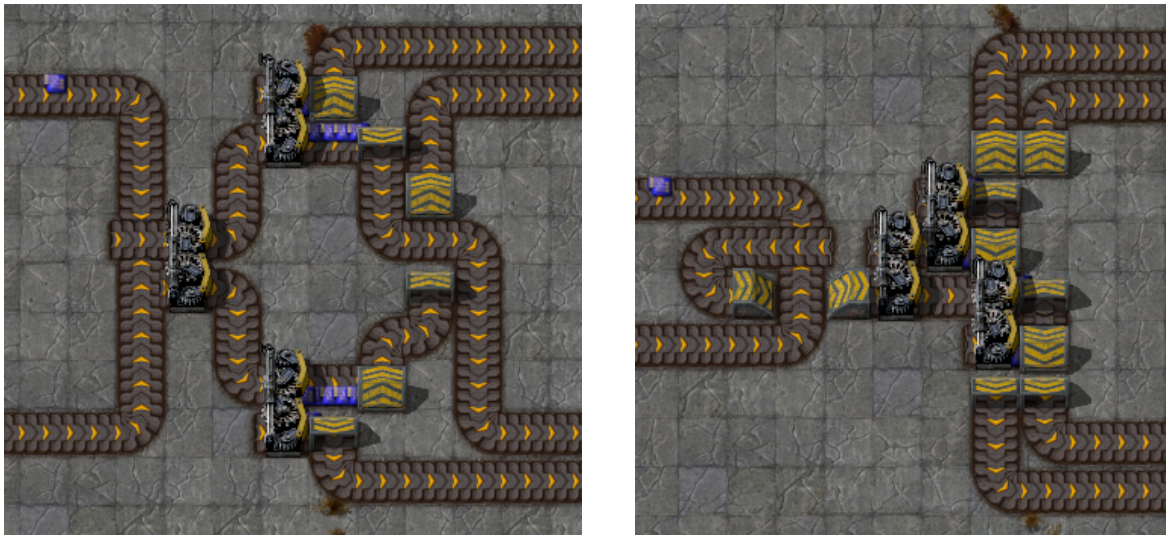


Figure 3.43: Two layouts of a toggle switch/toggle switch for transport belts in Factorio 0.15 and earlier. An incoming item is put on a lane depending on the input it enters. It passes through the leftmost splitter, which encodes the state of the gadget. The other two splitters help separate lanes: one lane of each output belt is blocked by items (by sideloading onto underground belts), so the output belt is determined by the lane of an item that enters that splitter. This gadget requires a constant number of immobile items: the layout on the left uses 16, and the layout on the right uses only 8 (but is harder to parse).

circuit"). The red circuit takes a natural path through the gadget, while the blue circuit shadows it to keep two splitters in the same state.

The two items take different amount of times to get through the gadget and may become separated. To fix this, after each toggle switch/toggle switch we place a *grouper*, shown in Fig. 3.45, which reduces the distance between the items by having item which arrives first take a longer path. The amount of separation from a single traversal of a gadget is bounded, so we can keep the items a bounded distance apart using an appropriate sequence of groupers after each gadget.

We place a single red circuit and a single blue circuit in the left lane at the start location. Both items will reach the goal location if and only if the agent does. □

Theorem 3.38. *In Factorio 0.16 and later, in a world with only transport belts, underground belts, splitters, and a single item, determining whether the item reaches a specified location is in NP ∩ coNP.*

Proof. As mentioned above, a splitter with the default settings is a pair of independent toggle switches, one for each lane. A splitter set to filter will always send the item to the same output belt. Splitters have another setting also added in 0.16: they can be set to prioritize a particular input or output belt, meaning it will always use that input or output unless it is empty or blocked, respectively, instead of alternating. With a single item in the world, a splitter in priority mode always sends the item to the same output belt. Thus this problem can be reduced to reachability with the toggle switch or equivalently ARRIVAL, which is in NP ∩ coNP. □

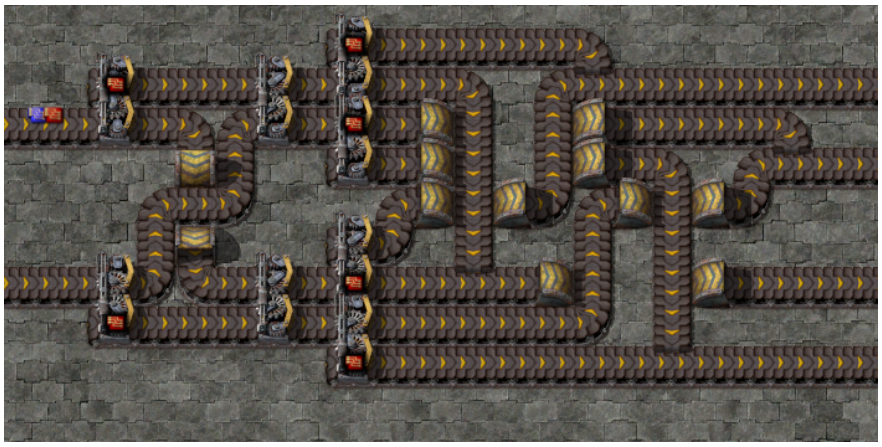


Figure 3.44: A toggle switch/toggle switch for transport belts in Factorio 0.16 and later. Both middle splitters encode the state of the gadget. Suppose the red and blue circuits enter the top input when the gadget is in the up state. The red circuit goes to the upper of the two middle splitters, takes the top exit belt, get sorted onto the topmost belt, and finally takes the topmost output. The blue circuit visits the lower middle splitter, takes the top exit, gets sorted onto the fourth belt from the bottom (just after the splitters), and also takes the topmost output. So both items took the topmost output, and both middle splitters flipped state. The other cases behave similarly. This construction is due to Twan van Laarhoven.

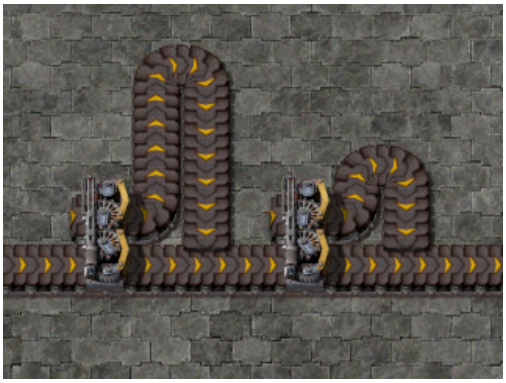


Figure 3.45: A grouper, which reduces the space between the red and blue circuits. The front item is delayed by about 8 tiles, and then the new front item is delayed by about 4 tiles. If the items are within about 16 tiles of each other when they enter, they exit with at most about 4 tiles between them. These distances are approximate; the actual distances are not integers since items take different amounts of time to traverse curved vs straight transport belts.

Chapter 4

Reversible Deterministic Gadgets

In this chapter, I describe another motion-planning gadgets offshoot, designed for proving hardness of systems that are deterministic and symmetric under time reversal. While the use case is quite different from the input/output gadgets of Chapter 3, the abstract framework is quite similar. Indeed, they have a natural common generalization more narrow than the full motion-planning gadgets framework: deterministic motion-planning gadgets, where ports are connected in pairs. Input/output gadgets have the additional property that they are always traversed from an input to an output, and we require that connections between gadgets respect this and allow fan-ins.

The gadgets of this chapter have the quite different additional property of symmetry under time reversal, making them useful for applying to reversible systems. The key result is a set of simple time-reversal-symmetric and deterministic gadgets that suffices for PSPACE-hardness, which is then used to prove hardness of four problems, including correcting an error in Hearn and Demaine's hardness proof for Deterministic Constraint logic [HD09].

This chapter represents joint work with Erik Demaine, Robert Hearn, and Jayson Lynch, which first appeared in [DHHL23].

4.1 Introduction

Reversible deterministic systems arise in various situations, some of the most important of which come from physics because fundamental existing physical theories are reversible and deterministic.[1] In particular, due to the thermodynamics of information, reversible computation can potentially use significantly less energy than irreversible computation because Landauer's Principle requires physical systems expend $k_B T \ln 2$ energy per bit of information lost.[2] Thus understanding how reversible systems can solve computationally difficult problems may help in designing general-purpose reversible computing hardware.

More precisely, a system is *deterministic* if its configuration at each time in the future is entirely determined by its current configuration. A system is *reversible* if, in addition, its

[1] The time evolution of the wave-function in the Standard Model is deterministic even if the observation of macroscopic phenomena is probabilistic.

[2] Here $k_B \approx 1.4 \cdot 10^{-23}$ is the Boltzmann constant and T is the temperature in kelvins. At room temperature, this is about $2.8 \cdot 10^{-21}$ joules per bit. Current chips are rapidly approaching this limit; see [Fra20, DLMT16].

configuration at each time in the past is entirely determined by its current configuration. The systems we consider all satisfy, or nearly satisfy, the stronger property of *time-reversal symmetry*: evolution forward in time and backward in time obey the same rules, so by looking at a sequence of configurations it is not possible to determine whether time is moving forwards or backwards. To reverse time, we simply need to reverse the direction of motion of each moving part in each of the systems we consider. In one system, we use a slightly more general symmetry by replacing each 'rotate clockwise' gadget with a 'rotate counterclockwise' gadget, and vice-versa. A physicist might call this parity–time (PT) symmetry; see, e.g., [ÖRNY19].

In this chapter, we first develop a framework for proving PSPACE-completeness of reversible deterministic systems in Section 4.2. Our framework extracts and simplifies a framework implicit in the work of Tsukiji and Hagiwara [TH11], who proved PSPACE-hardness for Langton's reversible 'ant' model of artificial life in two geometries, the square and hexagonal grids. Their hardness reductions construct five core gadgets in each grid, and show that these gadgets suffice for PSPACE-hardness by a reduction from QBF. Our framework decreases the number of required gadgets to just three, showing that some of the previous gadgets are redundant and others can be simplified. The framework also guarantees that the gadgets are connected together without crossings, making it well suited to reducing to planar systems (which all of our applications are).

We then apply our framework to analyzing the complexity of four reversible deterministic systems:

1. We prove in Section 4.3 that Deterministic Constraint Logic is PSPACE-complete. While this result was already claimed in 2008, [DH08, HD09], we describe in Section 4.3.1 an error in the previous reduction. The new framework enables a correct proof of the same result.

2. We develop in Section 4.6 a new PSPACE-hardness proof for the 'billiard ball' reversible model of computation, introduced and analyzed by Fredkin and Toffoli in 1982 [FT82]. In this model, unit-radius 2D balls move without friction and collide elastically with pinned or movable objects, according to classical physics. Unlike the previous proof, our PSPACE-hardness result works even in the case when only two balls ever move at once (and the rest are stationary), which results in a substantially simpler proof (no longer needing complex timing arguments to guarantee simultaneity).

3. We prove in Section 4.4 that zero-player motion planning through gadgets is PSPACE-complete when the gadgets include *any* reversible deterministic interacting k-tunnel gadget and a 'rotate clockwise' gadget (a 1-state 3-location gadget where an entering signal simply exits along the clockwise-next location). This result can be thought of as extending Table 1 from [DHL20], which summarizes the complexity of the motion-planning gadgets framework, to add a 'zero-player' column in the unbounded row, analogous to zero-player Deterministic Constraint Logic [HD09]. Our proof indeed uses the same simulations as for motion planning with a positive number of players [DHL20] to reduce to one core case — locking 2-toggles and rotate clockwise — and then shows that that case is PSPACE-complete.

4. We prove in Section 4.5 that zero-player motion planning with one very simple gadget called a '3-spinner' is PSPACE-complete. Specifically, a *3-spinner* has two states — 'clockwise' and 'counterclockwise' — and three locations at which the signal can enter; after entering, the gadget flips its state and the signal exits in the next port in the order given by the state. This result is weaker than Tsukiji and Hagiwara's PSPACE-hardness of Langton's ant on a hexagonal lattice [TH11], because the vertices in the lattice act exactly as 3-spinners. We effectively translate this result into the motion-planning gadgets framework of Demaine et al. [DHL20], and simplify it significantly while removing the geometric constraint.

All of the systems we consider can straightforwardly be simulated using polynomial space, so the decision problems are in PSPACE.

4.2 The framework

Our framework for proving PSPACE-hardness, which is a modest simplification of one due to Tsukiji and Hagiwara [TH11], can be understood in terms of the motion-planning gadgets framework described in Chapter 2. In particular, it is closely related to, and can be described in terms of, the input/output gadgets of Chapter 3. We will describe it independently.

The framework may apply to any setting with a single *signal* deterministically navigating a planar *system* of *gadgets* with the following properties. Each gadget has some designated *ports*. When the signal enters the gadget at one of its ports, it then exits the same gadget at one of its ports, which is determined by the entrance port and any previous traversals of that gadget. The system links gadgets by connecting the ports of the gadgets in disjoint pairs: when the signal exits at a port, it enters at the paired port.

To describe the "behavior" of a gadget, we define a *traversal* to be of the form $a \to b$ for any two ports a and b of the gadget. A gadget *implements* a sequence $[a_1 \to b_1, \dots, a_k \to b_k]$ of traversals if, when the sequence of the signal's entrance ports to the gadget is $[a_1, \dots, a_k]$, the sequence of exit ports from the gadget is $[b_1, \dots, b_k]$. Note that a gadget implements any prefix of a sequence it implements.

All of the gadgets we consider in this section are *symmetric under time-reversal* (or *reversible*, in the language of motion-planning gadgets [DGLR18]). This means that, starting from any state, if we perform a sequence of traversals followed by its time-reverse, the gadget is returned to the original state. For deterministic gadgets, we can express this without reference to states: if a gadget implements the sequences XY and XZ, then it also implements $XYY^{-1}Z$, where $Y^{-1} = [b_k \to a_k, \dots, b_1 \to a_1]$ is the time-reverse of $Y = [a_1 \to b_1, \dots, a_k \to b_k]$.

If every gadget in a system is symmetric under time reversal, then the entire system is as well: if we reverse the direction of the signal by returning it to the just-exited port instead of the port paired to just-exited port, it will retrace its steps in reverse, eventually returning to the initial configuration.[3]

[3]This is related to the 'implication properties' of [Hen21], but is not an implication property because X appears twice in the antecedent. Gizmos satisfying $XY, YZ \implies XYY^{-1}Z$ are not closed under simulation in general (because of 'superpositions'), but they are closed under simulation when everything is deterministic and we disallow branching.

4.2.1 Required gadgets

We are now ready to describe the gadgets which we will show suffice for PSPACE-hardness.

We describe each gadget by specifying some sequences it implements. The gadgets then also implement all prefixes of implemented sequences, and all sequences required for time-reversal symmetry. We don't fully specify the behavior of the gadgets: they are allowed to do anything if the signal arrives in an unspecified sequence, and this does not affect our PSPACE-hardness result because it never happens in the systems created by the reduction. The required behavior of our gadgets is summarized in Table 4.1. In addition, for each gadget G described below, we also allow our system to include G in a different state. In terms of implemented sequences, we allow the gadget G *after* X, which behaves like G would after having performed the traversals of X in order. That is, if G implements XY, then G after X implements Y.

Our first, and most complicated, gadget is the *Switch*. This corresponds to three of Tsukiji and Hagiwara's gadgets, the 'Switch & Pass,' 'Switch & Turn,' and 'Pseudo-Crossing,' which are all equivalent except for the cyclic order of ports in the planar embedding, and that Switch & Turn merges the ports we call Set and Out. The Switch has 5 ports, called 'Set,' 'Out,' 'Test,' 'T-Out,' and 'F-Out.' It implements [Set $\to$ Out, Test $\to$ T-Out] and [Test $\to$ F-Out]. Intuitively, it has an internal state which is initially False, and is set to True by the traversal Set $\to$ Out. Entering Test reveals the current state. Time-reversal symmetry implies that the Switch is reusable: for instance, it must also implement

$$[\text{Set} \to \text{Out}, \text{Test} \to \text{T-Out}, \text{T-Out} \to \text{Test}, \text{Test} \to \text{T-Out},$$
$$\text{T-Out} \to \text{Test}, \text{Out} \to \text{Set}, \text{Test} \to \text{F-Out}, \text{F-Out} \to \text{Test}].$$

There are really 12 different Switch gadgets (up to rotation and reflection), based on the cyclic order of the ports. We allow any cyclic order of the ports; our PSPACE-hardness applies to any individual order.

Our next gadget is the *Reversible Fan-in*. Tsukiji and Hagiwara call this gadget 'CONJ.' It has three ports a, b, and c, and implements $[a \to c]$ and $[b \to c]$. Intuitively, it is a fan-in that sends both a and b to c, but—as required by time-reversal symmetry—remembers which entrance was taken so that when the signal returns to c, it exits the port it originally entered.

Our final gadget is the *A/BA Crossover*. The A/BA Crossover has four ports A, B, a, and b in cyclic order, and implements $[A \to a]$ and $[B \to b, A \to a]$. Tsukiji and Hagiwara build a slightly more powerful crossover they call 'CROSS,' which also implements $[A \to a, B \to b]$. However, this is not necessary for PSPACE-hardness, and the A/BA Crossover can easily be constructed using Tsukiji and Hagiwara's Pseudo-Crossing (which is a particular planar embedding of a Switch) and CONJ.

4.2.2 PSPACE-hardness

We now prove PSPACE-hardness for the natural decision problem concerning these gadgets: given a planar system containing Switches, Reversible Fan-ins, and A/BA Crossovers (including these gadgets after some traversals), a starting port which the signal enters first, and

Gadget	Ports	Cyclic Order	Implements
Switch	Set Out Test T-Out F-Out	Any order	[Set → Out, Test → T-Out] [Test → F-Out]
Reversible Fan-in	a b c	(Only one possible)	$[a \to c]$ $[b \to c]$
A/BA Crossover	A B a b	A, B, a, b	$[A \to a]$ $[B \to b, A \to a]$

Table 4.1: Summary of time-reversal-symmetric gadgets required for PSPACE-hardness. Each gadget implements all sequences generated from those listed by prefixes and time-reversal symmetry ($XY, XZ \implies XYY^{-1}Z$).

a target port, does the signal ever reach the target port? We reduce from QBF, still following Tsukiji and Hagiwara [TH11] with some simplification and slightly different abstractions.

We first ignore the requirement of planarity, showing PSPACE-hardness for general systems containing just Switches and Reversible Fan-ins. Then we argue that A/BA Crossovers suffice for all required crossings in a planar embedding of the systems we construct.

Given a quantified formula $Q_1 x_1 : \cdots Q_n x_n : \phi(x_1, \ldots, x_n)$ where ϕ is a 3-CNF formula, we construct a system of Switches and Reversible Fan-ins. At a high level, the system consists of a series of 'quantifier gadgets,' ending in 'CNF evaluation.' When the signal arrives at a quantifier gadget, the quantifier gadget sets the variable it controls, and then queries the next quantifier. Depending on the response, it may perform a second query with the other setting of its variable, and then it sends a response to the previous quantifier. The final quantifier Q_n instead queries the CNF evaluation, which computes the value of ϕ under the current variable assignment. The structure of the reduction is shown in Fig. 4.1.

Because we are working with gadgets which are symmetric under time reversal, we need our quantifier gadgets have this symmetry as well. Quantifiers need to be used multiple times, so we will reset them in the way suggested by time-reversal symmetry: the signal needs to backtrack across its entire path through each quantifier gadget before returning to the previous quantifier. We will describe the desired behavior of quantifier gadgets which are symmetric under time reversal, and later show how to build them using Switches and Reversible Fan-ins.

We specifically discuss universal quantifiers; existential quantifiers require only a minor modification. A universal quantifier gadget Q_i has eight locations, named in cyclic order 'F-Out', 'T-Out', 'In', 'Write-Out', 'Write-In', 'Out', 'T-In', and 'F-In.'[4] The gadget is activated when the signal arrives at In, and the signal proceeds to Out to query the next quantifier;

[4]Tsukiji and Hagiwara call these '$\text{OUT}_{i,\text{FALSE}}$,' '$\text{OUT}_{i,\text{TRUE}}$,' '$\text{IN}_i$,' '$\text{I}_{x_i}$,' '$\text{O}_{x_i}$,' '$\text{IN}_{i+1}$,' '$\text{OUT}_{i+1,\text{TRUE}}$,' and '$\text{OUT}_{i+1,\text{FALSE}}$,' respectively.

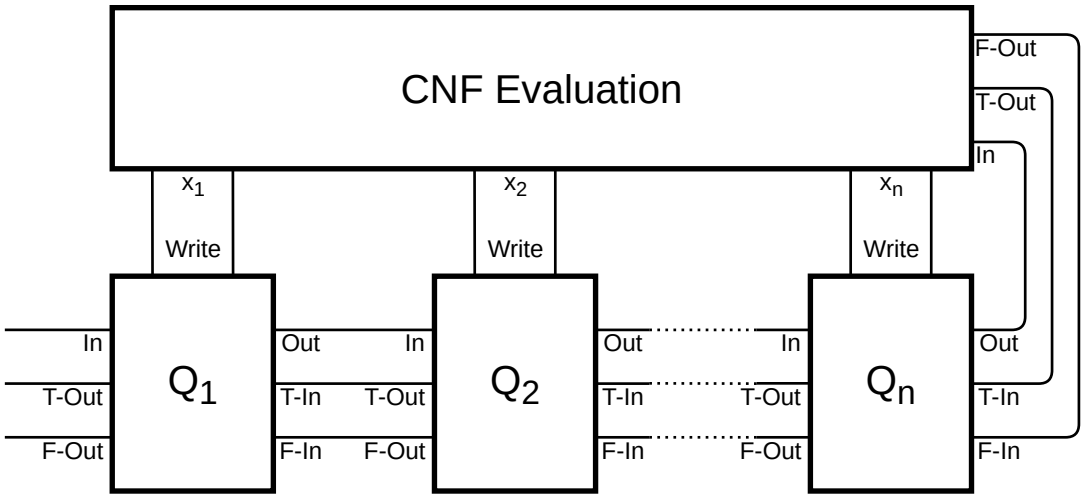


Figure 4.1: The high-level structure of the system produced by our reduction. The signal begins at In on Q_1, evaluates the formula, and eventually arrives at T-Out or F-Out on Q_1 depending on its truth value.

the variable x_i is currently set to False.

Eventually, the signal returns at either T-In or F-In, indicating the truth value of the remainder of the formula with the current variable assignment up to x_i. If it enters at F-In, the universally quantified formula is false, so it passes this along to Q_{i-1} by exiting at F-Out. If it enters at T-In, we need to try the other assignment, which means we need to reset the quantifiers after Q_i by backtracking through them. So the quantifier gadget 'remembers' that it received one True signal, and sends the signal back out T-In. Due to time-reversal symmetry, the signal eventually returns to Out, at which point it is sent to Write-Out to set x_i to True. The signal goes through a series of Switches in the CNF evaluation, and then returns at Write-In. Now Q_i sends the signal to Out, this time with the other setting of x_i. Eventually the signal returns again at either T-In or F-In, and it is sent straight to T-Out or F-Out to answer the query from Q_{i-1}.

Once Q_{i-1} has dealt with the response, the signal returns to Q_i at the same one of T-Out or F-Out it exited, at which point everything is reversed, ending with the signal exiting at In with x_i set to False, and Q_i and all later quantifiers in their initial configuration.

Formally, we need a universal quantifier to implement these sequences (and those implied by time-reversal symmetry), corresponding to the first query to Q_{i+1} returning False, the first query returning True but the second returning False, and both queries returning True, respectively:

- [In $\to$ Out, F-In $\to$ F-Out]
- [In $\to$ Out, T-In $\to$ T-In, Out $\to$ Write-Out, Write-In $\to$ Out, F-In $\to$ F-Out]
- [In $\to$ Out, T-In $\to$ T-In, Out $\to$ Write-Out, Write-In $\to$ Out, T-In $\to$ T-Out]

An existential quantifier gadget is constructed by swapping T-In with F-In and T-Out with F-Out on a universal quantifier gadget.

The signal starts at In on Q_1, which queries the truth value of the whole formula. It eventually arrives at either T-Out or F-Out depending on the answer; we make T-Out on Q_1 the target port. If we connect In, T-Out, and F-Out to themselves, then after evaluating the formula the signal will backtrack all the way to the beginning, and repeat this cycle.

The final quantifier Q_k interfaces directly with the CNF evaluation instead of another quantifier. The CNF evaluation maintains the current variable assignment, initially with all variables False. It has a path for each variable x_i which is connected to Write-Out and Write-In on Q_i; traversing this path forwards sets x_i True, and then traversing it backwards returns x_i to False. The CNF evaluation has three additional ports In, T-Out, and F-Out, analogous to those on a quantifier gadget. When the signal arrives at In, it exits at either T-Out or F-Out depending on the truth value of the formula under the current variable assignment. These ports are connected to Out, T-In, and T-Out on Q_k in the same way as other quantifiers.

By the designed behavior of quantifier gadgets and CNF evaluation, the signal arrives at T-Out on Q_1 if and only if the quantified formula is true. We still need to fill in the details: how do we build quantifier gadgets and CNF evaluation out of Switches and Reversible Fan-ins, and how do we handle crossings?

CNF evaluation

Our CNF evaluation is the same as Tsukiji and Hagiwara's, and is shown in Fig. 4.2. There is a switch for each literal in ϕ. For each variable x_i, there is a path that goes through all switches corresponding to instances of x_i (or $\neg x_i$) in ϕ, and traversing this path sets x_i to True. When the signal enters In, it checks each clause in series. For each clause, it goes through the switches corresponding to literals in the clause, and emerges in one of two locations depending on whether the clause is satisfied. If it is not satisfied, the signal exits at F-Out, and otherwise it proceeds to the next clause, exiting at T-Out once it has passed every clause. Later, it will return to either T-Out or F-Out and reverse its path back to In; the Reversible Fan-ins remember the path taken and necessarily send it back along the same path.

Quantifier gadgets

Our quantifier gadgets are essentially the same as Tsukiji and Hagiwara's; the only differences are due to planar arrangement and that we must build their Switch & Turn gadget out of a Switch and a Reversible Fan-in. The universal quantifier gadget is shown in Fig. 4.3. The existential quantifier gadget is constructed by exchanging the roles of T-In with F-In and T-Out with F-Out, so there is a direct path from T-In to T-Out which crosses some edges linking F-In and F-Out to the other ports. This similarity is sensible: for existential quantifiers if the formula is false we need to try again with the other value, but for universal quantifiers if the formula is true we are allowed to attempt the other required value for the variable.

We must check that the universal quantifier gadget correctly implements the behavior described above. Recall that the signal will first arrive at In. It proceeds to the upper left switch, taking [F-Out $\to$ Test] and leaving the Switch in its default state. Then signal takes

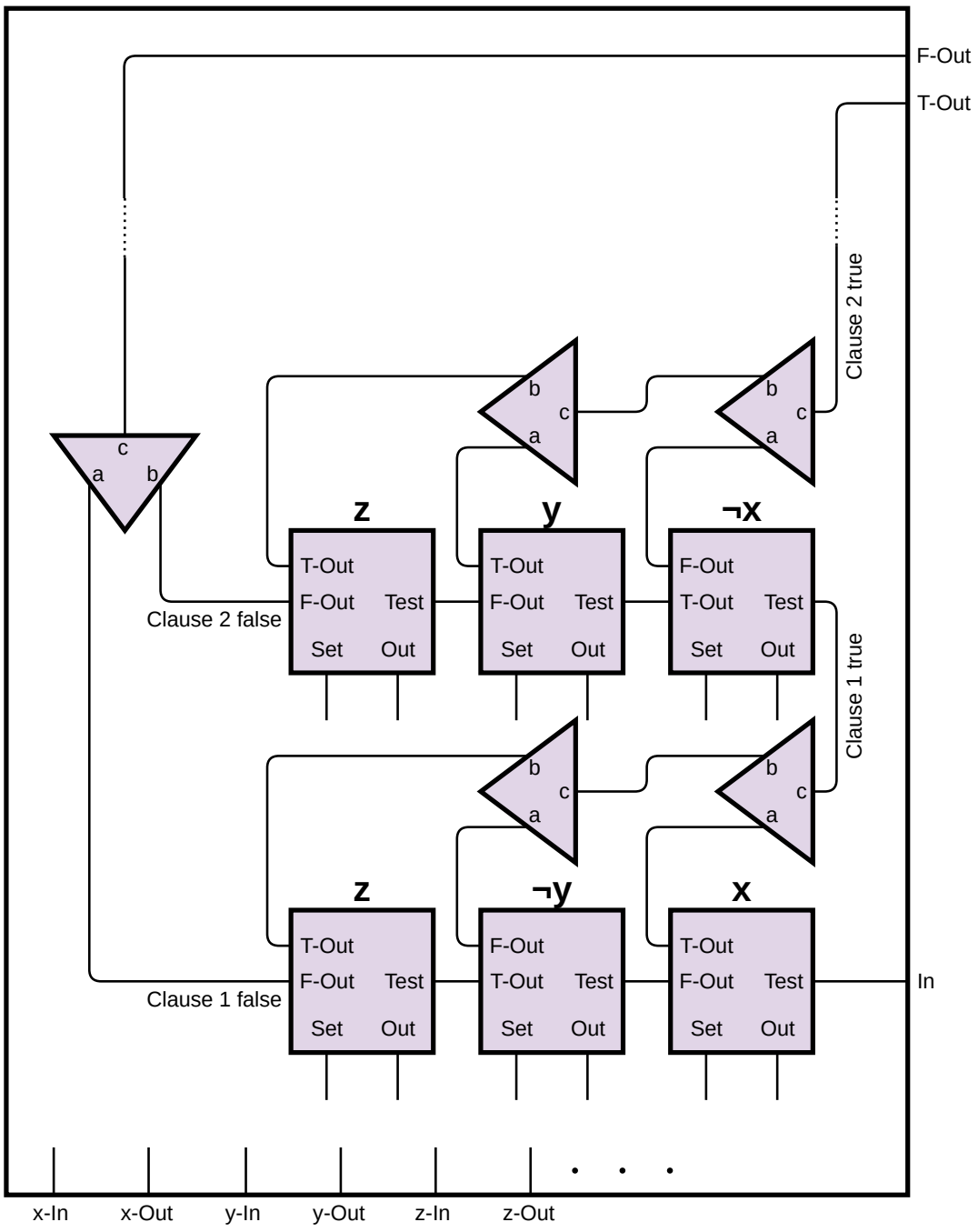


Figure 4.2: Our CNF evaluation. Each clause consists of three Switches corresponding to the literals in the clause, with Reversible Fan-ins to merge paths. A variable and its negation differ in the positions of T-Out and F-Out on the corresponding Switch. When the signal enters In, if any literal in the first clause is true it will take the edge labeled "Clause 1 true" and otherwise will take the edge labeled "Clause 1 false." All the exits for false clauses merge and lead to F-Out. If all clauses are true, the signal will traverse them in series and then exit T-Out. For each variable x_i, there is also a path from x_i-In to x_i-Out which goes through Set $\rightarrow$ Out on the switch corresponding to each instance of x_i or $\neg x_i$.

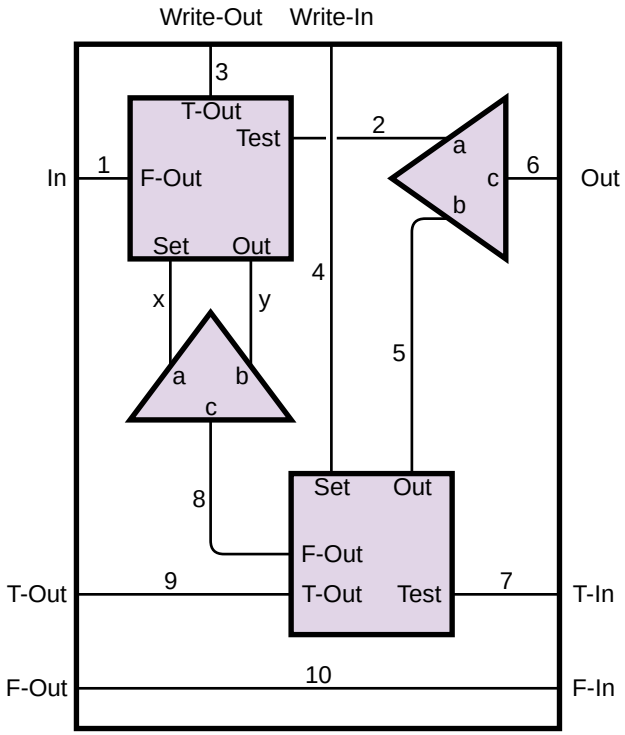


Figure 4.3: The universal quantifier gadget built from two Switches (squares) and two Reversible Fan-ins (triangles). The top Switch is after [Test → F-Out], the bottom left Reversible Fan-in is after $[a \to c]$, and the other two gadgets are their default versions. Edges between gadgets are labeled for later use.

