Exhaustive Search and Hardness Proofs for Games

by

Jeffrey Bosboom

B.S., University of California, Irvine (2011)
S.M., Massachusetts Institute of Technology (2014)

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

Doctor of Philosophy in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2020

Author .....
Department of Electrical Engineering and Computer Science
August 28, 2020

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

Exhaustive Search and Hardness Proofs for Games

by

Jeffrey Bosboom

Submitted to the Department of Electrical Engineering and Computer Science
on August 28, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

This thesis explores several games from two perspectives: exhaustive search and hardness proofs. First, we present an exhaustive search for hardness proofs: a system for finding motion planning simulations. Second, we prove that the pencil-and-paper puzzle Tatamibari is NP-complete, a proof developed using a Tatamibari solver we wrote based on the Z3 SMT solver. Third, we find by computer search that the board game Push Fight played on a board with one column (four squares) removed is a draw. Then we prove that mate-in-1 in generalized Push Fight is NP-complete and that determining the winner of a game in progress is PSPACE-hard. Fourth, we prove that path puzzles are NP-complete, ASP-complete, and #P-complete. We describe a solver for path puzzles based on depth-first search that solves 14 of 15 puzzles from the last chapter of the path puzzles book. Fifth, we present a nonogram solver based on automaton intersection. Relatedly, we prove that finding an optimal automaton intersection ordering is PSPACE-hard. Sixth, we analyze puzzles from the video game The Witness and obtain NP-completeness for most clue types and Σ_2-completeness for puzzles containing antibody clues. Finally, we propose a generic framework for parsing screenshots of grid-based video games.

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

Acknowledgments

I would like to thank my advisor, Erik Demaine, for personal as well as academic support during my time as his student.

I would like to thank my previous advisor, Saman Amarasinghe, for teaching me how to do research, and for gracefully letting go when my interests shifted.

I would like to thank everyone who participated in the open problem solving sessions that accompanied the two iterations of Erik's course on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.890 Fall 2014 and 6.892 Spring 2019). My experiences in 6.890 are a large part of why I switched fields.

Finally, I would like to thank my parents for their patience and support.

Contents

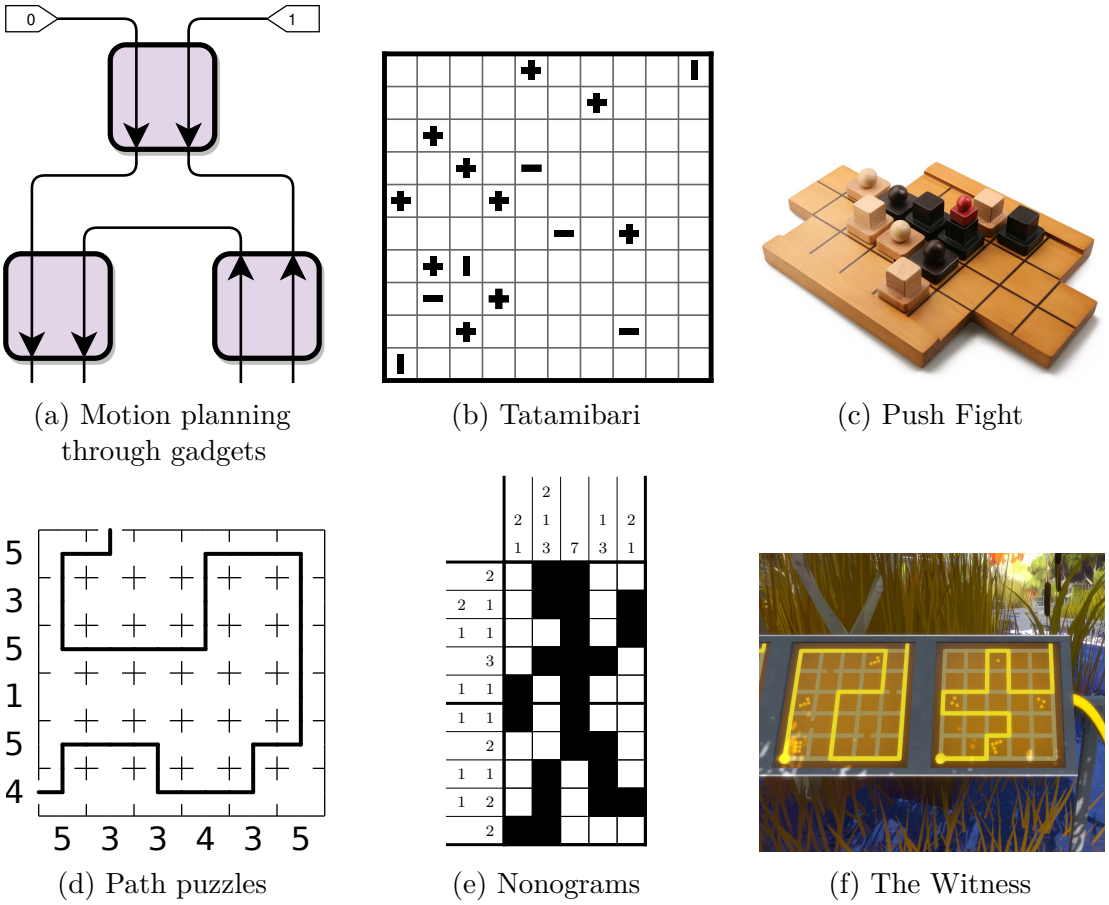
Chapter 1

Introduction

This thesis explores several games from two perspectives: exhaustive search and hardness proofs. Table 1.1 summarizes our results. This thesis is organized into chapters each considering a particular game (by row in the table), but this section introduces games and summarizes results by topic (by column in the table).

Games	Exhaustive search	Hardness proofs
Motion planning through gadgets (Chapter 2)	Exhaustive search for hardness proofs (Chapter 2)	
Tatamibari (Chapter 3)	Tatamibari solver using Z3 SAT solver (Section 3.4)	NP-hardness (Section 3.3)
Push Fight (Chapter 4)	Strongly solved reduced-board variant (Section 4.5)	Mate-in-1 NP-hardness (Section 4.3) 2-player game PSPACE-hardness (Section 4.4)
Path Puzzles (Chapter 5)	Solver based on depth-first search (Section 5.4)	NP-hardness ASP-completeness #P-completeness (Section 5.3)
Nonograms (Chapter 6)	Solver based on automaton intersection (Section 6.3)	NP- and PSPACE-hardness of automaton intersection (Section 6.4)
The Witness (Chapters 7 and 8)	Partial solver for the Challenge, inspiration for screenshot parsing (Section 8.1)	NP-hardness Σ_2-hardness PSPACE-hardness (Chapter 7)

Table 1.1: Summary of results in this thesis.



(a) Motion planning through gadgets

(b) Tatamibari

(c) Push Fight

(d) Path puzzles

(e) Nonograms

(f) The Witness

Figure 1-1: Games considered in this thesis.

1.1 Games

This thesis considers the following games shown in Figure 1-1, ranging from abstract motion planning and formula satisfaction to pencil-and-paper puzzles to two-player board games:

Motion planning through gadgets. The motion-planning-through-gadgets framework is an abstract model of reachability problems in which an agent is traversing a graph of gadgets, where each gadget has local state controlling how it can be traversed and possibly changing when traversed. The framework aims to characterize which types of gadgets make for hard reachability problems. This framework is a tool for proving hardness of games by constructing gadgets in those games.

Tatamibari. Tatamibari is a pencil-and-paper puzzle introduced in Puzzle Communication Nikoli volume 107 [108]. A Tatamibari puzzle is a rectangular grid with clues in some cells. The objective is to exactly cover the grid with rectangles such that the single clue in each rectangle matches the rectangle's relative aspect ratio (wider than it is tall, taller than it is wide, or square) and no four rectangles' corners meet at a

point.

Push Fight. Push Fight is an abstract board game for two players designed by Brett Picotte [115]. Each player has five pieces, three kings and two pawns. On each turn, the player moves up to two of their pieces along a simple path, then must push a line of pieces with one of their kings. A player loses when one of their pieces is pushed off the board or if they are unable to push on their turn.

Path puzzles. Path puzzles, designed by Roderick Kimball [86], are pencil-and-paper puzzles played on a rectangular grid with numerical clues assigned to the rows and columns of the grid. The objective is to draw a path through the grid between two designated endpoints such that the path occupies a number of cells in each row or column equal to that row or column's clue.

Nonograms. Nonograms are pencil-and-paper puzzles played on a rectangular grid where each row and column has a sequence of numerical clues. The objective is to fill some squares of the grid so that the sequence of lengths of contiguous filled squares in each row and column matches the corresponding clues.

The Witness. The Witness is a first-person adventure video game designed by Jonathan Blow (also known for designing Braid) whose primary mechanic is solving two-dimensional puzzles of a style similar to pencil-and-paper puzzles. The objective is always to draw a path on a grid from a circle to a semicircle, but puzzles contain a variety of other clues that constrain the path.

1.2 Exhaustive Search for Hardness Proofs

We developed an exhaustive search for hardness proofs in the motion-planning-through-gadgets framework. Our system searches for simulations of gadgets by graphs built from a set of given gadgets. When the system finds a simulation of a known-hard gadget by a set of gadgets S, we obtain a proof that gadget set S is also hard. Our system uses deterministic finite automata as specifications for the behavior of a gadget or gadget network. Instead of explicitly constructing all gadget networks up to a particular size, our system searches directly on specifications, enabling efficient canonicalization and deduplication. Simulations found by our system have been featured in a published paper on motion planning hardness [16].

1.3 Exhaustive Search

Tatamibari. We developed a Tatamibari solver based on the SAT solver Z3 [42]. We used this solver to check the correctness of our gadgets during the development of our hardness proofs, resulting in the development of a feature specifically for proof

development, checking for three-corner violations near reflex corners of nonrectangular puzzles.

Push Fight. We developed a system to strongly solve Push Fight, that is, to determine the winner under perfect play from every board position. Unlike most systems to solve games, we do not use retrograde analysis; instead our system repeats a forward search until it reaches a fixed point. We find that Push Fight played on a board with one column (four squares) removed is a draw under perfect play. We believe standard Push Fight is within reach of our current system given sufficient computation time.

Path puzzles. We developed a path puzzle solver based on depth-first search. We tested our solver against 15 puzzles from the last, hardest chapter of the path puzzles book [86]; our solver solves all but one puzzle and solves 10 of the 15 puzzles in less than a second.

Nonograms. We developed a nonogram solver based on computing the intersection of deterministic finite automata. We assign a position in the string to each cell of the grid, then build automata for each row and column that constrain the positions corresponding to the cells in that row or column. We then intersect all the clause automata and read off the solutions from the resulting automaton. Unfortunately, we find that our solver is not competitive with state-of-the-art solvers. We also find that the order of the automata intersections has a substantial impact on performance.

The Witness. The Challenge is an end-game area of The Witness containing a timed series of randomized puzzles. We wrote a game-playing program that can automatically solve most of the panel puzzles in the Challenge, though human input is required to navigate through the three-dimensional environment. The solver itself simply searches over all possible paths to find one that satisfies all the clues; the interesting part of the program is the puzzle parser that extracts the clues from screenshots. Our parser is entirely ad hoc, relying on hardcoded lists of colors and pixel coordinates. Our experience writing this program and several other game-playing programs leads us to propose a generic framework for parsing screenshots of grid-based puzzle games (see Chapter 8).

1.4 Hardness Proofs

Tatamibari. We prove Tatamibari is NP-complete by reduction from planar rectilinear monotone 3SAT. Our proof is an instance of a general framework for hardness proofs involving local gadgets whose behavior is characterized by area coverage.

Push Fight. We analyze both the problem of deciding whether the current player can win this turn (the mate-in-1 problem) and the problem of deciding the winner of

a game in progress for Push Fight generalized to larger boards with more pieces. If the number of moves per turn is a constant or unlimited, the mate-in-1 problem is in P, but if it is a variable provided as part of the problem instance, the problem is NP-complete. We prove that deciding the winner of a generalized Push Fight game is PSPACE-hard; it remains open whether it is in PSPACE or EXPTIME-hard.

Path puzzles. We prove path puzzles are NP-complete, ASP-complete, and #P-complete by parsimonious reduction from numerical three-dimensional matching.

Nonograms. We prove that finding an optimal automaton intersection order, that is, an order to intersect a set of given automata that minimizes the maximum size of the intermediate automata, is either NP-complete or PSPACE-hard, depending on the magnitude of the size limit, and hard to approximate. These hardness results are a fundamental barrier to improving on our automata-intersection approach to solving nonograms.

The Witness. We analyze many combinations of clues in The Witness, obtaining NP-completeness for most clue sets and Σ_2-completeness for some clue sets containing antibody clues (which eliminate another clue in their region and are satisfied only if this elimination was necessary). We also analyze the metapuzzles in The Witness, obtaining some PSPACE-completeness results. Along the way, we give polynomial-time algorithms for puzzles containing broken edges and only monomino clues or only hexagon clues.

Chapter 2

Exhaustive Search for Motion Planning Simulations[1]

2.1 Introduction

In the motion-planning-through-gadgets framework [47, 49, 16], we want to know whether an agent can traverse a network of gadgets to reach a goal location (*reachability*). Each gadget in the graph has local finite state that determines how the agent can traverse it. The complexity of the reachability problem depends on the types of gadgets allowed. In many cases, even one simple type of gadget results in the reachability problem being NP- or PSPACE-hard. These hard abstract motion-planning problems have formed the basis of proofs of hardness of concrete problems, such as winning in video games [47, 10, 16, 18] and reconfiguration in robotic motion planning [22, 36].

We can prove the hardness of a set of gadget types by constructing networks of those gadgets that simulate gadgets known to be hard. For example, a long chain of simulations shows that parallel and antiparallel 2-toggles simulate 2-toggle-locks [47]. In this way, we can gradually grow the set of gadgets known to have hard reachability problems, giving us more options for further hardness proofs.

The gadget simulations in [47] were constructed manually by drawing a gadget network on a whiteboard, checking whether it allows precisely the desired set of transitions, repeatedly adding or removing gadgets from the network until it behaves as desired, and finally proving correctness of the simulation. This process is laborious and error-prone. We would instead prefer to tell a computer what gadgets we have available and the gadget we wish to simulate and make the computer do the hard work.

It turns out that we can: this chapter describes a system for computer search for gadget simulations. We specify a gadget for the computer by writing a finite automaton whose language is the set of allowed sequences of traversals through the gadget. The key insight that makes the computer search practical is that the search occurs over the space of gadget specifications, by directly composing specifications using automata operations, instead of the space of gadget networks. In particular,

[1] This chapter is joint work with Erik D. Demaine and Jayson Lynch.

automata can be minimized and canonicalized cheaply, making it easy to discard duplicate constructions. Only once the search has found a simulation are the search operations replayed in gadget network space to produce the simulating gadget network.

Figure 2-1 gives an example of using the system to find a gadget network built from parallel 2-toggles (P2Ts) and unconstrained degree-3 vertices (3-splits) that simulates an antiparallel 2-toggle (A2T). A *2-toggle* is a gadget with two tunnels that can each only be traversed by the agent in one direction; when either tunnel is traversed, both tunnels flip direction. This behavior is specified in the transition tables for the two gadgets in Figures 2-1b and 2-1e, which give rise to the automata in Figures 2-1c and 2-1f. The search system finds a chain of automata operations that transform the automaton specifying P2T into the automaton specifying A2T. The output of the search system is the textual trace shown in Figure 2-1g; a separate visualization script replays the trace in gadget network space and automatically lays out the resulting network, resulting in Figure 2-1h. In this case, the search system has found the same simulation given in [47, Figure 13] (with the figure rotated 90∘).

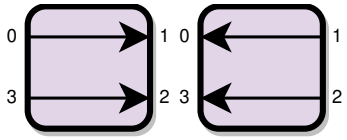
Section 2.2 describes the combine and connect operations our system uses to construct gadget networks. Section 2.3 describes the use of automata as gadget specifications, including how to perform combine and connect directly on automata. Section 2.4 describes the implementation of our search system, which performs a breadth-first search with parallel worker processes. Section 2.5 describes our visualization script.

In Section 2.6 we present simulations found by the system. First, we present simulations that are simpler than the hand-constructed simulations in [47]. Second, we present the subset of the simulations used to prove PSPACE-hardness of planar networks of door gadgets from [16] that were found using our system. Third, we investigate which output-disjoint input-output gadgets simulate crossovers, discharging [18]'s assumption of crossovers for a majority of gadgets. We also discuss the computational resources used to obtain these results.

2.1.1 Definitions

A *gadget* has a set of *locations* and a set of *states*. Each state specifies a set of *traversals* between locations the agent is allowed to make. Each traversal specifies the agent's entry and exit locations and the state of the gadget subsequent to the traversal. A gadget only changes state as the result of the agent traversing it. A *gadget network* is a collection of gadget instances with specified states where some locations of the instances are connected in a matching (following the definition in [47]). The agent can move from one location in a matched pair to the other, which does not change the state of any gadget. A gadget network having unmatched locations can be viewed as a gadget whose locations are the network's unconnected locations, whose set of states is a subset of the cross product of its component gadgets' states, and whose traversals are the transitive closure of the traversals of its component gadgets along with movements via the matching. Viewing a gadget network as a gadget effectively hides the network's internal states.

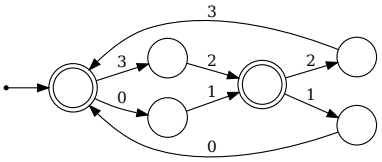
A *planar gadget* is a gadget that specifies a circular ordering of its locations. We



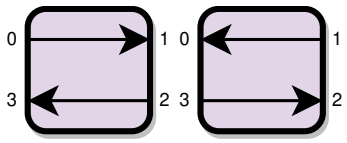
(a) Parallel 2-toggle (P2T)

state	in	out	next
0	0	1	1
0	3	2	1
1	1	0	0
1	2	3	0

(b) P2T transition table



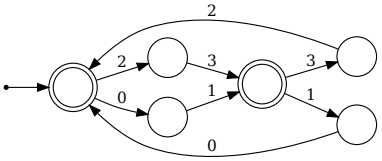
(c) Automaton specifying P2T



(d) Antiparallel 2-toggle (A2T)

state	in	out	next
0	0	1	1
0	2	3	1
1	1	0	0
1	3	2	0

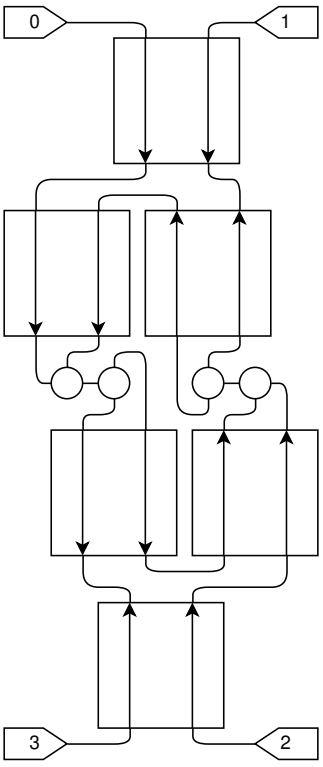
(e) A2T transition table



(f) Automaton specifying A2T

```
Target trace:
  217 = source {toggle-toggle-antiparallel
      -0-0}
Source trace:
  13 = source {3-split}
  224 = source {toggle-toggle-parallel}
  5131515 = combine 224 splice 0 with 13
      rotate 0 connect at 2 @0
  5131549 = connect 5131515 at 4 @0
  5131583 = combine 5131549 splice 0 with 13
      rotate 0 connect at 2 @0
  5134583 = combine 5131583 splice 1 with 224
      rotate 2 connect at 4 @3
  5149050 = connect 5134583 at 3 @3
  5221683 = combine 5149050 splice 1 with 224
      rotate 2 connect at 4 @3
  5446388 = combine 5221683 splice 0 with 13
      rotate 0 connect at 2 @0
  6593329 = connect 5446388 at 6 @0
  10347296 = combine 6593329 splice 0 with 13
      rotate 0 connect at 2 @0
  24814049 = combine 10347296 splice 1 with
      224 rotate 2 connect at 4 @3
  26628863 = connect 24814049 at 5 @5
  26809409 = connect 26628863 at 1 @0
  54722434 = combine 26809409 splice 0 with
      224 rotate 1 connect at 7 @3
  63906004 = connect 54722434 at 2 @2
  120322655 = combine 63906004 splice 0 with
      224 rotate 1 connect at 7 @3
  217 = connect 120322655 at 2 @0
```

(g) Search trace for simulating A2T by P2T



(h) Automatic visualization of search trace: a gadget network simulating A2T by P2T

Figure 2-1: Searching for a simulation of an antiparallel 2-toggle (A2T) by parallel 2-toggles (P2T) and unconstrained vertices

consider planar gadgets differing only by rotation of locations to be identical, but planar gadgets differing by reflection to be distinct. A *planar gadget network* is a gadget network of planar gadgets that specifies a combinatorial planar embedding of its gadgets and whose unconnected locations (if any) are on the exterior face of the network.

We use definitions of equivalence and simulation based on those in [47]. A *consecutive traversal sequence* is a sequence of traversals where the entry location of each traversal after the first is equal to the exit location of the previous traversal. Two gadgets are *equivalent* if there is a bijection between their locations, a partition of the states of each gadget, and a bijection between the groups of the partitions such that, for every consecutive traversal sequence of one gadget, and for every starting state for the other gadget in the corresponding group, there exists a consecutive traversal sequence of the other gadget such that the state of the gadgets and the location of the agent before and after the sequence correspond according to the bijections relabeling the locations and state groups, and considering states in the same group to be equivalent. For planar gadgets, the location bijection must preserve the cyclic order of the locations. A source gadget or set of source gadgets *simulates* a target gadget if there exists a gadget network consisting entirely of instances of the source gadget(s) that, when viewed as a gadget, is equivalent to the target gadget.

2.1.2 Related Work

Motion planning. An early investigation of motion planning through abstract gadgets is the door gadget metatheorem of [10], obtaining PSPACE-completeness for reachability in nonplanar networks of door gadgets. The first systematic study of a family of gadgets is the characterization of deterministic reversible 2-state gadgets on tunnels [47], again obtaining PSPACE-completeness and showing that all gadgets in this class simulate each other. A later paper analyzes gadgets whose transition graph is a directed acyclic graph along with deterministic reversible gadgets and obtains partial and full characterizations of 1-player, 2-player, and team reachability games on these these gadgets [49, 48].

Returning to the door gadget from [10], Ani et al. [16] analyze the possible planar embeddings of the door gadget, showing all but one are PSPACE-complete; some of the reductions in [16] were obtained through the search described in this chapter (see Section 2.6.2). Most recently, the complexity of 0-player reachability on deterministic output-disjoint input-output gadgets was classified [18]; we use our computer search to discharge the assumption of a crossover gadget/nonplanar gadget network (see Section 2.6.3).

Search for gadgets in other domains. While computer case analysis has been used to prove most of some theorems (e.g., the Four-Color Theorem, in 1977 [20, 21] and again in 1997 [118]), computer search for gadgets for reductions has concentrated on Boolean satisfiability and constraint satisfaction problems, especially for (in)approximability. One line of research is about finding a definition of a gadget [25, 137] that is simultaneously precise but general enough to cover many problems.

Based on this definition, Trevisan et al. [137] use linear programming to prune the gadget search space, leading to both new approximations and new inapproximability results. This technology does not cover all constraint problems (e.g., Wiman [147] can only iteratively approximate the linear program), much less to all search problems, though there is a recent proposal to extend to all search problems via category theory [100].

For a broader view of computer search for small examples, see [146]; in particular, the proposal to search for minimal Boolean circuits has a similar flavor to gadget search.

Canonicalization We depend crucially on the ability to quickly canonicalize an automaton over rotations of its alphabet (representing locations on the boundary of a planar gadget). If we were to consider automata as general directed graphs with labeled edges, we would have to compute a canonical labeling, which is at least as hard as graph isomorphism. Instead, after minimizing the automata using Hopcroft's algorithm for DFA minimization (implemented following Knuutila's exposition [88]), we can run a linear-time canonicalization algorithm [111], taking advantage of the fact that our automata already have a distinguished state (the initial state).

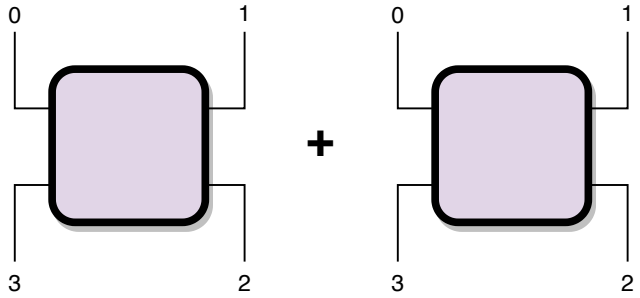
2.2 Building Gadget Networks with Combine and Connect

Our gadget search is based on two operations on planar gadget networks, combine and connect. The combine operation merges two networks and the connect operation connects locations within a single network. Together, combine and connect are sufficient to construct any planar gadget network. (Section 2.7.2 discusses modifications to search for nonplanar networks.)

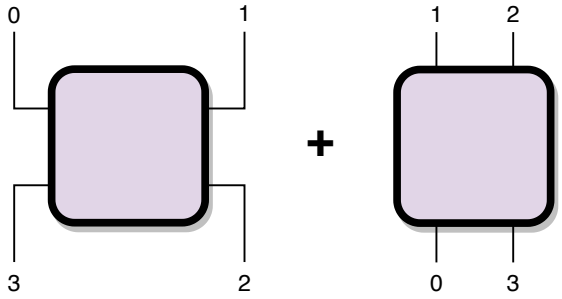
Combine. The combine operation combines two gadgets into a single gadget network (see Figures 2-2a to 2-2c). Combine has four operands: the left and right gadgets, the rotation applied to the right gadget's locations, and the index at which the right gadget's locations are spliced into the left gadget's location sequence. If the left gadget has m locations and the right gadget has n locations, the combined gadget has $m + n$ locations. The right gadget has n rotations and its locations can be spliced in at any of the m positions in the left gadget's location sequence, so there are mn ways to combine a given pair of gadgets.

Connect. The connect operation takes one input gadget and connects two adjacent locations to each other. There are n ways to combine a gadget with n locations, and each output has $n - 2$ locations. See Figure 2-2d.

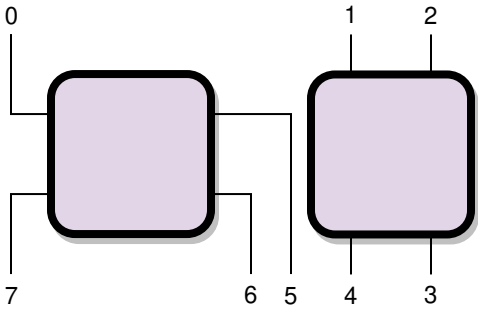
Universality. Next we show that combine and connect suffice to build any planar gadget network. Intuitively, we can find a valid ordering of combine operations by



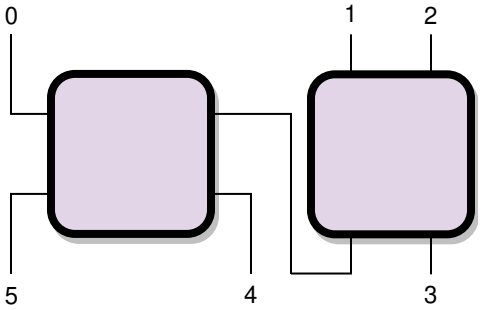
(a) The inputs to the combine operation, two four-location gadgets (or gadget networks). In this example combine, the right gadget will be rotated left one location and its locations spliced after location 0 in the left gadget's location sequence.



(b) As the first step of this combine operation, the right gadget is rotated.



(c) Then the right gadget's locations are spliced after location 0 in the left gadget's location sequence. This is the result of the combine operation.



(d) One possible connect operation on the resulting gadget network is to connect locations 4 and 5.

Figure 2-2: An example of the combine and connect operations.

topologically sorting the adjacency graph of gadgets (two gadgets are adjacent if a location of one is connected to a location of the other), and between each pair of combine operations we connect locations of the gadget just added to the existing gadgets it is connected to in the network we are building. More formally, we prove the universality of combine and connect by induction while deconstructing gadget networks.

Theorem 2.2.1. *Given a set S of base gadgets, any planar gadget network consisting only of gadgets in S can be constructed using a sequence of combine and connect operations, where at least one operand of every combine operation is a base gadget.*

Proof. We proceed by deconstructing gadget networks using the inverse of combine and connect until we reach base gadgets. Given an arbitrary planar gadget network, either it is a base gadget or a network of multiple gadgets. If it is a base gadget, we are done. Otherwise, choose a base gadget having a location incident to the outside face of the network and disconnect (the inverse operation of connect) all of its locations from the other locations in the gadget network. Because that base gadget is on the outside face, the now-disconnected locations are consecutive in the network's unmatched location ordering, so we can remove the base gadget from the network (the inverse operation of combine). The remaining gadget network has one fewer gadget, so this deconstruction procedure will terminate in a number of steps equal to the number of gadgets in the network minus one. Then reversing this sequence of inverse-combine and inverse-connect operations gives a sequence of combine and connect operations that constructs the gadget network, as desired. □

Search? So if combine and connect are universal for building gadget networks, can we use them as the basis of a search algorithm for simulations? We run into a problem when trying to decide whether a gadget network built by the search is the simulating network we are searching for. The obvious way to check whether two gadget networks are equivalent is a depth- or breadth-first search through the network from every input location in each combination of states of each of the network's gadgets to see whether they have the same behavior (possibly after a relabeling of locations or states). The sequence of combines and connects that builds a network is usually not unique, and to avoid duplicate work during the search we want to maintain a closed set of all the networks we have built so far and deduplicate each newly-built gadget against this closed set. But even deciding whether two networks are identical is equivalent to checking graph isomorphism, and to use a hash table for our closed set, we need to find a canonical labelling of the network.

What we need is a concise specification of a gadget's behavior that is easy to compute and to canonicalize. In the next section, we find that automata are the gadget specifications we need, and that we can perform combine and connect operations directly on specifications.

2.3 Automata as Gadget Specifications

The use of automata as gadget specifications is most easily explained through a specific example. The parallel 2-toggle (P2T) drawn in Figure 2-1a has four locations, which we will number circularly 0 to 3, and two states. In the state drawn on the left, the agent can traverse the gadget from location 0 to location 1 and from location 3 to location 2 (and in no other ways), in both cases leaving the gadget in the state on the right, in which the agent can traverse from location 1 to location 0 and location 2 to location 3, both returning to the left state. Numbering the left state 0 and the right state 1, the previous sentence is summarized by the transition table in Figure 2-1b.

We can write traversal sequences as strings of pairs of entry and exit locations. For example, the agent can traverse a P2T in state 0 from location 0 to location 1, then location 2 to location 3, then location 0 to location 1 again, which we can write as the string 012301. Then we can define the language of traversal sequences of a gadget (which we will call the language of the gadget) as the set of all such strings representing traversals of that gadget. Most interesting gadgets, including the P2T, have infinite languages of traversals because they can be traversed infinitely many times, but, for example, a single-use path gadget would have a finite language.

It turns out that the infinite language of the gadget is regular, so can be specified by a finite automaton accepting that language. We can build such an automaton from the transition table. In a minimal automaton, each gadget state corresponds to an accepting state of the automaton. The automaton's initial state corresponds to the initial gadget state, so we get one automaton per state of the gadget. The automaton accepting the language of the P2T is shown in Figure 2-1c.

One or two symbols per traversal. We could alternately represent each gadget traversal with a single symbol encoding both the entry and exit location. The alphabet size of the resulting automaton is the square of the gadget's location size, but the automaton has no non-accept states so its state size is smaller. In practice, we decided to use two-symbol traversals because our automata library represents a transition as a pair of the next state and a bitset of the symbols proceeding to that state, a representation that assumes the alphabet size is small.