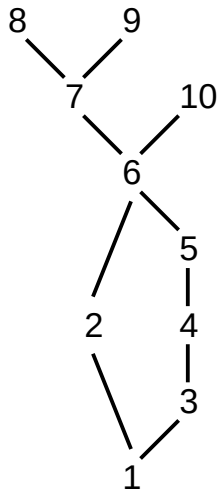


Figure 4.4: A Hasse diagram of the order relation on used edges in our quantifier gadgets (Fig. 4.3). That a is above b indicates that whenever both edges are used, a was used more recently and will be unused sooner.

$[a \to c]$ in the upper right Fan-in and leaves at Out. If it now enters F-In, it goes directly to F-Out. If instead it enters T-In, it goes from Test to F-Out on the bottom switch, goes along edge 8 to the Reversible Fan-In (which is after $[a \to c]$), and traverses $[c \to a]$. Then the signal traverses Set $\to$ Out on the top switch, and returns to the bottom switch via the Reversible Fan-in, leaving both the Switch and Reversible Fan-in in different states than before. The signal then backtracks from F-Out to Test on the bottom Switch, and exits T-In, where it just entered. Now if the signal enters Out, the Reversible Fan-in sends it back to Test on the top Switch along edge 2. But the top Switch has been activated, so the signal exits the Switch at T-Out and exits the quantifier at Write-Out. It next enters Write-In, at which point it traverses Set $\to$ Out on the bottom Switch, and exits Out. Finally, if the signal now enters F-In, it is still sent to F-Out, and if it enters T-In then it goes from Test to T-Out on the bottom switch (which has now been activated) and exits the quantifier at T-Out.

Planarity

Finally, we argue that we can use A/BA Crossovers to avoid crossings in the system produced by this reduction.

Note that each edge in the system is directed, in the sense that the first traversal across the edge is in a predetermined direction which we call *forwards*, and all future traversals alternate direction—we never traverse an edge twice consecutively in the same direction. At any time while running the system, we say an edge is *used* if it has been traversed forwards more recently than backwards. Initially no edges are used, and they are used and unused throughout the process. For two edges x and y which cross, an A/BA Crossover suffices for their crossing provided that whenever both x and y are used, always the same edge—say x—was traversed forwards more recently, and also x will be traversed backwards sooner in the future. In this case, we can set x to be the $A \to a$ tunnel and y to be the $B \to b$ tunnel of an A/BA Crossover. If x and y are never both used, either orientation of the A/BA Crossover will work.

So we just need to argue that there is a consistent order edges (other than the few we showed can avoid crossings) are used. There are no crossings outside the CNF evaluation and quantifier gadgets, so we need only check those gadgets. For the CNF evaluation, this is straightforward:

- For $i < j$, the path to set x_i is used before the path to set x_j.
- Within the path to set x_i, the edges are used in order.
- All paths for setting variables are used before edges involved in testing the current value.
- The edges involved in testing the current value are used in order. Specifically, there is a partial order on these edges based on when it is possible to traverse one and then another on the way from In to T-Out or F-Out. We arbitrarily extend this partial order to a total order, or equivalently, for two edges which can't both be used, we arbitrarily choose which is A and which is B in the A/BA Crossover.

For quantifier gadgets, the numbering listed in Fig. 4.3 works as an order for all edges other than x and y. More generally, a Hasse diagram of the "is sometimes used after" partial order on these edges is shown in Fig. 4.4, and positioning A/BA Crossovers to respect this order suffices for all crossings between these edges. It is straightforward to verify this partial order by considering the behavior of our quantifier gadgets.

For crossings inside a quantifier gadget which involve edge x or y, we need a different approach: for instance, if edge 2 crosses x, then the signal will sometimes traverse 2, then x, then 2 backwards, which isn't supported by the default A/BA Crossover. When x or y is involved in crossing, we use an A/BA crossover as follows:

- If x crosses y, make x the $B \to b$ tunnel since it is always used first.[5]
- If x or y crosses 1, make 1 the $B \to b$ tunnel since it is always used first.
- If x or y crosses 3, 4, 5, 9, or 10, make x or y the $B \to b$ tunnel since they are always used first.
- If x or y crosses 2, 6, 7, or 8, use an A/BA Crossover after $A \to a$, and make the numbered tunnel $a \to A$. By time-reversal symmetry, the A/BA Crossover after $A \to a$ implements $[a \to A, B \to b, A \to a]$, which corresponds for instance to traversing 2 forwards, x backwards, and then 2 backwards, which is what is needed.

To carefully check that this arrangement of A/BA Crossovers works for the quantifier gadget, we can consider the possible sequences of edge traversals. Using $\cdot^{-1}$ for backwards traversals, these are (generated by time-reversal symmetry from)

- $[1, 2, 6, 10]$
- $[1, 2, 6, 7, 8, x, y, 8^{-1}, 7^{-1}, 6^{-1}, 2^{-1}, 3, 4, 5, 6, 10]$
- $[1, 2, 6, 7, 8, x, y, 8^{-1}, 7^{-1}, 6^{-1}, 2^{-1}, 3, 4, 5, 6, 7, 9]$

which correspond to the sequences the quantifier gadget was built to implement. It is straightforward to verify, for each pair of edges, that an A/BA Crossover as described supports all of the ways that pair of tunnels is used. For instance, the possible sequences for just 2 and x are $[2]$ and $[2, x, 2^{-1}]$, which are $[a \to A]$ and $[a \to A, B \to b, A \to a]$ on the A/BA Crossover involved, and both of these are implemented by an A/BA Crossover after $A \to a$. It suffices to check just the sequences listed, since taking the closure under time-reversal symmetry does not give rise to any new intermediate configurations.

Hence we have the main result of this section:

Theorem 4.1. *Given a planar system of Switches, Reversible Fan-ins, A/BA Crossovers, and these gadgets after some traversals, a starting location, and a target location, it is PSPACE-complete to determine whether the signal ever reaches the target location from the starting location. This result holds even when all Switches have any particular cyclic order of ports.*

To apply this framework to a specific problem, we simply need to describe the signal and how it moves along wires, and then construct a Switch (with ports in any order), Reversible Fan-in, and A/BA Crossover.

[5] Alternatively, avoid this crossing by adjusting the Reversible Fan-in connecting x and y.

4.3 Deterministic Constraint Logic

Constraint Logic is a problem about graph orientation reconfiguration introduced by Hearn and Demaine [DH08, HD09] as a tool for proving hardness results. A *constraint graph* is a directed planar graph where each edge has *weight* 1 or 2, which are colored red and blue, respectively.[6] Each vertex in a constraint graph is either an AND vertex, which has two red and one blue edge, or an OR vertex, which has three blue edges. Each vertex is required to have at least 2 total weight in edges pointing towards it. Edges change orientation, while maintaining this constraint. Hearn and Demaine show how to 'tie up' loose edges, allowing the use of degree-2 vertices with any combination of colors, for which the required weight is only 1 (so a single red edge satisfies it).

In this chapter, we are specifically interested in *Deterministic Constraint Logic (DCL)*, in which edges flip according to the following deterministic rule. Each time step, an edge flips if it didn't flip in the previous time step and it can flip without violating the in-weight constraint of the vertex it is currently directed towards, or it did flip in the previous time step but no other edge pointing towards the vertex it is now directed towards can flip this time step.

Here are the basic behaviors that result from the deterministic rule:

- Begin with a path of edges of any color, all pointing to the left. If the leftmost edge flips, all the edges in the path will flip, one in each time step.
- If a blue edge flips to point towards an OR vertex, in the next time step the blue edge which was already pointing towards the OR vertex will flip.
- If a blue edge flips to point towards an AND vertex, in the next time step both red edges pointing towards that vertex will flip.
- If both red edges flip to point towards an AND vertex in the same time step, in the next time step the blue edge will flip.
- If one red edge but not the other flips to point towards an AND vertex, in the next time step the same red edge will flip again.

The decision problem in Deterministic Constraint Logic is whether some specified edge will eventually flip, given a constraint graph and the set of edges that are considered to have flipped in time step 0.

4.3.1 Issue with existing proof

Hearn and Demaine's proof of PSPACE-hardness for Deterministic Constraint Logic [HD09] has a subtle issue. When their universal quantifier receives a 'satisfied in' signal, it records this fact, much like our universal quantifier gadget. When it receives a second 'satisfied in' signal (assuming the signal did not enter 'try out' in between), it erases the record of the first one; this is by design, to reset the gadget for the next variable assignment.

[6] In grayscale, blue edges are darker than red edges. Figures also draw blue edges thicker than red edges.

The existential quantifier tries assigning its variable False, then True, and then False again, and passes every 'satisfied in' signal it gets to 'satisfied out' to inform the previous quantifier. If the existential quantifier is satisfied when its variable is False but not True, it sends two such signals instead of one. This is the problem: if the previous quantifier is universal, the second signal cancels the first one, and that quantifier behaves as though there was no signal. The simplest formula for which the reduction fails is $\forall x \exists y : \neg y$. Modifying the existential quantifier to test each assignment exactly once does not fix the problem, because then if the quantifier is satisfied by both values for its variable, it sends two signals to the previous quantifier. In particular, $\forall x \exists y : y \vee \neg y$ would fail.

The proof may be fixable by modifying the existential quantifier gadget to ensure it only ever sends one signal; it would likely be about as complicated as the universal quantifier. The approach our framework takes is different: it adds an additional query return line, so instead of just 'satisfied in' we have both T-in and F-in, and quantifier gadgets are guaranteed to receive exactly one response for each query.

4.3.2 PSPACE-hardness

Our PSPACE-hardness proof for Deterministic Constraint Logic uses many of the same elements as Hearn and Demaine's. The signal is a flipping edge, which propagates along paths in the direction opposite the orientation of the edges in the path. Like Hearn and Demaine, our gadgets will sometimes contain 'bouncing' edges which flip in a periodic way, and we ensure the length of each path through a gadget is a multiple of this period—for us, the period is 2, though Hearn and Demaine used a period of 4. The ports of our gadgets are always blue edges, which are connected by joining them with a degree-2 vertex. The target edge is the edge corresponding to the target port, and it flips if and only if the signal reaches the target port.

While DCL itself is symmetric under time reversal, it is possible to build a DCL gadget which is not, by including periodically bouncing edges calibrated such that the signal enters out of phase with when it exits. Some of Hearn and Demaine's gadgets [HD09] behave this way. However, all of our gadgets will be symmetric under time reversal in all of their relevant behavior, as is required for the framework we are using.

We simply need to build valid Switch, Reversible Fan-in, and A/BA Crossover gadgets. A Reversible Fan-in is simply an OR vertex, which always takes 2 time steps to traverse. We use Hearn and Demaine's A/BA crossover, which we reproduce in Fig. 4.5. This A/BA crossover always takes an even number of time steps to traverse, and contains bouncing edges with period 2.

Our Switch gadget is a bit more complicated, and is shown in Fig. 4.6. If the signal arrives at Set, it exits at Out and reflects the configuration by flipping the bottom four edges and setting the left red edge bouncing instead of the right red edge. If the signal arrives at Test, it exits either F-Out or T-Out based on which red edge is currently bouncing, and sets one of the top red edges bouncing. Every traversal through this gadget takes four time steps.

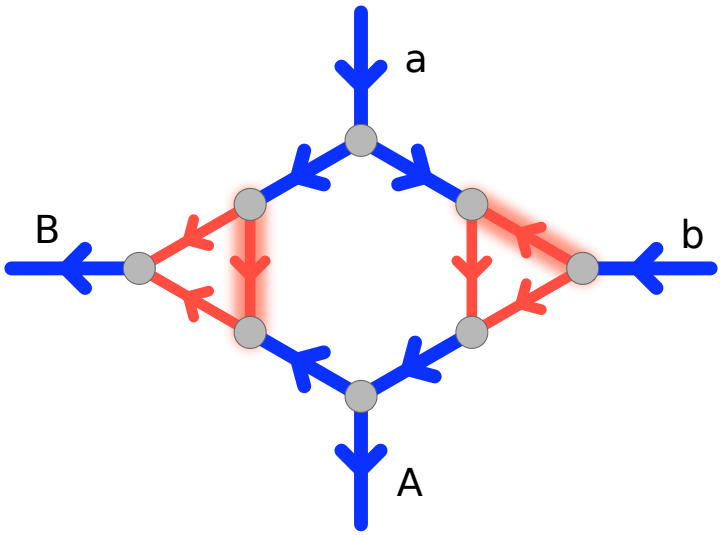


Figure 4.5: An A/BA Crossover for Deterministic Constraint Logic, from Hearn and Demaine [HD09]. Glowing auras indicate edges that flip every time step—the state shown is the state immediately before the signal enters the gadget, so that when the signal enters, the blue edge at the entered port and all glowing edges simultaneously flip from the shown configuration.

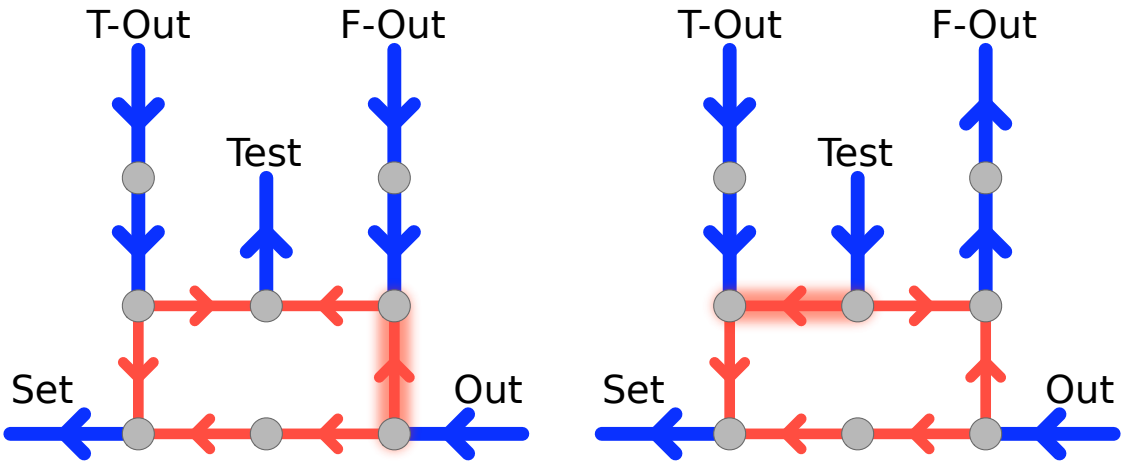


Figure 4.6: A Switch for Deterministic Constraint Logic. Left: the initial configuration. Right: the configuration after the traversal Test → F-Out.

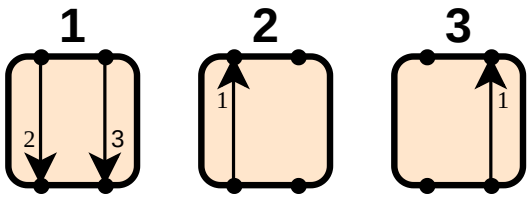


Figure 4.7: The locking 2-toggle. All the locking 2-toggles we show have this layout, which Demaine et al. [DHL20] call the 'parallel' locking 2-toggle (and Demaine et al.'s results imply that any single planar embedding is sufficient).

4.4 Locking 2-toggles

Our next application is a zero-player version of decision problems considered by Demaine et al. [DHL20], and also fits in the framework from Section 4.2. The *locking 2-toggle* is a gadget with two directed tunnels, where traversing either one flips its direction and disables the other tunnel until the traversed tunnel is traversed again in the opposite direction. A diagram is shown in Fig. 4.7. To adapt the locking 2-toggle to the fully deterministic setting, we say that if the signal arrives at a port where it cannot currently cross the tunnel, it 'bounces off,' exiting the same port. The locking 2-toggle is symmetric under time reversal.

The locking 2-toggle alone is boring in this setting: because it has separate tunnels, the signal is restricted to a linear path. Since the locking 2-toggle obeys time-reversal symmetry, when the signal bounces it will backtrack everything it has done, and not produce interesting new behavior. In particular, the reachability question can be solved in logarithmic space.

To make a more interesting problem, we introduce a zero-player analog of the 'branching hallways' used in one-player motion planning. This is a new gadget we call *rotate clockwise*, and the other enantiomer *rotate counterclockwise*. Rotate clockwise has three ports, and the signal always exits immediately clockwise of the port it entered. This is like a 3-spinner which doesn't change state. Rotate Clockwise and the notion of bouncing off of closed ports also appear in Asynchronous Ballistic Reversible Logic [Fra17], which is similar to the model just defined, but with multiple signals.

Rotate clockwise is not symmetric under time reversal: the time-reversed rotate clockwise is exactly rotate counterclockwise. But we will build gadgets which are symmetric under time reversal—at least, provided the sequence of input ports is one we need to account for—out of rotate clockwise and locking 2-toggles.

The decision problem is whether, in a given system of locking 2-toggles and rotate clockwise, the signal ever reaches some location. For PSPACE-hardness, we need to construct a Switch, a Reversible Fan-in, and an A/BA Crossover. These gadgets are shown in Fig. 4.8. The A/BA Crossover is the same as the one built by Demaine et al. [DHL20] for the nondeterministic setting, with branching hallways replaced by rotate clockwise. We are able to construct all of these gadgets with just rotate clockwise, without using rotate counterclockwise.

Demaine et al. [DHL20] showed that any gadget from a large class—what they call "interacting-tunnels reversible deterministic gadgets"—can simulate the locking 2-toggle.

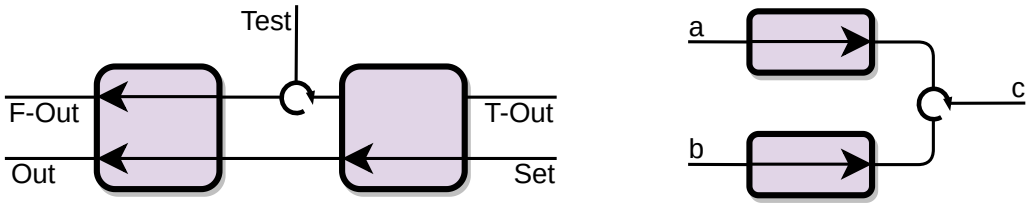


Figure 4.8: The Switch and Reversible Fan-in built out of locking 2-toggles and rotate clockwise. It is easy to verify correctness. Note that we cannot rely on time-reversal symmetry; we must check each desired sequence followed by its time-reverse, and this needs to return the gadget to its initial state. The A/BA Crossover from Demaine et al. [DHL20] is complicated to build with parallel locking 2-toggles, so we omit it.

These simulations do not use any branching hallways, so they work in our fully deterministic model as well. Hence zero-player motion planning with any interacting-tunnels reversible deterministic gadget and rotate clockwise is PSPACE-complete.

Our gadgets for locking 2-toggles, including those from Demaine et al. involved in building the A/BA Crossover, don't rely on the precise behavior of rotate clockwise: any instances of rotate clockwise could be replaced with rotate counterclockwise, and the reduction would still work. In fact—though this is more complicated to verify because the resulting gadgets are nondeterministic (in particular, the player can choose to turn around at any time, but this is the only resulting nondeterminism)—all instances of rotate clockwise can be replaced with branching hallways, yielding a reduction to one-player motion planning with locking 2-toggles. This is a new and arguably simpler proof of PSPACE-hardness than the original by Demaine et al. [DHL20].

4.5 3-spinners

One additional application of our framework is a strictly more general decision problem, and thus a strictly weaker hardness result, than the hexagonal-lattice version of Langton's ant which Tsukiji and Hagiwara prove PSPACE-complete [TH11]. We include it because it resolves a question posed by Demaine et al. [DGLR18], who were not aware of Tsukiji and Hagiwara's work: this result implies that one-player motion planning with 3-spinners is PSPACE-complete, since the player would never have nontrivial decisions to make because 3-spinners obey time-reversal symmetry. We are able to simplify the gadgets involved a bit because we will not restrict to a hexagonal lattice like Tsukiji and Hagiwara do.

The *3-spinner* is a particular gadget which fits in the framework described in Section 4.2 and is symmetric under time reversal. It has three locations a_1, a_2, and a_3, and implements all sequences which alternate between traversals of the form $a_i \to a_{i+1}$ and $a_i \to a_{i-1}$ with indices mod 3. That is, the first time the signal enters a 3-spinner, it exits one position 'clockwise' of the entrance, the next time it exits one position 'counterclockwise,' and this alternates. A diagram of the 3-spinner is shown in Fig. 4.9.

Zero-player motion planning with 3-spinners asks whether the signal ever reaches some location in a system of 3-spinners. To prove PSPACE-hardness even in planar systems, we

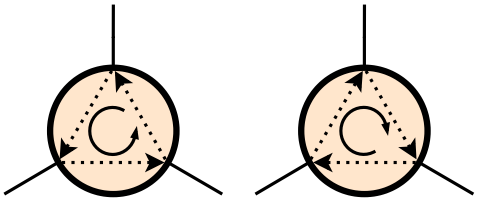


Figure 4.9: The 3-spinner. In the left state, the signal is sent one port counterclockwise, and in the right state it is sent one port clockwise, as indicating by the dotted arrows. Each traversal flips the state. The circular arrow indicates the current state.

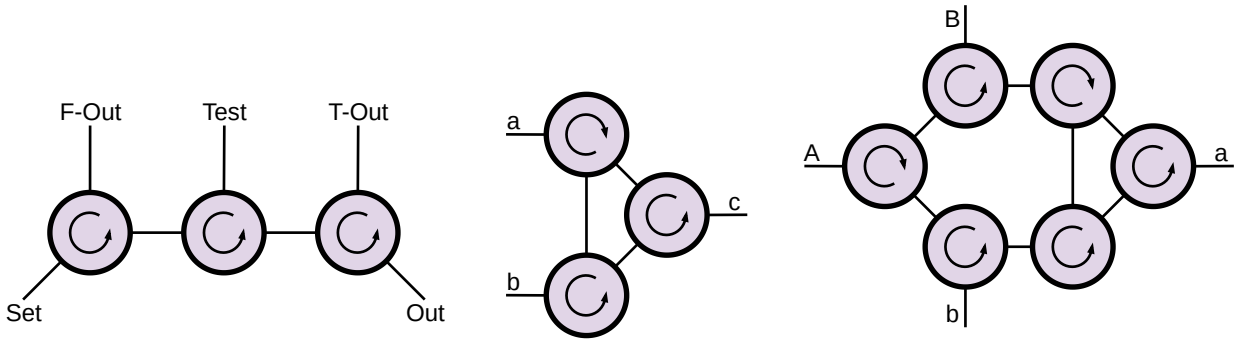


Figure 4.10: The Switch, Reversible Fan-in, and A/BA Crossover built out of 3-spinners, based on gadgets from Tsukiji and Hagiwara [TH11]. Correctness is easily verified by testing each desired sequence of input ports. The A/BA Crossover is the combination of a (differently laid out) Switch and a Reversible Fan-in.

just need to show how to build a Switch, a Reversible Fan-in, and an A/BA Crossover out of 3-spinners. These constructions are simplified versions of gadgets by Tsukiji and Hagiwara. Our gadgets are shown in Fig. 4.10.

4.6 Billiard balls

Our final application is the billiard ball model, which was introduced by Fredkin and Toffoli [FT82] and is one of the best known reversible models of computation. In the billiard ball model, there are circular *balls* colliding elastically with each other and with fixed *mirrors*. For simplicity, all balls have the same size and mass, and will only move at a single nonzero speed. This model is based on classical physics, and in fact exactly matches the classical kinetic theory of perfect gasses.

The decision problem we consider is whether a ball ever reaches a particular position, given a configuration of mirrors and initial positions and velocities of balls. Fredkin and Toffoli [FT82] proved that this model can perform arbitrary computation by showing how to build and string together Fredkin gates; it follows that the decision problem is PSPACE-complete.

We present a new proof of PSPACE-hardness using our framework. The primary ad-

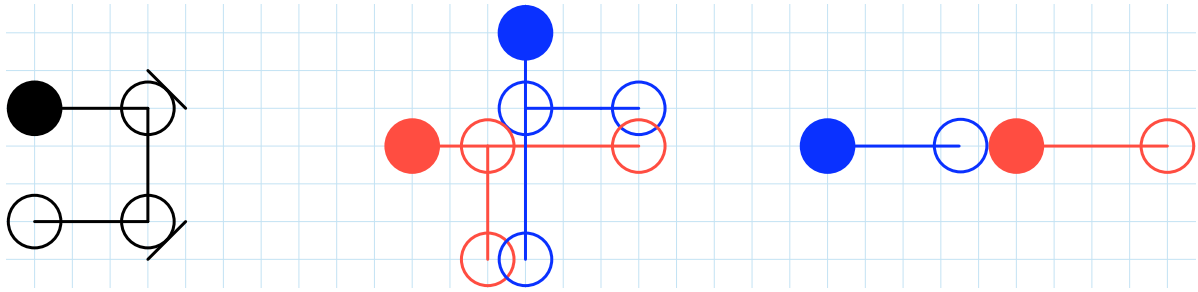


Figure 4.11: The billiard ball model. Filled circles depict initial positions of balls, and empty circles depict intermediate or final positions. Diagonal lines are mirrors, and horizontal or vertical lines are paths taken by balls. Left: a ball bounces off of mirrors. Middle: two moving balls collide. If only one ball arrives, it goes straight through, but if both balls arrive simultaneously, they bounce off each other. Right: A moving blue ball collides with a stationary red ball, transferring its momentum and leaving the blue ball not grid-aligned.

vantage this proof has over Fredkin and Toffoli's is that only a constant number—namely two—of balls will be moving at any time, and the two moving balls will always be in close proximity. This means there are fewer details to work out relating to issues like timing; Fredkin and Toffoli had to ensure that signals from disparate parts of the construction arrive at a logic gate simultaneously.

The balls in our construction all have a radius of $\frac{1}{\sqrt{2}}$, and will move only horizontally or vertically. The types of collisions that will occur are shown in Fig. 4.11. One can think of a head-on collision with a stationary ball as moving the stationary ball backwards by the ball diameter, and teleporting the moving ball forwards by the same amount.

The signal will be represented by two balls moving along parallel paths $2\sqrt{2}$ (i.e. twice the diameter) apart. This signal is easy to route, as demonstrated by Fig. 4.12. We will always have the two balls aligned with each other when the signal enters a gadget. Full crossovers, and in particular A/BA crossovers, are trivial: simply have two paths the signal might take cross each other. For simplicity, our diagrams show the paths separated by 3 units, rather than the actual distance $2\sqrt{2} \approx 2.8$.

All that remains is constructing the Switch and Reversible Fan-In. Our Switch is shown in its initial state Fig. 4.13. The key idea is that stationary balls inside the gadget might (depending on the state) be in the way of one of the balls in the signal entering at Test, effectively making that ball arrive slightly earlier. This change in timing affects whether that ball collides with the other ball in the signal, resulting in two possible places for the signal to end up.

The three relevant traversals are shown in Fig. 4.14. Since the model has time-reversal symmetry, any gadget built in it also has time-reversal symmetry, so we only need to check that the sequences listed in Table 4.1 are implemented correctly.

Finally, our Reversible Fan-in is shown in Fig. 4.15. It works in a very similar way to Switch, but in reverse, and essentially combining the Set traversal with one of the Test traversals. If the signal enters at a, the balls collide and arrive at c. If the signal enters at b, the signal balls do not collide, and arrive at c with a slightly different timing. To correct the timing, we have the signal entering at b first remove two balls from the path near c.

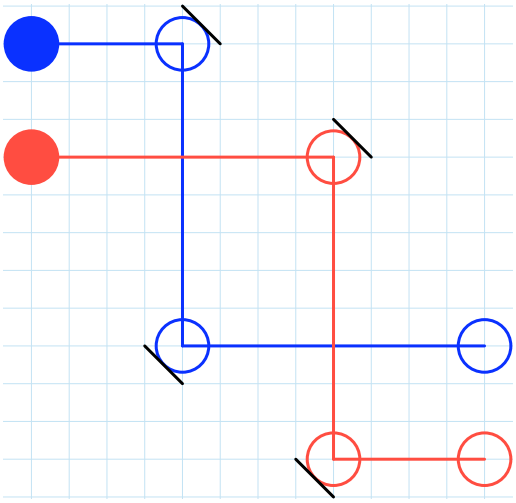


Figure 4.12: A signal consisting of two billiard balls is sent from the top left to the bottom right. The paths of the two balls have the same length.

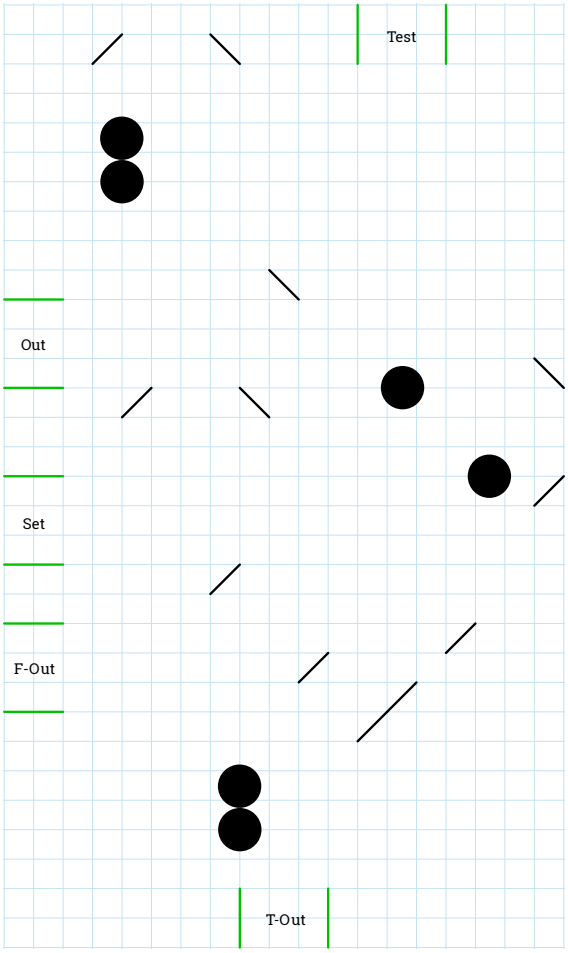


Figure 4.13: The Switch for the billiard ball model. Each port is marked with a pair of green lines, along which the two balls of the signal may enter or exit.

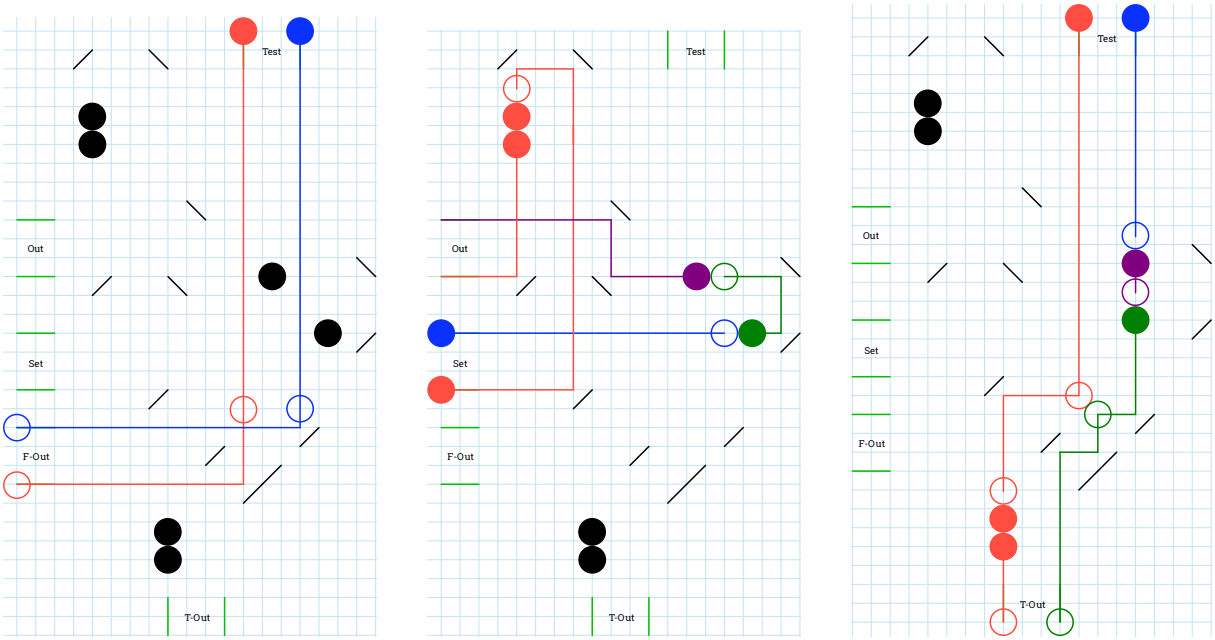


Figure 4.14: The ways a signal moves through the switch. Left: in the initial state, the signal bounces straight from Test to F-Out. The two balls don't collide where there paths cross. Middle: the blue ball hits the green, which hits the purple, leaving two balls in the path of the Test port. The red ball's path is extended north so that two balls exit at Out simultaneously; the two red balls in its path save the same amount of time as the two balls in the blue ball's path. Right: with the purple and green balls in the way of the signal entering Test, the green ball arrives soon enough to collide with the red ball, resulting in the signal exiting at F-Out. The two additional red balls are again to help synchronize the exit signal.