Our gadget search operates by iterations of combining two gadgets, then connecting some of their locations. Besides combine and connect, we have three additional operations on gadget specifications that serve to merge equivalent gadgets, reducing the amount of work done by the search: taking the transitive closure of the gadget's traversals, and canonicalizing the automaton. The following subsections describe the automata operations that realize these basic search operations.

2.3.1 Combine

Given two gadgets, a right gadget rotation amount, and a left gadget location splice index (the four operands of combine), the combined gadget can be traversed in any interleaving of the traversals of the input gadgets, appropriately renumbered based

on the rotation and splice point. There are no traversals entering the left gadget but leaving the right gadget or vice versa, so the language of the combined gadget is a pair-granularity interleaving of the languages of the input gadgets. For example, if 0123 is a traversal sequence of the left gadget and 4567 is a traversal sequence of the right gadget (after location renumbering), then 01234567, 45670123, 01452367, and 01456723 are all traversal sequences of the combined gadget, but 04567123 is not (because there is no traversal entering location 0 and exiting location 4).

At the automaton level, this pairwise interleaving results in a combined automaton having states $Q_L \times Q_R \times \{L, R\}$, where Q_L and Q_R are the states of the left and right input automaton and $\{L, R\}$ is a flag indicating which automaton we are currently simulating. We allow the simulation to switch automata using ε-transitions between (q_l, q_r, L) and (q_l, q_r, R) when both q_l and q_r are accepting, in which case the combined automaton state is accepting; because the input automata are bipartite between accepting and non-accepting states, this ensures pairs of symbols cannot be split up. The combined automaton begins with ε-transitions to states S_L, S_R, L and S_L, S_R, R, where S_L and S_R are the initial states of the left and right input automaton, because the first traversal can be in either gadget.

2.3.2 Connect

Given a gadget and a location b to connect to the gadget's next location c, whenever a traversal would exit the gadget at b, the connected gadget must immediately be traversed again from location c, and vice versa. The connected locations b and c become unavailable for traversals from outside the gadget (they are removed from the gadget's location sequence). In terms of the language of the gadget, wherever $abcd$ appears in a traversal string, ad can appear instead. Then all occurrences of b and c are deleted from all traversal strings.

At the automaton level, for every non-accepting state s that has a transition on b to an accept state t having a transition on c to non-accepting state u, we add an ε-transition from state s to state u, and the same with b and c swapped. Then we renumber the alphabet to delete b and c, dropping any transitions on b and c and compressing the remaining symbols to the locations 0 to $n - 2$.

2.3.3 Closure

The transitive closure operation takes a gadget as input and produces the transitively-closed gadget as output. In terms of the language of the gadget, whenever $abcd$ appears in a traversal string for any a, b, c, d, transitive closure allows ad to appear instead. This is similar to connect, but with no restriction on b or c, and without deleting any locations afterward.

At the automaton level, for every non-accepting state s having a transition on any symbol b to an accept state t having a transition on c to non-accepting state u, we add an ε-transition from state s to state u. This is just like the connect implementation, except with no constraints on b and c.

Replacing a gadget with its transitive closure does not give the agent any extra capability to reach the goal. Whenever the agent uses a traversal added by transitive closure, the agent could have reached the same location with the gadget in the same state by simply making multiple traversals of the gadget. For this reason, it is not useful to distinguish gadgets that differ only in traversals added by transitive closure, so our search always transitively closes its base gadgets.

2.3.4 Canonicalization

The search wants to canonicalize automata to efficiently check whether the search has already produced that automaton. We first minimize the automaton using Hopcroft's algorithm (implemented following Knuutila's exposition [88]). Then we run a canonicalization algorithm similar to the full canonical labelling algorithm for site graphs [111], but simplified by taking advantage of the automaton's distinguished initial state.

For each of the permutations that are rotations of the automaton's symbols, we perform a lazy breadth-first search starting from the initial state, processing each transition in order of the permutation. That is, the search using the identity permutation attempts a transition first on symbol 0, then on symbol 1, and so on, while the search using the next permutation attempts a transition first on symbol 1, then symbol 2, coming back to symbol 0 only after trying the rest of the symbols. One of these searches follows the lexicographically least sequence of edges; we apply this permutation to canonicalize the location numbering. Each search records the order in which it first visits the automaton's states, which we take as the canonical labelling of the states. Canonicalization thus takes at most $O(k(n+m))$ time for an automaton with k symbols, n states, and m transitions.

All automata having the same canonical form specify equivalent gadgets. The bijection between locations is given by the rotation permutation and the bijection between groups of gadget states is given by the accepting states in the equivalence classes of automaton states found by minimization.

Nop traversals. Traversals from a location to itself that do not change the state of the gadget are called *nop (no-operation) traversals*. Nop traversals can be introduced into the language of a gadget during connect or transitive closure. Transitively closing a parallel 2-toggle (transition table given in Figure 2-1b), for example, introduces nop traversals for each location: starting in state 0, the agent can traverse from location 0 to 1, then from 1 to 0, so transitive closure introduces a nop traversal from 0 to 0. The output of connect may have locations that only participate in nop traversals or that have no traversals at all; as part of canonicalizing the connect output, we delete these nop-only or unused locations, renumbering further locations as needed to keep the location numbering dense.

Gadgets with nop traversals are essentially the same as gadgets without, but nop traversals affect the edge enumeration used to compute the canonical labelling. To ensure nop traversals do not distinguish gadgets, we add the maximal set of nop traversals (one for each location in each gadget state) to each automaton before

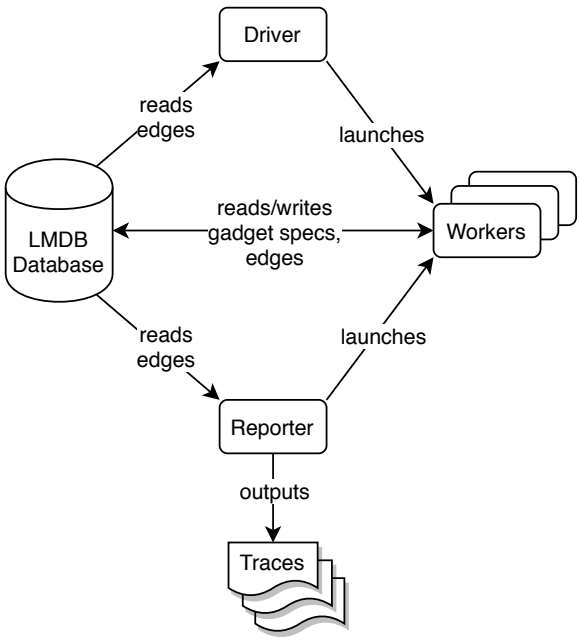


Figure 2-3: Relationships between the parts of the gadget search system.

canonicalizing it. At the automaton level, adding a nop traversal in a gadget state means adding a transition from the accepting automaton state corresponding to that gadget state to a new non-accepting automaton state and adding a transition from the non-accept state back to the accept state.

2.4 The Gadget Search System

The gadget search system consists of three main components:

- the driver, which decides what needs to be computed;
- one or more workers that do the work in parallel;
- and the reporter, which prints traces that witness how a gadget specification was constructed.

The three components communicate through a database storing gadget specifications, edges recording how gadgets combine and connect to form other gadgets, and completion intervals recording what has already been computed.

Input and output. The input to the system is multiple JSON or YAML files containing gadget specifications with human-meaningful names, which are used to seed the database with the specifications to be searched from or found by the search. The format is very close to the transition tables shown in Figures 2-1b and 2-1e, with the addition that pairs of reversible transitions can be abbreviated into "undirected"

```
3-split:
  uedges: [[0, 0, 1, 0], [0, 0, 2, 0], [0, 1, 2, 0]]

toggle-toggle-parallel:
  uedges:
  - [0, 0, 1, 1]
  - [0, 3, 2, 1]
toggle-toggle-antiparallel:
  uedges:
  - [0, 0, 1, 1]
  - [0, 2, 3, 1]

door-0xd12d34:
  uedges:
  - [0, 0, 0, 0]
  dedges:
  - [1, 0, 0, 0]
  - [0, 1, 2, 0]
  - [0, 3, 4, 1]
  - [1, 3, 4, 1]
  state-names: {0: open, 1: closed}
```

Figure 2-4: YAML gadget definitions for 3-split, parallel and antiparallel 2-toggles, and a directed door gadget

transitions. Some of these input files are themselves computer-generated as combinations of simpler gadgets. For example, the parallel and antiparallel 2-toggle gadgets shown in the introduction are part of a family of gadgets "on tunnels" investigated in [47]; the definitions of P2T and A2T, shown in Figure 2-4, are generated by a script that pairs all of the tunnels (including the toggle, tripwire and lock tunnels investigated in [47]) in all possible ways.

The output of the system consists of the traces printed by the reporter that witness how a gadget specification was built using combine, connect, and close operations; see Figure 2-1g for an example. These traces can be replayed directly on gadgets (instead of on gadget specifications) by a separate visualizer script to produce gadget network drawings like Figure 2-1h.

Generational search. The search is a breadth-first search where each vertex is a gadget specification and the edges store the parameters of the operation performed. For example, a connect operation is stored in the database as an edge from an input gadget specification (specified by an integer ID) to an output gadget specification, along with the location being connected (the other is always the circularly next location). A combine operation is stored in the database as a (hyper)edge from its two inputs to its output gadget specification, along with the splice point and right gadget rotation (the two degrees of freedom described in Section 2.3.1), plus the connect location from the immediately following connect operation.

Each generation in this breadth-first search begins with one combine operation,

with all gadget specifications in the frontier being used as the left input, but with the right gadget only being from the initial set of gadget specifications the search started with. The remainder of the generation consists of connect operations (which always shrink their input by at least two locations), and transitive closure operations. This generational structure of combine-then-connect allows the system to make a partial completeness guarantee: if a gadget is simulated by a gadget network consisting of k copies of the input gadgets, that simulation will be found in generation $k-1$ (after $k-1$ combines, each introducing a copy of an input gadget to the network, and the first introducing two).

We also use this generational structure for lack of a distance-to-solution heuristic in specification space. We could define a notion of a difference of specifications, but it is not clear how to use it to guide the search. We experimented with a search structure that put all gadget specifications in a priority queue sorted by size, and operating (combining and connecting) the smallest gadgets first, but in practice most interesting simulations have an intermediate state whose specification is relatively large and complex, so smallest-first is not an effective heuristic. The lack of a good ordering heuristic is a weakness of the current system.

2.4.1 The Database

We use the LMDB key-value store [38, 74] for our database. LMDB is a B+-tree-based database using copy-on-write and multiversion concurrency control (MVCC) to provide full ACID semantics for unlimited parallel readers and a single parallel writer. LMDB provides multiple "subdatabases", separate key namespaces that can be operated on in a single transaction. LMDB is a memory-mapped database, so it provides very high read throughput when the database fits in memory; when the database is larger than memory, read throughput from SSDs is reasonable if there are enough parallel reader threads to use the SSDs at high queue depth. Section 2.4.5 explains how we work around the single-writer limitation.

Early versions of our system used PostgreSQL, which was useful for prototyping because declarative SQL queries are easy to iterate on. However, we had to abandon PostgreSQL because it does not support "select, or insert if not present" queries; additionally, PostgreSQL has high space overhead because data is duplicated in indices, and PostgreSQL's concurrency support was ineffective as most transactions rolled back and were replayed due to serialization failures.

Gadgets. Gadget specifications are stored in the database as byte strings. The first two bytes of the string store the number of locations of the gadget (up to a 16 location maximum) and a bitfield coding the size of the following variable-length fields storing the gadget's number of states, the number of undirected and directed transitions in the following transition lists, and the number of strongly connected components in the gadget's state space.[2] The gadget's transitions are split into undirected (where

[2]For example, a single-use path gadget has two states, one where the path is available and one where it has been used. There is a transition from the first to the second state, but not the other

each transition's reverse transition is also present) and directed groups. Transitions are 4-vectors (start and end states and from and to locations), which we can map into integers by multiplying the coordinates by the number of locations and states. For undirected transitions, only the smaller of the two directions is stored. Each group of integers is sorted and delta-coded, with the deltas stored as variable-length integers using SQLite's scheme.[3] (This delta-varint encoding is similar to how search engine postings lists are compressed.)

Gadget specifications are usually looked up by a 64-bit hash (to determine whether the gadget has been seen before and to insert it if not), but it is also convenient for gadget specifications to have an integer ID for use in edges. Accordingly, gadget specifications are stored using two subdatabases, the gadget hashtable and the gadget index. The hashtable, as its name suggests, maps the hashes of gadget specification byte strings to the byte strings followed by their 8-byte[4] integer ID. The hashtable uses linear probing to resolve collisions. The gadget index maps gadget specification IDs to the hash the gadget specification is stored under in the hashtable. Thus, given a gadget string, we can quickly check whether we've seen this gadget specification before and get its integer ID if so by querying the hashtable, and given a gadget specification ID, we can get the corresponding gadget string by looking up the hash in the gadget index, then using that hash in the gadget hashtable. Both of these databases have 8-byte integer keys, which LMDB provides special support for (`MDB_INTEGERKEY`).

Edges. For each operation, there are many output gadget specifications for each input gadget specification: connecting an n-location gadget can yield up to n distinct gadgets, and combining an m-location gadget with an n-location gadget and connecting the resulting $m+n$ location gadget can yield up to $mn(m+n)$ distinct gadgets. To save space, most edges are stored only in "skinny" form, in subdatabases mapping the input gadget specification ID to a list of output gadget IDs, delta-varint encoded similarly to gadget transitions.[5] Because combine takes two inputs, there is a separate subdatabase for each combine right operand, in which the keys name the left operand. If the reporter finds a skinny edge participates in a trace, the operation is repeated and the resulting full edges (with the parameters of the operation) are stored in separate full edge subdatabases so the parameters can be printed in the trace.

Other subdatabases. The database also contains subdatabases storing completion and predicate interval lists used by the driver, described in Section 2.4.2 and Section 2.4.6. There is also a subdatabase mapping the human-readable gadget names from the input files to gadget specification IDs, used by the driver to parse the initial set of gadgets for a search and by the reporter to print names in traces.

---

direction, so the gadget's state space has two strongly connected components.

[3] https://sqlite.org/src4/doc/trunk/www/varint.wiki

[4] In practice, 5 bytes would suffice for all databases we have computed, but working with 5-byte integers is annoying in most languages.

[5] Compressing these lists of integer IDs is the primary benefit to assigning gadget specification IDs instead of using hashes as identifiers.

2.4.2 The Driver

The driver implements the generational search, maintaining a closed set, current generation, and current subgeneration, all of which are lists of intervals of gadget specification IDs in the set. The generational search does one combine operation followed by repeated connect operations until there are no gadget specifications to connect, so the current generation contains all gadget specifications produced by and since the last combine operation, while the current subgeneration contains only the gadget specifications produced by the last operation of either type (i.e., the inputs to the next connect operation). The search starts from a set of initial gadget specifications specified on the command line when invoking the driver, either as gadget specification IDs or as names. These gadget specifications are transitively closed to form the first subgeneration; they are figuratively "combined against nothing" in generation 0. The first subgeneration is also sorted as the set of combine right inputs to be used for the rest of the search.

During the search, every generation's first subgeneration is produced by combining all gadget specifications in the previous generation as the left operand with all gadget specifications in the set of combine right operands. Subsequent subgenerations are produced by connecting all gadget specifications in the previous subgeneration. Every gadget produced is added to the closed set, and only gadget specifications not already in the closed set are retained in the subgeneration (to avoid repeated work). Connecting always produces a gadget specification with at least two fewer locations, so eventually the subgeneration will be empty, at which point the next generation begins.

The driver does not carry out automata operations directly. Instead it packages the gadget specification IDs of the operands and the path to the database into a MessagePack message and launches a worker process to do the work. The worker process retrieves the gadget specifications from the database, performs the operation, commits the output gadget specifications and edges to the database, and returns a MessagePack message to the driver. The driver then follows edges from the gadget specifications in the (sub)generation to find the produced gadget specification IDs to add to the next subgeneration.

In early versions of our system, the MessagePack messages also included the encoded gadget specification byte strings, allowing the worker processes to run on remote computers or in a batch queueing system. This batch mode of operation is still possible, but we now have sufficiently powerful hardware that it is not necessary.

It is common to run the search for n generations, run the reporter to see what simulations the search found, and then run the search again for $n + 1$ generations. To speed up passing the first n generations, as part of committing an operation result to the database, the worker also commits intervals spanning the input gadget specification IDs to a subdatabase of completion intervals for that operation type. Then before each operation, the driver subtracts the completion intervals from the subgeneration intervals, and only the remaining intervals (if any) need to be operated on. As an additional benefit, storing separate completion information allows us to omit storing edges to the empty or 1-location gadget specifications without causing the operations producing the omitted edges to be repeated.

2.4.3 The Worker

The worker is the component that actually performs automata operations. Each worker process reads a MessagePack message as input that directs it to perform some operation, and produces a MessagePack message as output (besides storing any computed results in the database). The driver and reporter spawn worker processes to perform automata operations, as described in their sections. The worker also performs some utility functions, including initializing new databases (which involves transitively closing and canonicalizing automata from input files).

2.4.4 The Reporter

The reporter prints traces that witness the sequence of operations that transformed the input gadget(s) into an output gadget. To do this, the reporter does a breadth-first search through the edges in the database, recording for each gadget found the source of the edge; that is, the reporter is inverting the graph, saving only one edge per vertex. Starting from the an initial set of gadget specifications in the same way as the driver, the reporter maintains sets (as lists of intervals of gadget specification IDs) of gadget specifications awaiting close edges, connect edges, and combine edges. In that order, the reporter takes the first nonempty set and follows the corresponding edges. For any gadget not already in the reporter's closed set, the reporter adds the inverse of one edge leading to the new gadget to the inverse edge set. Following edge types in this order means the reporter will find a trace with the fewest combine operations. When the trace is replayed in gadget network space, the produced gadget network will contain the smallest number of gadgets.

Whenever a newly found gadget has a name, the reporter prints a trace by following the inverse edges until reaching the gadgets in the initial set. For example, in the trace shown in Figure 2-1g, the found named gadget is gadget 217. Reading from the bottom of the trace up, we can see that gadget 217 was the result of connecting gadget 120322655 at location 2, and that gadget 120322655 was the result of combining gadget 63906004 with gadget 224, and so on. The reporter will print traces for all named gadgets it reaches in its search – we do not have to specify in advance which gadget we are searching for.

The reporter initially follows the edges from the "skinny" edge databases, which store only the input and output gadget specifications to the corresponding operation. Besides saving space in the database, using skinny edges also saves memory in the reporter process; to further save memory, inverse edges are compressed using delta-varint coding. To recover the parameters of the operation (e.g., the connect location), the reporter spawns a worker process that performs the operation with all possible parameters and stores the resulting full edges in the database, which the reporter then retrieves from the database.

2.4.5 Effective Parallelism with LMDB

As described in Section 2.4.1, LMDB is a many-reader, single-writer database. The readers and the writer do not block each other, but on high thread count machines (in our experience, after about 16 threads), writing results back to the database becomes a bottleneck, with multiple workers waiting to begin a write transaction. To work around this limitation, the driver can instruct workers to write their results (gadget specifications and edges) to a temporary file instead of to the database. Gadget specifications in the temporary file are arranged in 256 slices based on the high-order byte of the encoded gadget byte string's hash. Periodically, the driver spawns a worker process that merges these temporary files and writes their results into the database. The writing worker merges files by slice, which results in better locality for LMDB's B+-tree updates; instead of randomly writing across the entire keyspace of the gadget hashtable, writes are localized to one of the 256 slices of the keyspace. While one slice is being committed, other threads in the writing worker read ahead for the next slice by looking up the hashes of the gadget specifications in that slice. Whether the gadgets are already in the database or not, this readahead reduces the number of page faults taken by the writing thread. Once all of the slices have been processed, the edges from each temporary file are committed together after mapping temporary-file-local gadget IDs to the global gadget IDs assigned as the slices were committed.

When the database fits in main memory, the writing worker can usually keep up with the computing workers, but when the database does not fit, the writing worker usually falls behind. Currently our system does not incorporate backpressure, but it would be easy to modify the driver to stop spawning new workers once the pending temporary files reach some size threshold.

2.4.6 Partial Search with Predicate Indices

Empirically, we have found it useful to limit the search space by applying predicates to the list of gadget specifications to be combined or connected in each subgeneration. We can limit the search on any of the statistics stored in the header bytes of the encoded gadget specification byte strings (see Section 2.4.1, Gadgets). To support partial search, the database contains a predicates subdatabase whose keys are a statistic kind and a number, mapped to an interval list of all gadget specifications whose value for that statistic are less than or equal to that number. For example, the key `states<=4` maps to an interval list containing all the IDs of gadget specifications having four or fewer states. To keep these interval lists small, gadget IDs are assigned after sorting the gadget specifications being inserted into the database by their statistics, so that there are fewer, larger intervals in the predicate lists. The predicates subdatabase also stores a special `valid_before` entry whose value is one more than the last ID included in the indices; the indices are extended on demand by the driver whenever a query involves a gadget specification ID greater than or equal to `valid_before`'s value.

When running with a limited search space, all gadget specifications found are stored in the database, but each time the driver computes the set of gadget specifications

needing to be operated on, it applies predicates by intersecting that list of IDs with the interval list from the predicates subdatabase. Only gadget specifications in this intersection are operated on. This limits the partial completeness guarantee (see Generational search). When the limiting predicate is the number of strongly connected components in the gadget's state space, the search is complete regarding targets with fewer than that number of connected components. For example, if the limit on connected components is 1, the search will still find all simulations of gadgets with one connected component in their state space, but may fail to find simulations for gadgets having more components. This is because none of the automata operations can decrease the number of components in the state space. On the other hand, the modifications to the guarantee resulting from limiting the number of locations, states, or edges are harder to visualize. If we limit the number of edges, for example, we fail to find gadgets where every sequence of automata operations had an intermediate gadget specification having more than that number of edges.

Limiting the number of components in the state space to 1 is useful when the gadgets we are hoping to find simulations of have only one component in the state space (which is most gadgets, because we are usually looking for reusable gadgets). We found it useful to limit the number of states and edges, because we observed that gadgets above some complexity limit rarely simplify and automata operations on complex gadgets are more expensive than on simple gadgets. We also limited the number of locations of the left input to a combine operation, usually to the maximum number of locations in the initial set of gadgets plus one (to allow a 3-split gadget to duplicate a location). Many of the planar door simulations presented in Section 2.6.2 were found with searches limited to 8 states, 50 edges, and 7 max combine left locations, in exchange for an increased search depth. But deciding on these limits is entirely empirical. It is possible to repeat a search with extended limits because the completions subdatabase reflects that some gadgets were skipped (not present in the completion interval lists), so we can take an approach of progressively expanding the limits, but we have no way to tell which limit should be expanded or if we should instead decrease the limits to improve the search depth.

2.5 Visualization

The trace is a certificate of simulation of a gadget by a gadget network, but manually interpreting a trace is difficult. To this end, our system includes a visualization script that parses the trace, replays the combine and connect operations in gadget network space, and draws the resulting gadget network. The script maintains location numbers for each gadget (a numbering global to the network), a list of pairs of connected locations, and the numbers of the unmatched locations on the boundary of the gadget network (the network's *interface*). To replay a combine operation, the script recursively builds the gadget networks of the left and right operands,[6] renumbers all locations in the right network to not collide with the left network's numbers, rotates

[6]Combines in traces produced by our system always have a base gadget as their right operand, but our script does not depend on this.

the interface of the right network according to the rotation parameter of the combine operation, and splices the right network's interface into the left's interface at the splice point parameter of the combine operation. To replay a connect operation, the script recursively builds the gadget network of the operand, records the location at the index specified by the combine operation's parameter and its adjacent index as being connected, and deletes them from the interface. After both combine and connect operations, the output network's interface is rotated according to the rotation applied during canonicalization (Section 2.3.4). The result is a gadget network stating which global locations are unconnected (the last output network's interface) and connections between all other locations. By looking through the connection edge list in order of each gadget's location numbers, we obtain a combinatorial embedding of the gadget network (the order of edges around each vertex).

One theoretically appealing option for drawing gadget networks is Tutte embedding [138], but it produces drawings with poor angular resolution . The widely used graph drawing package GraphViz does not preserve the order of edges at a vertex, but edge order is important for drawing gadget networks because locations are ordered. To work around this limitation, we could explode each gadget into a cycle, one vertex per location, use GraphViz to lay out the graph, and then take the average position of the vertices in the cycle as the position of the gadget, but doing this averaging well is not obvious, so we did not attempt this. GraphViz's layout algorithms are also not guaranteed to produce a planar drawing when the graph is planar, which we would like.

Instead of implementing a graph drawing or place-and-route algorithm, we define the problem as a system of linear inequalities and solve the system using the Z3 SMT (satisfiability modulo theories) solver [42]. Each gadget is a small rectangle (8×8 for most gadgets, 2×2 for some simple gadgets including splits, diodes, and fanin). Locations (points on the boundary of each rectangle) are connected by wires, where each wire is composed of a configurable number of line segments constrained to share endpoints. Constraints ensure gadget rectangles and/or wires do not overlap or cross and that the unmatched locations of the network are on the boundary of the drawing. We find minimal-area drawings by repeatedly solving the system and using binary search to find the minimum width and height; e.g., we first solve the system with all objects constrained to be in an 80×80 box, then if that succeeds, we try a 40×40 box, and if that fails we try a 60×60 box, then after finding the smallest box we try to minimize either the height or width leaving the other parameter fixed. Optionally, we minimize the number of wire bends using Z3 soft constraints constraining the wire segments to have zero length.

Despite our system of linear inequalities not encoding any of the combinatorial embedding of the graph that we built while replaying the trace, Z3 produces a usable result within a few minutes even for networks containing up to 15 gadgets when not minimizing bends. When minimizing bends, Z3 is less performant, taking up to an hour. Beyond minimizing area and optionally minimizing bends, our model has no knowledge of what a good graph layout is, but the result is generally interpretable with some effort by a human familiar with the gadgets in the network, and sometimes the solver produces a nice-looking graph like Figure 2-1h.

2.6 Results

In this section we present three types of results obtained through the use of the system. First, we present simulations between reversible deterministic 2-tunnel gadgets that are simpler than the manually-constructed simulations in [47]. Second, we present the simulations used to show PSPACE-hardness of planar door gadgets, previously published in [16]. Third, we present a partial classification of which output-disjoint input-output gadgets build crossovers, discharging [18]'s assumption of crossovers for a majority of gadgets. After presenting these results, we discuss the computational resources used to achieve them.

2.6.1 Reversible Deterministic 2-Tunnel Simulations

A *tunnel* is a gadget building block consisting of a pair of matched locations. A 2-tunnel gadget is a composition of two tunnels sharing the same state; for example, both toggles of a parallel 2-toggle flip direction when either is traversed. There are three nontrivial tunnel types:

- *Toggles* are traversable in a single direction depending on the state of the gadget and flip the state of the gadget when traversed. A toggle is symbolized by an arrow pointing in the direction it is currently traversable in.
- *Tripwires* are always traversable in both directions and flip the state of the gadget when traversed. A tripwire is symbolized by a line with a shorter perpendicular line at its midpoint.
- *Locks* are traversable in both directions in one state (unlocked or open) and not traversable in the other state (locked or closed) and do *not* flip the state when traversed. A lock is symbolized by a line with a circle at its midpoint; the circle is empty when the lock is open and contains an X when the lock is closed.

2-toggle gadgets can be combined in three ways (parallel, antiparallel or crossing); gadgets containing at least one tripwire or lock can be combined in two ways (noncrossing or crossing).

Demaine et al. [47] proves that all nontrivial reversible deterministic 2-tunnel gadgets simulate each other and reachability in networks built from them is PSPACE-complete. In this section, we present simpler simulations found by our search.

2-toggles simulate toggle-lock. Figure 6 of [47] shows a simulation of a noncrossing toggle-lock by parallel, antiparallel, and crossing 2-toggles. Our search found that each type of 2-toggle by itself, plus unconstrained vertices (called "branching hallways" in [47] and "3-split" in our system's traces), suffices to simulate a noncrossing toggle-lock.

Theorem 2.6.1. *The gadget network in Figure 2-5 simulates a noncrossing toggle-lock.*

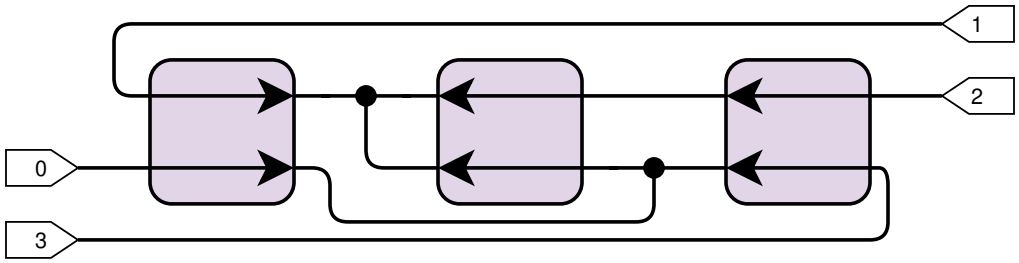


Figure 2-5: Simulation of a noncrossing toggle-lock with parallel 2-toggles.

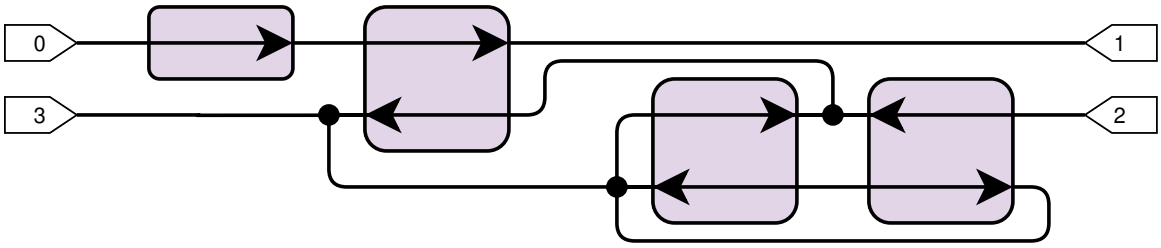


Figure 2-6: 1-player simulation of a noncrossing toggle-lock with antiparallel 2-toggles and one 1-toggle. (Note that a 2-toggle can be used as a 1-toggle by ignoring one tunnel.)

Proof. The state of the toggle-lock is stored in the center 2-toggle; the other two 2-toggles serve to isolate the toggle tunnel from the lock tunnel. In the state shown in Figure 2-5, the toggle points right from location 0 to location 1 and the lock is open between locations 2 and 3. When the agent traverses from location 2 to 3 or vice versa, it passes through two 2-toggles twice, leaving them in the same state before the traversal; the left 2-toggle blocks exiting to location 0 or 1.

If the agent takes the toggle path from location 0 to 1, it traverses the center 2-toggle only once, flipping the state of the toggle-lock. Now the lock path is closed because the center and right 2-toggles point in opposite directions, and the toggle path is traversable from location 1 to location 0. □

Theorem 2.6.2. *The gadget network in Figure 2-6 simulates a noncrossing toggle-lock in one-player.*

Proof. The state of the toggle is stored in the 1-toggle. The lock is open when it is in the state shown in the figure, and otherwise closed. When the toggle path is traversed, the 1-toggle and left 2-toggle flip, flipping the toggle; this also closes the lock because any traversal from location 2 or 3 is blocked at some point by a 2-toggle pointing in the wrong direction.