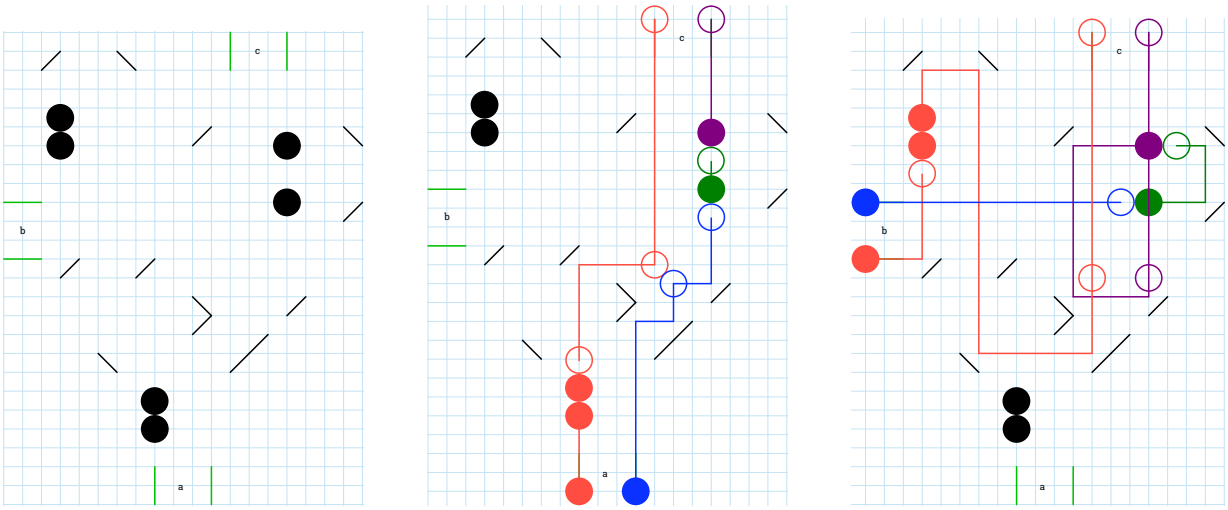


Figure 4.15: The Reversible Fan-in for the billiard ball model. Left: the gadget in its initial state. Middle: the signal enters at a. The signal balls ricochet off each other, and then exit at c. They each collide with two stationary balls, so the balls exiting c get there at the same time. Right: The signal enters at b. The blue ball knocks the green ball, which knocks the purple ball, clearing the vertical path to c. The red ball and the purple ball then exit at c without colliding. The red zigzag to the north and two additional red balls are to make the timing correct.

Chapter 5

Graph Orientation Gadgets and Minesweeper

In this chapter, I present a version of the graph orientation gadget framework designed for hidden-information puzzles like Minesweeper. This framework provides tools to prove hardness of three natural decision problems for such a puzzle: consistency, inference, and solvability. I then apply the framework to many puzzles from the game *14 Minesweeper Variants*, and prove that (standard) Minesweeper solvability is coNP-complete even from an initial configuration. One of the results fixes an error in prior work proving coNP-hardness of Minesweeper inference [SSvR11].

This chapter represents joint work with Andy Tockman, which first appeared in [GHT24].

5.1 Introduction

The puzzle-based video game Minesweeper was popularized by its inclusion in the default installation of Windows 3.1 in 1992, and to this day is one of the most widely recognizable computer games. Though the premise is very simple, it has spawned an endless stream of spinoffs, and a number of communities dedicated to challenges such as completing a randomly generated game as quickly as possible.

From a computational complexity perspective, Minesweeper is interesting in that there are multiple natural decision problems to study. The simplest is *consistency* ('given some clues, is there a possible arrangement of mines?'), which was proved NP-complete in 2000 [Kay00].

But consistency doesn't capture the essence of Minesweeper as a game, where the player has some partial information and tries to find a cell that they can click on safely, leading to more information. This suggests the *inference* problem ('given some clues, is there a cell that is provably safe?'), which was shown coNP-complete in 2011 [SSvR11]. Unfortunately, this proof of coNP-hardness of Minesweeper inference is incorrect, and Thieme and Basten's proof [TB22], which is designed to use very small gadgets, suffers from the same issue.

While inference asks about a single 'turn' of Minesweeper, it is also natural to ask a related question about the entire game. We introduce the *solvability* decision problem, in which we are given both the current state of a Minesweeper game and also the secret arrangement of

mines, and asked whether the player can make a sequence of safe clicks to solve the game.

In this chapter, we develop a framework based on graph orientation to prove coNP-completeness of Minesweeper inference and solvability. As a side effect, the same gadgets also suffice to prove NP-hardness of Minesweeper consistency. Of particular note, we prove that Minesweeper solvability is coNP-complete even 'after a single click': from a nearly empty initial state with only one cell revealed.

Specifically, we define three graph orientation decision problems (consistency, promise-inference, and uniqueness) related to the Minesweeper problems, and show that each is hard with a particular set of simple abstract gadgets. It follows that finding well-behaved constructions in Minesweeper which behave like those gadgets is enough to prove hardness for all three Minesweeper problems.

In Section 5.2, we define each decision problem carefully, summarize the previously known results, and explain the flaw in the existing reduction for inference. In Section 5.3, we develop a framework of gadgets that makes it easy to prove hardness results for all three decision problems. Finally, in Section 5.4 we apply the framework to Minesweeper and to many variants of Minesweeper from the video game *14 Minesweeper Variants*. For the most part, each application consists entirely of constructions of the relevant gadgets in the Minesweeper variant under consideration.

5.2 Prior work and definitions

5.2.1 Consistency

Research on the computational complexity of Minesweeper began when Sadie Kaye [Kay00] posed the Minesweeper consistency problem. Informally, this problem asks whether a partially completed Minesweeper board has a legal arrangement of mines. We provide a formal definition here.

Definition 5.1. A *partial board* is a two-dimensional rectangular array, where each entry is either a *covered cell* or an *uncovered cell*. A covered cell is an unknown cell which may or may not be a mine. An uncovered cell has an integer (and is known to not contain a mine), representing a Minesweeper clue. For a partial board B, we denote the set of covered cells $\mathcal{C}(B)$, and the set of uncovered cells $\mathcal{U}(B)$.

Definition 5.2 (Minesweeper consistency problem)**.** A partial board B is *consistent* if there exists a set $M \subseteq \mathcal{C}(B)$, representing all locations of mines, such that $M \cap \mathcal{U}(B) = \emptyset$ and the integer in each uncovered cell $c \in \mathcal{U}(B)$ counts the number of cells in M which are orthogonally or diagonally adjacent to c. Otherwise, B is *inconsistent*. See Figs. 5.1 and 5.2. The input to the *Minesweeper consistency problem* is a partial board, and the problem asks whether it is consistent.

Kaye proves that the consistency problem is NP-complete [Kay00]. We also prove NP-completeness as a side effect of our framework.

Proposition 5.3. *The Minesweeper consistency problem is in NP.*

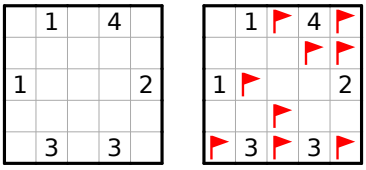


Figure 5.1: Left: an example of a consistent board. Right: one possible way to satisfy all clues.

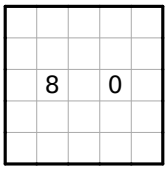


Figure 5.2: An example of an inconsistent board.

Proof. Given a partial board B, a certificate of consistency is the M of Definition 5.2. We can iterate over $\mathcal{U}(B)$ and check that each clue is satisfied in polynomial time. □

We leave the proof of hardness to Section 5.3.

5.2.2 Inference

The Minesweeper inference problem was first posed by Allan Scott, Ulrike Stege, and Iris van Rooij [SSvR11]. Informally, this problem asks whether a partially completed Minesweeper board has a logical deduction available to the player that lets them click on a cell which is guaranteed to not have a mine. We provide a formal definition of a slight variation on the problem as originally posed.

Definition 5.4 (Minesweeper inference problem)**.** Given a partial board B, an *inference* is a cell $c \in \mathcal{U}(B)$ such that for all consistent arrangements of mines $M \subseteq \mathcal{C}(B)$, we have $c \notin M$. The input to the *Minesweeper inference problem* is a partial board, and it asks whether there is an inference. See Figs. 5.3 and 5.4.

Note that this definition has two key differences from the one originally given by Scott, Stege, and van Rooij [SSvR11]:

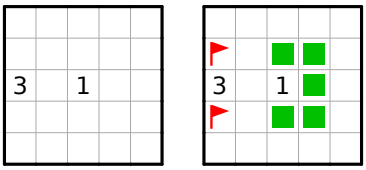


Figure 5.3: Left: an example of a board with an inference. Right: green squares mark cells that can be inferred.

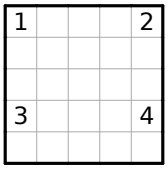


Figure 5.4: An example of a board that doesn't have an inference.

- Deducing the location of a mine does not count as an inference.
- No positions of known mines are given in the input.

We prefer Definition 5.4 because it eliminates the need to place conditions on the input such as "all given mines must be deducible from the clues," which is otherwise necessary to avoid placing mines that could never be deduced in a real game. Furthermore, the flags representing known mines can be viewed as simply a player aid—one could in principle play a full game of Minesweeper without marking a single mine. Though we will proceed with Definition 5.4 only, we remark that all results in this chapter work under either definition.

Proposition 5.5. *The Minesweeper inference problem is in coNP.*

Proof. Given a partial board B, a certificate of noninference is, for each cell $c \in \mathcal{C}(B)$, a consistent arrangement of mines M with $c \in M$. We can iterate over the polynomially many arrangements given by the certificate and check consistency in polynomial time. □

Again, the proof of hardness will be in Section 5.3.

Error in existing proofs. Scott, Stege, and Rooij's proof of coNP-hardness [SSvR11] has an issue where there is sometimes an unintended inference. Their OR gate is shown in Fig. 5.5. A cell has a mine when the literal marking it is true. The inputs are u and v, and the output is a. The 6 enforces $a + b = u + v$, and the section at the bottom enforces $a \geq b$. Together this forces $a = u \vee v$ and $b = u \wedge v$.

The issue occurs when we know that u and v can't both be true. This might happen if the OR gate is used inside an AND gate which merges two clauses which can't simultaneously be false, such as if one contains x and the other contains $\neg x$.

In this case, we can deduce that $b = u \wedge v$ is false, and thus the (higher) cell labeled b is safe to click. This is an inference. One strategy for resolving this issue would be to reveal that cell in the input, assuming one can find all inferences like this. But this doesn't work: That cell has either a 3 or a 4 depending on the value of u, so revealing it tells the player the value of u, allowing them to make more inferences and possibly learning further information.

Thieme and Basten's more compact proof [TB22] has a very similar issue.

Our approach to avoiding this issue is twofold. First, we design our gadgets to be 'silent', meaning clicking an inferred cell never reveals information. This allows us to eliminate inferences by revealing those cells in the input. This handles unintended 'local' inferences, which we are able to detect.

Second, to prevent unintended larger-scale inferences, we design the network of gadgets carefully. Specifically, we ensure that for every gadget (OR gate, crossover, etc.), every

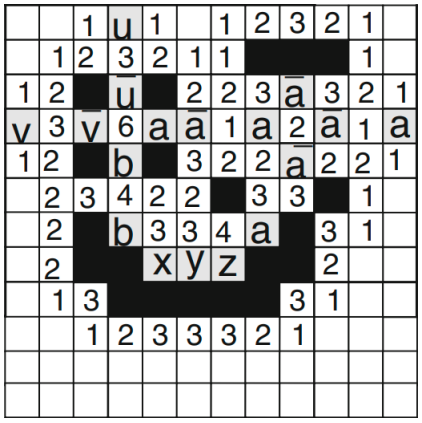


Figure 5.5: The OR gate from [SSvR11] Figure 16. There is a minor typo: the lower cell with b should have $\overline{b}$.

locally valid combination of values can be achieved. The only exception is the 'final' gate, which has a forced output when the input formula is unsatisfiable. We maintain this property across simulations by introducing a problem we call 'promise-inference', which also partially relaxes the constraint that every locally valid solution is achievable.

5.2.3 Solvability

Informally, the Minesweeper solvability problem asks whether a player can make a sequence of inferences from a partially completed Minesweeper game and win by clicking on all non-mine cells. We provide a formal definition of this problem, which to our knowledge has not been studied.

Definition 5.6 (Minesweeper solvability problem)**.** Let B be a partial board with uncovered cells $K = \mathcal{U}(B)$, and let M be a set of mines consistent with B. Note that B is determined by M and K. Here M represents the secret set of mines, which is thought of as unknown to the player, and K represents the set of known cells.

Consider an ordering O of $G \setminus (M \cup K)$, meaning a list containing each covered non-mine cell exactly once. For such a cell $o \in G \setminus (M \cup K)$, let $O = O_{init} +\!\!+ [o] +\!\!+ O_{tail}$, where $+\!\!+$ denotes concatenation. We say M is *solvable* from K if there exists an O such that for all $o \in O$, o is a inference for the (unique) partial board consistent with M whose uncovered cells are $K \cup O_{init}$. Otherwise, M is *unsolvable* from K. See Fig. 5.6.

The input to the *Minesweeper solvability problem* consists of the dimensions of a rectangular grid G and two disjoint subsets of cells $M \subseteq G$ and $K \subseteq G$. The problem asks whether M is solvable from K.

Intuitively, solvability simulates a player who is given a Minesweeper puzzle on their computer to solve. If such a player wishes to guarantee that they do not click a mine (to win without any luck, or because they have antagonistic implementation which repositions mines to make the player lose if they click a cell that isn't provably safe, such as 'expert mode' in *14 Minesweeper Variants*), they must click covered cells in an order O that satisfies Definition 5.6.

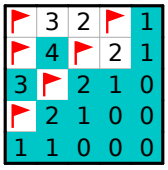


Figure 5.6: An example of a solvable board. Cyan cells indicate elements of K, and white cells are unknown to the player. Red flags indicate elements of M. From the initial state, the only provably safe cell is the fourth cell in the second row. After clicking it, the player can now deduce the third cell in the first row. Finally, this reveals enough information to deduce the second cell in the first row.

Proposition 5.7. *The Minesweeper solvability problem is in coNP.*

Proof. Given a game (M, K), a certificate of unsolvability is a set $K' \supseteq K$ together with a certificate of noninference for K'.

Suppose there is such a certificate. As long as the information available to the player is a subset of K', they cannot infer that a cell outside K' is safe. Any order has a first cell outside K', which by assumption is not an inference.

Conversely, suppose an instance is unsolvable. Starting from K, repeatedly make inferences and add them to the known information. By assumption, this must get stuck before all non-mine cells are uncovered, meaning we reach a point with no inferences. Then take K' to be the uncovered cells at that point. □

Once again, we prove hardness in Section 5.3.

5.3 The Minesweeper gadget framework

In this section, we describe an abstract framework which we will later apply to Minesweeper. The wires in Minesweeper hardness proofs generally have two states, which can be thought of as two orientations of an edge, so our framework is a general form of graph orientation. Other work on abstract graph orientation gadgets has focused on our first decision problem, consistency, and there is a fairly complete characterization of the complexity with certain types of vertex or gadget [GAD+26].

Definition 5.8. A *gadget* is an abstract object which has

- a finite set of *ports*, in a specified cyclic order (we will generally list the ports in this order, and describe it more explicitly when relevant).
- a *constraint*, which is a set of subsets of the ports.

Gadgets will interact via directed edges connecting ports. The constraint says which sets of edges pointing towards the gadget should be considered legal.

The gadgets we name are collected in Table 5.1, and will also be described as they come up.

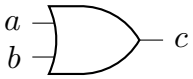
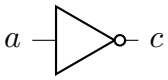
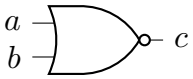
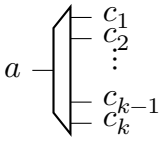
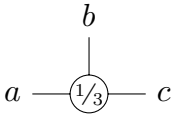
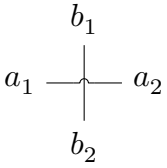
Name	Icon	Constraint
fixed (true) terminal	a — F	$\{\{\}\}$
fixed (false) terminal	a — T	$\{\{a\}\}$
free terminal	a —	$\{\{\}, \{a\}\}$
AND gate		$\{\{c\}, \{a, c\}, \{b, c\}, \{a, b\}\}$
OR gate		$\{\{c\}, \{a\}, \{b\}, \{a, b\}\}$
NOT gate		$\{\{\}, \{a, c\}\}$
NOR gate		$\{\{\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$
(k-way) fanout gate		$\{\{a\}, \{c_1, \ldots, c_k\}\}$
1-in-3 gadget		$\{\{a\}, \{b\}, \{c\}\}$
crossover		$\{\{a_1, b_1\}, \{a_2, b_1\}, \{a_1, b_2\}, \{a_2, b_2\}\}$

Table 5.1: The gadgets we define in this chapter, the icons we use to draw them (for those we show in networks), and their constraints.

Definition 5.9. A *network* of gadgets from a set S is an undirected graph where

- each vertex is labeled with a gadget from S
- if $G \in S$ has k ports, each vertex labeled G has degree k and its edge incidencies are labeled in a bijection with ports of G.

A *planar network* is such a graph equipped with a planar embedding, such that the cyclic order of edges around each vertex matches the order of the ports of the corresponding gadget.

We often equivocate between vertices and gadgets, and between edge incidencies and ports—think of each vertex as a copy of its label, and think of edges as connecting ports to ports in a matching.

We draw planar networks using the icons in Table 5.1, which also indicate the correspondence between edges and ports (except for when it doesn't matter by symmetry).

Definition 5.10. An *assignment* to a (planar) network assigns a direction to each edge. A vertex is *satisfied* if the set of (labels of) edges pointing into it is in its (label's) constraint. An assignment is *satisfying* if every vertex is satisfied.

Planar Graph Orientation (PGO) is the study of satisfying assignments of planar networks.

5.3.1 Gates as gadgets

One important kind of gadget is logic gates, which can be interpreted as gadgets: inputs are edges entering from the left and output are edges exiting on the right. Interpret pointing right as true. The gadget's constraint allows the inputs to be arbitrary, but forces the correct outputs for each combination of inputs. More precisely, for each subset S of input ports, the constraint contains $S \cup T$, where T is the set of outputs that are *false* when precisely the inputs in S are true.

It's important to keep in mind distinction between gate gadgets, which compute a value, and "normal" gadgets, which enforce a constraint.

For example, the *OR gate* has ports $\{a, b, c\}$ and constraint $\{\{c\}, \{a\}, \{b\}, \{a, b\}\}$. This constraint allows any subset of the input ports a and b to have edges pointing in, and enforces that the edge incident to the output port c to point out exactly when at least one input port points in. In other words, it computes the OR of a and b, and outputs it at c.

On the other hand, the *OR gadget* (which we don't need beyond this example) has only two ports $\{a, b\}$ and constraint $\{\{a\}, \{b\}, \{a, b\}\}$. It enforces that at least one edge points in, or that at least one input is true. This is equivalent to an OR gate with the output forced to be "true" (pointing away).

Any boolean circuit can be represented as a network of gate gadgets, by attaching input ports to output ports in the natural way. Allow us to leave inputs and outputs to the full circuit as "dangling" edges for now. It follows from the definition of gate gadgets—formally, one can induct on the depth of the circuit—that this network has exactly one satisfying assignment for each combination of orientations of the dangling input edges, and in each the

orientations of the output edges encode the output of the circuit (and internal edges encode the internal state of the circuit).

If an output connects to $k > 1$ inputs, we need to use a k-way *fanout gate*, which has ports $\{a, c_1, \dots, c_k\}$ and constraint $\{\{a\}, \{c_1, \dots, c_k\}\}$. This is the gadget representing the gate that duplicates its input k times.

A drawing of a circuit in the plane becomes a planar network of gate gadgets. A crossing pair of wires is translated to the *crossover gate*, which has two inputs and two outputs which match the inputs in the opposite order. As a gadget, the crossover gate has ports $\{a_1, b_1, a_2, b_2\}$, importantly in that cyclic order. It has constraint

$$\{\{a_1, b_1\}, \{a_1, b_2\}, \{a_2, b_1\}, \{a_2, b_2\}\}.$$

One can think of it as two wires $a_1 \to a_2$ and $b_1 \to b_2$ that cross in a circuit. However, the directions of these wires are arbitrary because the gadget is highly symmetric. As a gadget outside the perspective of networks of logic gates, the crossover gate is two edges that can independently be assigned orientations, and which cross each other. So we will also call it simply the *crossover*.

5.3.2 Decision problems

We consider three decision problems about planar graph orientation, corresponding to the Minesweeper decision problems we're interested in. Each of them is parameterized by a set of gadgets S—throughout we consider only finite sets of gadgets—and takes as input a planar network N of gadgets from S.

The problem corresponding to Minesweeper consistency is straightforward.

Problem 5.11. PGO *consistency* with S asks whether N has a satisfying assignment.

For Minesweeper inference, we need to avoid a subtle issue, which is the error in prior claims of coNP-hardness [SSvR11, TB22]. If there is a gadget for which we can deduce that some legal combination of edge orientations can't be extended to a full satisfying assignment, this information may allow us to infer the value of a Minesweeper cell internal to the gadget. This can happen even if we can't deduce the orientation of any particular edge, so the most obvious PGO inference problem fails to reduce to Minesweeper, and thus isn't useful.

Our strategy for resolving this issue will ultimately be to "click" on all cells in the Minesweeper instance that could be deduced in this way. To make this work, we will need the values of those cells to not reveal any additional information (a property we will call "silence"), and we need the reduction to find all such cells in polynomial time. For our gadget framework to help here, we need to define the inference problem carefully, and in particular it needs to be aware of which combinations of edges are ruled out by "semilocal" deductions.

Problem 5.12. PGO *promise-inference* with S is given a network N as well as, for each vertex in N, a *semilocal constraint* which is a subset of its constraint. We say an edge e is *locally forced* if the semilocal constraint of one of its vertices requires it has a particular orientation (because it's either in every element or in no element), and *locally free* otherwise.

We are promised that

- in every satisfying assignment, the set of edges pointing into a vertex is in its semilocal constraint; and
- either
  - there is a locally free edge to which every satisfying assignment assigns the same direction; or
  - for each vertex v and set of edges in its semilocal constraint, there's a satisfying assignment that makes exactly those edges point towards v.

We are asked to determine whether there is an edge whose orientation can be inferred but isn't immediate from semilocal constraints; that is, whether we are in the first option above.

Note that the two options can't simultaneously be true, since the existence of such an edge would imply that its incident vertices' semilocal constraints have elements that can't be extended to satisfying assignments.

The intent of semilocal constraints is to be between the levels of individual gadgets' "local" constraints and "global" deductions that require understanding the entire network. A semilocal constraint typically contains the sets of in-pointing edges that are attainable when looking at some constant-size neighborhood of a gadget.

The first part of the promise ensures that semilocal constraints are actually enforced by the structure of the network. In the case where there's no inferable edge, the second part says that it isn't possible to deduce more about the immediate neighborhood of a gadget than its semilocal constraint.

Finally, the PGO decision problem analogous to Minesweeper solvability is simpler.

Problem 5.13. PGO *uniqueness* with S is given N as well as a satisfying assignment of N, and asks whether it is the unique satisfying assignment.

PGO uniqueness is related to Minesweeper solvability because it's possible to deduce the orientations of all edges if and only if there's a unique satisfying assignment.

Lemma 5.14. *For any set S of gadgets,*

1. *PGO consistency with S is in NP.*
2. *PGO promise-inference with S is in promise-coNP.*
3. *PGO uniqueness with S is in coNP.*

Proof. Each problem has a straightforward certificate:

1. A satisfying assignment serves as a certificate of consistency.
2. A certificate that there is no inference consists of, for each locally free edge and each orientation, a satisfying assignment that assigns the orientation to the edge.
3. A second satisfying assignment serves as a certificate of nonuniqueness.

□

5.3.3 Hardness

We now prove hardness of each PGO decision problem for the appropriate class, with a specific set of gadgets. The gadget sets are chosen to make the hardness proofs simple; there are easy ways to reduce the number of different gadgets needed, and we will further simplify our gadget sets using simulation in Section 5.3.5. With Lemma 5.14, we have completeness in each case.

Theorem 5.15. *PGO satisfiability with free terminals, fanout gates, and 1-in-3 gadgets is NP-hard.*

Some of these gadgets are new: the *1-in-3 gadget* has three ports, and its constraint says that exactly one of them must point in. The *free terminal* has one port, which is allowed to point in either direction.

Proof. We reduce from planar positive 1-in-3SAT, which is NP-hard [Lar93]. Each variable with k occurrences becomes a free terminal that leads to a k-way fanout gate. Each clause becomes a 1-in-3 gadget. We connect the outputs of the fanout gates to 1-in-3 gadgets as in the 1-in-3SAT formula, which is planar. Satisfying assignments of this network correspond to satisfying assignments of the formula. □

Theorem 5.16. *PGO promise-inference with free terminals, fanout gates, crossovers, OR gates, and AND gates is coNP-hard.*

Proof. We reduce from the complement of monotone 3SAT. Monotone 3SAT is 3SAT with the additional requirement that each clause contains either only positive literals or only negative literals, and is NP-hard [Gol78]. The layout of gadgets is depicted in Fig. 5.7. The green-outlined region is an example of a variable, and the red-outlined region is an example of a clause. We will place all positive clauses on the top half of the network, and all negative clauses on the bottom half. Before constructing the circuit, we remove duplicate literals within a clause, and we remove any redundant clause, meaning one that has a superset of the literals of another clause. The semilocal constraint of each gadget is its full constraint set.

The output of the formula is given by r, which is connected to a free terminal. If the formula is unsatisfiable, there is an inference; namely, r is false (points towards the AND gate). So we just need to show that if the formula is satisfiable, there is no inference; that is, for each gadget in the reduction, every set of edges in its constraint is possible to achieve (pointing in) in some consistent assignment. Because each gadget is a gate, this is equivalent to every combination of input values being attainable.

There is a unique consistent assignment for each choice of values for variables, z_1, and z_2. We will describe such a choice that achieves each combination of input values for each gadget.

Each fanout depends on a single variable, which can have either value. Each crossover is between two wires connected to different variables, so all combinations are possible. The inputs of each OR gate in a clause can be chosen by setting the input variables as desired.

Consider any of the AND gates combining the outputs of the clauses on one half of the circuit, and assume for simplicity that it's on the positive half. We can make both

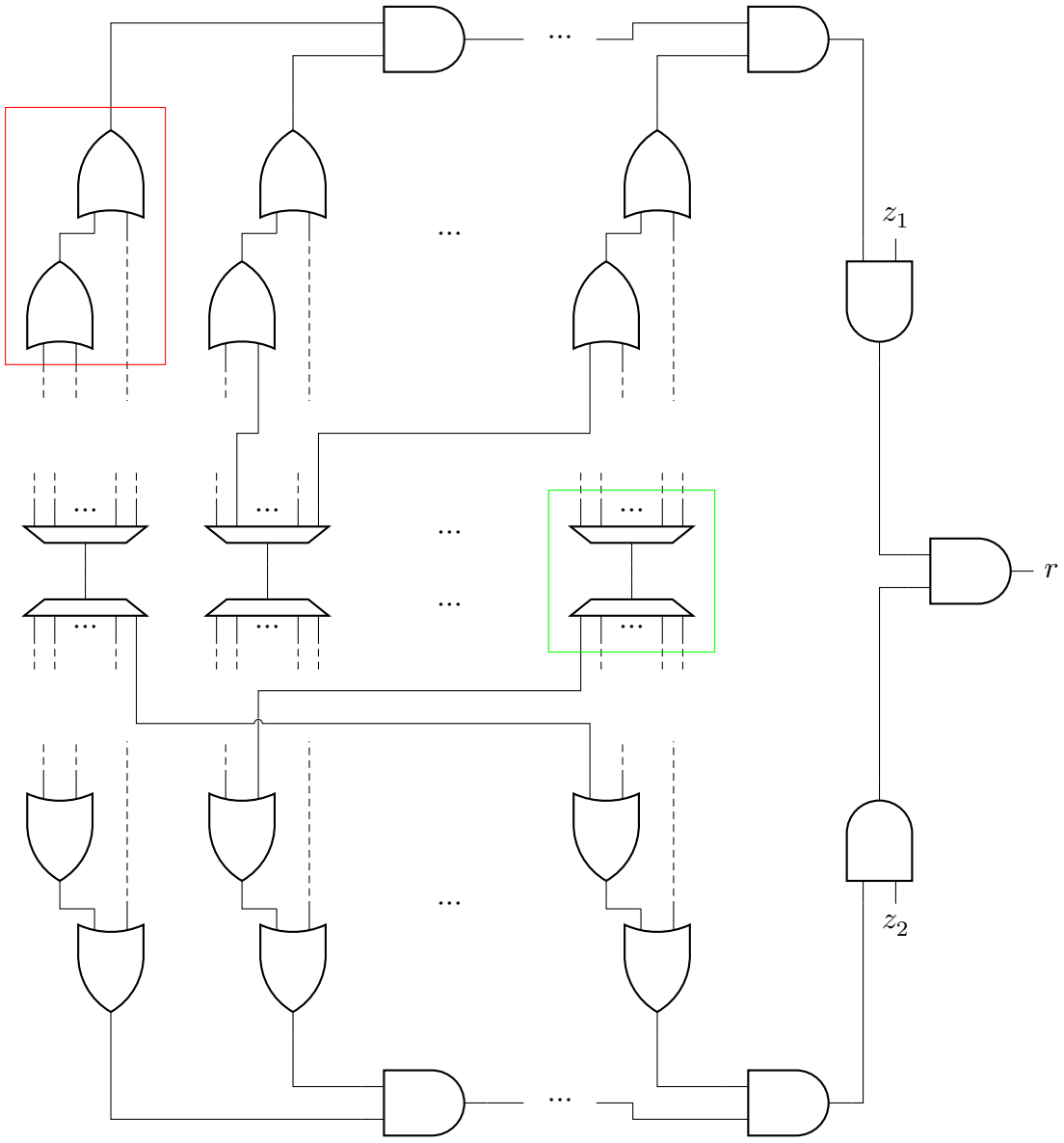


Figure 5.7: Our reduction for coNP-hardness of PGO promise-inference. See Table 5.1 for gadget notation.

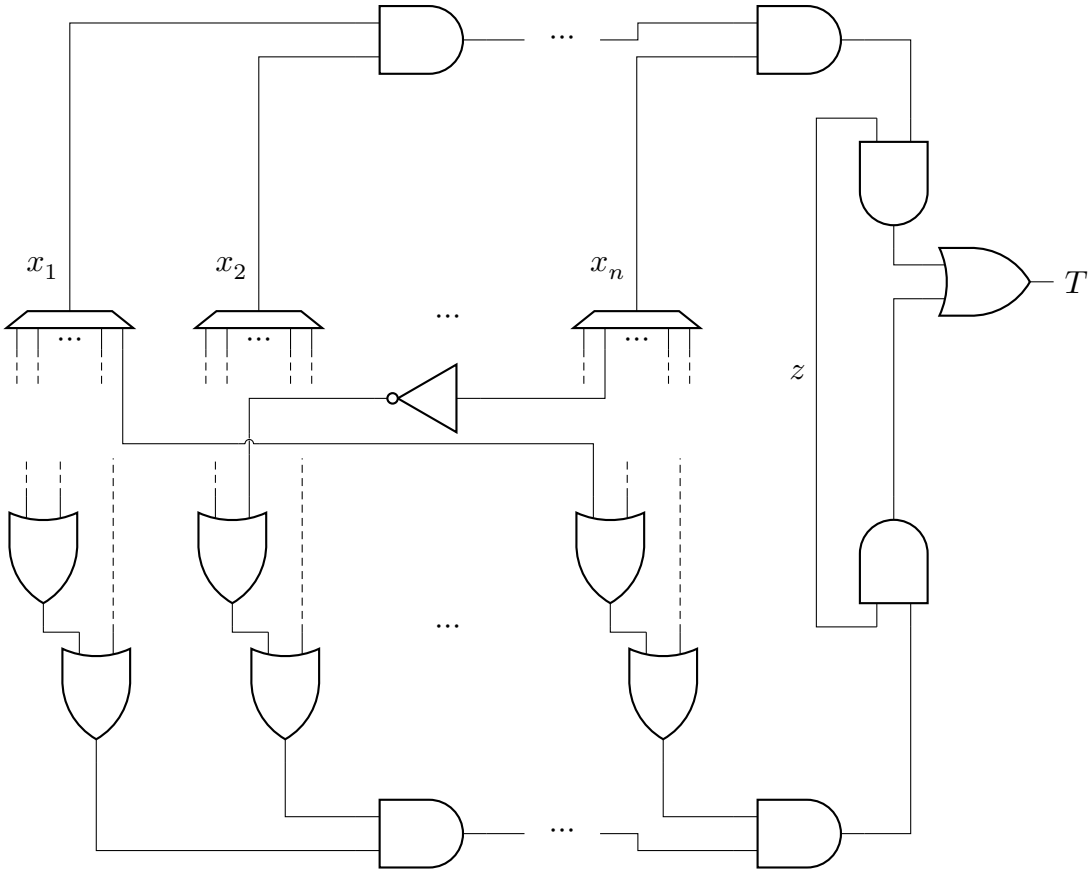


Figure 5.8: Our reduction for coNP-hardness of PGO uniqueness. See Table 5.1 for gadget notation.

inputs true (pointing in) or both inputs false (pointing out) by setting all variables true or all variables false, respectively. To make exactly one input true, note that we can choose a single (positive) clause to be unsatisfied: make the variables in that clause false and all other variables true. Then the clause is unsatisfied, but, since there are no redundant clauses every other positive clause is satisfied. By choosing a clause that feeds into either side of the AND gate in question, we can make either of its inputs false and the other true. The same argument shows that the AND gates with z_1 and z_2 can also have any combination of inputs.