In the state shown in the figure, the lock path can be traversed in either direction passing through each 2-toggle twice, leaving the network in the same state (so the lock remains open). From location 2, the agent traverses the top tunnel of the right 2-toggle, the bottom tunnel of the left 2-toggle, the bottom tunnel of the right 2-toggle, the bottom tunnel of the center 2-toggle, the top tunnel of the center 2-toggle, the bottom tunnel of the left 2-toggle and the top tunnel of the center 2-toggle before

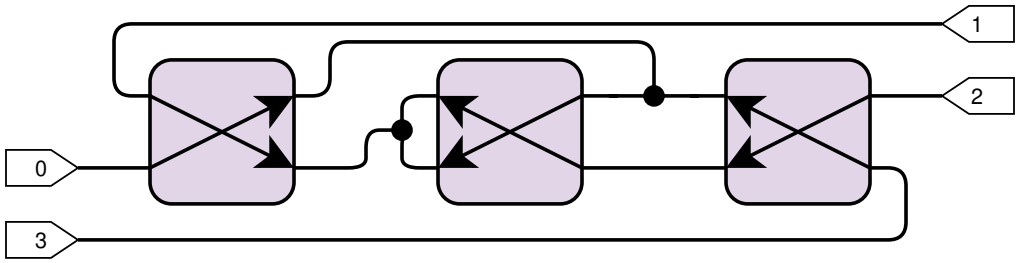


Figure 2-7: Simulation of a noncrossing toggle-lock with crossing 2-toggles.

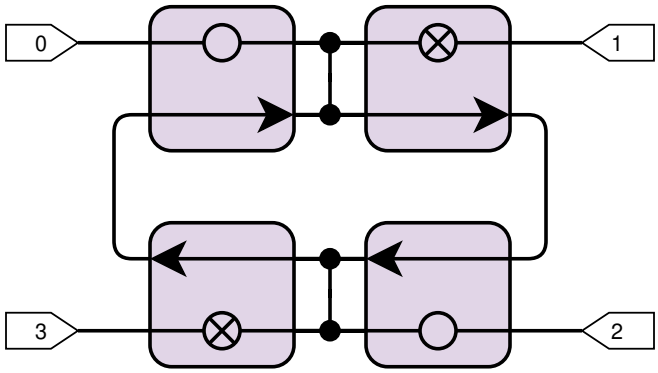


Figure 2-8: Simulation of an antiparallel 2-toggle by noncrossing toggle-locks. Compared to [47, Figure 14], eliminates two 1-toggles.

exiting at location 3. From location 3, the agent traverses the top tunnel of the center 2-toggle, the bottom tunnel of the left 2-toggle, the bottom tunnel of the center 2-toggle, the bottom tunnel of the right 2-toggle, the bottom tunnel of the left 2-toggle and the top tunnel of the right 2-toggle before exiting at location 2. □

Theorem 2.6.3. *The gadget network in Figure 2-7 simulates a noncrossing toggle-lock.*

Proof. This gadget network is just like the network in Figure 2-5 except that the unconstrained vertices have been placed on another face. Because each traversal of the network passes through two 2-toggles, the internal crossing is immediately undone, so the network still simulates a noncrossing 2-toggle.[7] □

Three other simulations. Here are three other simulations, all improved by the elimination of 1-toggles.

Theorem 2.6.4. *The gadget network in Figure 2-8 simulates an antiparallel 2-toggle.*

Proof. The agent can enter the gadget at either location 0 or location 2. Either way, the agent is forced to complete a full cycle through the toggles before being able to leave at location 1 or location 3, respectively. After this traversal, all gadgets have changed state once, and the network is traversable from 1 to 0 or 3 to 2, as desired. □

[7] Crossing 2-toggles can simulate a crossing toggle-lock, but we do not present the simulation here.

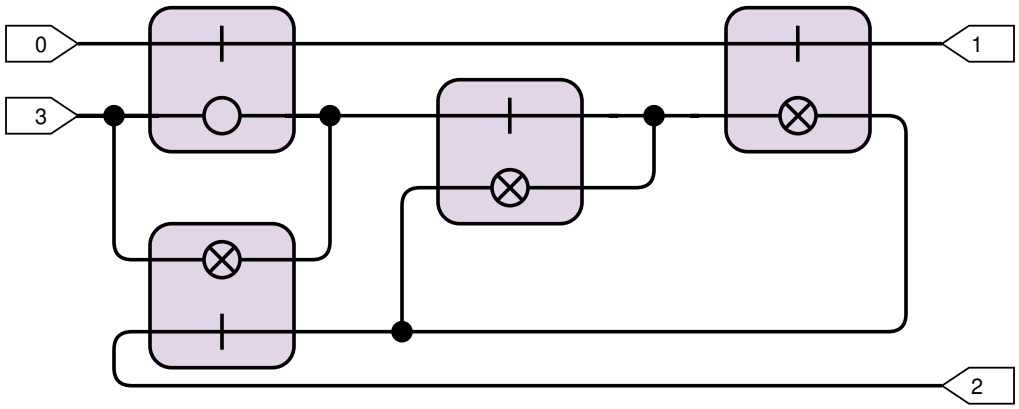


Figure 2-9: Simulation of a tripwire-toggle by noncrossing tripwire-locks. Compared to [47, Figure 12], eliminates one 1-toggle and two crossovers.

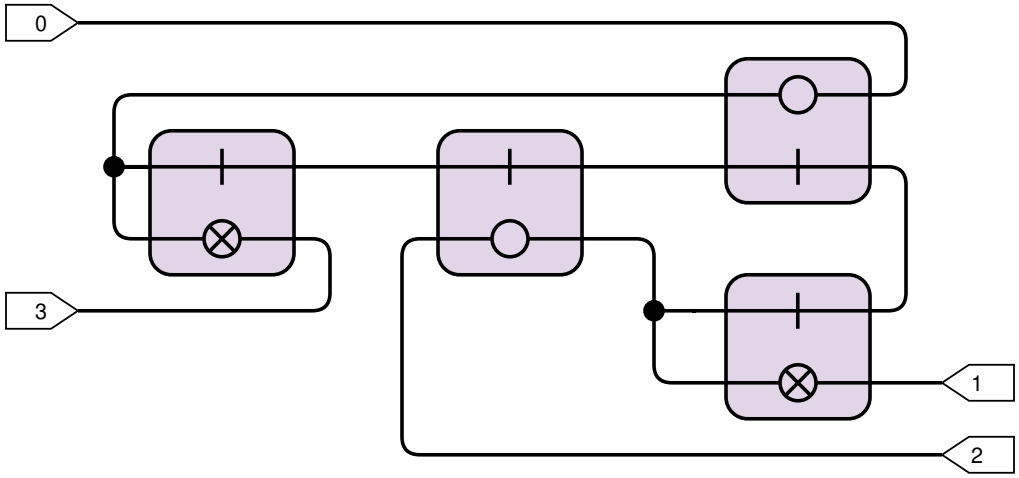


Figure 2-10: Simulation of an antiparallel 2-toggle by noncrossing tripwire-locks. Compared to [47, Figure 16], eliminates two 1-toggles.

Theorem 2.6.5. *The gadget network in Figure 2-9 simulates a noncrossing tripwire-toggle.*

Proof. In the state shown in the figure, the toggle is pointing right from location 3 to location 2; it cannot be traversed in the other direction because the center and right tripwire-locks are locked. After traversal, the middle tripwire-lock is open, so the only the reverse traversal is possible. Thus this path is a toggle, as desired.

The path from location 0 to 1 is the tripwire path. Traversing the tripwire path closes the upper-left tripwire-lock, making the traversal from location 3 to location 2 impossible, and opens the upper-right tripwire-lock, making the traversal from location 2 to location 3 possible. After the latter traversal, the middle tripwire-lock is open, allowing the reverse traversal. Thus the tripwire path flips the state of the gadget, as desired. □

Theorem 2.6.6. *The gadget network in Figure 2-10 simulates an antiparallel 2-toggle.*

Proof. In the state shown in the figure, the upper toggle points right and the lower toggle points left. When traversing from location 0, the agent cannot exit at location 3 because that tripwire-lock is locked until the agent traverses the tripwire, nor can the agent exit at 2 because that tripwire-lock locks when the agent traverses the tripwire, so the only possible exit is at location 1. By a similar argument, if the agent enters at location 2 it can only exit at location 3.

After either traversal, all four tripwire-locks have flipped their state. Then traversals from 0 and 2 are impossible because the first lock they would pass through is locked. Instead traversal is possible from location 1 to location 0 and location 3 to location 2, as desired. □

2.6.2 Planar Directed Door Simulations[8]

A door is a gadget with two states, named open and closed, and three tunnels, named open, traverse and close. Traversing the open or close tunnel sets the door's state to open or closed, respectively. The traverse tunnel is traversable only when the door is in the open state. Doors were introduced in [144] to prove PSPACE-hardness of Lemmings and used soon after in [10] to prove PSPACE-hardness of all three Donkey Kong Country games and The Legend of Zelda: A Link to the Past. These proofs used a separate crossover gadget, so the planar arrangement of the open, traverse and close tunnels is irrelevant. A recent paper on planar door gadgets [16] seeks to dispense with the crossover gadget. After proving that all doors with only undirected tunnels or a mix of directed and undirected tunnels simulate some door with only directed tunnels, Ani et al. [16] proves that 11 of the 12 fully directed doors (shown in Figure 2-11) build a crossover gadget. Three of those cases are trivial; simulations for the other eight cases were found through the use of our computer search, and we re-present those simulations here.

First, we need to introduce two other gadgets that build crossovers that will be the targets of some of our simulations. A self-closing door, introduced in [16], has two states, open and closed, but only two tunnels, open and traverse. The open tunnel sets the door's state to open; the traverse tunnel sets the door's state to closed. Self-closing doors build crossover gadgets [16, Theorem 4.1].

Another gadget that builds a crossover gadget is the ditripwire-dilock, which is like the tripwire-lock presented in the previous section except both tunnels are directed. The parallel ditripwire-dilock builds the antiparallel ditripwire-dilock via a simulation also found by our computer search and presented in the following proof. Together, the parallel and antiparallel ditripwire-dilock build a crossover gadget using the same construction as for the tripwire-lock.

Lemma 2.6.7. *1-player planar motion planning with the parallel directed tripwire-lock is PSPACE-hard.*

Proof. The parallel directed tripwire lock can simulate the antiparallel directed tripwire lock, as in Figure 2-12. Crossing the tripwire in gadget 2 forces the agent to cross

[8] Proofs and figures in this section are from [16] (also available on arXiv [17]), which is joint work with Joshua Ani, Erik D. Demaine, Yevhenii Diomidov, Dylan Hendrickson and Jayson Lynch.

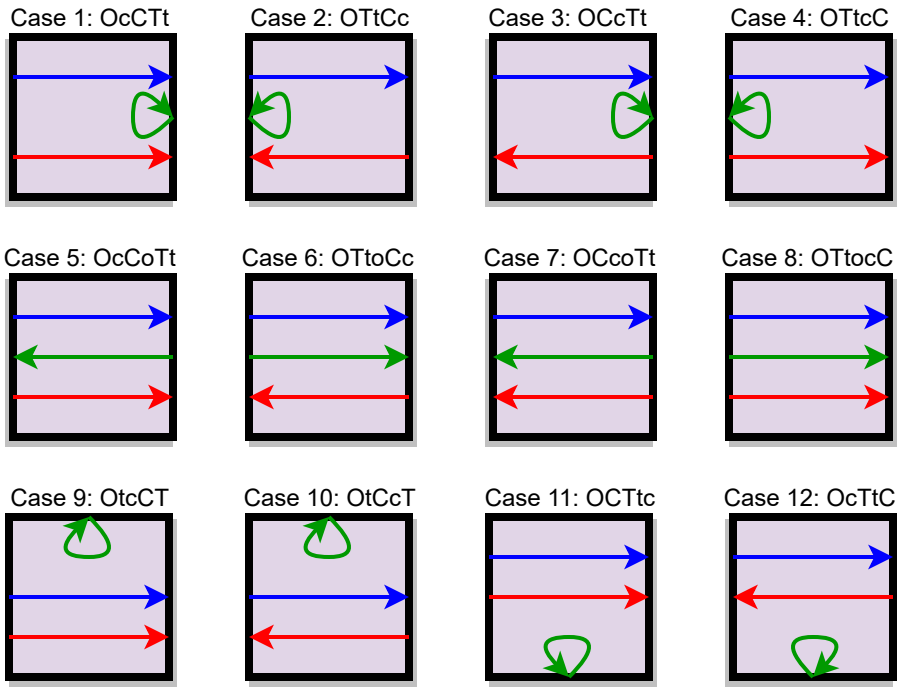


Figure 2-11: The twelve cases of a planar directed door without internal crossings. Opening tunnels with adjacent ports are merged into opening ports.

the tripwire in gadget 1, so exactly one of the locks of gadgets 1 and 2 are locked. Similarly, exactly one of the locks of gadgets 3 and 5 are locked. Crossing the tripwire in gadget 4 forces the agent to exit the simulation (or be stuck), and is also the only way to exit when the agent comes from the top left port, so the lock in gadget 4 is unlocked after an even number of crossings of the top path and locked after an odd number of crossings. Since the locks of gadgets 1 and 2 are anti-correlated, if the agent wants to unlock the lock in gadget 1, it must afterward cross the lock in gadget 3. But said lock must be unlocked by going in a loop through the lock in gadget 4, which is unlocked only after an even number of crossings of the top path. So during an even-indexed (second, fourth, etc.) crossing of the top path, the lock in gadget 1 must be locked before the agent can leave. During an odd-indexed crossing, the agent can take the loop through the lock in gadget 4, unlock the lock in gadget 1, take the loop again to unlock the exit, and exit. So the top path behaves like a directed tripwire for the lock in gadget 1, which is the bottom path.

The parallel and antiparallel directed tripwire locks together simulate a directed tripwire-lock-tripwire with antiparallel tripwires (Figure 2-13). In this gadget going through the top or bottom pairs of tripwires flips which of the middle locks is closed. If the top and bottom set of locked tunnels are paired, then it blocks the middle pathway; however, if they are opposite, the connection in the middle can be used to traverse them.

This gadget in turn simulates a crossover (Figure 2-14), removing the planarity constraint. Here both pathways lock two of the gadgets before reaching the center

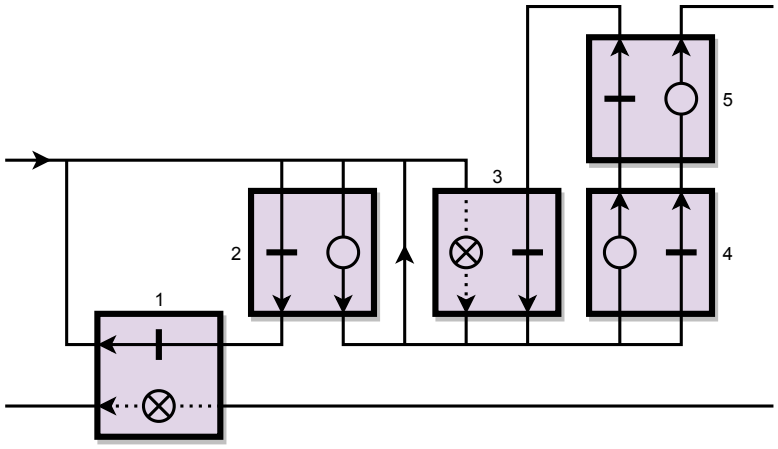


Figure 2-12: Simulation of an antiparallel ditripwire-dilock with a parallel ditripwire-dilock. Gadgets are labelled to aid explanation.