Finally, consider the rightmost AND gate, with output r. By hypothesis, the formula is satisfiable, so both inputs can be made true by satisfying the formula and setting z_1 and z_2 to both be true. We can then independently choose to make either input false by flipping z_1 or z_2 as appropriate. □

Theorem 5.17. *PGO uniqueness with free terminals, fanout gates, crossovers, NOT gates, OR gates, AND gates, and fixed terminals is coNP-hard.*

The *fixed terminal* has one port, and its constraint forces the port's value. There are two kinds of fixed terminal; we will use the *fixed true terminal*, which forces the edge to point in.

Proof. We reduce from the complement of 3SAT, which is NP-hard [Kar72]. Refer to Fig. 5.8.

Each variable becomes an edge connected to a fanout gate pointing down: that edge pointing down represents the variable being true. Each clause becomes a pair of OR gates connected in the natural way. We place edges connecting variables to clauses in the structure of the formula. If edges cross, we use a crossover gadget. For negated literals, we place a NOT gate along the edge.

The outputs of the clauses are merged with AND gates, so the edge in the bottom right of Fig. 5.8 points right (and up) exactly when the assignment (based on the orientations of edges representing variables) is satisfying.

In the top section of Fig. 5.8, we merge the other ends of the variable edges with AND gates. The top right edge points right (and down) exactly when all variables are false.

On the right, there is an edge z which points towards one of two AND gates, with the results merged by an OR gate and then run into a fixed true terminal. For the output of that OR gate to be true (point right), either

- z points up, and all variables are false; or
- z points down, and the 3SAT formula is satisfied.

Note that the orientations of all edges are uniquely determined by those of z and variables, even ignoring the fixed terminal. In particular, the network has exactly one satisfying assignment (with z pointing down) for each satisfying assignment of the 3SAT formula, plus exactly one more, which has z pointing up and all variable edges pointing up.

The input to PGO uniqueness is the network described above and the satisfying assignment with z pointing up. This is the unique satisfying assignment exactly when the 3SAT formula is not satisfiable. □

5.3.4 Simulation

A key feature of our framework is that it allows us to abstractly construct gadgets out of other gadgets, greatly reducing the complexity of the gadgets we need to actually implement in Minesweeper. In particular, the results in Section 5.3.3 use many gadgets, some of which are hard to build directly, particularly with the properties our hardness proofs require.

Definition 5.18. A *simulation* using gadgets from S is a network of gadgets from S, except it may have some 'dangling' edges incident to only one vertex. Equivalently, the graph contains one special vertex called the 'outside world' (which has the trivial constraint).

For *planar* simulations, we require that the dangling edges are in the external face, or equivalently that the graph including the outside world is planar.

See Section 5.3.5 for several examples of simulations.

Definition 5.19. Given a simulation, the *simulated gadget* has dangling edges as ports, and its constraint contains each set of dangling edges for which there is a satisfying assignment making exactly those edges point into the simulation (away from the outside world).

In the planar case, the order of the ports is the order of the dangling edges around the simulation, or the reverse of their order around the outside world.

Definition 5.20. We say S *simulates* G if there is a simulation using gadgets from S where the simulated gadget is G. For a set T, we say that S *simulates* T if S simulates each gadget in T.

For PGO uniqueness and Minesweeper solvability, we will need our simulations to be even better behaved.

Definition 5.21. A simulation of G is *parsimonious* if for each legal configuration of the edges of G, there is exactly one satisfying assignment of the simulation that orients the dangling edges that way.

We say that S *parsimoniously* simulates G if the relevant simulation is parsimonious, and S *parsimoniously* simulates T if S parsimoniously simulates each element of T.

Much of the point of having a theory of simulations is that they can be composed. This reduces conceptual complexity by letting us break down complicated simulations into a sequence of simpler ones.

Lemma 5.22. *If S simulates T and T simulates G, then S simulates G. Moreover, this composition preserves parsimony.*

Proof. In the simulation of G using gadgets from T, replace each gadget with its simulation using gadgets from S. In any satisfying assignment of the new simulation, each component simulation is consistent with the gadget it's simulating, so we can construct a satisfying assignment of the original simulation with the same orientations for dangling edges by looking at only the edges between simulated gadgets.

Conversely, any satisfying assignment of the original simulation can be extended to one of the new simulation by filling in each simulated gadget with an appropriate satisfying assignment. Thus we have a simulation of G using gadgets from S.

If each gadget is replaced with a parsimonious simulation, there is only one way to fill in simulated gadgets this way. So we have defined a bijection between satisfying assignments of the original and new simulations of G. If the original is parsimonious, so is the new simulation. □

The other half of the point of simulations is to simplify hardness proofs, so we need them to preserve hardness. This is straightforward for satisfiability.

Lemma 5.23. *If S simulates T, then there is a polynomial-time reduction from PGO satisfiability with T to PGO satisfiability with S.*

Proof. Given a network N of gadgets from T, replace each gadget with a (constant-size) simulation using gadgets from S to construct a network N' of gadgets from S.

If there is a satisfying assignment of N', looking only at the edges connecting simulated gadgets gives a satisfying assignment of N.

If there is a satisfying assignment of N, we can set the edges connecting simulated gadgets to match it, and then each simulated gadget has a local solution compatible with those edges. □

This is somewhat more complicated for promise-inference, but it's not so bad with the right definition for the decision problem.

Lemma 5.24. *If S simulates T, then there is a polynomial-time reduction from PGO promise-inference with T to PGO promise-inference with S*

Proof. We are given an instance of promise-inference with T, which is a network N of gadgets from T and semilocal constraints. Construct a network N' of gadgets from S in the same way as above.

To complete the instance of PGO promise-inference with S, we must define semilocal constraints for the vertices in N'. Consider a vertex v, which is inside a simulation of $G \in T$. The semilocal constraint of v shall contain the sets the of edges that point into v in satisfying assignments of the simulation that are compatible with the semilocal constraint of G (as a vertex of N). These semilocal constraints can be computed in polynomial time because each one requires considering only a simulation, and simulations have constant size.

It remains to check that this instance satisfies the promise, and falls into the same option as the input instance. For the first part of the promise, consider a vertex v in N' and a satisfying assignment. Then v is inside a simulation of some $G \in T$, and the corresponding assignment of N (which has only edges between simulated gadgets) is compatible with the semilocal constraint of G. So we have a satisfying assignment of the simulation of G compatible with the semilocal constraint of G, and thus by construction it is compatible with the semilocal constraint of v.

Suppose N has a locally free edge whose orientation is forced. Since every satisfying assignment of N' contains a satisfying assignment of N in the inter-simulation edges, the corresponding edge of N' also has forced orientation. It must also be locally unforced: each orientation is compatible with the semilocal constraints of its vertices in N, and therefore each orientation is compatible with some appropriate satisfying assignment of the simulations of its vertices. Hence N' also has a locally free edge with forced orientation.

Now suppose N falls into the second option, namely there are satisfying assignments achieving every element of its semilocal constraints. Consider a vertex v in N' which is inside a simulation of G, and consider a set L in the semilocal constraint of v. By definition, there is a satisfying assignment of the simulation which makes exactly the edges in L point towards v, and which is compatible with the semilocal constraint of G as a vertex of N. By assumption, there is a satisfying assignment of N which orients the dangling edges of the simulation in the same way. Combining these, and filling in the assignment for other simulations, we obtain a satisfying assignment of N' which directs exactly the edges in L towards v, as desired. □

To prove an analogous result for PGO uniqueness, we need our simulations to preserve unique solutions. That is, they should be parsimonious.

Lemma 5.25. *If S parsimoniously simulates T, then there is a polynomial-time reduction from PGO uniqueness with T to PGO uniqueness with S*

Proof. We are given a network N of gadgets from T and a satisfying assignment A. Construct a network N' of gadgets from S in the same way as the two previous proofs—replace each gadget with its simulation.

Our instance of PGO uniqueness with S also needs a satisfying assignment of N'. To construct one, direct inter-simulation edges to match A, and extend this to a full satisfying

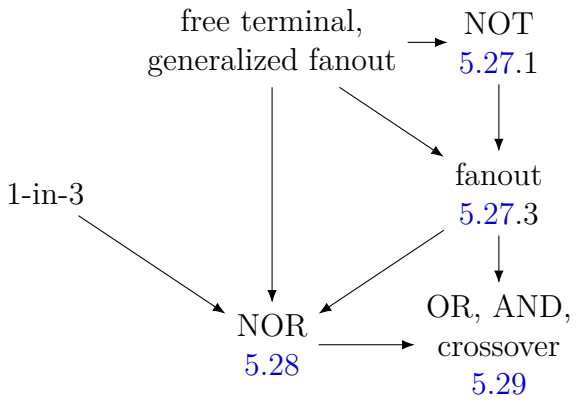


Figure 5.9: The (parsimonious) simulations we use to simplify our gadget set. Each gadget is simulated by the collection of gadgets pointing towards it, except that we will need to build 1-in-3 gadgets, free terminals, and any generalized fanout directly.

assignment A' by consistently orienting edges inside simulations. Since the simulations are parsimonious, there is a unique way way to do this for each simulation, and since simulations are constant-size, A' can be computed in polynomial time.

As before, there is a correspondence between satisfying assignments of N and satisfying assignments of N'. This time, however, parsimony ensures that the correspondence is a bijection, as illustrated by the well-definedness of A'. In particular, there is a satisfying assignment of N other than A if and only if there is a satisfying assignment of N' other than A'. □

5.3.5 Simpler gadget sets

Now we put the theory of simulations into practice: we will demonstrate several simulations and use them with the results of Section 5.3.4 to improve the results of Section 5.3.3 to use more convenient sets of gadgets. The simulations we use are summarized in Fig. 5.9.

When we build gadgets in variants of Minesweeper, we will construct gadgets that are like the fanout gate, but don't have a clearly designated input port, and may have some ports inverted.

Definition 5.26. A *generalized fanout* is a gadget with at least three ports whose constraint has exactly two sets, which are complements. That is, it has two legal configurations, which differ by flipping all edges.

For instance, the fanout gate is a generalized fanout, and the NOT gate is almost a generalized fanout, except it has only two ports.

Lemma 5.27. *Let F be a generalized fanout. Then F and free terminals parsimoniously simulate*

1. *the NOT gate*
2. *any generalized fanout F'*

3. *fanout gates (with any number of outputs).*

Proof. We describe each simulation.

1. Let F have ports P and constraint $\{T, F\}$, where $T \sqcup F = P$. Since $|P| \geq 3$, at least one of T and F has size at least 2; assume without loss of generality that $|T| \geq 2$, and $a, b \in T$. Attach free terminals to all ports of F other than a and b. This simulation has two satisfying assignments, corresponding to T and F. The simulated gadget has ports a and b, and constraint $\{\{a, b\}, \emptyset\}$, so it is the NOT gate.

2. Connect copies of F in a tree until there are at least as many dangling edges as ports of F'. Note that there are exactly two satisfying assignments; all copies of F must flip state together. Now assign each port of F' to a dangling edge, respecting cyclic order. If there are extra dangling edges, put free terminals on them.

   The result is a generalized fanout with the same ports as F', but the partition into the two legal configurations may be wrong. For each port on the wrong side of the partition, attach a NOT gate (composing simulations using Lemma 5.22). This changes which of the two satisfying assignments has the edge at that port pointing in. Now the simulated gadget partition its ports into two legal configurations in the same way as F', so it is in fact F'.

3. This is a special case of the above.

□

Lemma 5.28. *1-in-3 gates, free terminals, and fanouts parsimoniously simulate the NOR gate.*

Proof. The simulation consists of three 1-in-3 gadgets, four free terminals, two 2-way fanouts, and a 3-way fanout, and is shown in Fig. 5.10 (the fanouts could be simplified a bit; we draw it this way for clarity). The inputs are a and b and the output is c. True means 'pointing right', for inputs, outputs, and labeled free terminals.

The simulation enforces a 1-in-3 constraint on each of $\{a, x, c\}$, $\{b, y, c\}$, and $\{x, y, z\}$. The easiest way to verify that this is a parsimonious simulation of a NOR gate is to list all satisfying assignments (recall that 'true' means pointing right):

a	b	x	y	z	c
T	T	F	F	T	F
T	F	F	T	F	F
F	T	T	F	F	F
F	F	F	F	T	T

We see that c is always $\neg(a \vee b)$, and there is a unique assignment for each combination of values for a and b. When comparing against the definition of the NOR gate in Table 5.1, recall that the correspondence between true/false and in/out is reversed for c relative to a and b.

□

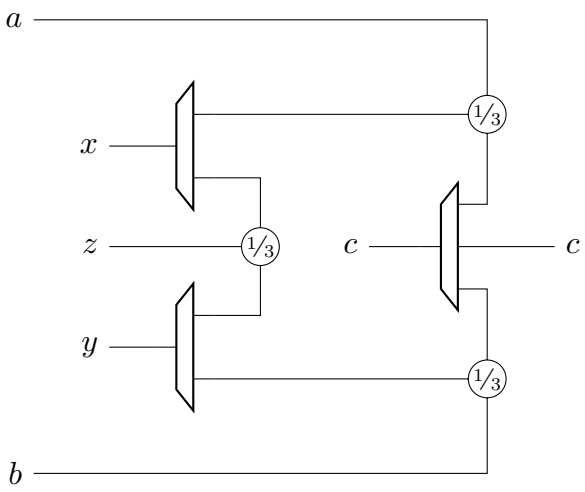


Figure 5.10: A simulation of a NOR gate using 1-in-3 gadgets, free terminals, and fanouts.

Lemma 5.29. *NOR gates and fanout gates parsimoniously simulate OR gates, AND gates, and crossovers.*

Proof. In Section 5.3.1 we observed that boolean circuits can be converted to gadget networks; these networks can be thought of as parsimonious simulations. The NOR operation is logically complete and can build crossovers in planar circuits [Gol77]. Hence for each gate we want to simulate, we can find a planar circuit of NOR gates that computes it, then convert this into a planar network of NOR-gate gadgets and fanout-gate gadgets. □

We now pull everything together by applying the results on simulation in Section 5.3.4 to the hardness results of Section 5.3.3, using the simulations in this section. We can either compose reductions or compose simulations using Lemma 5.22; either way, we obtain our main result about planar graph orientation decision problems.

Corollary 5.30. *PGO consistency with any generalized fanout, fixed terminals, free terminals, and 1-in-3 gadgets is NP-hard. PGO promise-inference and PGO uniqueness with the same set of gadgets are coNP-hard.*

5.3.6 Applying the framework

We now conclude the development of our gadget framework by discussing what it takes to use it to prove hardness for a game like Minesweeper. We do not attempt to precisely define "like Minesweeper", and this results of this section are not stated precisely. Ultimately, one needs a polynomial-time reduction from the appropriate PGO problem to the corresponding question for the game under consideration, but the games we will consider are similar enough that the reductions share most of their structure, and the discussion here applies to many or all of them.

It is conceptually helpful to distinguish between gadgets as abstract formal objects and the constructions we build in concrete games to embed gadgets in them. We call the latter 'implementations', to parallel the distinction between the specification and the implementation of a function.

Definition 5.31. An *implementation* of a gadget is a local construction in a game like Minesweeper which behaves like the gadget, in that there are covered cells (usually on the boundary of the construction) corresponding to ports, and the construction has a solution exactly when the configuration of edges represented by the values of those cells is legal for the gadget.

Unfortunately, because it matches how the term is typically used, we will frequently call implementations "gadgets" when the intended meaning is clear from context.

For implementations to interact in a way that simulates gadget networks, we need some kind of *wire*. Each of our wires will have two possible states, corresponding to the two orientations of the edge it represents. We must be able to route our wires in an arbitrary planar graph—the ability to extend and turn them is sufficient. We need to be able to plug wires into the ports of implementations, which sometimes requires the ability to adjust the alignment of a wire by a single cell.

Our wires and implementations also need to have the following properties.

- Every solution uses the same number of mines. This means the total number of mines in the puzzle doesn't reveal any information that might break our reduction.
- Every uncovered cell has a clue, as in most implementations of Minesweeper. There are versions of Minesweeper which allow cells that are known to be safe but don't have additional information (e.g. showing a question mark instead of a number), but we don't want to rely on this feature.
- All given mines can be deduced from uncovered non-mine cells. This provides robustness against changes in the definition, e.g. whether the locations of mines can be provided as part of a puzzle in addition to uncovered cells without mines.

Implementing some gadgets in a game allows us to build a network of those gadgets in the game. The resulting instance of the game has a solution if and only if the network is satisfiable.

Claim 5.32. *If a game like Minesweeper has implementations of the gadgets in S, then there is a polynomial-time reduction from PGO consistency with S to the game's consistency problem.*

Note that our definition of consistency in Section 5.2 is specific to Minesweeper. For other games or variants, 'partial board' and 'consistent' need to be defined appropriately, but the definitions that depend on these (for inference and solvability) work as is.

For the other two decision problems, we need well-behaved implementations.

Definition 5.33. An implementation is *silent* if clicking a known-safe internal cell can never reveal information that wasn't already known. In other words, for each covered internal cell, the information that cell provides when clicked (e.g. the number of adjacent mines) is the same in every solution in which it doesn't have a mine.

Claim 5.34. *If a game like Minesweeper has silent implementations of the gadgets in S, then there is a polynomial-time reduction from PGO promise-inference with S to the game's inference problem.*

Proof. Embed a network of gadgets from an instance of PGO promise-inference with S in the game. For each (constant size) gadget in the network, consider all solutions to its implementation which are consistent with its semilocal constraint. If there are *commonalities*, or cells which are safe in all such solutions (or mines in all such solutions), "click on" them, revealing them in the instance of the game. Silence guarantees that the information revealed is the same in all such solutions, so the information to put on that cell can be determined from the semilocal constraint. If a commonality dictates the orientation of an edge, we click on all covered cells in the edge in the same way. It doesn't matter whether this includes the cell(s) representing the port on the other end of the edge—if that gadget's semilocal constraint doesn't force the edge in the same way, we must be in the first case of the promise so there's an inferable locally unforced edge anyway.

If the network has an inferable locally unforced edge, then cells inside the wire representing that edge (or cells representing the ports that edge connects to) can be inferred.

Otherwise, we are in the second case of the promise. We've already clicked all cells that can be deduced from each semilocal constraint—for any cell that is still covered, the gadget (or wire) it's in has solutions compatible with its semilocal constraint in which that cell has a mine and in which it doesn't. Thus the entire instance also has both of these kinds of solutions because we are promised that every element of a semilocal constraint is achieved by a satisfying assignment. That is, the cell in question can't be inferred. This is where it is crucial that we use promise-inference, and not a simpler PGO inference decision problem. □

For PGO uniqueness and Minesweeper solvability, we also need parsimony as we did for Lemma 5.25.

Definition 5.35. An implementation is *parsimonious* if it has exactly one solution corresponding to each legal configuration of the gadget.

Claim 5.36. *If a game like Minesweeper has silent parsimonious implementations of the gadgets in S, then there is a polynomial-time reduction from PGO uniqueness with S to the game's solvability problem.*

Proof. Embed a network N of gadgets from an instance of PGO uniqueness with S in the game. Thanks to parsimony, satisfying assignments of the network are in bijection with consistent solutions to this instance of the game. The secret solution is the solution corresponding to the given satisfying assignment of N, but cells are only uncovered if they are uncovered in the embedding of N, which doesn't depend on the satisfying assignment.

If N has another satisfying assignment, the game instance has multiple solutions. The player may be able to safely click some cells—perhaps the orientation of some edge is forced, or they can deduce that a cell inside a gadget is safe. However, since our implementations are silent, the player can't gain any information by doing this. In particular, all solutions consistent with the initial state of the game are also consistent after the player makes any sequence of safe clicks. In order to solve the instance, they would need to distinguish between these consistent solutions, which is impossible.

Conversely, suppose the only satisfying assignment to N is the one given to the reduction. Then the secret solution is the only solution consistent with the initial state of the game (we need parsimony to ensure there is only one). Thus the value of every cell can be deduced, and all cells without mines can be safely clicked in any order. □

It follows that if we can silently and parsimoniously implement the gadgets needed by Corollary 5.30, we have hardness of consistency, inference, and solvability for the game in question. To simplify a little further, fixed terminals are trivial to construct: just let a wire end, and reveal some cells in it to force its orientation. Alternatively, modify a free terminal (possibly including a bit of wire) by revealing cells that determine its configuration.

Corollary 5.37. *Suppose that for some game like Minesweeper, we have silent parsimonious implementations of any generalized fanout, free terminals, and 1-in-3 gadgets, and we are able to route, turn, and adjust the alignment of wires enough to embed any planar network of those gadgets. Then consistency is NP-hard, and inference and solvability are coNP-hard.*

We now go through one final layer of abstraction, which will handle routing of wires and filling empty space. We lay a network of the gadgets above on a square grid, where edges run horizontally and vertically and can turn. In particular, each tile contains either one of the gadgets above, a straight edge section, a turning edge section, or nothing. Tiles have up to one port on each edge, usually at the center.

This lets us reduce from problems about planar networks on grids, so for wire routing we need only construct tiles with straight and turning wires.

It turns out we don't need turning wires, and can instead turn using any generalized fanout and the other tiles. As a first attempt, pick two ports on adjacent sides of the generalized fanout and cover its other ports with free terminals. This sometimes makes a turning wire, but if the chosen ports have the same 'polarity', meaning they point in or out together, it instead makes a turning NOT gate. If we can only build turning NOT gates this way, all ports of the generalized fanout must have the same polarity, so its legal configurations are all-in and all-out. But the generalized fanout must have two ports on opposite sides, so we can make a straight NOT gate by covering the other ports with free terminals. A turning NOT gate and a straight NOT gate in series gives a turning wire.

Finally, for a few of our applications, instead of building a 1-in-3 gadget we build a 2-in-3, 1-in-4, or 3-in-4. With NOT gates and fixed terminals, each of these can easily simulate a 1-in-3.

Corollary 5.38. *Suppose that for some game like Minesweeper, we have silent parsimonious implementations of gadgets which are all square tiles that fit in a grid and interact appropriately with adjacent tiles. Suppose in particular that we have empty tiles, straight wires, any generalized fanout, free terminals, and any one of 1-in-3, 2-in-3, 1-in-4, or 3-in-4 gadgets. Then consistency is NP-hard, and inference and solvability are coNP-hard.*

5.4 Hardness proofs

To demonstrate the utility of this PGO framework and the ease with which it facilitates writing hardness proofs, we provide a number of example applications that prove hardness of Minesweeper and Minesweeper-like games. In particular, our examples are drawn from the video game *14 Minesweeper Variants*, which despite its name actually implements significantly more than 14 Minesweeper variants.

The variants in the game are identified by letters surrounded with brackets. For instance, [V] stands for *vanilla* Minesweeper with no variants. We first briefly summarize the rules of each variant in the game.

The rules are separated into two categories. The first category, which we refer to as "left rules" (because they appear on the left half of the menu), contains rules that add global constraints to the board without changing the meanings of clues:

[Q] **Quad**:	there must be at least 1 mine in every 2×2 region
[C] **Connected**:	all mines are orthogonally or diagonally connected
[T] **Triplet**:	there can be no 3 mines in a row (orthogonally or diagonally)
[O] **Outside**:	all non-mines are orthogonally connected; all mines are orthogonally connected to any edge of the grid
[D] **Dual**:	all mines form non-orthogonally-touching 1×2 dominoes
[S] **Snake**:	all mines form an orthogonally connected path which does not touch itself orthogonally
[B] **Balance**:	the number of mines in each row and column is the same
[T'] **Triplet'**:	mines must be part of a 3-in-a-row orthogonally or diagonally
[D'] **Battleship**:	all mines form non-touching (even diagonally) 1×1's, 1×2's, 1×3's, or 1×4's
[A] **Anti-knight**:	no two mines can be a knight's move away from each other
[H] **Horizontal**:	no two mines can touch horizontally
[U] **Unary**:	no two mines can touch orthogonally

The second category, which we refer to as "right rules," contains variants that apply to the clues themselves (which by default count the number of mines in the 8 orthogonally or diagonally adjacent cells):

[M] **Multiple**:	the grid is checkerboard colored; each mine in a shaded cell counts as 2
[L] **Liar**:	each clue is off by exactly 1
[W] **Wall**:	clues give the multiset of lengths of contiguous runs of mines in the 8 orthogonally or diagonally adjacent cells
[N] **Negation**:	the grid is checkerboard colored; clues give the difference between the number of mines in shaded and unshaded cells
[X] **Cross**:	clues give the number of mines up to 2 away in all 4 orthogonal directions
[P] **Partition**:	clues give the number of groups of mines (i.e. number of [W] clues there would be)
[E] **Eyesight**:	clues give the number of non-mines seen in all 4 orthogonal directions (including the clue itself), where mines block line of sight
[X'] **Mini-cross**:	clues give the number of mines 1 away in all 4 orthogonal directions
[K] **Knight**:	clues give the number of mines a knight's move away
[W'] **Longest wall**:	clues give the largest length of contiguous mines (i.e. the largest [W] clue there would be)
[E'] **Eyesight'**:	clues give the difference of non-mines seen horizontally and vertically, where mines block line of sight

In most cases, our reductions will "incidentally" satisfy some number of rules, in that the reduction is unaffected or can be easily adapted to work whether the rule is included or not. This is very efficient, as a reduction that incidentally satisfies n extra rules constitutes a hardness proof for 2^n different variants.

The notation used in the below gadgets is as follows:

- Yellow-highlighted cells represent wires. It is enforced by the given clues that e.g. for the wire A, either all cells marked A are mines and all cells marked $\overline{\mathsf{A}}$ are empty, or all cells marked A are empty and all cells marked $\overline{\mathsf{A}}$ are mines.
- Black cells represent mines which can be locally derived from the clues contained within the same gadget (or in some cases the connecting regions between gadgets). Usually, it is immediately obvious how to derive these (they come directly from a single clue); mines that require deductions that are any more complicated than that are marked with red stars.
- White cells represent pre-revealed cells, which contain the appropriate type of clue. In most cases, the appropriate number is clear from the surrounding cells; a few numbers are given explicitly for clarity.

When two of our gadgets meet, there will be a pair of mines, with one in each gadget, which interact in that exactly one of them must be a mine. We think of the edge between them as pointing towards the mine. For most versions, tiles are glued together as depicted in Fig. 5.11.

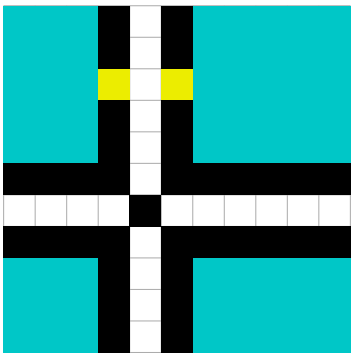
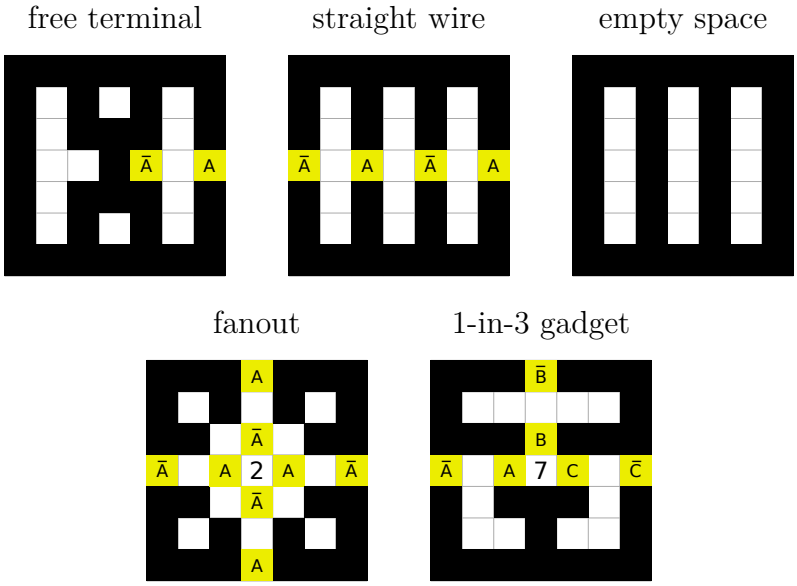


Figure 5.11: Relative positioning of tile-based gadgets. Cyan represents the interior of each gadget, which differs between gadgets, and everything else shows the common "frame." The extra mine outside the frames is placed to satisfy [C] constraints. Details differ between variants (e.g. dimensions of each tile), but the structure is the same.

All of our gadgets satisfy the properties listed in Section 5.3.6, so we have hardness even when the number of mines is known, when all uncovered cells have clues, and when the input can't contain given mines.

5.4.1 Vanilla clues

These gadgets work with vanilla clues together with any combination of [M], [L], [Q], [C], and [T'].



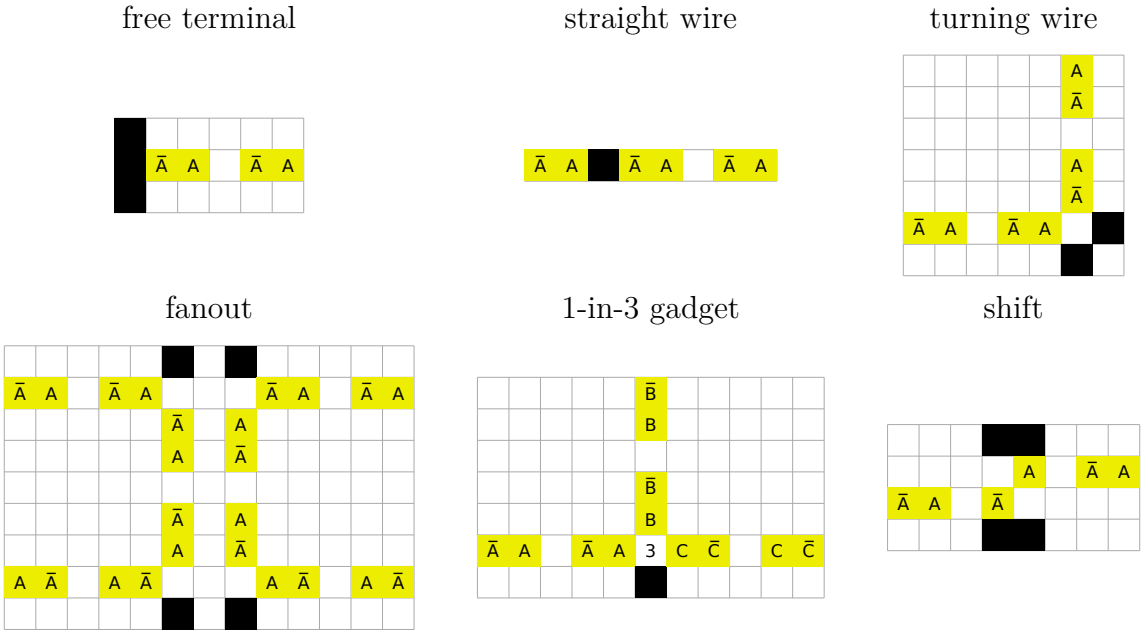
5.4.2 Solvability from an empty board

In the original Minesweeper game, the player does not start with any cells revealed, and the player starts by clicking any cell on the entirely unrevealed board (which the game guarantees

to be safe). The above gadgets do not work in this setting, because they rely on a large number of clues being pre-revealed. However, we also construct the following "transparent" gadgets, where all clues can be deduced starting from this initial state. Away from wires and gadgets, there are no mines—each face of the network graph mostly consists of a large field of 0s.

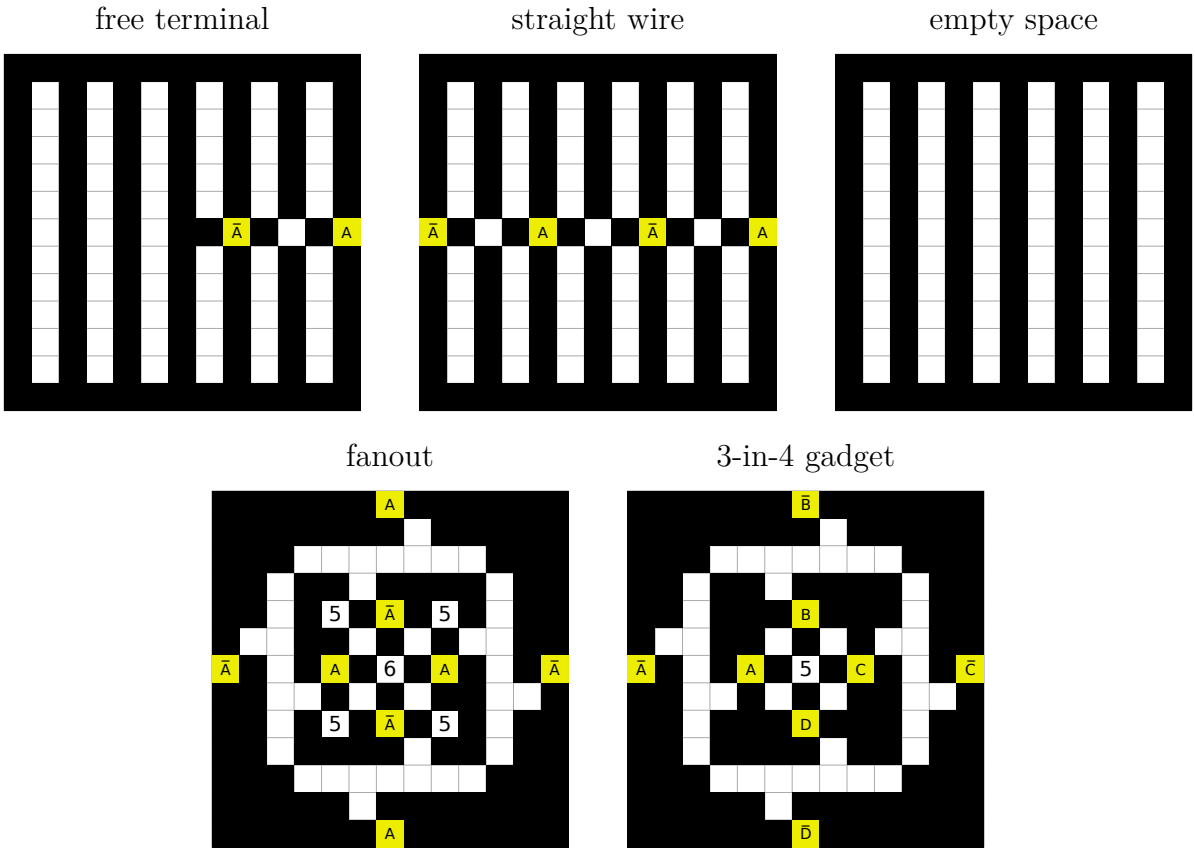
We place a mine in each wire as shown, which ensures that the player can 'cross' the wire: if they have revealed cells on one side of the wire, they can deduce that mine and the empty cell three tiles away, which then allows them to reveal cells on the other side of the wire, including the 0s in that face. Once each face is uncovered, the player can deduce all the other white cells inside gadgets.

This shows that Minesweeper solvability is coNP-complete even in the setting where the player is initially given no clues. (To guarantee the first click is safe, we can double the width of the grid, and delay the decision of which half to place the circuit on until after the click is made.)



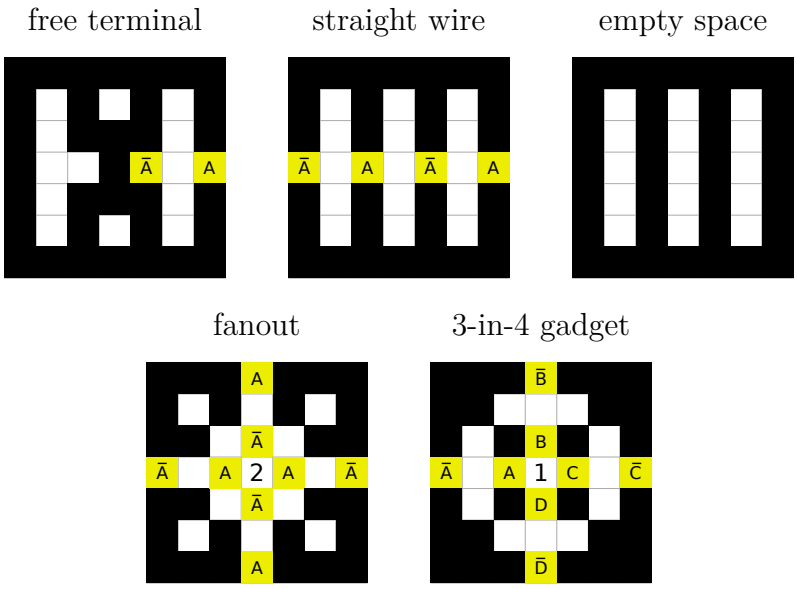
5.4.3 Cross clues

These gadgets work with [X] together with any combination of [M], [L], [Q], [C], and [T'].



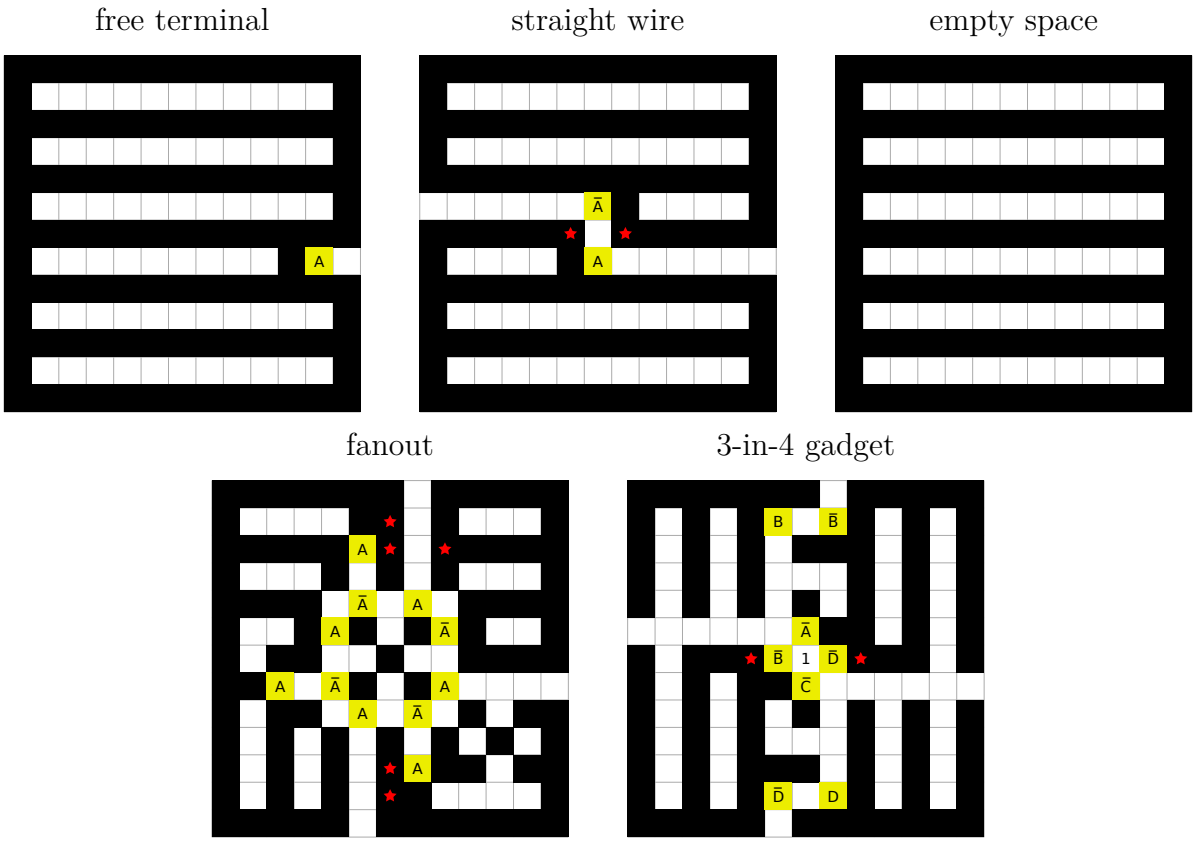
5.4.4 Mini-cross clues

These gadgets work with [X'] together with any combination of [M], [L], [Q], [C], [T'], and [N].



5.4.5 Eyesight clues

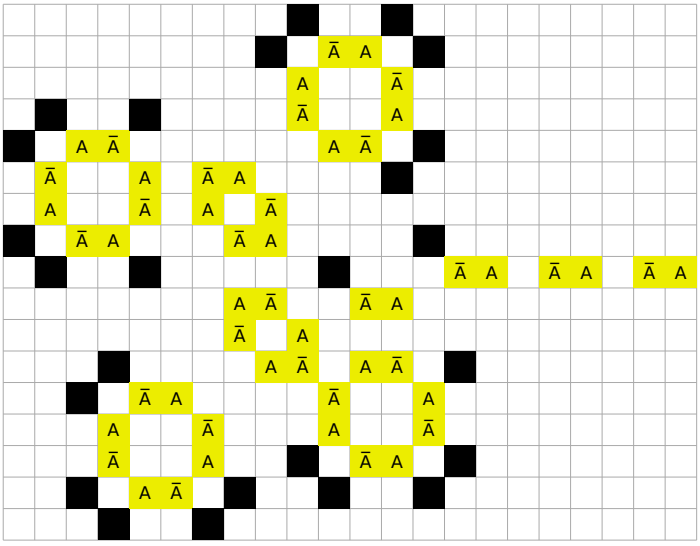
These gadgets work with [E] together with any combination of [L], [Q], [C], and [T']. Alternating tiles are reflected to make ports align. Solutions don't all have the same number of mines in a single tile, but they do globally: the two cells interacting between tiles contain exactly one mine, and ignoring those cells each tile has a constant number of mines.



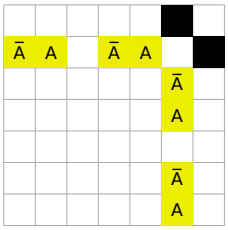
5.4.6 Wall and partition clues

These gadgets work with either [W] or [P], together with any combination of [U], [T], and [L].

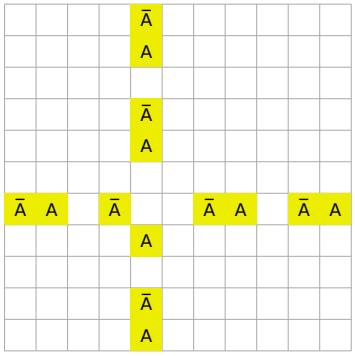
free terminal



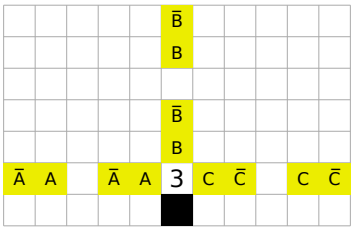
turning wire



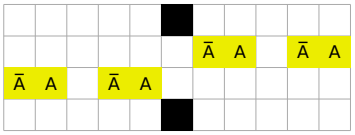
fanout



1-in-3 gadget



shift



Chapter 6

Grid Graphs and T-Metacells

In this chapter, I describe a gadget framework designed for proving NP-hardness of logic puzzles involving drawing a loop. This framework can be thought of as an extension of the Hamiltonian cycle 'gadget framework', refined so only one gadget is needed: the T-metacell. This gadget framework has been particularly successful in making it easy to mass-produce hardness proofs, as indicated by the sheer number of applications, including some that are not directly related to drawing cycles.

This chapter represents joint work with Josh Brunner, Lily Chung, Erik Demaine, and Andy Tockman. These results first appeared in [GBC+24], with the exception of the proof in Section 6.6.4 that Dotchi-Loop is ASP-hard with no black circles.

6.1 Introduction

Hamiltonicity is one of the core NP-complete problems, used as the basis for countless NP-hardness reductions. It accounts for two of Karp's 21 NP-complete problems [Kar72]: directed and undirected Hamiltonian cycle. It has been shown to remain NP-complete for many restricted graph classes: undirected maximum-degree-3 graphs [GJS74], undirected bipartite graphs [Kri75], undirected 3-connected 3-regular bipartite graphs [ANS80], undirected 2-connected 3-regular bipartite planar graphs [ANS80], undirected 3-connected 3-regular planar graphs of minimum face degree 5 [GJT76], directed planar graphs with indegree and outdegree at most 2 and total degree at most 3 [Ple79], and so on.

One of the most useful special cases of Hamiltonicity is (square) *grid graphs*: graphs whose vertices are a subset of the 2D integer lattice, with an edge connecting two vertices exactly when they have distance 1. Itai et al. [IPS82] proved that finding a Hamiltonian cycle is NP-complete in grid graphs, Papadimitriou and Vazirani [PV84] improved this result by proving Hamiltonian cycle NP-complete in grid graphs of maximum degree 3. Together, these results strengthen most of the special graph classes mentioned above (as grid graphs are necessarily planar and bipartite), with a stronger geometric guarantee. Other papers extend these results to other 2D grids [AFI+09, DR17, HL18]. Hamiltonicity in grid graphs is the foundation for NP-hardness proofs of countless games and puzzles, from video games [For10, DLL18, ABC+20] to pencil-and-paper puzzles [Yat00, And09], as well as practical problems such as lawn mowing and milling [AFM00, ABD+05].

But what about *parsimonious* reductions that preserve the number of solutions? A particularly strong form of this notion is ASP-completeness: an NP search problem P is *ASP-complete* [YS03] if there is a polynomial-time reduction from every NP search problem Q to P along with a polynomial-time bijection converting every solution of P to a unique solution of Q and vice versa. If P is ASP-complete, then the decision version of P is NP-complete, counting solutions to P is #P-complete, and the *k-ASP* P problem—given an instance of P and k solutions, find another solution—is NP-complete for any $k \geq 0$ [YS03].

Only a few versions of Hamiltonicity are known to be ASP-complete, or weaker, #P-complete. Liśkiewicz, Ogihara, and Toda [LOT03] proved #P-completeness of Hamiltonian cycle in undirected 3-regular planar graphs (based on [GJT76]). Seta [Set02] proved ASP-completeness of Hamiltonian cycle in undirected maximum-degree-3 planar graphs (based on [Ple79]). Bosboom et al. [BCC+20] proved ASP-completeness of Hamiltonian cycle in *directed* 3-regular (indegree 2 and outdegree 1 or vice versa) planar graphs (based on [Ple79]). But what about grid graphs?

6.1.1 Our results

In this chapter, we prove that Hamiltonian cycle in maximum-degree-3 grid graphs is ASP-complete. Thus this popular problem can serve as a foundation for ASP-completeness proofs as well. The same result holds for Hamiltonian cycle in *directed* maximum-degree-3 grid graphs, where each edge has a specified direction. As mentioned above, grid graphs are bipartite and planar, so these results roughly strengthen the ASP-completeness results mentioned above, except that we can guarantee "maximum-degree-3" but not "3-regular". (No grid graphs are 3-regular; consider the top left corner. Furthermore, undirected 3-regular graphs have an even number of Hamiltonian cycles by Smith's Theorem [Tut46], so we cannot hope for ASP-completeness in this case: the 1-ASP decision problem is trivial, while the 1-ASP construction problem is in PPA [Pap94].)

The basis for this result is another form of Hamiltonicity called *Tree-Residue Vertex-Breaking (TRVB)* [DR18], previously used to analyze Hamiltonicity in grid graphs [DR17]. In TRVB, we are given a graph where some vertices are breakable, and the goal is to *break* a subset of the breakable vertices—replacing each broken degree-k vertex with k degree-1 vertices—to make the graph into a tree. This problem has a known characterization of what degrees of breakable or unbreakable vertices make the problem polynomial vs. NP-complete [DR18]. We prove that several forms of TRVB are in fact ASP-complete, including planar multigraphs with degree-6 breakable vertices, and planar multigraphs with degree-4 breakable and degree-1 unbreakable vertices.

We also study even more geometric forms of grid-graph Hamiltonicity. Suppose that instead of allowing an arbitrary set of vertices on the square grid, we require the vertex set to be an entire $m \times n$ rectangle of integer points. Such graphs are known as *rectangular grid graphs* [IPS82]. In this case, undirected Hamiltonian cycle is known to be easy [IPS82]. But we show that *directed* Hamiltonian cycle in rectangular grid graphs is ASP-complete. Alternatively, if the graph is undirected but we allow removing some edges (but not vertices) from the rectangular grid—a spanning subgraph of a rectangular grid graph—then Hamiltonicity is also ASP-complete. Table 6.1 summarizes these results.

Rectangular grid graphs are useful because many (if not most) pencil-and-paper puzzles

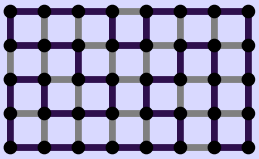
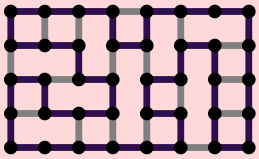
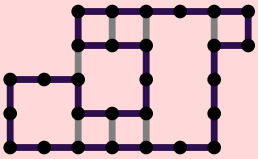
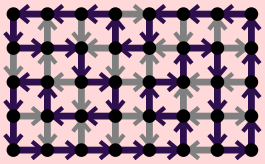
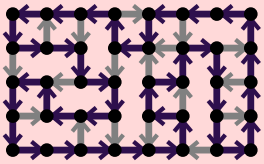
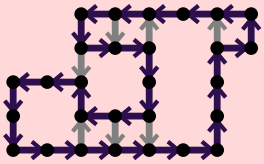
	Rectangular	Max-degree-3 spanning subgraph of rectangular	Max-degree-3
Undir.	P [IPS82] 	ASP-complete [§6.4.2] 	ASP-complete [§6.4.3] 
Directed	ASP-complete [§6.4.1] 	ASP-complete [§6.4.2] 	ASP-complete [§6.4.3] 

Table 6.1: Complexity of Hamiltonian cycle in various types of grid graphs. Each cell shows an example of a Hamiltonian graph of the specified type, with a darkened Hamiltonian cycle. The first and third column concern true grid graphs, where there is an edge between each pair of vertices at distance 1. In the first and second columns, the vertices form exactly an $m \times n$ rectangle, whereas the third column allows an induced subgraph of a rectangular grid graph. The middle column concerns graphs constructed from a rectangular grid graph by removing some edges (but no vertices) so that each vertex has degree at most 3. The second and third columns have maximum degree 3.

Games	#	New ASP-Hardness	New Reduction	New NP-Hardness
Slalom/Suraromu [KT15, Tan22], Onsen-meguri [Tan22], Mejilink [Tan22], Detour [Tan20, Tan22], Tapa-Like Loop [Tan22], Kouchoku [Tan22], Icelom [Tan22]	7	yes	no	no
Masyu [Fri02, Tan22], Yajilin [ISI12, Tan22], Nagareru [II22, Tan22], Castle Wall [Tan22], Moon or Sun [II22, Tan22], Country Road [ISI12, Tan22], Geradeweg [Tan22], Maxi Loop [Tan22], Mid-loop [Tan22], Balance Loop [Tan22], Simple Loop [IPS82, Tan22], Haisu [Tan20, Tan22], Reflect Link [Tan22], Linesweeper [Maa19]	14	yes	yes	no
Vertex/Touch Slitherlink, Dotchi-Loop, Ovotovata, Building Walk, Rail Pool, Disorderly Loop, Ant Mill, Koburin, Mukkonn Enn, Rassi Silai, (Crossing) Ichimaga, Tapa, Canal View, Aqre, Paintarea	17	yes	yes	yes

Table 6.2: Our results on pencil-and-paper puzzles. All ASP-completeness results are new; some are via an existing reduction [Tan22] and some are via a new reduction; and some puzzles were not even known to be NP-hard. (Puzzles known to be NP-hard have corresponding citations.)

take place on a full rectangular grid. In particular, the *T-metacell framework* of Tang [Tan22] shows how NP-hardness for a pencil-and-paper puzzle often follows from building a single gadget, essentially representing a degree-3 vertex that must be visited at least once. In Section 6.5, we extend this framework to prove ASP-completeness as well. We also extend the framework to allow for T-metacells where some exits are directed (usable in only one direction) and up to one exit is forced (must be used). In some cases, we need to build more than one T-metacell to handle different orientations of directions and/or forced edges.

Finally, in Section 6.6, we apply this framework to prove ASP-completeness of 38 pencil-and-paper puzzles, listed in Table 6.2. Five of these results use the same reduction from [Tan22], while the remainder involve creating new T-metacell gadget(s). For fourteen of the analyzed puzzles, even our NP-hardness result is new.

6.2 Connections between problems

We collect together some useful equivalences between problems on plane graphs, which are variously present in the literature [FS06, DR18].

Definition 6.1 ([DR18])**.** The *Tree-Residue Vertex-Breaking* (TRVB) problem takes place on an undirected multigraph with vertices marked as either 'breakable' or 'unbreakable'. The goal is to *break* a subset S of the breakable vertices to leave a tree—to break a vertex of degree d, replace it with d new leaves attached to its incident edges. In other words, the graph obtained from G by subdividing every edge and deleting the vertices in S must be a tree.

Definition 6.2 ([BM87, FS06])**.** Given a plane multigraph, a *kiki Euler tour* is a cycle which traverses every edge exactly once, such that any time the cycle enters a vertex via an edge e, it leaves by an edge adjacent to e in the cyclic order.[1]

The following is a well-known result with a long history; see [TW11].

Theorem 6.3. *Every Eulerian plane graph where every face is a triangle, except possibly the exterior face (a "near-triangulation"), has a proper vertex 3-coloring.*

Let G be a connected 3-regular bipartite plane multigraph, and let $\widetilde{G}$ be its plane dual. By Theorem 6.3, $\widetilde{G}$ is 3-colorable; equivalently it is possible to 3-color the faces of G so that adjacent faces have different colors, where faces are regarded as adjacent if they share an edge. Note that in such a 3-coloring, the three faces around a single vertex contain each color exactly once.

Let us fix such a coloring using the colors {white, blue, yellow} such that the exterior face is colored white. Define the following graphs:

- G_1 is the directed plane multigraph obtained from G by orienting every blue face clockwise and every white face counterclockwise. This fully determines the orientation.
- G_2 is the plane multigraph obtained from G by contracting every yellow face to a single vertex.
- G_3 is the subgraph of $\widetilde{G}$ induced by the non-white vertices.

Lemma 6.4. *There are bijections between the following sets:*

(i) Assignments of colors {white, blue} to each yellow vertex of $\widetilde{G}$ such that the white induced subgraph is connected and the blue induced subgraph is also connected.

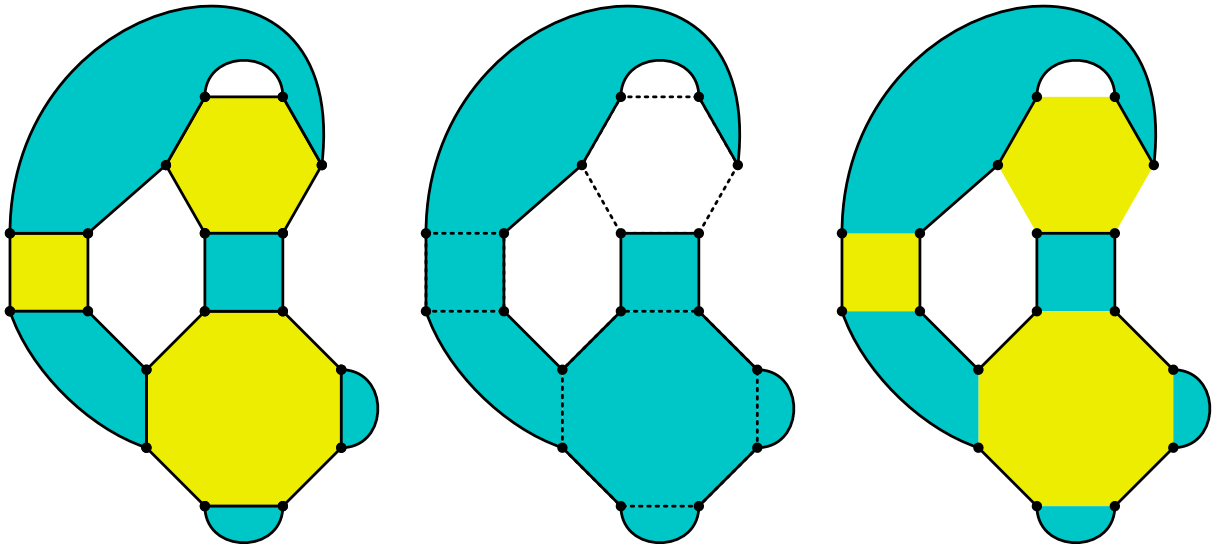
(ii) Hamiltonian cycles of G which contain all blue faces and no white faces.

(iii) Hamiltonian cycles of G which use every edge separating white faces from blue faces.

(iv) Directed Hamiltonian cycles of G_1.

(v) Kiki Euler tours of G_2.

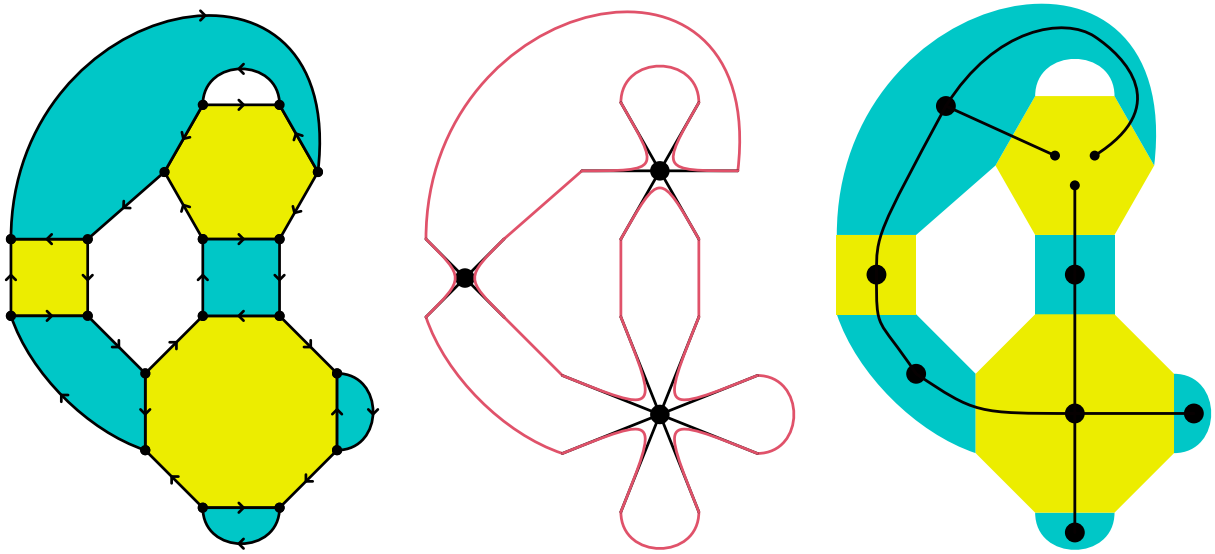
[1] This notion is one of two definitions of "nonintersecting" or "noncrossing Euler tour". We avoid this term to avoid confusion with the other definition, where an Euler tour is has a *crossing* if there are four edges e, e', f, f' adjacent to a single vertex so that e' follows e and f' follows f in the tour, and $\{e, e'\}$ alternates with $\{f, f'\}$ in the cyclic order [TW11]. Noncrossing Euler tours in this sense always exist, whereas kiki is a stricter condition.



(a) Face 3-coloring of G.

(b) Assignment of colors with blue and white connected.

(c) Cycle containing all blue faces and no white faces.



(d) Directed graph G_1.

(e) Kiki Euler tour of G_2.

(f) Tree-Residue Vertex-Breaking of G_3.

Figure 6.1: Illustration of Lemma 6.4.

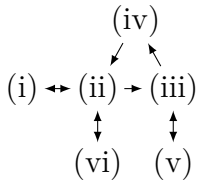


Figure 6.2: The bijections we define for Lemma 6.4.

(vi) Tree-Residue Vertex-Breakings of G_3, where yellow vertices are breakable and blue vertices are unbreakable.

Proof. Refer to Fig. 6.1. We give explicit transformations between the sets; it can be checked that these transformations invert each other as needed. Fig. 6.2 summarizes the transformations we describe, which form a strongly connected graph.

(i) → (ii): Consider an assignment of colors to faces of G. For each vertex, two of the faces around it are one color and the third is the other color, so exactly two edges incident to it separate blue from white. The set of all edges separating blue from white thus forms a collection of cycles visiting each vertex once.

We claim that this is actually a single cycle. If it were multiple cycles, they would divide the plane into more than two regions. Two of those regions must be the same color (blue or white), violating the assumption that each color is connected.

So we have a Hamiltonian cycle separating blue from white, and since the exterior face is white, it contains all blue faces and no white faces of G.

(ii) → (i): Given a cycle, assign blue to exactly the faces it contains. Since the cycle is Hamiltonian, it does not intersect itself, so the blue faces are connected and the white faces are connected.

(ii) → (iii): If C contains all blue faces and no white faces, then it must use every edge separating white from blue.

(iii) → (iv): If C is a cycle on G_1 which uses every edge separating white from blue, then at each individual vertex it is impossible for C to reverse directions; thus it is always consistent with the orientations, so it is a directed Hamiltonian cycle.

(iv) → (ii): Suppose C is a directed Hamiltonian cycle of G_1. Since C visits every vertex, it contains at least one edge of every face. Because C contains an edge of the exterior face its orientation must be consistently clockwise. Therefore C it encounters every blue face on its right side and every white face on the left, meaning it contains every blue face and does not contain any white faces.

(iii) → (v): The edges separating white and blue faces are exactly the edges of G_3 remaining after contracting the yellow faces. Let C be a Hamiltonian cycle of G containing every white-blue edge, and let C' be the Euler tour of G_3 obtained from C by the contraction. It must be the case that C contains exactly half of the edges incident to each yellow face, each of which connects two adjacent white-blue edges; so C' is kiki.

(v) → (iii): Suppose C' is a kiki Euler tour of G_3. Let C be the set of edges of G consisting of all white-blue edges, together with those that connect consecutive edges in C'; then C is a Hamiltonian cycle of G containing every white-blue edge.

(ii) → (vi): Note that G_3 does not have any edges between two breakable vertices, so breaking a vertex is equivalent to removing it and all incident edges. Thus TRVB becomes "find an induced subgraph of G_3 containing all unbreakable vertices which is a tree".

Given a cycle C, break all yellow vertices which are outside C, or equivalently take the induced subgraph on vertices inside C. This subgraph is clearly connected. If it has a cycle, there is a face of $\widetilde{G}$ inside that cycle, which corresponds to a vertex v of G. Then v is strictly inside C. But v must touch a white face, contradicting the fact that all white faces are outside C. Hence the induced subgraph on vertices inside C is a tree.

(vi) → (ii): Take C to be the boundary of the tree containing blue faces and nonbroken yellow faces. Then C is a cycle because it bounds a tree, its interior contains all blue faces (which cannot be broken) and no white faces (which are not present in G_3. Finally, C is Hamiltonian because every vertex is incident to an edge separating blue from white, which must be in C. □

Furthermore, given any of the graphs G_i, equivalents to the others can be obtained by analogous transformations. So these various problems can be regarded as equivalent.

An important special case of TRVB is when every breakable vertex has degree at most 3. For planar graphs this condition is equivalent to requiring that every yellow face of the graph G in the preceding discussion is a digon or triangle; it is also equivalent to kiki Euler tour with vertices of degree at most 6. In this case, the problem can be solved in polynomial time by reducing it to a matroid parity problem.[FS06][DR18] In the next section we will discuss breakable vertices with higher degrees, with which the problem turns out to be ASP-complete.