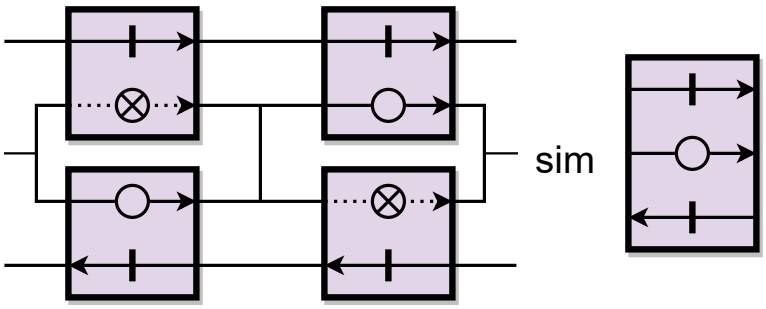


Figure 2-13: Simulation of a directed tripwire-lock-tripwire with antiparallel tripwires using both parallel and antiparallel ditripwire-dilock. Inspired by Figure 10 of [47].

preventing the agent from exiting via a disallowed path, and then they route through those gadgets again opening them back up and restoring the directed crossover to its original state. □

We now present the simulations proving PSPACE-hardness of 1-player motion planning through the doors in Figure 2-11. We give each door a name based on the cyclic orientation of its ports/tunnels, with entrances using uppercase letters and exits using lowercase letters. When an open tunnel's entrance and exit are next to one another, we combine them into a single location, an open port.

Theorem 2.6.8. *1-player planar motion planning with any directed door without internal crossings except the Case 8: OTtocC door is PSPACE-hard.*

Proof. We divide into multiple cases. Note the cases are numbered according to Figure 2-11, not in the order they are addressed in this proof.

Case 2: OTtCc, Case 10: OTcCt, and Case 12: OcTtC doors. In all these doors the opening port/tunnel is a port, and the traverse tunnel output is adjacent to the closing tunnel input. Thus, we can simulate a directed open-optional self-closing

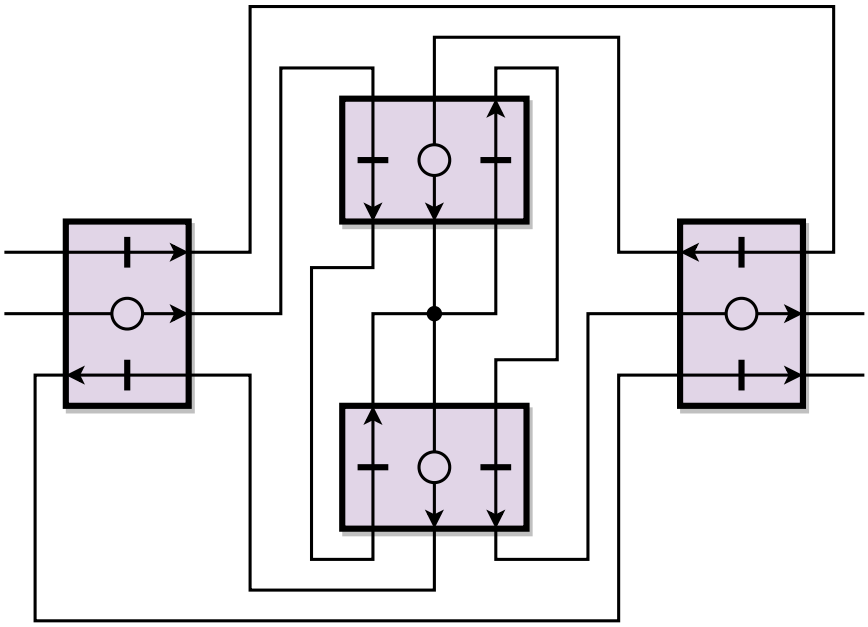


Figure 2-14: Simulation of a crossover using a directed tripwire-lock-tripwire with antiparallel tripwires. Inspired by Figure 11 of [47].

door by wiring the traverse tunnel output to the closing tunnel input and by attaching a wire to the open port, and these wires do not cross each other. Then this reduces to [16, Theorem 4.1]. These are the three trivial cases; the simulations for the rest of this proof were found by our computer search.

Case 1: OcCTt door. can simulate the directed version of the tripwire lock, as shown in Figure 2-15. We will refer to the gadgets numbered left to right. The lock is simply the traverse tunnel on door 1. In the two simulated states we will either have doors 1 and 3 open or door 2 open. If door 2 is open, when traversing the tripwire tunnel we can go through the traverse tunnel allowing us to open doors 1 and 4. With door 4 now open, we can go through its traverse tunnel opening door 3, and then closing door 4 on the way out. This leaves us with doors 1 and 3 open. Going through the tripwire tunnel again closes door 1 but allows us to go through the traverse tunnel of door 3, allowing us to open door 2. Doors 3 and 4 are then closed on the way out. There are states where we could fail to open all of these doors while traversing the close tunnel, but this will leave the gadget with strictly less traversability and thus the agent will never want to take such a path. Thus the Case 1: OcCTt door is PSPACE-complete by Lemma 2.6.7.

Case 3: OCcTt door. This door can simulate a directed open-optional normal self-closing door (Figure 2-16). If the agent enters from port O (the opening port), they can open doors 2 and 3. If they then leave, they have accomplished nothing because door 2 was already open, and door 3 can be opened from port O anyway and cannot be traversed from port T_0 or T_1 as we will see later. So they close door

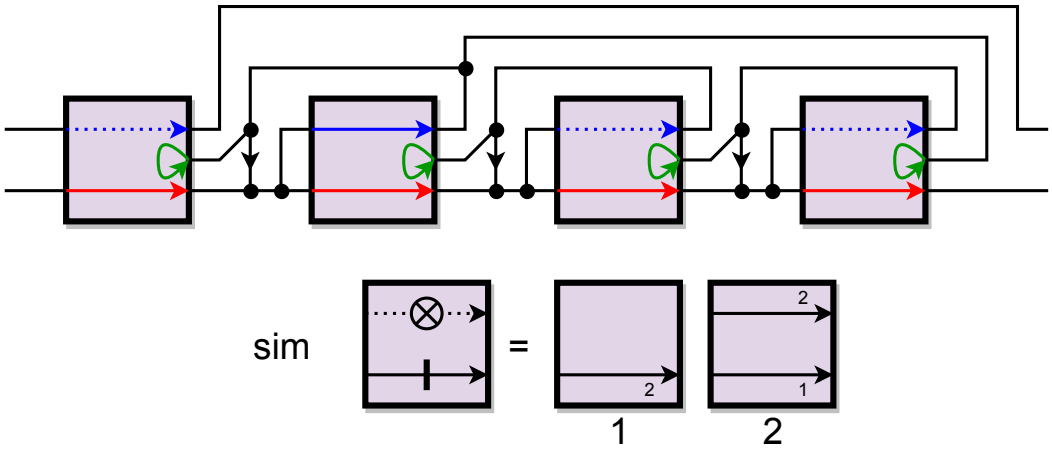


Figure 2-15: The Case 1: OcCTt door simulates the parallel directed tripwire lock. In addition, the state diagram of the directed tripwire lock. Arrows are drawn directly on wires to represent diodes.

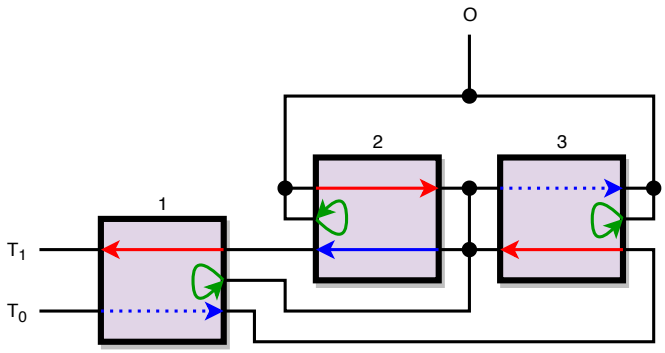


Figure 2-16: Simulation of a self-closing door with the Case 3: OCcTt door. The simulation starts in the closed state. Ports and gadgets are labelled.

2 instead. Then they can open door 1 and they are forced to traverse door 3. The agent can then reopen door 2 and return to port O. Now all the doors are open. If the agent then enters from port T_0, then they are forced to close door 3. They can then open door 1 (useless), and then they are forced to traverse door 2 and close door 1, leading to port T_1. The agent could not have taken this path initially because door 1 was closed, and they cannot take it again without visiting port O because they just closed door 1.

Case 4: OTtcC door. This door can simulate a directed open-optional normal self-closing door (Figure 2-17). If the agent enters from port O, they can open door 1, and then they are forced to close door 2. Continuing this loop does nothing, so the agent then returns to port O. Now door 1 is open and door 2 is closed. If the agent then enters from T_0, then they are forced to traverse door 1. They can then open door 2 and then they are forced to close door 1. Continuing the loop does nothing, so the agent has no other option but to traverse door 2 to port T_1. The agent could not

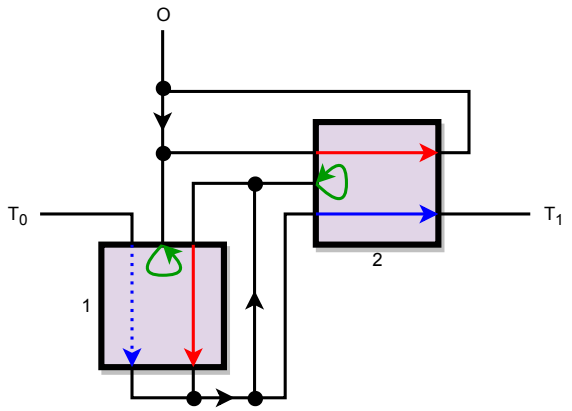


Figure 2-17: Simulation of a self-closing door with the Case 4: OTtcC door. The simulation starts in the closed state. Ports and gadgets are labelled.

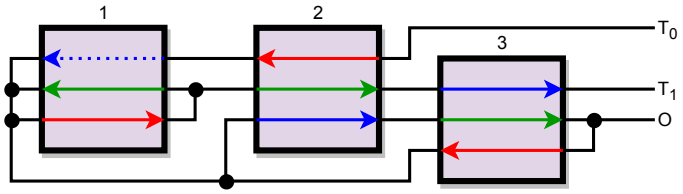


Figure 2-18: Simulation of a self-closing door with the Case 6: OTtoCc door. The simulation starts in the closed state. Ports and gadgets are labelled.

have taken this path initially since door 1 was closed, and they cannot take it again without visiting port O because they closed door 1.

Case 6: OTtoCc door. This door can simulate a directed open-optional normal self-closing door (Figure 2-18). If the agent enters from port O, they are forced to close door 3. If the agent then traverses door 2, they are forced to open door 3 and return to port O, accomplishing nothing. So the agent has no other option but to close door 1. If the agent tries to open door 2, they get stuck, so they instead open door 1. Continuing the loop involving door 1 does nothing, so the agent then traverses door 2, opens door 3, and returns to port O. Now door 1 is open. If the agent enters from port T_0, then they are forced to close door 2, traverse door 1, and close door 1. Reopening door 1 puts the agent back into the situation of being forced to close door 1, so the agent instead opens door 2 and traverses door 3 to port T_1. The agent could not have taken this path initially since door 1 was closed, and they cannot take it again without visiting port O because they closed door 1.

Case 5: OtToCc door. This door can simulate the Case 6: OTtoCc door, which has been covered, by effectively flipping the traverse tunnel (Figure 2-19). Door 1 is the gadget that we flip the traverse tunnel of. If the agent enters from port T_0, they must open door 2, the close door 2. If door 1 is open and the agent then traverses it, they are back to a previous position with nothing changed. Instead, the agent opens

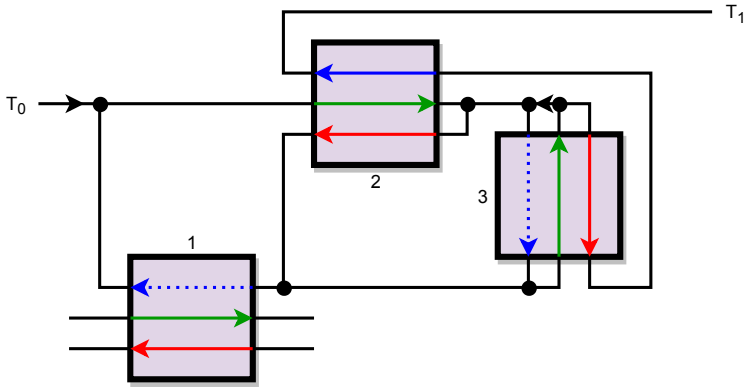


Figure 2-19: Simulation of the Case 6: OTtoCc with the Case 5: OcCoTt door. The traverse tunnel of the leftmost gadget is effectively flipped.

door 3. If the agent then closes door 3, they get stuck because door 2 is closed. So they must close door 2 (again) or traverse door 3. These actions lead to the same situation. If the agent opens door 3 (again), they are back to the same situation that occurred after opening door 3 the first time. If door 1 is open, the agent then traverses door 1. Then they must open door 2. Closing door 2 leads to a previous situation, so the agent then traverses door 3. If the agent then traverses door 1 (again), they must open door 2 (again), leading to a previous situation. So they instead open door 3. Closing door 2 and traversing door 3 lead to different previous situations, so the agent then closes door 3, and then is forced to traverse door 2 to port T_1, leaving all the doors unchanged. If door 1 is not open, then the agent is unable to leave.

Case 7: OtTocC door. This door can simulate a directed open-optional normal self-closing door (Figure 2-20). If the agent enters from port O, they must open door 1, then close door 2. If the agent then closes door 3, they get stuck because door 2 is closed. The agent can traverse door 1 and leave via port O, but they can also open and then traverse door 3 and then do the same thing, which is advantageous. So the agent opens and traverses door 3, then traverses door 1 to port O. Now door 1 is open, door 2 is closed, and door 3 is open. If the agent enters from port T_0, they must close door 1, then open door 2, then traverse door 3. Opening door 3 and then traversing it is a no-op, and door 1 is closed, so the agent closes door 3 and then must traverse door 2 to port T_1. This leaves door 1 closed, door 2 open, and door 3 closed. The agent could not have taken this path initially because door 3 was closed, and cannot take it again without visiting port O first for the same reason.

Case 9: OtcCT door. This door can simulate a directed open-optional normal self-closing door (Figure 2-21). If the agent enters from port O, they can open door 1 and must close door 2. If the agent later enters from port T_0, then they must traverse door 1. They then can open door 2 (and must, since that is the only way out) and must close door 1. Then the agent traverses door 2 to port T_1. The agent could not have taken this path initially because door 1 was closed, and cannot take the path

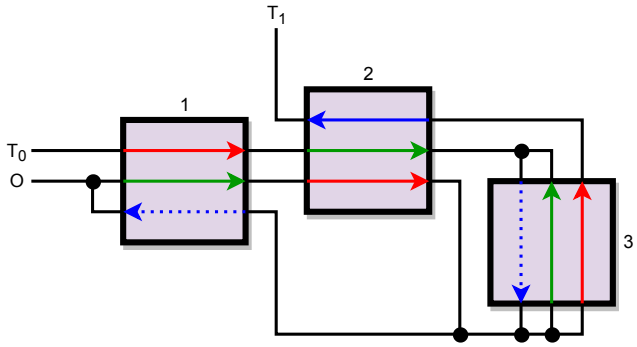


Figure 2-20: Simulation of a self-closing door with the Case 7: OCcoTt door.

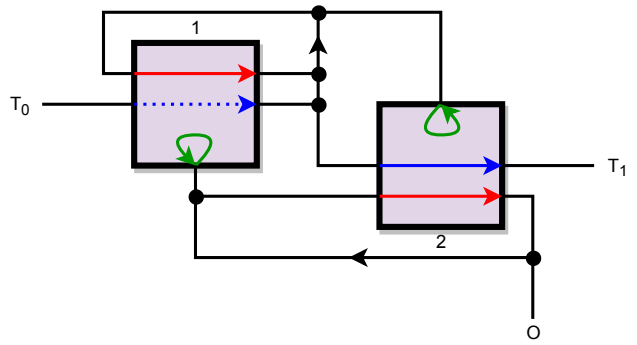


Figure 2-21: Simulation of a self-closing door with the Case 9: OtcCT door.

again without visiting port O first for the same reason.