The above characterization in Lemma 6.4 is general enough to usefully characterize all directed Hamiltonian max-degree-3 plane graphs. In particular, for any directed max-degree-3 plane graph G, it is possible to construct in polynomial time a spanning subgraph H which contains all of the Hamiltonian cycles of G, and is essentially of the form of G_1 in Lemma 6.4. We first give two useful facts about directed planar Hamiltonian cycles before showing how to construct H.

Lemma 6.5. *Let G be a directed plane multigraph, C be a directed cycle of G, and F be a face of G. Then every edge of C that borders F must have the same direction (clockwise or counterclockwise) around F.*

Proof. A cycle in a plane graph splits the plane into two regions: inside the plane and outside the plane. Every face must lie either entirely inside or entirely outside the cycle. Any time a face touches the cycle, it must have the same orientation as the cycle: if the cycle is clockwise, then wherever it touches a face inside the cycle that edge must be oriented clockwise with respect to the face, and wherever it touches a face outside the cycle that edge

must be oriented counterclockwise with respect to the face. Since every face lies entirely inside or entirely outside the cycle, all of the orientations of edges touching the face that are part of the cycle must be consistent. □

Observation 6.6. Let G be a Hamiltonian directed plane multigraph, and let C be a Hamiltonian cycle. Then any edge which is the only incoming or only outgoing edge from a vertex must be present in C.

Lemma 6.7. *Let G be a directed max-degree-3 plane multigraph. Then there exists a polynomial-time algorithm which either reports that G has no Hamiltonian cycles, or computes a directed spanning subgraph H of G so that the faces of H can be 3-colored with {blue, white, yellow} so that every blue face is oriented clockwise and every white face counterclockwise, such that every Hamiltonian cycle of G is contained in H.*

Proof. We describe a polynomial-time algorithm to compute H from G.

Call an edge *forced* if it must be in every Hamiltonian cycle by Observation 6.6. If a vertex has two forced edges, the third edge can never be taken.

We now give 2 rules to remove edges from G. To get H, repeatedly apply these rules until they do not remove any edges; the resulting graph is H. If at any point a vertex has indegree 0 or outdegree 0, then terminate and report that G has no Hamiltonian cycles.

1. For every vertex with two forced edges, delete any other edge incident to that vertex.
2. For every face, delete all edges whose orientation is not consistent with every forced edge on that face.

The first rule clearly does not remove any Hamiltonian cycles. By Lemma 6.5, the second rule will also never delete an edge that is part of any Hamiltonian cycle. Thus, H and G have the same Hamiltonian cycles.

All that remains is to show that the faces of H can be colored appropriately. We will first show that every face is one of three types:

1. Every edge is oriented clockwise (blue faces)
2. Every edge is oriented counterclockwise (white faces)
3. The edges alternate orientation around the face (yellow faces)

Consider a face F. Suppose F has at least one forced edge. Then because the 2nd rule does not remove any edges from H, so every edge on F must have the same orientation.

Now suppose F has no forced edges. Consider any vertex incident to F. Then since neither of the edges of F incident to that vertex are forced, they must either both point into or both point out of that vertex. This means that these edges must be oriented opposite ways (i.e. one clockwise and one counterclockwise) around F. Since this is true at every vertex of F, the edges alternate around F.

Finally, we need to show that if two faces share an edge, they must be of different types. It's never possible for two blue faces to share an edge, because if an edge is clockwise according to one of the faces, it must be counterclockwise according to the other. A similar argument applies for white faces. For yellow faces, consider a vertex v incident to the shared edge. Since edges alternate orientation around yellow faces, it follows that v has either indegree 0 or outdegree 0, which is impossible. □

6.3 ASP-completeness of TRVB

Demaine and Rudoy [DR18] prove several NP-hardness results for TRVB using reductions from finding Hamiltonian cycles on a max-degree-3 planar directed graph. At the time, this Hamiltonian cycle problem was not known ASP-complete, so they did not consider ASP-completeness.

More recently, Bosboom et al. [BCC+20] showed that finding Hamiltonian cycles on a directed max-degree-3 planar graph is ASP-complete, using a reduction from positive 1-in-3SAT.

Several of the reductions used by Demaine and Rudoy [DR18] are easily verified to be parsimonious, proving ASP-completeness. We are specifically interested in the results of Section 4, on planar $(\{k\}, \{4\})$-TRVB.

They first reduce finding Hamiltonian cycles on a max-degree-3 planar directed graph to finding Hamiltonian cycles on a planar graph where all vertices have indegree and outdegree 2 and vertices have their two in-edges and their two out-edges adjacent in the planar embedding. This last condition is called *non-alternating*, because vertices are not allowed to alternate in-edges and out-edges. The reduction is by contracting forced edges, and is straightforwardly parsimonious.

Theorem 6.8. *Finding Hamiltonian cycles on non-alternating indegree-2 outdegree-2 planar graphs is ASP-complete.*

Next, Demaine and Rudoy reduce this problem to a version of Tree-Residue Vertex-Breaking. Specifically, Demaine and Rudoy [DR18] prove NP-hardness of TRVB on a planar graph where each unbreakable vertex has degree 4 and each breakable vertex has degree k, for any constant $k \geq 4$. This is *planar* $(\{k\}, \{4\})$*-Tree-Residue Vertex-Breaking*. This reduction is a bit more complicated (see Section 4.2 and in particular Figures 11 through 13 of [DR18]) but it is again parsimonious; indeed, [DR18, Lemmas 4.14 and 4.15] show that there is a bijection between Hamiltonian cycles in the input problem and solutions to the TRVB instance.

Theorem 6.9. *Planar* $(\{k\}, \{4\})$*-TRVB is ASP-complete, for each* $k \geq 4$*.*

To further simplify our reductions, we will use a slightly simpler version of TRVB: degree-4 breakable vertices and degree-1 unbreakable vertices.

Theorem 6.10. *Planar* $(\{4\}, \{1\})$*-TRVB is ASP-complete.*

Proof. It suffices to parsimoniously simulate a degree-4 unbreakable vertex. Such a simulation is shown in Fig. 6.3. No vertex in the simulation can be broken in a solution to TRVB. □

Theorem 6.11. *Planar* $(\{6\}, \emptyset)$*-TRVB is ASP-complete.*

Proof. It again suffices to simulate a degree-4 breakable vertex. Such a simulation is shown in Fig. 6.4. If the top vertex is not broken, both others must be broken, disconnecting the middle edge. So the top vertex must be broken, and then the other two vertices must not be. □

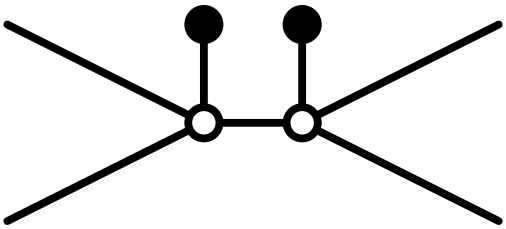


Figure 6.3: Simulating a degree-4 unbreakable vertex using degree-4 breakable vertices (white) and degree-1 unbreakable vertices (black).

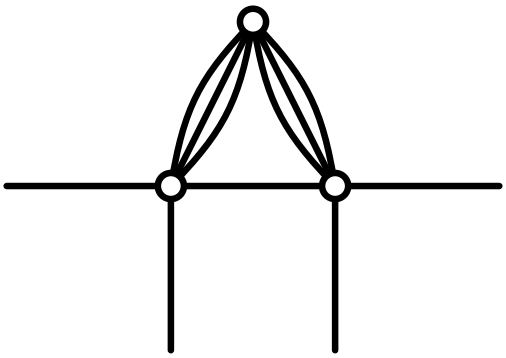


Figure 6.4: Simulating a degree-4 unbreakable vertex using degree-6 breakable vertices.

6.4 Hamiltonian cycles in grid graphs

In this section, we prove ASP-completeness of finding Hamiltonian cycles in several natural classes of grid graphs. We begin by defining the types of graph that appear in our results.

Definition 6.12. A *grid graph* is an induced subgraph of the square lattice. That is, its vertices are a subset of $\mathbb{Z}^2$, and it has an edge between each pair of vertices at distance 1. In a *directed grid graph*, each edge has a direction, so there is exactly one edge between each pair of vertices at distance 1.

Definition 6.13. A *rectangular* grid graph is one whose vertex set consists of all lattice points within a rectangle.

Definition 6.14. A graph is *max-degree-3* if each of its vertices have degree at most 3.

Definition 6.15. A *spanning subgraph* of G is a subgraph of G which contains all of the vertices (and some subset of the edges) of G.

Note that grid graphs contain all possible edges: graphs that contain only some of the edges are (spanning) subgraphs of grid graphs.

We consider three types of graph for each of undirected and directed. Our results are summarized in Table 6.1.

Most of our ASP-completeness results are by reductions from planar $(\{4\},\{1\})$-TRVB, and use the same core idea illustrated in Fig. 6.5. This is a breakable degree-8 vertex, with the yellow square in the middle representing the vertex itself and the blue tentacles

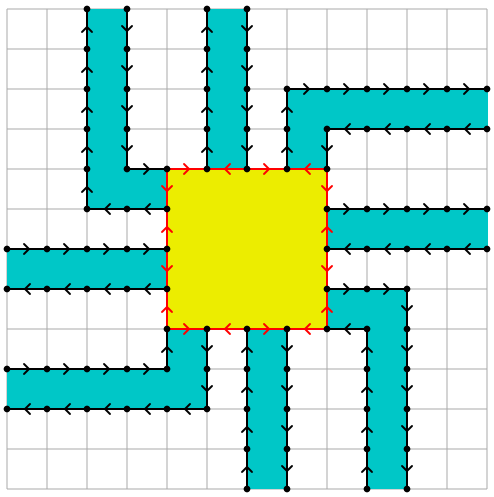


Figure 6.5: An example showing how reductions from TRVB to Hamiltonian cycle work.

representing edges. We replace every vertex in the TRVB instance with a vertex like the one shown, and connect the tentacles of adjacent vertices. By Lemma 6.4, Hamiltonian cycles of the resulting graph correspond to solutions of the original TRVB instance.

This idea works equally well for directed and undirected graphs. To apply this idea to each of the five types of graph we prove ASP-completeness for, we need to show how to draw gadgets for degree-4 breakable and degree-1 unbreakable vertices in that type of graph, while ensuring that the tentacles representing edges do not interfere with each other.

6.4.1 Rectangular grid graphs

Theorem 6.16 ([IPS82])**.** *Finding Hamiltonian cycles on an undirected rectangular grid graph is in P.*

Theorem 6.17. *Finding Hamiltonian cycles on a directed rectangular grid graph is ASP-complete.*

Proof. We first consider directed grid graphs, and later fill in holes to make them rectangular. Everything we need for this is shown in Fig. 6.6. The yellow rectangles are degree-4 breakable vertices with exactly two local solutions, and the dead end in the bottom left is a degree-1 unbreakable vertex. As before, blue is inside the loop and yellow might be inside the loop depending on the choice made for a vertex gadget. If we ignore the gray edges, this is essentially the same as Fig. 6.5.

We just need to ensure that gray edges cannot be used, which we can do by orienting them carefully. Ignoring the H-shaped construction in the center for the moment, each black edge is either the only edge pointing towards or the only edge pointing away from some vertex (depending on which side of the tentacle it's on), and thus must be used in a Hamiltonian cycle. We call such an edge *forced*. Each gray edge (still ignoring the H) shares either its source or its target with a black edge, and thus cannot be used. We call such an edge *unusable*.

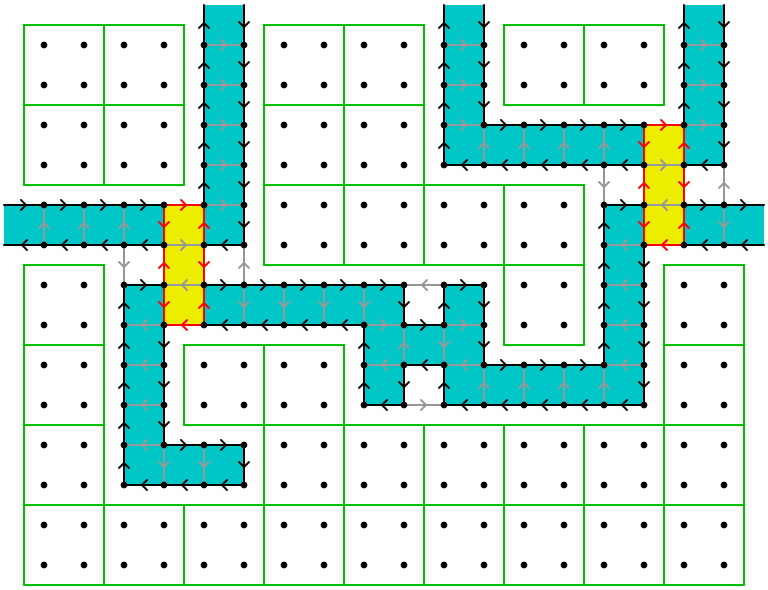


Figure 6.6: TRVB gadgets for directed grid graphs, showing two breakable degree-4 vertices connected by an edge and an unbreakable degree-1 vertex.

This requires the orientation of the gray edges relative to a tentacle to be different on the two ends of the tentacle, which is what the H achieves: one can verify by repeatedly finding forced edges and deleting unusable edges that any Hamiltonian cycle must use all black edges and no gray edges in the H. Each tentacle representing an edge between two degree-4 breakable vertices will have such an H.

This reduction proves a weaker version of the theorem: Finding Hamiltonian cycles on a directed grid graph is ASP-complete. It remains to fill all of the unused space to make a rectangular grid graph.

If we place each vertex gadget, H, and turn on the same parity, the construction lies neatly on a 2×2 grid, and in particular the holes are made of 2×2 squares. Fig. 6.6 indicates these squares in green. In addition, in each hole at least one of these squares is adjacent to a forced edge: all black edges except a few in each H are forced,[2] and each hole is adjacent to a non-H section of tentacle provided we do not use any extremely short tentacles.

Pick one such 2×2 square, and add four new vertices to fill it. Assume that the adjacent forced edge is the only outgoing edge from its source; the case where it is the only edge pointing towards its target is similar but with directions reversed. This situation is illustrated in Fig. 6.7 (left), with the forced edge in blue. Now reverse the forced edge, and add new edges as shown on the right of Fig. 6.7 (omitting any edges between a vertex in the square and a vertex outside it which doesn't yet exist). It is straightforward to check that all gray edges are unusable, so any Hamiltonian cycle must follow the blue path, which is equivalent to the original forced edge but consumes the added vertices.

Filling this small portion of hole preserves the fact that every hole has a 2×2 square adjacent to a forced edge, since the three relevant blue edges are forced. Thus we can repeat

[2]They all become forced after deleting some unusable edges, but it's simpler to argue that hole filling works with directly forced edges.

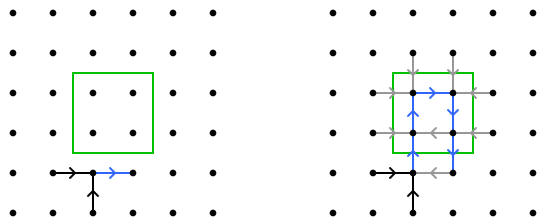


Figure 6.7: Filling holes in a directed rectangular grid graph.

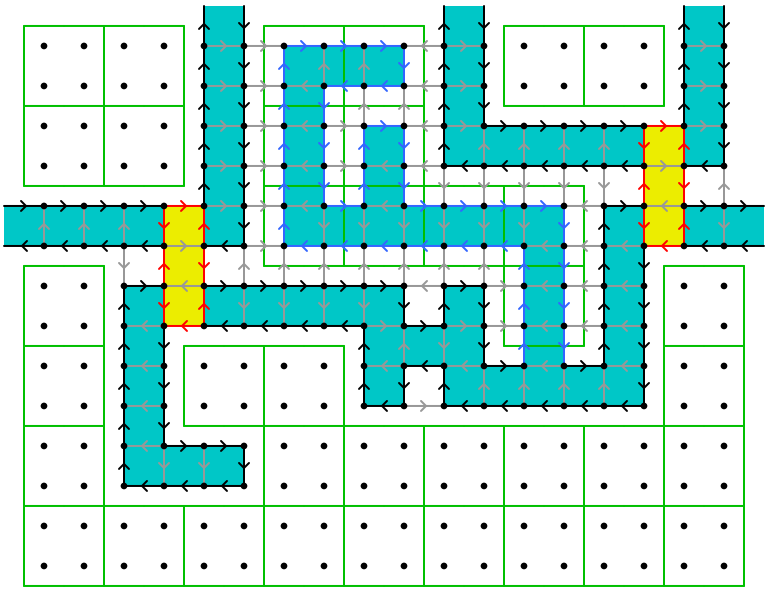


Figure 6.8: Fig. 6.6 after some hole filling.

this process until all holes are filled, ultimately filling each hole with paths that outline a spanning forest of the 2×2 squares. Fig. 6.8 shows what this looks like after filling (the visible portion of) the top middle hole in Fig. 6.6.

The result is a directed rectangular grid graph which is equivalent to the original directed grid graph for the purposes of Hamiltonian cycles. Hence Hamiltonian cycles in the final graph correspond to solutions to the instance of TRVB. □

6.4.2 Max-degree-3 spanning subgraphs of rectangular grid graphs

Theorem 6.18. *Let G be a directed max-degree-3 spanning subgraph of a rectangular grid graph. Consider the promise problem of finding an* undirected *Hamiltonian cycle on G, subject to the promise that all such cycles respect the given edge directions; that is, they would also be valid directed Hamiltonian cycles of G. This promise problem is ASP-complete.*

Proof. We modify the construction from Theorem 6.17 by simply removing all of the gray edges. Inspection of Fig. 6.8 reveals that every vertex is incident to at most three non-gray edges: vertices along tentacles have two forced edges, and vertices in degree-4 vertex gadgets have one forced edge and two optional red edges. Filling holes preserves the non-gray degree of existing vertices and adds vertices with two non-gray edges.

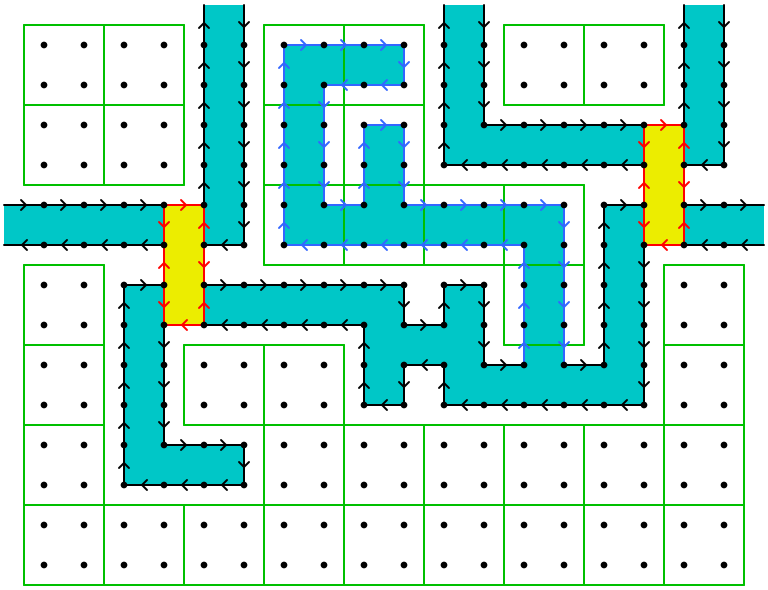


Figure 6.9: Fig. 6.8 after removing gray edges.

In the previous proofs, all of the possible solutions only used non-gray edges. Thus, we can adapt the previous reduction by simply deleting all gray edges, obtaining a directed max-degree-3 spanning subgraph of a rectangular grid graph. For instance, doing this to Fig. 6.8 yields Fig. 6.9, which also has the advantage of being easier to read.

By the proof of Lemma 6.4, directed Hamiltonian cycles on G are the same as undirected Hamiltonian cycles on G, and the set of such cycles is in bijection with solutions of the original TRVB instance. □

Corollary 6.19. *Finding Hamiltonian cycles on a directed max-degree-3 spanning subgraph of a rectangular grid graph is ASP-complete.*

Later, we will take advantage of the fact that, in these constructions, every vertex has a forced edge (for undirected graphs, an edge is *forced* if it is incident to a degree-2 vertex). A degree-3 vertex in a subgraph of a grid graph has two edges in opposite directions, which we call *side* edges, and a third edge between them, which we call the *center* edge. In the undirected case, we can assume not only that each degree-3 vertex has a forced edge, but that this forced edge is a side edge, further reducing the number of distinct vertices we need to simulate for an application.

Theorem 6.20. *Finding Hamiltonian cycles on an undirected max-degree-3 spanning subgraph of a rectangular grid graph is ASP-complete, even when every degree-3 vertex has a forced side edge.*

Proof. We are not able to directly build breakable degree-4 TRVB vertices under these constraints. However, we are able to build a breakable degree-6 vertex, so we reduce from planar $(\{6\}, \emptyset)$-TRVB, which was shown ASP-complete in Theorem 6.11.

Our breakable degree-6 vertex gadget is shown in Fig. 6.11. Black edges are forced, and red edges are optional. Note that vertices in tentacles all have degree 2, and each degree-3

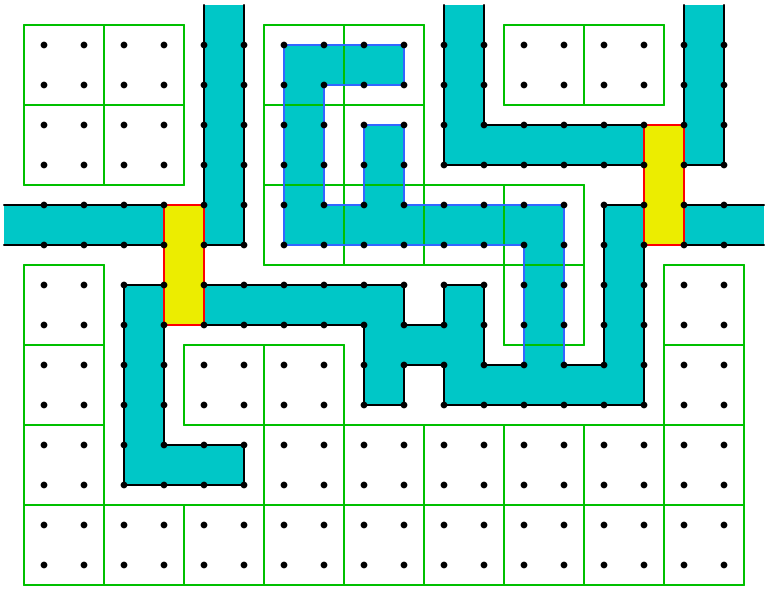


Figure 6.10: Fig. 6.9 after forgetting directions of edges.

vertex inside the vertex gadget has a forced side edge. This is equivalent to the cycle of red edges turning at every vertex. The vertex gadget has exactly two local solutions, which each use alternating red edges.

As before, blue tentacles are inside the cycle, and the yellow region is inside the cycle in one of the local solutions, corresponding to not breaking the TRVB vertex. We have new color as well: the green squares are inside the cycle in the other solution, when the TRVB vertex is broken. It is clear by inspection that the yellow local solution connects all six tentacles, and the green local solution disconnects them all.

Finally, we connect vertex gadgets along tentacles and fill holes in exactly the same way as before. Filling holes uses only degree-2 vertices, so it does not introduce degree-3 vertices without forced side edges. □

6.4.3 Max-degree-3 grid graphs

The existing proof of NP-hardness for finding Hamiltonian cycles in max-degree-3 grid graphs [PV84] has only one nonparsimonious gadget, the 'fork connection'. However, the reduction can be simplified to avoid this gadget, making it parsimonious. We will sketch the parsimonious version, but we will also provide a different proof using TRVB which yields the useful property that every vertex has a forced edge.

For the parsimonious adaptation of Papadimitriou and Vazirani's [PV84] proof, we reduce from finding Hamiltonian cycles on non-alternating indegree-2 outdegree-2 planar graphs, which is ASP-complete (Theorem 6.8) [DR18]. See Fig. 6.12. Given such a graph, replace each vertex with a *dumbbell*, which consists of two small loops called *in* and *out* connected by a path. Each edge is represented by a *tentacle* which connects the out-loop of its source to the in-loop of its target. These connections are slightly different, and are shown with blue and red tentacles, respectively, in Fig. 6.12.

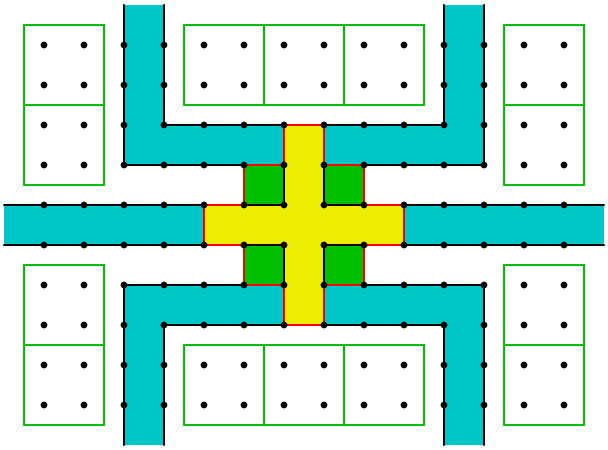


Figure 6.11: A breakable degree-6 TRVB vertex gadget for undirected max-degree-3 spanning subgraphs of rectangular grid graphs.

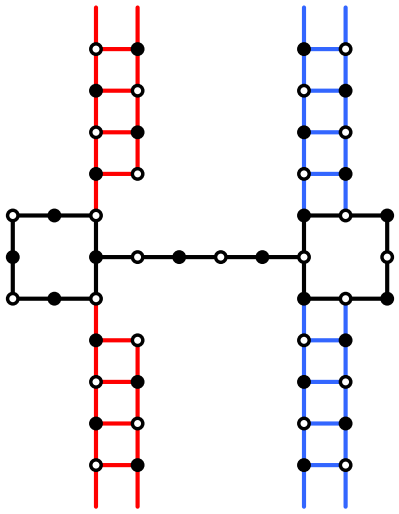


Figure 6.12: A dumbbell (black) with two red tentacles attached to its in-loop and two blue tentacles attached to its out-loop. Vertices are colored black and white in a checkerboard.

There is a lot of freedom in the placement of dumbbells, but the parity of the position of each loop is important: for tentacles to connect properly, the in-loop must have white corners and the out-loop must have black corners.

This works because each tentacle has only two solutions: one in which the Hamiltonian cycle zigzags along it, and one in which it loops back to the out-loop. These correspond to the edge being used and unused, respectively. Any Hamiltonian cycle in the grid graph must: arrive at a dumbbell from a red tentacle, go around the in-loop, cross the dumbbell to the out-loop, go down and back one of the blue tentacles back to this dumbbell, go around the out-loop to the other blue tentacle, zigzag down that blue tentacle, and finally arrive at the next dumbbell. The sequence of dumbbells gives a Hamiltonian cycle on the original graph.

Theorem 6.21. *Finding Hamiltonian cycles on an undirected max-degree-3 grid graph is*

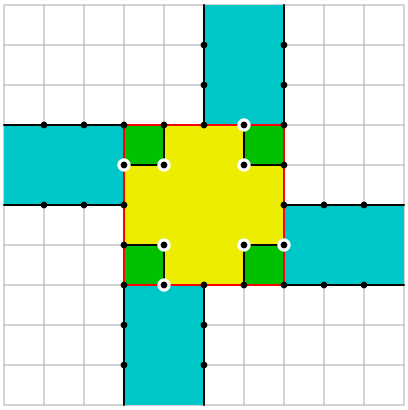


Figure 6.13: A breakable degree-4 TRVB vertex gadget for undirected max-degree-3 grid graphs. Removing the vertices highlighted in white gives an unbreakable degree-4 vertex gadget.

ASP-complete, even when every vertex has a forced edge.

Proof. This proof is sketched, and its key gadget is shown, by Demaine and Rudoy [DR18], but at the time TRVB was not known to be ASP-complete, so it was purely a simpler proof of NP-hardness used to motivate the usefulness of TRVB.

Like most of our other proofs, we reduce from planar $(\{4\},\{1\})$-TRVB. Our breakable degree-4 vertex gadget is shown in Fig. 6.13. The main difficulty in this case is that we need the paths on each side of a tentacle to be separated by distance at least 2, so that the cycle cannot cross between the two sides (and all tentacle edges are forced). As usual, black edges are forced, and there are exactly two solutions which each use alternating red edges. One solution puts the green region inside the cycle, and one puts the yellow region inside the cycle, corresponding to breaking and not breaking the vertex, respectively.

A degree-1 unbreakable vertex can be made by simply 'capping off' a tentacle. Alternatively, we could reduce from $(\{4\},\{4\})$-TRVB, and construct a degree-4 unbreakable vertex gadget by removing the vertices highlighted in white from Fig. 6.13. □

Theorem 6.22. *Finding Hamiltonian cycles on a directed max-degree-3 grid graph is ASP-complete, even when every vertex has a forced edge.*

Proof. The proof is extremely similar to the previous proof. We again reduce from $(\{4\},\{1\})$-TRVB. Our degree-4 breakable vertex gadget is shown in Fig. 6.14, and a degree-1 unbreakable vertex can again be made by capping off a tentacle. Black edges are forced and gray edges are unusable. We again keep the sides of a tentacle apart from each other (away from vertex gadgets) so that a cycle cannot leak between them.

As before, there are exactly two solutions to the vertex gadget, one of which puts the yellow square inside the cycle corresponding, to leaving the TRVB vertex unbroken. □

6.5 T-metacells

Many puzzle genres which involve drawing a single loop are proven hard using reductions from various forms of grid graph Hamiltonicity. Tang [Tan22] described a simple "T-metacell"

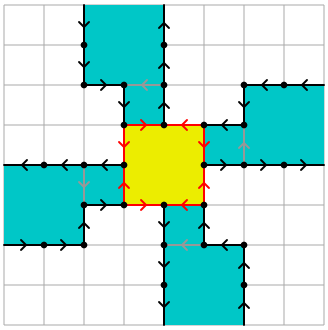


Figure 6.14: A breakable degree-4 TRVB vertex gadget for directed max-degree-3 grid graphs.

framework for proving NP-hardness of these puzzles using grid graph Hamiltonicity. A T-metacell is a gadget which represents a single degree-3 vertex in a grid graph. Each T-metacell is a (usually square) tile with 3 exits (on 3 of the 4 sides) such that the loop may traverse the gadget between any pair of exits. The gadget should be reflectable and rotatable, and the loop may travel between adjacent T-metacells only when both have exits along their shared border. Finally, the loop must be required to visit every T-metacell.

It's straightforward to see how T-metacells can simulate degree-3 vertices in a Hamiltonicity reduction; Tang showed that they can also simulate degree-2 vertices, as follows. Let G be a subgraph of a grid graph in which every vertex has degree 2 or 3. Degree-3 vertices of G can be replaced directly with T-metacells. To handle degree-2 vertices, consider the graph H on the same vertex set as G which has an edge between two lattice-adjacent vertices precisely when G is missing that edge. Then H consists of degree-1 and degree-2 vertices. Orient the edges of H into directed paths and cycles such that each vertex has a maximum indegree and outdegree of 1. Each degree-2 vertex of G can now be replaced by a T-metacell with its extra edge facing in the direction of the outward-pointing edge from that vertex in H. This ensures that this extra exit will always be facing a non-exit in the adjacent cell, so only the intended edges of G may be used by the loop.

We apply our results from Section 6.4 to show that solving T-metacell problems is ASP-complete, instead of just NP-hard. We extend the framework to allow for some exits of a T-metacell to be *directed*, meaning that the loop must have a consistent orientation which agree with the directions of the exits it uses. We also allow for T-metacells to have one *required edge* which the loop is required to use. Note that a directed T-metacell automatically has a required edge: there must be either a lone exit directed inwards or a lone exit directed outwards, which in either case must be used. When a T-metacell has a required center edge, we call it *symmetric*, and when it has a required side edge, we call it *asymmetric*.

Thus we can describe several types of T-metacell, based on whether they are directed, whether they have a required edge, and if so whether they are symmetric. We use this to reduce the number of distinct gadgets needed to apply the framework, by proving that certain small collections of types of T-metacell are sufficient for ASP-hardness.

Corollary 6.23. *Finding Hamiltonian cycles on a rectangular grid of directed T-metacells is ASP-complete.*

Proof. We reduce from finding Hamiltonian cycles on directed max-degree-3 spanning sub-

graphs of rectangular grid graphs (Corollary 6.19). Place a T-metacell for each degree-3 vertex, and handle degree-2 vertices in the same way as above. The direction of the unusable edge on a T-metacell at a degree-2 vertex can be arbitrary. □

Corollary 6.24. *Finding Hamiltonian cycles on a rectangular grid of undirected T-metacells (with no required edge) is ASP-complete.*

Proof. We reduce from finding Hamiltonian cycles on max-degree-3 spanning subgraphs of rectangular grid graphs (Theorem 6.20). Replace each vertex with a undirected T-metacell, handling degree-2 vertices as described above. □

Corollary 6.25. *Finding Hamiltonian cycles on a rectangular grid of asymmetric required-edge undirected T-metacells is ASP-complete.*

Proof. The reduction is the same as above, but now we take advantage of the requirement in Theorem 6.20 that every degree-3 vertex has a forced side edge. We place the required edge of the T-metacell for such a vertex at its forced edge. Degree-2 vertices require a bit more care, but are not an obstruction: after deciding how to orient T-metacells as described above, note that for each degree-2 vertex, at least one of its edges is a side edge of the T-metacell. So we can simply place a T-metacell with that side edge required. □

Corollary 6.26. *Finding Hamiltonian cycles on a rectangular grid of both asymmetric required-edge directed T-metacells and symmetric required-edge undirected T-metacells is ASP-complete.*

Proof. We reduce from the promise problem of finding a Hamiltonian cycle of a directed max-degree-3 spanning subgraph of a rectangular grid graph, with the promise that every undirected Hamiltonian cycle is a valid directed Hamiltonian cycle (Theorem 6.18). We perform the same replacement of vertices with T-metacells as in Corollary 6.23, except that the symmetric T-metacells are undirected. We claim that Hamiltonian cycles of the original graph are in bijection with solutions to the T-metacell instance. A directed Hamiltonian cycle of the original graph clearly solves the T-metacell instance, since it correctly passes through the directions on the directed T-metacells. On the other hand, a solution to the T-metacell instance is necessarily an undirected Hamiltonian cycle of the original graph; and by the promise, directed Hamiltonian cycles and undirected Hamiltonian cycles are the same. □

6.6 Applications

In this section, we apply our improved T-metacell framework to a variety of pencil-and-paper logic puzzles implemented by the online puzzle-solving interface "puzz.link" [KVS+]. This web resource implements more than 240 different logic puzzles. It includes most genres published by the Japanese publisher Nikoli, whose puzzles have a long history of analysis from a computational complexity perspective [Set02, YS03, And09, USO17, Maa19, Tan22], as well as many others in a similar style.

We improve existing NP-hardness results for pencil-and-paper logic puzzles to ASP-completeness, and give new ASP-completeness results. Many of the ASP-completeness proofs

consist of just a single T-metacell, demonstrating the ease of applying the framework for proving ASP-completeness. The main additional requirement when designing a T-metacell gadget for ASP-completeness proofs is that it be "parsimonious": for each pair of exits, there must be a *unique* local solution where the loop passes through those exits.

6.6.1 Loop-drawing paper-and-pencil logic puzzles

The rules of these logic puzzle genres share many commonalities. In order to avoid repetition in describing them, we first define some terminology.

The vast majority of the puzzles we analyze are loop-drawing puzzles. This is a natural setting to apply the T-metacell framework, as drawing a cycle is already part of the rules.

- All of the puzzles we analyze take place on a rectangular grid graph. The solver can draw *segments* between centers of orthogonally adjacent cells.
- In a *loop* genre, the goal is to draw a set of segments that form a single cycle. In geometric terms, they must form a simple closed curve.
- A *full* loop genre is one in which the loop is required to visit every cell.
- A *crossing* loop genre is one in which the loop may cross itself, violating the standard "simple" constraint.
- A *directed* loop genre is one in which the loop is additionally given a direction. By default, loop puzzles are assumed to be undirected.
- Considering each individual cell and how it connects to its neighbors, there are three possibilities up to rotation: it is either a *turn* (if it connects to one vertically adjacent cell and one horizontally adjacent cell), goes *straight* (if it connects to two cells on opposite sides), or is *unused*.
- We can also decompose the loop into *lines*, which are contiguous runs of segments in the same direction. A loop always alternates between horizontal and vertical lines. The length of a line is the number of segments it contains.
- Some puzzles divide the grid into a set of *regions*. In these puzzles, each region is a set of orthogonally connected cells, and the set of regions is a partition of the set of cells in the grid.

Although most of our results apply to loop genres, the methods also work for some other classes of puzzles:

- In a *path* genre, the goal is to draw a set of segments that trace a single path. The start and end points of the path may be given, or they may be to be determined by the solver. All considerations that apply to loop puzzles (full, crossing, etc.) also apply to path puzzles.

  The framework applies fairly directly to path puzzles, since it is almost always easy to construct a metacell that is forced to contain both endpoints of the path. This

simulates the Hamiltonian cycle problem, as a path that starts in this metacell, visits every other metacell, and ends in the original metacell is the same as a cycle at the metacell level.

- In a *shading* genre, the goal is to mark some subset of the grid cells as shaded to satisfy some set of constraints.
- The constraint found in shading genres that allows us to apply our framework is *connectivity*, which says that some set of cells (typically all of the shaded cells) must form a single connected component.

  If we can enforce that the shaded cells leave each T-metacell by at most two exits via other constraints, the set of shaded cells simulates a path, and our framework thus applies as described above.
- Some genres do not fit into an overarching archetype like loop, path, or shading. We refer to such puzzles as *variety* puzzles, which occasionally include a set of constraints conducive to our framework.

6.6.2 Prior ASP-completeness results

Some of the T-metacell gadgets in [Tan22] are already parsimonious, and thus automatically serve as ASP-completeness proofs for their respective genres given our new results. These genres are Slalom, Onsen-meguri, Mejilink, Detour, Tapa-Like Loop, Kouchoku, and Icelom.

For some other puzzle genres, ASP-completeness proofs exist outside of the T-metacell framework. Yato and Seta [YS03] prove that Slitherlink is ASP-complete in the paper originally defining the ASP class. Uejima, Suzuki, and Okada [USO17] prove that Pipelink is ASP-complete, which also proves ASP-completeness of the generalized versions Pipelink Returns and Loop Special.

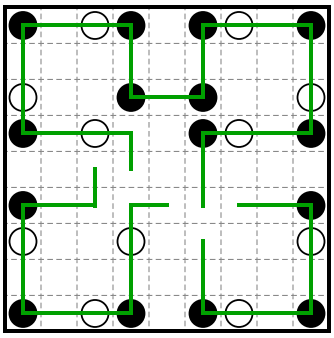
6.6.3 Prior NP-hardness improved to ASP-completeness

Most of the gadgets in this section consist of minor adjustments to existing T-metacells in [Tan22] to ensure parsimony.

Masyu

Masyu [Fri02, Tan22] is a loop genre. Some cells contain pearls, which can be either black or white. In any cell with a black pearl, the loop must turn, and it must go straight through both of the two adjacent cells. In any cell with a white pearl, the loop must go straight, and it must turn in at least one of the two adjacent cells.

We construct an undirected T-metacell to show ASP-completeness of Masyu by Corollary 6.24.



Yajilin

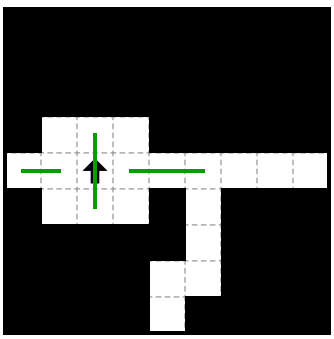
Yajilin [ISI12, Tan22] is a loop genre. Some cells contain numbered arrow clues, which count the number of unused cells from the clue to the edge of the grid (not including the clue itself). The loop cannot go through the clues, and of the remaining cells, unused cells may not be orthogonally adjacent.

There is a straightforward reduction from Hamiltonian cycle in undirected grid graphs, which by Theorem 6.21 is ASP-complete: start with the bounding box of the grid graph as a rectangular grid graph, place a "0" clue pointing in an arbitrary direction in every cell excluded from the original graph, and then add a row above the grid entirely filled with "0" clues pointing down, forcing every other cell to be visited.

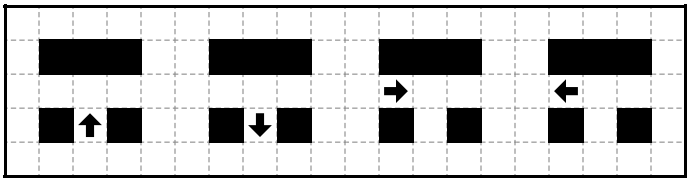
Nagareru

Nagareru (a.k.a. Nagareru-Loop) [II22, Tan22] is a directed loop genre. Shaded cells cannot be visited. Some unshaded cells contain arrows, which must contain a straight segment and indicate the direction of the loop along that segment. Some shaded cells contain arrows, which exert a "wind" on all unshaded cells in the direction of the arrow up to the next shaded cell. Whenever the loop enters a cell experiencing wind, it must immediately turn in the direction of the wind.

We construct an asymmetric required-edge undirected T-metacell to show ASP-completeness of Nagareru by Corollary 6.25. Note that the resulting construction has two solutions for every solution of the original undirected Hamiltonian cycle problem—we must also fix a global direction by inserting an arrow in the center cell of one T-metacell.



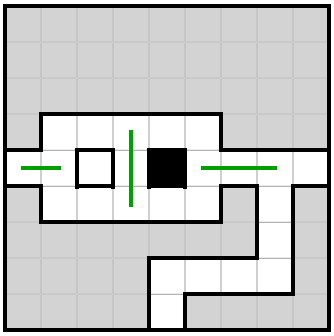
We note that we can alternatively construct a set of required-edge directed T-metacells to show ASP-completeness by Corollary 6.23.



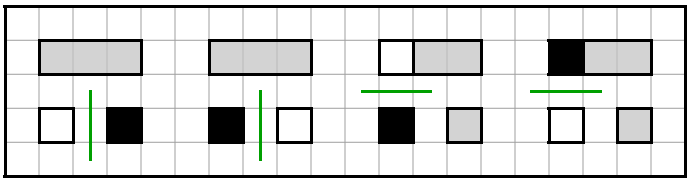
Castle Wall

Castle Wall [Tan22] is a loop genre. Some cells are shaded white, black, or gray. Shaded cells cannot be visited. White cells must be contained in the interior of the loop, and black cells must be in the exterior of the loop. Additionally, some shaded cells have numbered arrows, which count the number of segments in the direction of the arrow up to the edge of the grid with the same orientation as the arrow (vertical or horizontal).

We construct an asymmetric required-edge undirected T-metacell to show ASP-completeness of Castle Wall by Corollary 6.25.



We note that we can alternatively construct a set of required-edge directed T-metacells to show ASP-completeness by Corollary 6.23. This works because if the global orientation of the loop is arbitrarily thought of as travelling clockwise, it always "sees" white squares on its right and black squares on its left, and these T-metacells thereby force its direction.



Moon or Sun

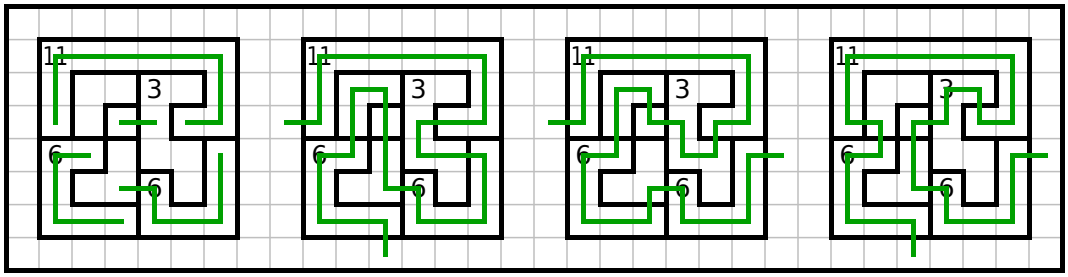
Moon or Sun [II22, Tan22] is a loop genre with regions. Some cells contain moons, and some cells contain suns. The loop must visit each region exactly once, and within each region must visit either all moons and no suns, or all suns and no moons. Furthermore, the loop cannot visit the same symbol in two consecutively used regions, so it must alternate between "sun-regions" and "moon-regions."

There is a straightforward reduction from Hamiltonian cycle in undirected grid graphs, which by Theorem 6.21 is ASP-complete: take the bounding box of the grid graph as a rectangular grid graph, place a sun in every cell contained in the original graph, place a moon in every other cell, and finally replace any sun with a 1×1 region containing a moon.

Country Road

Country Road [ISI12, Tan22] is a loop genre with regions. Some regions contain numbers, which count the number of cells the loop passes through in that region. Furthermore, no pair of orthogonally adjacent cells in different regions can both be unused.

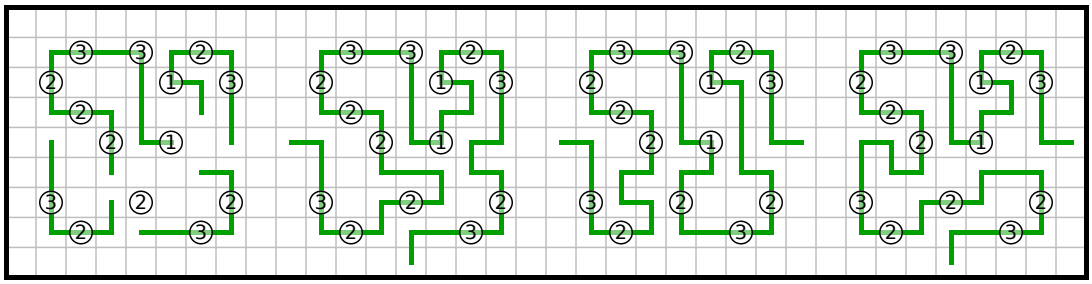
We construct an undirected T-metacell to show ASP-completeness of Country Road by Corollary 6.24.



Geradeweg

Geradeweg [Tan22] is a loop genre. Some cells contain numbers, which count the length of all lines touching that cell. All numbers must be visited.

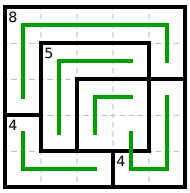
We construct an undirected T-metacell to show ASP-completeness of Geradeweg by Corollary 6.24.



Maxi Loop

Maxi Loop [Tan22] is a full loop genre with regions. Some regions contain numbers, which give the number of cells occupied by the maximal set of contiguous segments contained in that region.

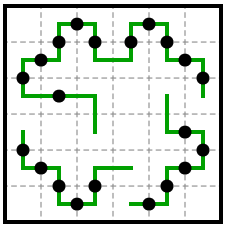
We construct an undirected T-metacell to show ASP-completeness of Maxi Loop by Corollary 6.24.



Mid-loop

Mid-loop [Tan22] is a loop genre. Dots can be placed on cell borders or at the centers of cells, and every dot must mark the midpoint of some line in the loop.

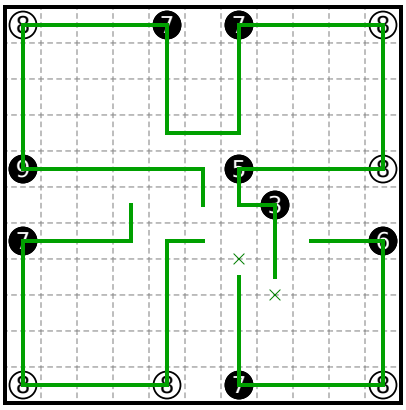
We construct an undirected T-metacell to show ASP-completeness of Mid-loop by Corollary 6.24.



Balance Loop

Balance Loop [Tan22] is a loop genre. Some cells contain either black or white circles. All circles must be visited. Circles give information about the two lines emanating from the circle, in the direction of each incident segment (which may be both-horizontal or both-vertical). For a white circle, their lengths must be the same, and for a black circle, their lengths must be different. Additionally, if the circle has a number, it gives the sum of the two lengths.

We construct an undirected T-metacell to show ASP-completeness of Balance Loop by Corollary 6.24.



Simple Loop

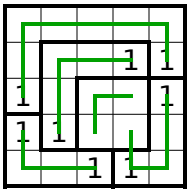
Simple Loop [IPS82, Tan22] is a loop genre. Some cells are shaded and cannot be visited, but all other cells must be visited.

Simple Loop is directly ASP-complete from Theorem 6.21.

Haisu

Haisu [Tan20, Tan22] is a path genre with regions. Some cells contain numbers. The first time the path visits a region, it must visit all the 1s; the second time the path visits a region, it must visit all the 2s; and so on.

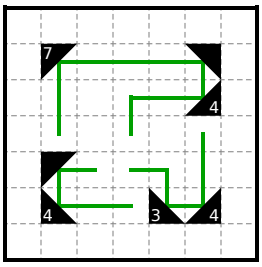
We construct an undirected T-metacell to show ASP-completeness of Haisu by Corollary 6.24. Note that we must break the loop into a path by e.g. picking an arbitrary T-metacell, replacing its center cell with an S, and replacing the cell to the right with a G.



Reflect Link

Reflect Link [Tan22] is a crossing loop genre. Some cells contain mirrors, which must contain a turn in the depicted orientation. Some mirrors contain numbers, which are 1 greater than the sum of the two incident lines (in other words, they count the total number of cells occupied by the two incident lines). The loop can only cross on marked crossings.

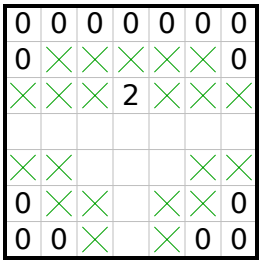
We construct an undirected T-metacell to show ASP-completeness of Reflect Link by Corollary 6.24.



Linesweeper

Linesweeper [Maa19] is a loop genre. Some cells contain numbers, which count the total number of cells the loop visits among the 8 orthogonally and diagonally adjacent cells. Numbers must not be visited.

We construct an undirected T-metacell to show ASP-completeness of Linesweeper by Corollary 6.24. Note that puzz.link does not contain a Linesweeper implementation, so it was not analyzed in Tang's paper; the known NP-completeness result was obtained independently [Maa19].

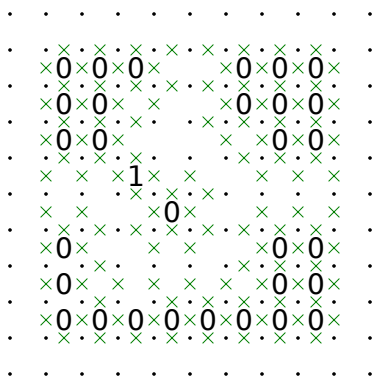


6.6.4 New NP- and ASP-completeness Results

Vertex/Touch Slitherlink

Vertex Slitherlink and Touch Slitherlink are loop genres. The presentation differs slightly from most other loop genres in that lines are drawn between dots instead of between centers of cells. Clues refer to the four surrounding vertices: for Vertex Slitherlink, they count the number of vertices visited by the loop, and for Touch Slitherlink, they count the number of distinct times the loop visits any of the four surrounding vertices (where a segment between two of them does not count as a separate visit).

We construct an undirected T-metacell to show ASP-completeness of both versions by Corollary 6.24.



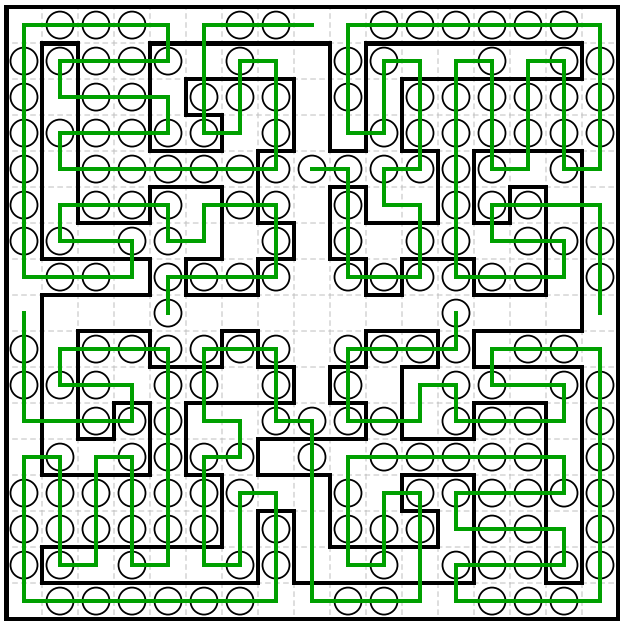
Dotchi-Loop

Dotchi-Loop is a loop genre with regions. Some cells contain black circles, which mean they cannot be visited. Some cells contain white circles, which must be visited. Additionally, within each region, the shape of the loop (whether it is a turn or goes straight) must be the same on all white circles.

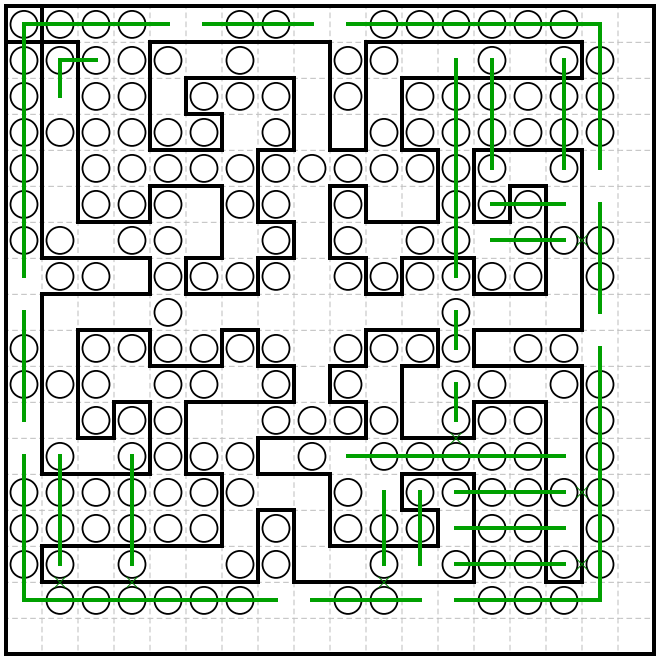
There is a straightforward reduction from Hamiltonian cycle in undirected grid graphs, which by Theorem 6.21 is ASP-complete: take the bounding box of the grid graph as a rectangular grid graph, place a 1×1 region with a white circle in every cell contained in the original graph, and place a black circle in every other cell.

Concurrently with the original publication of this result [GBC+24] in 2024, Deurloo et al. [DDM+24] used a very similar framework to prove that Dotchi-Loop is NP-hard even when black circles aren't adjacent. We now prove ASP-hardness of an even more restricted problem: Dotchi-Loop with no black circles.

We construct an undirected T-metacell. The outsides of T-metacells are connected, forming a single large region that spans the entire puzzle. Each T-metacell also contains one smaller region. Finally, we place one more white circle in the top left cell of the puzzle, in its own region.



It is relatively complicated to argue that everything drawn in the T-metacell above is forced. First, note that the top left corner forces the large outside region to have the loop go straight through white circles. Consider a particular T-metacell, and suppose we know that the loop runs parallel to its boundary in all of the white circles along its top and left sides. The white circle in the top left then must have a turn, which means every white circle in the region also has a turn. We can then make a sequence of deductions, eventually finding that the loop runs parallel to the boundary of the T-metacell in all of the white circles around it. Here is the path of this logic, showing the T-metacell in the top left corner:



It then follows by induction that the loop runs parallel to the boundary of T-metacells in every white circle in the 'channels' between them, and every small region in a T-metacell has

the loop turn on white circles. Deducing the rest of the edges drawn above is straightforward from this point.

Ovotovata

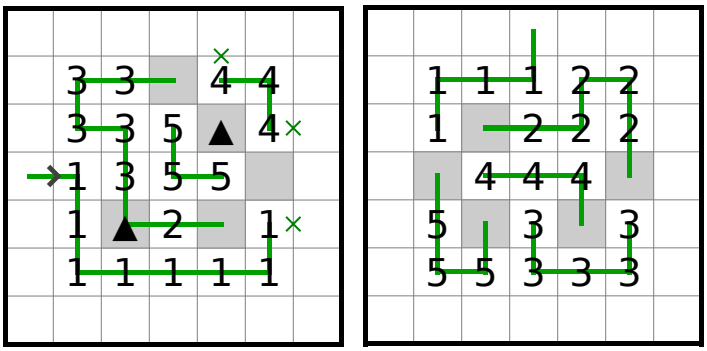
Ovotovata is a loop genre with regions. Some regions contain numbers, which for every time the loop exits that region in any direction count the number of additional cells it travels. Additionally, some regions are shaded, which means they must be visited.

There is a straightforward reduction from Hamiltonian cycle in undirected grid graphs, which by Theorem 6.21 is ASP-complete: take the bounding box of the grid graph as a rectangular grid graph, place a 1×1 shaded region in every cell contained in the original graph, and place an unshaded region containing a number larger than the size of the grid in every other cell.

Building Walk

Building Walk is a path genre. Some cells are shaded, representing elevators; and some unshaded cells have numbers: every number and elevator must be visited. Numbers on unshaded cells indicate which "floor" the path must be on when it reaches that number. Whenever the path reaches an elevator, it must change floors, and it cannot change floors except at elevators. Elevators may be marked with an arrow indicating whether the floor increases or decreases at that elevator. The path cannot go below the 1st floor or above the nth floor, where n is the maximum number on the grid. The start and end of the path are designated by 'S' and 'G' on unshaded cells, which may also have a number indicating which floor the path starts or ends on.

We construct an asymmetric required-edge directed T-metacell and a symmetric required-edge undirected T-metacell to show ASP-completeness of Building Walk by Corollary 6.26.

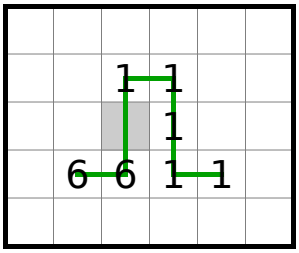


For now we assume each number appears in at most one T-metacell, so that edges cannot be drawn between numbered cells in different T-metacells. The left diagram shows a directed asymmetric T-metacell, which has a required edge on the left and cannot exit through the bottom. Its orientation may be reversed by inverting the arrows on the elevators and applying the involution $x \mapsto 6 - x$ to its numbered cells. The right diagram shows a symmetric required-edge undirected T-metacell.

There is one technical issue with these constructions, which is that two adjacent T-metacells whose required edges both point toward each other must share a number where the required edges connect.

If the two T-metacells' required-edge paths (i.e. the paths labeled "1" in the above diagrams) turn in opposite directions, then we can just label the two paths with the same number. However, if they turn towards the same direction, then this might allow for unintended solutions.

The following diagram shows how to resolve the situation in this case. It shows the boundary between two T-metacells whose required-edge paths (labeled "1" and "6") turn towards the same direction. In this case, replacing two cells of the "6" path with a "1" and an elevator as shown forces the paths to connect properly without allowing extra solutions.

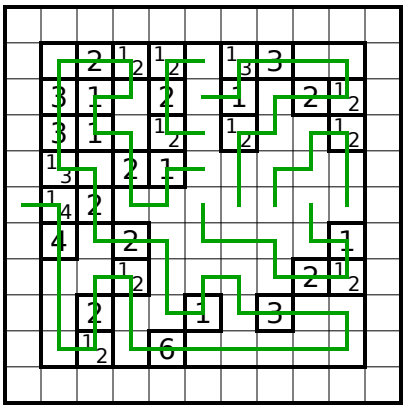


Finally we must also break the loop into a path; this can easily be done by picking any two unshaded cells which are forced to be adjacent in the loop, and labeling one with 'S' and one with 'G'.

Rail Pool

Rail Pool is a full loop genre with regions. For each region, consider all lines that overlap any cell of that region, and take the set of their lengths. This set must match the set of numbers in the region, ignoring duplicates. Unnumbered regions are unconstrained.

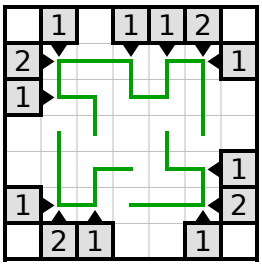
We construct an asymmetric required-edge undirected T-metacell to show ASP-completeness of Rail Pool by Corollary 6.25.



Disorderly Loop

Disorderly Loop is a loop genre. Each clue contains a multiset of numbers and points to a cell. The clue cannot be visited, the cell pointed to must contain a turn, and the multiset contains the lengths of the next n lines along the loop in the direction of the arrow.

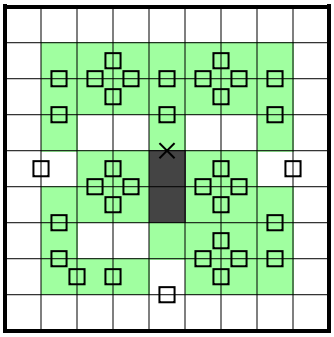
We construct an undirected T-metacell to show ASP-completeness of Disorderly Loop by Corollary 6.24.



Ant Mill

Ant Mill is a shading genre. The goal is to draw a loop composed of dominoes of shaded cells, connected by diagonal adjacency. Some edges have squares, which indicate that the two incident cells are either both shaded or both unshaded. Some edges have X marks, which indicate that exactly one of the two incident cells is shaded.

We construct an undirected T-metacell to show ASP-completeness of Ant Mill by Corollary 6.24.



Koburin

Koburin is a loop genre. Some cells contain numbers, which count the number of orthogonally adjacent unused cells. Numbers cannot be visited, and unused cells cannot be orthogonally adjacent to each other.

There is a straightforward reduction from Hamiltonian cycle in undirected grid graphs, which by Theorem 6.21 is ASP-complete: take the bounding box of the grid graph as a rectangular grid graph, and place a "0" clue in every cell excluded from the original graph. Since the original graph has max degree 3, every cell is adjacent to at least one unused cell, and the clues therefore force every other cell to be visited.

Mukkonn Enn

Mukkonn Enn is a full loop genre. Some cells contain numbers in one of their four quadrants, which counts the length of the line extending from the cell in that direction if a segment in that direction is present. (If no such segment is present, the number can be ignored.)

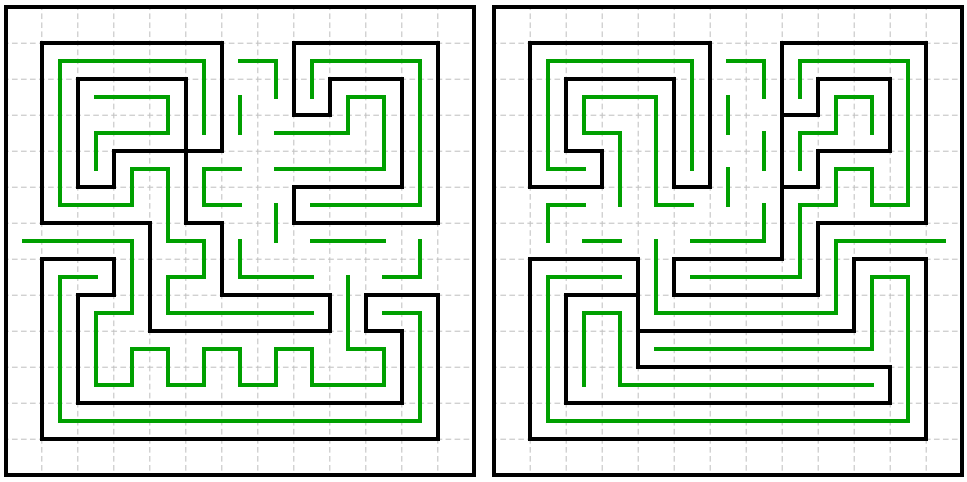
We construct an undirected T-metacell to show ASP-completeness of Mukkonn Enn by Corollary 6.24.

Rassi Silai

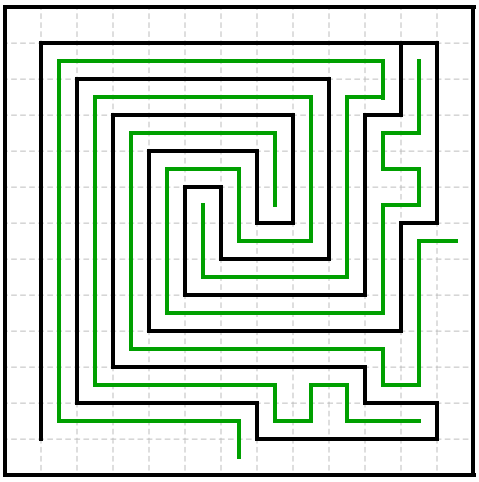
Rassi Silai is a variety genre with regions. Each region must contain exactly one path, which visits all cells in that region. Additionally, no two endpoints of lines can be orthogonally or diagonally adjacent.[3]

We construct an asymmetric required-edge undirected T-metacell to show ASP-completeness of Rassi Silai by Corollary 6.25. Our proof works even without any borders between two cells in the same region.

Note that this T-metacell cannot be reflected, as it relies on the borders present in adjacent cells. So we actually must make both a 'right-handed' and a 'left-handed' version. We break the loop into a path by replacing the top left cell with the fourth image shown.