Case 11: OCTtc door. This door can simulate the Case 12: OcTtC door, which has been covered, by effectively flipping the traverse tunnel (Figure 2-22). Door 1 is the gadget that we flip the traverse tunnel of. If the agent enters from port T_0, then they must traverse the bottom-left diode. The agent can then open doors 2 and 3 but must pick one to close. If they close door 2, they get stuck. So the agent closes door 3. Going to open doors 2 and 3 again leads to a previous situation, so the agent instead traverses door 2. Traversing door 1 (if it is open) leads to a previous situation, and traversing door 5 leads to being stuck. The agent traverse the right diode, and can open doors 4 and 5 but must pick one to close. Closing door 4 leads to a previous situation, so the agent then closes door 5, then must traverse door 4. Going to open doors 4 and 5 again leads to a previous situation. If door 1 is open, then the agent traverses door 1. Traversing door 2 leads to a previous situation, so the agent then opens doors 2 and 3. Closing door 3 leads to a previous situation, so the agent closes door 2, then must traverse door 4. Traversing door 1 leads to a previous situation, so the agent instead opens doors 4 and 5. Closing door 5 leads to a previous situation, so the agent then closes door 4. Traversing door 1 or going to open doors 4 and 5 both lead to previous situations, so the agent then traverses door 5 and must traverse door 4, leaving via port T_1 and leaving all the doors unchanged. If door 1 is not open, however, then the agent cannot leave because door 3 is closed and the only way to

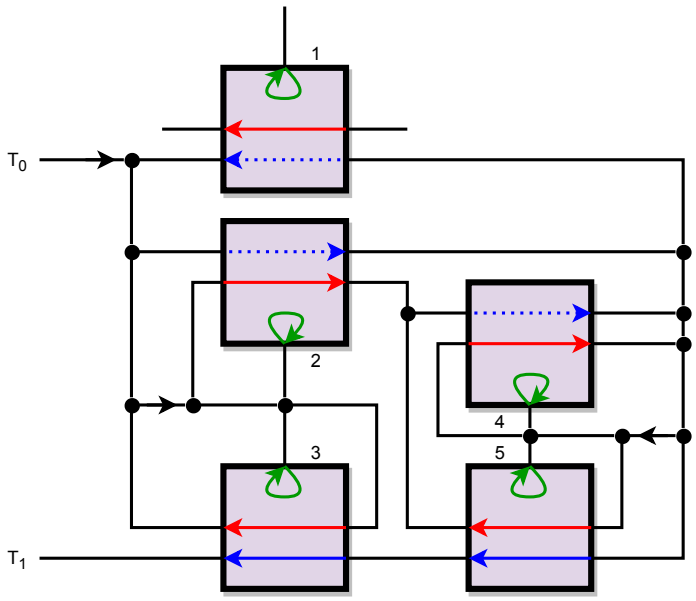


Figure 2-22: Simulation of the Case 12: OcTtC with the Case 11: OCTtc door. The traverse tunnel of the topmost gadget is effectively flipped.

open it is through door 1's traverse tunnel.

This covers all the planar directed doors without internal crossings except the OTtocC door, finishing the proof. □

2.6.3 Input-Output Crossover Classification

Ani et al. [18] proved PSPACE-hardness for zero-player reachability in networks built from unbounded output-disjoint deterministic 2-state input/output gadgets, but they did so without concern for the planar arrangement of the input and output locations of each gadget. That is, they studied six types of gadgets, but there are actually 144 distinct gadgets in this class after taking arrangement of locations into account. We can reframe their unconcern for planarity as an assumption that all these gadgets build (directed) crossovers. Using our computer search, we discharge this assumption for all but 10 of the gadgets by finding a crossover simulation.

We do not attempt to present and argue correctness of 134 simulations here. Instead we take these results as a computer-aided proof: if we believe the gadget search system's operations indeed correspond to operations on gadget networks, and if we believe the implementation is correct, we can depend on traces output by the system to witness the corresponding simulation. (The traces are available on GitHub.[9])

We use different terms to make it easier to generate and parse formulaic names, particularly to avoid assigning multiple meanings to "toggle":

[9] https://github.com/jbosboom/gadget-search-traces

I/O paper's name	Our name
Set-Up Line	diupwire
Set-Down Line	didownwire
Toggle Line	ditripwire
Switch	point
Set-Up Switch	uppoint
Set-Down Switch	downpoint
Toggle Switch	trippoint

We name gadgets by listing the port locations of each component of the gadget. For example, a 01-diupwire-234-downpoint has a diupwire from location 0 to location 1 and a downpoint from location 2 to location 3 in the up state and location 4 in the down state.

Figure 2-23 lists the gadgets known to simulate crossovers by themselves (along with a fanin gadget, as is standard in the input/output gadget model). Figure 2-24 lists the gadgets known to simulate another input/output gadget known to simulate a crossover, or a gadget known to simulate a gadget known to simulate a crossover, and so on. Figure 2-25 lists the gadgets not known to build a crossover. The gadget search cannot prove they do not simulate crossovers, only that, if such a simulation exists, it is beyond the horizon of the search. We searched 13 generations for the diupwire-downpoint, 12 generations for the ditripwire-point, 7 generations for the ditripwire-uppoint, and 9 generations for the diupwire-didownwire-point.

2.6.4 Computational Resources

As an example of the computational resources used during the search, consider the OCTtc door. Our search for gadgets this door simulates ran for 20 generations, with predicate limitations (Section 2.4.6) to 1 component in the state space and 8 states for all operations and 6 locations for combine operations. The search used 618.4 CPU-hours over 13.87 wall-clock hours on our 64-thread AMD Ryzen Threadripper 2990WX with a base clock of 3.0 GHz and a boost clock of 4.2 GHz and 128 GiB of DDR4-2933 RAM. At the end of the search, the database contained 2,524,692,664 gadget specifications and 24,731,118,894 edges, occupying 379.9 GB of storage space.

2.7 Future Work

2.7.1 Mirror

We could change our definition of planar gadget equivalence to only require the location bijection to preserve the cyclic order of locations up to reflection. Then the search could canonicalize away reflections as well as rotations by including the reflected rotation permutations as well. We chose instead to keep enantiomorphs distinct because when reducing to a problem it is not always possible to build both enantiomorphs. For example, endless runner games with gravity fix both left and right

01−diupwire−234−downpoint
02−diupwire−134−downpoint
02−diupwire−143−downpoint
03−diupwire−124−downpoint
03−diupwire−142−downpoint
03−diupwire−214−downpoint
03−diupwire−241−downpoint
04−diupwire−123−downpoint
04−diupwire−231−downpoint

01−diupwire−234−trippoint
02−diupwire−134−trippoint
02−diupwire−143−trippoint
03−diupwire−124−trippoint
03−diupwire−142−trippoint
03−diupwire−214−trippoint
03−diupwire−241−trippoint
04−diupwire−132−trippoint
04−diupwire−231−trippoint

02−ditripwire−134−point
03−ditripwire−124−point
03−ditripwire−214−point

01−ditripwire−243−uppoint
02−ditripwire−134−uppoint
02−ditripwire−143−uppoint
03−ditripwire−124−uppoint
03−ditripwire−142−uppoint
03−ditripwire−214−uppoint
03−ditripwire−241−uppoint
04−ditripwire−132−uppoint
04−ditripwire−213−uppoint

02−ditripwire−134−trippoint
03−ditripwire−124−trippoint
03−ditripwire−214−trippoint

01−diupwire−23−didownwire−465−point
01−diupwire−25−didownwire−436−point
02−diupwire−13−didownwire−456−point
02−diupwire−13−didownwire−465−point
02−diupwire−14−didownwire−356−point
02−diupwire−14−didownwire−365−point
02−diupwire−15−didownwire−346−point
02−diupwire−15−didownwire−364−point
02−diupwire−15−didownwire−436−point
02−diupwire−15−didownwire−463−point
02−diupwire−16−didownwire−345−point
02−diupwire−16−didownwire−354−point
02−diupwire−16−didownwire−435−point
02−diupwire−16−didownwire−453−point
03−diupwire−12−didownwire−456−point
03−diupwire−14−didownwire−256−point
03−diupwire−14−didownwire−265−point
03−diupwire−15−didownwire−246−point
03−diupwire−15−didownwire−264−point
03−diupwire−15−didownwire−426−point
03−diupwire−15−didownwire−462−point
03−diupwire−16−didownwire−245−point
03−diupwire−16−didownwire−254−point
03−diupwire−16−didownwire−425−point
03−diupwire−16−didownwire−452−point
03−diupwire−21−didownwire−456−point
03−diupwire−25−didownwire−416−point
03−diupwire−25−didownwire−461−point
03−diupwire−26−didownwire−415−point
03−diupwire−26−didownwire−451−point
04−diupwire−12−didownwire−356−point
04−diupwire−12−didownwire−365−point
04−diupwire−13−didownwire−256−point
04−diupwire−13−didownwire−265−point
04−diupwire−15−didownwire−236−point
04−diupwire−15−didownwire−263−point
04−diupwire−15−didownwire−326−point
04−diupwire−15−didownwire−362−point
04−diupwire−16−didownwire−235−point
04−diupwire−16−didownwire−253−point
04−diupwire−16−didownwire−325−point
04−diupwire−16−didownwire−352−point
05−diupwire−12−didownwire−346−point
05−diupwire−12−didownwire−364−point
05−diupwire−12−didownwire−436−point
05−diupwire−12−didownwire−463−point
05−diupwire−13−didownwire−246−point
05−diupwire−13−didownwire−264−point
05−diupwire−13−didownwire−426−point
05−diupwire−13−didownwire−462−point
05−diupwire−14−didownwire−236−point
05−diupwire−14−didownwire−263−point
05−diupwire−14−didownwire−326−point
05−diupwire−14−didownwire−362−point
05−diupwire−16−didownwire−234−point
05−diupwire−16−didownwire−243−point
05−diupwire−16−didownwire−324−point
05−diupwire−16−didownwire−342−point
05−diupwire−16−didownwire−423−point
05−diupwire−16−didownwire−432−point
05−diupwire−21−didownwire−436−point
05−diupwire−21−didownwire−463−point
05−diupwire−23−didownwire−416−point
05−diupwire−23−didownwire−461−point
05−diupwire−26−didownwire−413−point
05−diupwire−26−didownwire−431−point
06−diupwire−12−didownwire−354−point
06−diupwire−13−didownwire−245−point
06−diupwire−13−didownwire−254−point
06−diupwire−13−didownwire−425−point
06−diupwire−13−didownwire−452−point
06−diupwire−14−didownwire−235−point
06−diupwire−14−didownwire−253−point
06−diupwire−14−didownwire−325−point
06−diupwire−14−didownwire−352−point
06−diupwire−15−didownwire−423−point
06−diupwire−23−didownwire−415−point
06−diupwire−25−didownwire−413−point
06−diupwire−25−didownwire−431−point

Figure 2-23: Unbounded output-disjoint deterministic 2-state input/output gadgets known to simulate directed crossovers.

Simulating gadget	Simulated gadget
04-diupwire-132-downpoint	04-diupwire-132-trippoint
04-diupwire-213-downpoint	04-diupwire-231-downpoint
01-diupwire-243-trippoint	04-diupwire-132-trippoint
04-diupwire-123-trippoint	04-diupwire-132-trippoint
04-diupwire-213-trippoint	04-diupwire-132-trippoint
01-ditripwire-234-point	04-diupwire-231-downpoint
04-ditripwire-213-point	04-diupwire-132-downpoint
01-ditripwire-234-uppoint	04-diupwire-132-downpoint
04-ditripwire-231-uppoint	04-diupwire-231-downpoint
01-ditripwire-234-trippoint	04-diupwire-132-trippoint
04-ditripwire-123-trippoint	01-ditripwire-234-trippoint
04-ditripwire-213-trippoint	04-ditripwire-213-uppoint
01-diupwire-23-didownwire-456-point	06-diupwire-12-didownwire-354-point
01-diupwire-25-didownwire-463-point	01-diupwire-26-didownwire-453-point
01-diupwire-26-didownwire-435-point	04-diupwire-231-downpoint
01-diupwire-26-didownwire-453-point	04-diupwire-213-downpoint
03-diupwire-12-didownwire-465-point	06-diupwire-14-didownwire-253-point
03-diupwire-21-didownwire-465-point	04-diupwire-132-downpoint
06-diupwire-15-didownwire-324-point	04-diupwire-132-downpoint
06-diupwire-15-didownwire-432-point	06-diupwire-14-didownwire-253-point
06-diupwire-21-didownwire-435-point	04-diupwire-231-downpoint
06-diupwire-23-didownwire-451-point	04-diupwire-132-downpoint

Figure 2-24: Unbounded output-disjoint deterministic 2-state input/output gadgets known to simulate another such gadget in a chain of simulations that eventually leads to a directed crossover. Each gadget in the left column simulates gadgets in the right column in that row.

01−diupwire−243−downpoint

04−ditripwire−123−point

04−ditripwire−123−uppoint

06−diupwire−12−didownwire−345−point
06−diupwire−12−didownwire−435−point
06−diupwire−12−didownwire−453−point
06−diupwire−15−didownwire−234−point
06−diupwire−15−didownwire−243−point
06−diupwire−15−didownwire−342−point
06−diupwire−21−didownwire−453−point

Figure 2-25: Unbounded output-disjoint deterministic 2-state input/output gadgets not known to simulate a directed crossover, even indirectly.

and up and down. This makes it useful to know whether both morphs are necessary for a simulation.

2.7.2 Nonplanar Simulations

The implementation of all the operations described earlier in this chapter are planarity-preserving: gadget networks built from these operations can always be drawn in the plane. This is important when we are seeking planar simulations, as we did for the door gadgets presented in Section 2.6.2, but means the search will not find any nonplanar simulations.

The simplest way to search for nonplanar simulations, and the one we used in some initial experiments, is to include a crossover gadget in the set of gadgets available to build simulations with. This requires a combine generation for each crossing, so strongly favors finding simulations with few crossings.

In nonplanar simulations, the locations of a gadget or gadget network are an unordered set, instead of being in circular order around the gadget's exterior. Combine just merges the locations of the two gadgets instead of splicing the right gadget's locations into the left's at a specific point, and connect can connect any two locations.

The implementations of combine, connect and canonicalize can be modified to natively allow nonplanar simulations. During combine, instead of splicing in a rotation of the right gadget's locations into the left gadget's locations, simply append the right gadget's locations to the left gadget's. Then during connect, connect all pairs of locations, not just adjacent ones. Canonicalization for nonplanar searches uses all permutations of locations, not just rotations, so that gadgets differing only in the numbering of their locations have the same canonical form. With these alternate implementations, the search does fewer combine operations but more connect operations, but should be faster overall because gadgets will canonicalize together more often.

2.7.3 Two-Player Simulations

For one-player and zero-player reachability problems, the number of time steps taken by the agent is irrelevant; the problem asks only if the agent can reach the goal location in any number of steps, or not. Thus we do not need to, and in fact prefer not to, encode timing information in our specifications; when we compose gadget specifications in the search, we do not record the number of individual gadget traversals making up a traversal of the composed gadget.

In two-player reachability problems, the two players are racing to reach their respective goal locations. A precise definition of gadget simulation for two-player problems remains an open problem, but the most obvious definition is that substituting the simulating gadget network with or for the gadget being simulated should preserve the winner, which means preserving precise timing information is important. Thus our specifications do not permit us to find simulations guaranteed to be valid for two-player reachability.

Our search can find simulations that are only valid under the assumption that the agent will maximize the number of available traversals. This assumption is not valid for two-player reachability because one player may want to block the other. Theorem 2.6.2 is an example of a simulation produced by our search that is not valid in two-player. From the state shown in Figure 2-6, the agent can enter at location 3 and traverse through the top tunnel of the center 2-toggle and the bottom tunnel of the left 2-toggle and exit back at location 3. Now the 1-toggle and the top tunnel of the left 2-toggle point towards each other, making the toggle path of the simulated toggle-lock impassable. There are also complete traversals of the lock path that fail to reset the network to the open state, locking the lock in one or both directions. Because making these traversals makes the network strictly less traversable, there is never a reason for the agent in a single-player reachability problem to make them; it is strictly better to leave the toggle traversable in one or the other direction and/or the lock open. In two-player, however, one player might want to block the other player, so for two-player purposes this network does not simulate a noncrossing toggle-lock.