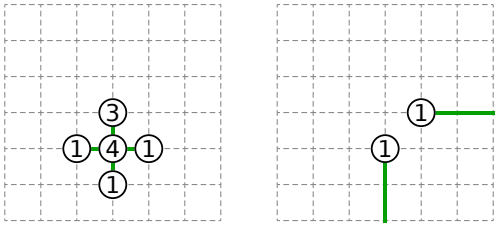
[3]The standard rules of Rassi Silai also allow the grid to contain shaded cells, which cannot be visited, but this makes the reduction trivial from Theorem 6.21.



(Crossing) Ichimaga

Ichimaga is a variety genre. Some gridpoints contain circled numbers, which count the number of adjacent segments used. All drawn segments must form lines that connect two circles and turn at most once. Additionally, the resulting graph of circles joined by lines must be connected. The Crossing Ichimaga variant allows these connecting lines to cross.

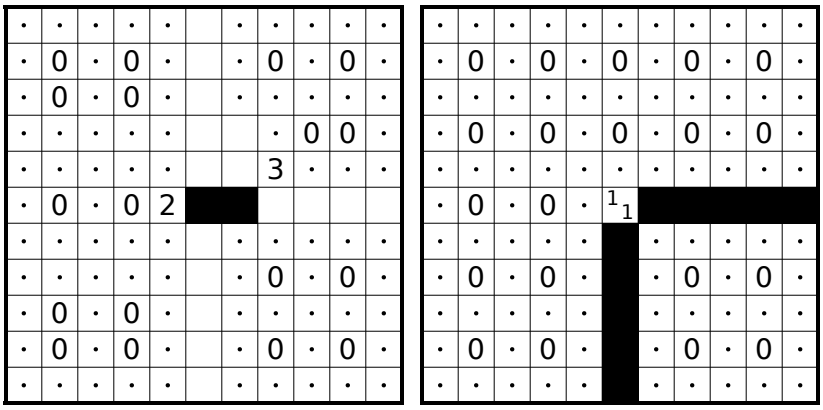
We construct an undirected T-metacell to show ASP-completeness of both versions by Corollary 6.24. Note that due to the global structure of the construction, it is never possible for lines to turn or cross.



Tapa

Tapa is a shading genre. Some cells contain multisets of numbers, which must match the multiset of lengths of contiguous runs of shaded cells in the 8 surrounding cells. Additionally, the shaded cells must be connected, and 2×2 squares of shaded cells are forbidden.

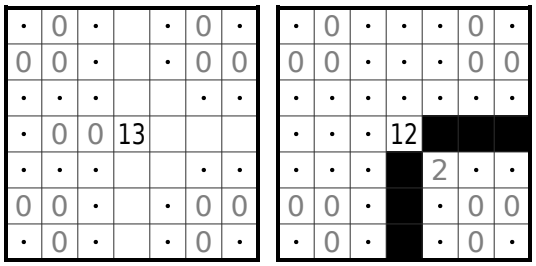
We construct an undirected T-metacell to show ASP-completeness of Tapa by Corollary 6.24. Note that we must break the loop into a path by replacing the top left cell with the second image shown.



Canal View

Canal View is a shading genre. Some cells contain numbers, which the sum of lengths of runs of shaded cells in all 4 orthogonal directions. Additionally, the shaded cells must be connected, and 2×2 squares of shaded cells are forbidden.

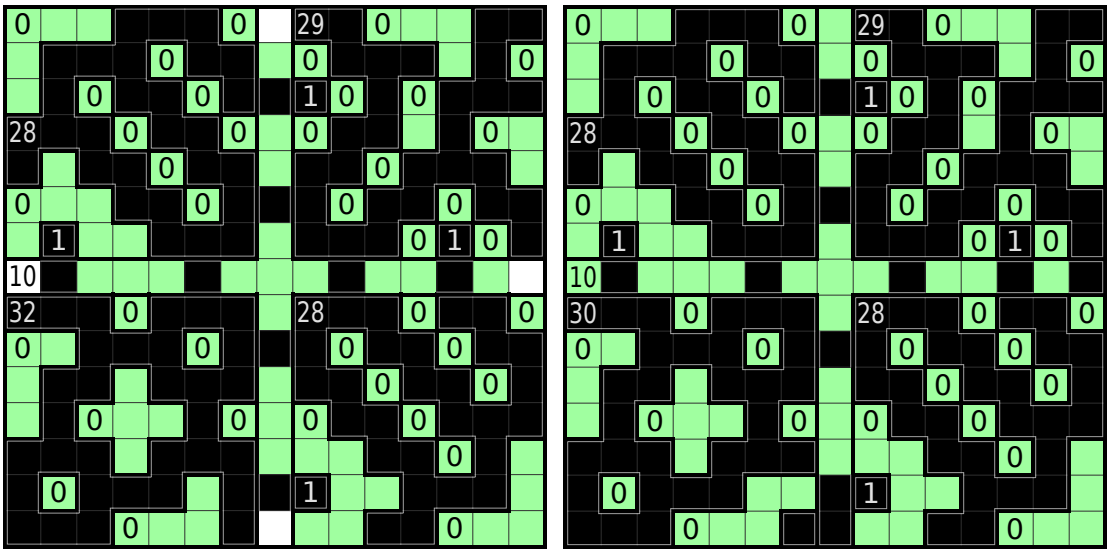
We construct an undirected T-metacell to show ASP-completeness of Canal View by Corollary 6.24. We replace the top left cell with the second image shown to break the loop into a path.



Aqre

Aqre is a shading genre with regions. Some regions contain numbers, which count the number of shaded cells in that region. Additionally, the shaded cells must be connected, and no horizontal or vertical line of 4 cells in a row can be all shaded or all unshaded.

We can embed any max-degree-3 spanning subgraph of a rectangular grid graph in Aqre to get ASP-completeness directly from Theorem 6.20. First, expand every cell into the first image shown to embed a rectangular grid graph. Then, remove the appropriate edges from each metacell by placing "0" clues in any of the 4 undetermined cells. Finally, replace the top left cell with the second image shown to break the loop into a path.

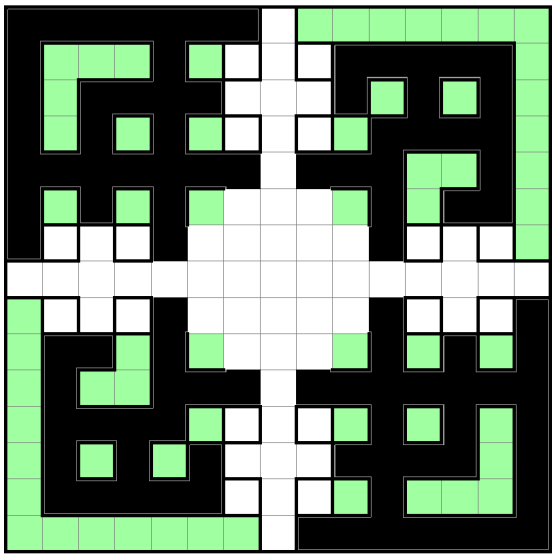


Paintarea

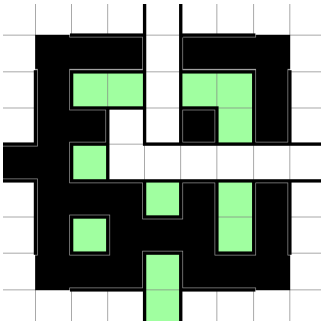
Paintarea is a shading genre. The grid is divided into regions, each of which must be either entirely shaded or entirely unshaded. The shaded cells must be connected, and no 2×2 square may be entirely shaded or entirely unshaded. Numbered cells indicate the number of orthogonally adjacent shaded cells.

We construct an asymmetric required-edge undirected T-metacell to show ASP-completeness of Paintarea by Corollary 6.25. Our proof works even without any numbered cells.

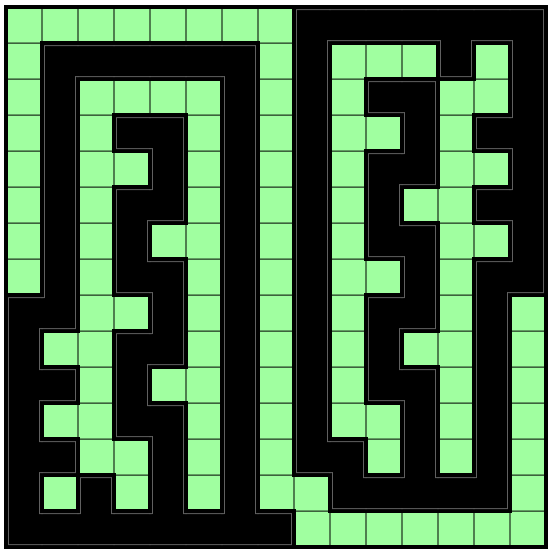
The base for our T-metacell is this 15×15 outer frame, which ensures that shaded cells can connect only in the middle of their shared edges.



The center of the frame is then filled with the following 7×7 core, which can be individually rotated and reflected (while the outer frame is fixed) to produce a T-metacell with the desired orientation.



We also need to break the loop into a path, which is accomplished by using this tile for the top left corner:



6.6.5 Open ASP-completeness Questions

Some puzzle genres were proved NP-complete by Tang, but we have not yet found parsimonious adaptations of the corresponding T-metacells. These genres are Angle Loop, Double Back, Scrin, Icebarn, and Icelom 2.

References

[ABC+20] Zachary Abel, Jeffrey Bosboom, Michael Coulombe, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, Mikhail Rudoy, and Clemens Thielen. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. *Theoretical Computer Science*, 839:41–102, November 2020.

[ABD+05] Esther M. Arkin, Michael A. Bender, Erik D. Demaine, Sándor P. Fekete, Joseph S. B. Mitchell, and Saurabh Sethia. Optimal covering tours with turn costs. *SIAM Journal on Computing*, 35(3):531–566, 2005.

[ADD+20] Sualeh Asif, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, Hayashi Layers, Jayson Lynch, Sarah Scheffler, and Adam Suhl. PSPACE-completeness of pulling blocks to reach a goal. *Journal of Information Processing*, 28:929–941, 2020.

[ADG+21] Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Della Hendrickson, Adam Hesterberg, Matias Korman, Oliver Korten, Jayson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots. In *37th International Symposium on Computational Geometry (SoCG 2021)*, pages 10–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 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.

[AFI+09] Esther M. Arkin, Sándor P. Fekete, Kamrul Islam, Henk Meijer, Joseph S. B. Mitchell, Yurai Núñez-Rodríguez, Valentin Polishchuk, David Rappaport, and Henry Xiao. Not being (super)thin or solid is hard: A study of grid hamiltonicity. *Computational Geometry: Theory and Applications*, 42(6–7):582–605, 2009.

[AFM00] Esther M. Arkin, Sándor P. Fekete, and Joseph S.B. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry: Theory and Applications*, 17(1–2):25–50, 2000.

[ALP18a] Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Tracks from hell – when finding a proof may be easier than checking it. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *Proceedings of the 9th*

International Conference on Fun with Algorithms (FUN 2018), volume 100 of *LIPIcs*, pages 4:1–4:13, La Maddalena, Italy, June 2018.

[ALP18b] Matteo Almanza, Stefano Leucci, and Alessandro Panconesi. Trainyard is NP-hard. *Theoretical Computer Science*, 748:66–76, 2018.

[And09] Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109(19):1145–1146, 2009.

[ANS80] Takanori Akiyama, Takao Nishizeki, and Nobuji Saito. NP-completeness of the Hamiltonian cycle problem for bipartite graphs. *Journal of Information Processing*, 3(2):73–76, 1980.

[BB12] Kevin Buchin and Maike Buchin. Rolling block mazes are PSPACE-complete. *Information and Media Technologies*, 7(3):1025–1028, 2012.

[BBC+23] Jeffrey Bosboom, Josh Brunner, Michael Coulombe, Erik D. Demaine, Della Hendrickson, Jayson Lynch, and Elle Najt. The Legend of Zelda: The complexity of mechanics: Discrete and computational geometry, graphs, and games. *Thai Journal of Mathematics*, 21(4):687–716, 2023.

[BCC+20] Jeffrey Bosboom, Charlotte Chen, Lily Chung, Spencer Compton, Michael Coulombe, Erik D. Demaine, Martin L. Demaine, Ivan Tadeu Ferreira Antunes Filho, Della Hendrickson, Adam Hesterberg, Calvin Hsu, William Hu, Oliver Korten, Zhezheng Luo, and Lillian Zhang. Edge matching with inequalities, triangles, unknown shape, and two players. *Journal of Information Processing*, 28:987–1007, 2020.

[BCC+23] Josh Brunner, Lily Chung, Michael Coulombe, Erik D. Demaine, Timothy Gomez, and Jayson Lynch. Complexity of solo chess with unlimited moves. *arXiv preprint arXiv:2302.01405*, 2023.

[BCD+20] Josh Brunner, Lily Chung, Erik D. Demaine, Della Hendrickson, Adam Hesterberg, Adam Suhl, and Avi Zeff. 1 x 1 Rush Hour with fixed blocks is PSPACE-complete. In *10th International Conference on Fun with Algorithms (FUN 2021)*, pages 7–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020.

[BDD+20] Jeffrey Bosboom, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, Hayashi Layers, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In *10th International Conference on Fun with Algorithms (FUN 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[BM87] Samuel W. Bent and Udi Manber. On non-intersecting Eulerian circuits. *Discrete Applied Mathematics*, 18(1):87–94, 1987.

[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 Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2689–2708. SIAM, 2019.

[CD23] Lily Chung and Erik D. Demaine. Celeste is PSPACE-hard: Discrete and computational geometry, graphs, and games. *Thai Journal of Mathematics*, 21(4):671–686, 2023.

[CDD+22] Lily Chung, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, Hayashi Layers, and Jayson Lynch. Pushing blocks via checkable gadgets: PSPACE-completeness of Push-1F and Block/Box Dude. In *11th International Conference on Fun with Algorithms (FUN 2022)*, pages 3–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2022.

[CDD+23] Michael Coulombe, Erik D. Demaine, Jenny Diomidova, Timothy Gomez, Della Hendrickson, Hayashi Layers, and Jayson Lynch. Complexity of motion planning of arbitrarily many robots: Gadgets, petri nets, and counter machines. In *2nd Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2023)*, pages 5–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2023.

[CG94] Adam Chalcraft and Michael Greene. Train sets. *Eureka*, 53:5–12, 1994.

[CHP82] Gerard Cornuéjols, David Hartvigsen, and William Pulleyblank. Packing subgraphs in a graph. *Operations Research Letters*, 1(4):139–143, 1982.

[Cul97] Joseph C. Culberson. Sokoban is PSPACE-complete. *IEICE Technical Report*, 1997.

[dBK10] Mark de Berg and Amirali Khosravi. Optimal binary space partitions in the plane. In *Computing and Combinatorics: 16th Annual International Conference, COCOON 2010, Nha Trang, Vietnam, July 19-21, 2010. Proceedings 16*, pages 216–225. Springer, 2010.

[DDD18] Andreas Darmann, Janosch Döcker, and Britta Dorn. The monotone satisfiability problem with bounded variable appearances. *International Journal of Foundations of Computer Science*, 29(06):979–993, 2018.

[DDH+23] Erik D. Demaine, Jenny Diomidova, Della Hendrickson, Hayashi Layers, and Jayson Lynch. Traversability, reconfiguration, and reachability in the gadget framework. *Algorithmica*, 85(11):3453–3486, 2023.

[DDHO03] Erik D. Demaine, Martin L. Demaine, Michael Hoffmann, and Joseph O'Rourke. Pushing blocks is hard. *Computational Geometry*, 26(1):21–36, 2003.

[DDM+24] Marnix Deurloo, Mitchell Donkers, Mieke Maarse, Benjamin G Rin, and Karen Schutte. Hamiltonian paths and cycles in NP-complete puzzles. In *12th International Conference on Fun with Algorithms (FUN 2024)*, pages 11–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[DDO00] Erik D. Demaine, Martin L. Demaine, and Joseph O'Rourke. PushPush and Push-1 are NP-hard in 2D. *arXiv preprint cs/0007021*, 2000.

[DGK+17] Jérôme Dohrau, Bernd Gärtner, Manuel Kohler, Jiří Matoušek, and Emo Welzl. Arrival: A zero-player graph game in NP ∩ coNP. In *A journey through discrete mathematics*, pages 367–374. Springer, 2017.

[DGL17] Erik D. Demaine, Isaac Grosof, and Jayson Lynch. Push-pull block puzzles are hard. In *International Conference on Algorithms and Complexity*, pages 177–195. Springer, 2017.

[DGLR18] Erik D. Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *9th International Conference on Fun with Algorithms (FUN 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[DH08] Erik D. Demaine and Robert A. Hearn. Constraint Logic: A uniform framework for modeling computation as games. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity*, pages 149–162, June 2008.

[DHH02] Erik D. Demaine, Robert A. Hearn, and Michael Hoffmann. Push-2-F is PSPACE-complete. In *CCCG*, pages 31–35, 2002.

[DHH04] Erik D. Demaine, Michael Hoffmann, and Markus Holzer. PushPush-k is PSPACE-complete. In *Proceedings of the 3rd International Conference on FUN with Algorithms*, pages 159–170, 2004.

[DHHL23] Erik D. Demaine, Robert A. Hearn, Della Hendrickson, and Jayson Lynch. PSPACE-completeness of reversible deterministic systems. *International Journal of Foundations of Computer Science*, pages 1–22, 2023.

[DHL20] Erik D. Demaine, Della Hendrickson, and Jayson Lynch. Toward a general complexity theory of motion planning: Characterizing which gadgets make games hard. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[DHLL23] Erik D. Demaine, Della Hendrickson, Hayashi Layers, and Jayson Lynch. Trains, games, and complexity: 0/1/2-player motion planning through input/output gadgets. *Theoretical Computer Science*, 969:113945, 2023.

[DL25] Zachary DeStefano and Bufang Liang. Push-1 is PSPACE-complete, and the automated verification of motion planning gadgets. *arXiv preprint arXiv:2508.17602*, 2025.

[DLL18] Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. The computational complexity of portal and other 3d video games. In *9th International Conference on Fun with Algorithms (FUN 2018)*, pages 19–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

[DLMT16] Erik D. Demaine, Jayson Lynch, Geronimo J. Mirano, and Nirvan Tyagi. Energy-efficient algorithms. In *Proceedings of the 7th Annual ACM Conference on Innovations in Theoretical Computer Science (ITCS 2016)*, pages 321–332, Cambridge, Massachusetts, January 14–16 2016.

[DO92] Arundhati Dhagat and Joseph O'Rourke. *Motion planning amidst movable square blocks*. PhD thesis, Smith College, Northampton, Mass., 1992.

[DR17] Erik D. Demaine and Mikhail Rudoy. Hamiltonicity is hard in thin or polygonal grid graphs, but easy in thin polygonal grid graphs. arXiv:1706.10046, 2017.

[DR18] Erik D. Demaine and Mikhail Rudoy. Tree-Residue Vertex-Breaking: a new tool for proving hardness. In *16th Scandinavian Symposium and Workshops on Algorithm Theory (SWAT 2018)*, pages 32–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2018.

[DVW16] Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder/easier than we thought. In *8th International Conference on Fun with Algorithms (FUN 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[DZ99] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[FGMS20] John Fearnley, Spencer Gordon, Ruta Mehta, and Rahul Savani. Unique end of potential line. *Journal of Computer and System Sciences*, 114:1–35, 2020.

[FGMS21] John Fearnley, Martin Gairing, Matthias Mnich, and Rahul Savani. Reachability switching games. *Logical Methods in Computer Science*, 17(2):10:1–10:29, 2021.

[For10] Michal Forišek. Computational complexity of two-dimensional platform games. In *International Conference on Fun with Algorithms*, pages 214–227. Springer, 2010.

[Fra17] Michael P. Frank. Asynchronous ballistic reversible computing. In *Proceedings of the IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8, Washington, DC, November 2017.

[Fra20] Michael P Frank. Fundamental physics of reversible computing–an introduction. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2020.

[Fri02] Erich Friedman. Pearl puzzles are NP-complete. Manuscript, August 2002. https://erich-friedman.github.io/papers/pearl.pdf.

[FS06] Tomás Feder and Carlos Subi. On Barnette's conjecture. *Electronic Colloquium on Computational Complexity (ECCC)*, 01 2006.

[FT82] Edward Fredkin and Tommaso Toffoli. Conservative logic. *International Journal of theoretical physics*, 21(3):219–253, 1982.

[GAD+26] MIT Hardness Group, Zachary Abel, Erik D Demaine, Jenny Diomidova, Jeffery Li, and Zixiang Zhou. Planar graph orientation frameworks, applied to kplumber and polyomino tiling. *arXiv preprint arXiv:2603.03488*, 2026.

[GBC+24] MIT Hardness Group, Josh Brunner, Lily Chung, Erik D. Demaine, Della Hendrickson, and Andy Tockman. Asp-completeness of Hamiltonicity in grid graphs, with applications to loop puzzles. In *12th International Conference on Fun with Algorithms (FUN 2024)*, pages 23–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[GBC+25] MIT Hardness Group, Josh Brunner, Lily Chung, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Pushing blocks without fixed blocks via checkable gizmos: Push-1 is PSPACE-complete. *arXiv preprint arXiv:2508.19759*, 2025.

[GBD+24] MIT Hardness Group, Josh Brunner, Erik D. Demaine, Jenny Diomidova, Timothy Gomez, Markus Hecher, Frederick Stock, and Zixiang Zhou. Easier ways to prove counting hard: A dichotomy for generalized #SAT, applied to constraint graphs. In *35th International Symposium on Algorithms and Computation (ISAAC 2024)*, pages 51–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[GDD+25] MIT Gadgets Group, Erik D. Demaine, Jenny Diomidova, Timothy Gomez, Markus Hecher, and Jayson Lynch. Hardness of traversing gadget systems with small bandwidth. In *4th Symposium on Algorithmic Foundations of Dynamic Networks (SAND 2025)*, pages 11–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2025.

[GDH+24] MIT Hardness Group, Erik D. Demaine, Lilly Hall, Matias Korman, and Hayashi Layers. PSPACE-hard 2D Super Mario games: Thirteen doors. In *12th International Conference on Fun with Algorithms (FUN 2024)*, pages 21–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[GDH+25] MIT Hardness Group, Erik D. Demaine, Lilly Hall, Ricardo Ruiz, and Naveen Venkat. You can't solve these Super Mario Bros. levels: Undecidable Mario games. *Theoretical Computer Science*, page 115549, 2025.

[GHH+18] Bernd Gärtner, Thomas Dueholm Hansen, Pavel Hubáček, Karel Král, Hagar Mosaad, and Veronika Slívová. ARRIVAL: next stop in CLS. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *LIPIcs*, pages 60:1–60:13, Prague, Czech Republic, July 2018.

[GHH21a] Bernd Gärtner, Sebastian Haslebacher, and Hung P Hoang. A subexponential algorithm for ARRIVAL. *Leibniz International Proceedings in Informatics (LIPIcs)*, 198:69, 2021.

[GHH+21b] Aster Greenblatt, Oscar I Hernandez, Robert A Hearn, Yichao Hou, Hiro Ito, Minwoo Kang, Aaron Williams, and Andrew Winslow. Turning around and around: Motion planning through thick and thin turnstiles. In *CCCG*, pages 377–387, 2021.

[GHR95] Raymond Greenlaw, H James Hoover, and Walter L Ruzzo. *Limits to parallel computation: P-completeness theory*. Oxford university press, 1995.

[GHT24] MIT Hardness Group, Della Hendrickson, and Andy Tockman. Complexity of planar graph orientation consistency, promise-inference, and uniqueness, with applications to Minesweeper variants. In *12th International Conference on Fun with Algorithms (FUN 2024)*, pages 25–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024.

[GJ02] Michael R. Garey and David S. Johnson. *Computers and intractability*, volume 29. wh freeman New York, 2002.

[GJS74] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the 6th Annual ACM Symposium on Theory of Computing*, pages 47–63, Seattle, Washington, 1974.

[GJT76] Michael R. Garey, David S. Johnson, and R. Endre Tarjan. The planar Hamiltonian circuit problem is NP-complete. *SIAM Journal on Computing*, 5(4):704–714, 1976.

[Gol77] Leslie M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *ACM SIGACT news*, 9(2):25–29, 1977.

[Gol78] E. Mark Gold. Complexity of automaton identification from given data. *Information and control*, 37(3):302–320, 1978.

[HD09] Robert A. Hearn and Erik D. Demaine. *Games, puzzles, and computation*. CRC Press, 2009.

[Hen21] Della Hendrickson. Gadgets and gizmos: A formal model of simulation in the gadget framework for motion planning. Master's thesis, Massachusetts Institute of Technology, 2021.

[HL18] Kaiying Hou and Jayson Lynch. The computational complexity of finding Hamiltonian cycles in grid graphs of semiregular tessellations. In Stephane Durocher and Shahin Kamali, editors, *Proceedings of the 30th Canadian Conference on Computational Geometry (CCCG 2018)*, pages 114–128, Winnipeg, Canada, August 2018.

[HLH03] Jeffrey R. Hartline and Ran Libeskind-Hadas. The computational complexity of motion planning. *SIAM review*, 45(3):543–557, 2003.

[HS04] Markus Holzer and Stefan Schwoon. Assembling molecules in ATOMIX is hard. *Theoretical Computer Science*, 313(3):447 – 462, 2004. Algorithmic Combinatorial Game Theory.

[HSS84] John E. Hopcroft, Jacob Theodore Schwartz, and Micha Sharir. On the complexity of motion planning for multiple independent objects; PSPACE-hardness of the "warehouseman's problem". *The international journal of robotics research*, 3(4):76–88, 1984.

[II22] Chuzo Iwamoto and Tatsuya Ide. Moon-or-Sun, Nagareru, and Nurimeizu are NP-complete. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 105(9):1187–1194, 2022.

[IPS82] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, 1982.

[ISI12] Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. NP-completeness of two pencil puzzles: Yajilin and Country Road. *Utilitas Mathematica*, 88, June 2012.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations*, page 85, 1972.

[Kar17] Karthik C. S. Did the train reach its destination: The complexity of finding a witness. *Information Processing Letters*, 121:17–21, 2017.

[Kay00] Sadie Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[KKR18] Alexandr Kazda, Vladimir Kolmogorov, and Michal Rolínek. Even delta-matroids and the complexity of planar boolean csps. *ACM Transactions on Algorithms (TALG)*, 15(2):1–33, 2018.

[Kri75] Mukkai S Krishnamoorthy. An NP-hard problem in bipartite graphs. *ACM SIGACT News*, 7(1):26–26, 1975.

[KT15] Shohei Kanehiro and Yasuhiko Takenaga. Satogaeri, Hebi, and Suraromu are NP-complete. In *Proceedings of the 3rd International Conference on Applied Computing and Information Technology/2nd International Conference on Computational Science and Intelligence*, pages 46–51, 2015.

[KVS+] Daisuke Kobayashi, Robert Vollmert, Lennard Sprong, et al. puzz.link. https://puzz.link.

[Lar93] Philippe Laroche. Planar 1-in-3 satisfiability is NP-complete. *Comptes Rendus de L Academie des Sciences Serie I-Mathematique*, 316(4):389–392, 1993.

[Lay23] Hayashi Layers. Unsimulability, universality, and undecidability in the gizmo framework. Master's thesis, Massachusetts Institute of Technology, 2023.

[Lic82] David Lichtenstein. Planar formulae and their uses. *SIAM Journal on Computing*, 11(2):329–343, 1982.

[Loe93] Martin Loebl. Gadget classification. *Graphs and Combinatorics*, 9(1):57–62, 1993.

[LOT03] Maciej Liśkiewicz, Mitsunori Ogihara, and Seinosuke Toda. The complexity of counting self-avoiding walks in subgraphs of two-dimensional grids and hypercubes. *Theoretical Computer Science*, 304(1):129–156, 2003.

[Lyn20] Jayson Lynch. *A Framework for Proving the Computational Intractability of Motion Planning Problems*. PhD thesis, Massachusetts Institute of Technology, 2020.

[Maa19] Mieke Maarse. The NP-completeness of some lesser known logic puzzles. Bachelor's thesis, Utrecht University, June 2019.

[Min20] MiniBetrayal. Rule 110 train Turing machine explained. https://www.youtube.com/watch?v=Ubc7iNZ7MV8, Feb 2020.

[MR08] Wolfgang Mulzer and Günter Rote. Minimum-weight triangulation is NP-hard. *Journal of the ACM (JACM)*, 55(2):1–29, 2008.

[ÖRNY19] Ş. K. Özdemir, S. Rotter, F Nori, and L. Yang. Parity–time symmetry and exceptional points in photonics. *Nature Materials*, 18:783–798, 2019.

[Pap94] Christos H. Papadimitriou. On the complexity of the parity argument and other inefficient proofs of existence. *Journal of Computer and System Sciences*, 48(3):498–532, 1994.

[Pil19] Alexander Pilz. Planar 3-sat with a clause/variable cycle. *Discrete Mathematics & Theoretical Computer Science*, 21(Discrete Algorithms), 2019.

[Ple79] Ján Plesník. The NP-completeness of the Hamiltonian cycle problem in planar digraphs with degree bound two. *Information Processing Letters*, 8(4):199–201, April 1979.

[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, 1984.

[Rit10] Marcus Ritt. Motion planning with pull moves. *arXiv preprint arXiv:1008.2952*, 2010.

[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. The complexity of satisfiability problems. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 216–226, 1978.

[Set02] Takahiro Seta. The complexities of puzzles, Cross Sum, and their Another Solution Problems (ASP). Senior thesis, University of Tokyo, 2002.

[SSvR11] Allan Scott, Ulrike Stege, and Iris van Rooij. Minesweeper may not be NP-complete but is hard nonetheless. *Mathematical Intelligencer*, 33(4), 2011.

[Ste94] Ian Stewart. A subway named Turing. *Scientific American*, 271(3):104–107, 1994.

[Sza09] Jácint Szabó. Good characterizations for some degree constrained subgraphs. *Journal of Combinatorial Theory, Series B*, 99(2):436 – 446, 2009.

[Tan20] Hadyn Tang. On the NP-completeness of satisfying certain path and loop puzzles. arXiv:2004.12849, 2020. https://arXiv.org/abs/2004.12849.

[Tan22] Hadyn Tang. A framework for loop and path puzzle satisfiability NP-hardness results. *arXiv preprint arXiv:2202.02046*, 2022.

[TB22] Alex Thieme and Twan Basten. Minesweeper is difficult indeed! technology scaling for Minesweeper circuits. In *A Journey from Process Algebra via Timed Automata to Model Learning: Essays Dedicated to Frits Vaandrager on the Occasion of His 60th Birthday*, pages 472–490. Springer, 2022.

[TFAF19] Ivan Tadeu Ferreira Antunes Filho. *Characterizing Boolean satisfiability variants*. PhD thesis, Massachusetts Institute of Technology, 2019.

[TH11] Tatsuie Tsukiji and Takeo Hagiwara. Recognizing the repeatable configurations of time-reversible generalized Langton's ant is PSPACE-hard. *Algorithms*, 4(1):1–15, 2011.

[Tut46] W. T. Tutte. On Hamiltonian circuits. *Journal of the London Mathematical Society, Series 1*, 21(2):98–101, 1946.

[Tut54] W. T. Tutte. A short proof of the factor theorem for finite graphs. *Canadian Journal of Mathematics*, 6:347–352, 1954.

[TW11] Mu-Tsun Tsai and Douglas B. West. A new proof of 3-colorability of Eulerian triangulations. *Ars Mathematica Contemporanea*, 4(1):73–77, 2011.

[USO17] Akihiro Uejima, Hiroaki Suzuki, and Atsuki Okada. The complexity of generalized pipe link puzzles. *Journal of Information Processing*, 25:724–729, 2017.

[vdZ15] Tom C. van der Zanden. Parameterized complexity of graph Constraint Logic. In *10th International Symposium on Parameterized and Exact Computation (IPEC 2015)*, pages 282–293. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015.

[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.

[Vig14] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

[Vig15] Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

[Wig92] Avi Wigderson. The complexity of graph connectivity. In Ivan M. Havel and Václav Koubek, editors, *Proceedings of the 17th International Symposium on Mathematical Foundations of Computer Science (MFCS 1992)*, pages 112–132, Prague, Czechoslovakia, 1992.

[Wil88] Gordon Wilfong. Motion planning in the presence of movable obstacles. In *Proceedings of the fourth annual symposium on Computational geometry*, pages 279–288, 1988.

[Yat00] Takayuki Yato. On the NP-completeness of the Slither Link puzzle. *IPSJ SIGNotes ALgorithms*, 74:25–32, 2000.

[YS03] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences*, E86-A(5):1052–1060, 2003. Also IPSJ SIG Notes 2002-AL-87-2, 2002.

[Zha19] Zhujun Zhang. A note on hardness frameworks and computational complexity of xiangqi and janggi. *arXiv preprint arXiv:1904.00200*, 2019.