Chapter 3

Tatamibari[1]

3.1 Introduction

Nikoli is perhaps the world leading publisher of pencil-and-paper logic puzzles, having invented and/or popularized hundreds of different puzzles through their *Puzzle Communication Nikoli* magazine and hundreds of books. Their English website [110] currently lists 38 puzzle types, while their "omopa list" [109] currently lists 456 puzzle types and their corresponding first appearance in the magazine.

Nikoli's puzzles have drawn extensive interest by theoretical computer scientists (including the FUN community): whenever a new puzzle type gets released, researchers tackle its computational complexity. For example, the following puzzles are all NP-complete: Bag / Corral [61], Country Road [82], Fillomino [153], Hashiwokakero [15], Heyawake [79], Hiroimono / Goishi Hiroi [14], Hitori [73, Section 9.2], Kakuro / Cross Sum [154], Kurodoko [89], Light Up / Akari [102], LITS [103], Masyu / Pearl [62], Nonogram / Paint By Numbers [139], Numberlink [91, 4], Nurikabe [101, 78], Shakashaka [50, 5], Slitherlink [154, 153, 3], Spiral Galaxies / Tentai Show [63], Sudoku [154, 153], Yajilin [82], and Yosenabe [84].

Allen et al. [8] defined the ***Nikoli gap*** to be the amount of time between the first publication of a Nikoli puzzle and a hardness result for that puzzle type. They observed that, while early Nikoli puzzles have a gap of 10–20 years, puzzles released within the past ten years tend to have a gap of < 5 years.

In this chapter, we prove NP-completeness of a Nikoli puzzle first published in 2004 [108] (according to [109]), establishing a Nikoli gap of 16 years. Specifically, we prove NP-completeness of the Nikoli puzzle ***Tatamibari*** (タタミバリ), named after Japanese tatami mats. A Tatamibari ***puzzle*** consists of a rectangular $m \times n$ grid of unit-square cells, some k of which contain one of three different kinds of clues:  , , and . (The remaining $m \cdot n - k$ cells are empty, i.e., contain no clue.) A ***solution*** to such a puzzle is a set of k grid-aligned rectangles satisfying the following constraints:

1. The rectangles are disjoint.

[1] This chapter, except for Sections 3.4 and 3.5, is from [6] (also available on arXiv [7]), which is joint work with Aviv Adler, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu and Jayson Lynch.

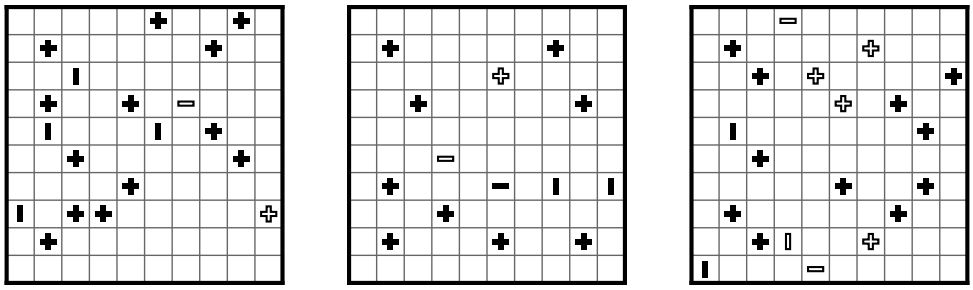


Figure 3-1: What do these Tatamibari puzzles spell when solved and the dark clues' rectangles are filled in? Figure 3-16 gives a solution.

2. The rectangles together cover all cells of the puzzle.

3. Each rectangle contains exactly one symbol in it.

4. The rectangle containing a ⊞ ("square") symbol is a square, i.e., has equal width (horizontal dimension) and height (vertical dimension).

5. The rectangle containing a ⊟ ("horizontal") symbol has greater width than height.

6. The rectangle containing a ▮ ("vertical") symbol has greater height than width.

7. No four rectangles share the same corner (***four-corner constraint***).

To prove our hardness result, we first introduce in Section 3.2 a general "gadget area hardness framework" for arguing about (assemblies of) local gadgets whose logical behavior is characterized by area coverage. Then we apply this framework to prove Tatamibari NP-hard in Section 3.3. In Section 3.8, we show that our framework applies to at least one existing NP-hardness proof, for the Nikoli puzzle Spiral Galaxies [63].

We also present in Section 3.6 a mathematical puzzle font [45] for Tatamibari, consisting of 26 Tatamibari puzzles whose solutions draw each letter of the alphabet. This font enables writing secret messages, such as the one in Figure 3-1, that can be decoded by solving the Tatamibari puzzles. This font complements a similar font for another Nikoli puzzle, Spiral Galaxies [13].

3.2 Gadget Area Hardness Framework

Puzzles. The ***gadget area hardness framework*** applies to a general ***puzzle type*** (e.g., Tatamibari or Spiral Galaxies) that defines puzzle-specific mechanics. In general, a ***subpuzzle*** is defined by an embedded planar graph, whose finite faces are called ***cells***, together with an optional ***clue*** (e.g., number or symbol) in each cell. The puzzle type defines which subpuzzles are valid ***puzzles***, in particular, which clue types and planar graphs are permitted, as well as any additional **properties** guaranteed by a hardness reduction producing the puzzles.

We will use the unrestricted notion of subpuzzles to define gadgets. Define an ***area*** of a puzzle to be a connected set of cells. An ***instance*** of a subpuzzle in a puzzle is an area of the puzzle such that the restriction of the puzzle to that area (discarding all cells and clues outside the area) is exactly the subpuzzle.

Solutions. An ***area assignment*** (potential solution) for a (sub)puzzle is a mapping from clues to areas such that (1) the areas disjointly partition the cells of the (sub)puzzle, and (2) each area contains the cell with the corresponding clue. The puzzle type defines when an area assignment is an actual ***solution*** to a puzzle or a ***local solution*** to a subpuzzle.

Gadgets. A ***gadget*** is a subpuzzle plus a partition of its ***entire*** area (all of its cells) into one ***mandatory*** area and two or more ***optional*** areas, where all clues are in the mandatory area. A hardness reduction using this framework should compose puzzles from instances of gadgets that overlap only in optional areas, and provide a ***filling algorithm*** that defines which clues are in the areas exterior to all gadgets. Each gadget thus defines the entire set of clues of the puzzle within the gadget's (mandatory) area.

For a given gadget, a ***gadget area assignment*** is an area assignment for the subpuzzle that satisfies three additional properties:

1. the assigned areas cover the gadget's mandatory area;
2. every optional area is either fully covered or fully uncovered by assigned areas; and
3. every assigned area lies within the gadget's entire area.

A ***gadget solution*** is a gadget area assignment that is a local solution as defined by the puzzle type.

Profiles. A ***profile*** of a gadget is a subset of the gadget's entire area. A profile is ***proper*** if it satisfies two additional properties:

1. the profile contains the mandatory area of the gadget; and
2. every optional area of the gadget is either fully contained or disjoint from the profile.

Every gadget area assignment induces a proper profile, namely, the union of the assigned areas.

A profile is ***locally solvable*** if there is a gadget solution with that profile. A profile is ***locally impossible*** if, in any puzzle containing an instance of the gadget, there is no solution to the entire puzzle such that the union of the areas assigned to the clues of the gadget instance is that profile. These notions might not be negations of each other because of differences between local solutions of a subpuzzle and solutions of a puzzle.

Each gadget is characterized by a ***profile table*** (like a truth table) that lists all profiles that are locally solvable, and for each such profile, gives a gadget solution. A profile table is ***proper*** if it contains only proper profiles. A profile table is ***complete*** if every profile not in the table is locally impossible. A hardness reduction using this framework should prove that the profile table of each gadget is proper and complete, in particular, that any improper profile is locally impossible.

Given a puzzle containing some gadget instances, a ***profile assignment*** specifies a profile for each gadget such that the profiles are pairwise disjoint and the union of the profiles covers the union of the entire areas of the gadgets. In particular, such an assignment decides which overlapping optional areas are covered by which gadgets. A profile assignment is ***valid*** if every gadget is locally solvable with its assigned profile, i.e., every assigned profile is in the profile table of the corresponding gadget.

A hardness reduction using this framework should prove that every valid profile assignment can be extended to a solution of the entire puzzle by giving a ***composition algorithm*** for composing local solutions from the profile tables of the gadgets, possibly modifying these local solutions to be globally consistent, and extending these solutions to assign areas to clues from the filling algorithm (exterior to all gadgets).

3.3 Tatamibari is NP-hard

In this section, we prove Tatamibari NP-hard by a reduction from planar rectilinear monotone 3SAT. In Section 3.3.1 we briefly discuss a more constrained (but still NP-hard) variant of the classic 3SAT problem from which we will make our reduction; in Section 3.3.2, Section 3.3.3, and Section 3.3.4, we describe the gadgets (wires, variables, and clauses) from which we build the reduction; in Section 3.3.5, we discuss how the spaces between the gadgets are filled; and in Section 3.3.6 we use everything to show the main result.

3.3.1 Reduction Overview

We reduce from *planar rectilinear monotone 3SAT*, proved NP-hard in [41]. An instance of planar rectilinear monotone 3SAT comes with a planar rectilinear drawing of the clause-variable graph in which each variable is a horizontal segment on the x-axis and each clause is a horizontal segment above or below the axis, with rectilinear edges connecting variables to the clauses in which they appear. Each clause contains only positive or negative literals (i.e., is monotone); clauses containing positive (negative) literals appear above (below) the variables. We can always lengthen the variable and clause segments to remove bends in the edges, so we assume the edges are vertical line segments. We can further assume that each clause consists of exactly three (not necessarily distinct) literals: if a clause has $k < 3$ literals, we can just duplicate one of the clauses $3 - k$ times, which is easy to do while preserving the tri-legged rectilinear layout.

We create and arrange our gadgets directly following the drawing, possibly after scaling it up; see Figure 3-2. Edges between variables and clauses are represented by

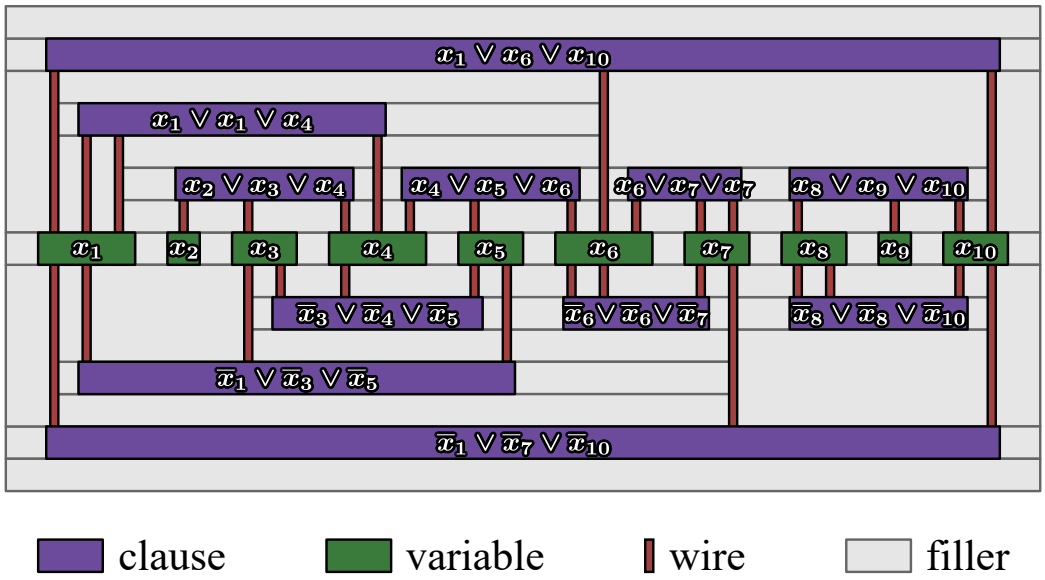


Figure 3-2: The overall layout of the Tatamibari puzzles produced by our reduction follows the input planar rectilinear monotone 3SAT instance. Clause, variable, and wire gadgets are represented by purple, green, and red rectangles. Not drawn are terminator gadgets at the base of all unused copies of variables. Grey rectangles correspond to individual filler clues. Figure inspired by [41, Figure 2].

wire gadgets that communicate a truth value in the parity of their covering. For each variable, we create a *variable gadget*, which is essentially a block of wires forced to have the same value, and place it to fill the variable's line segment in the drawing. For each clause, we create a *clause gadget* with three wire connection points and place it to fill the clause's line segment. Negative clauses and wires representing negative literals are mirrored vertically. Both the variable and clause gadgets can telescope to any width to match the drawing; unused wires from the variable gadgets are terminated at a *terminator*. By our assumption that the edges are vertical segments, we do not need a turn gadget.

Covering a clause gadget without double-covering or committing a four-corner violation requires at least one of its attached wires to be covered with the satisfying parity (the true parity for positive clauses and the false parity for negative clauses).

To ensure clues in one gadget do not interfere with other gadgets, the wire gadget is surrounded on its left and right sides by sheathing of ▯ clue rectangles and the clause gadget is surrounded on three sides by a line of ▯ clues forced to form 1×1 rectangles. Wire sheathing also ensures neighboring wires do not constrain each other, except in variable gadgets where the sheathing is deliberately punctured.

In our construction, gadgets will not overlap in their mandatory areas, so in the intended solutions, the mandatory area will be fully covered by rectangles satisfying the gadget's clues. Also in our construction, every optional area will belong to exactly two gadgets, and in the intended solutions, such an area will be covered by clues in exactly one of those gadgets.

To apply the gadget area hardness framework, we define a ***local solution*** of

a subpuzzle to be a disjoint set of rectangles satisfying the gadget's clues and the property that no four of these rectangles share a corner. (At the boundary of the subpuzzle, there is no constraint.) Our composition algorithm will combine these local solutions by staggering rectangles to avoid four-corner violations on the boundary of and exterior to gadgets. We will prove that valid profile assignments correspond one-to-one to satisfying truth assignments of the 3SAT instance.

We developed our gadgets using a Tatamibari solver based on the SMT solver Z3 [42]. Section 3.4 describes the solver and Section 3.5 shows the solver-guided development of the clause gadget. The solver and machine-readable gadget diagrams are available on GitHub.[2] Unfortunately, the solver can only verify the correctness of constant-size instances of the gadgets, but the variable and clause gadgets must telescope to arbitrary width. Thus we still need to give manual proofs of correctness.

3.3.2 Wire Gadgets and Terminators

The wire gadget, shown in Figure 3-3, consists of a column of + clues surrounded by | clues which encodes a truth value in the parity of whether the squares are oriented with the + clues in their upper left or lower left corners. We will call this the *wire parity* or *wire value*. In this construction, only vertical wires are needed, and thus we do not give a turn gadget or horizontal wire. We call the column containing the + clues and the empty column next to it the *inner wire*. The inner wire is covered by columns of alternating | clues, called the *(inner) sheathing*. In the overall reduction, | clues in columns just outside the wire at its ends (in the variable gadget and either the clause or terminator gadget) add a further layer of sheathing (called the *outer sheathing*) outside the wire gadget, ensuring neighboring wires do not constrain each other.

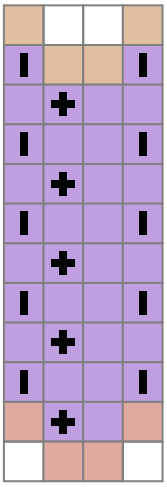
The following lemmas will show that the + clues in the wire must be covered by 2×2 squares, the squares must all have the same parity, and the wire does not impart any significant constraints onto the surrounding region. These lemmas assume that no rectangle from a | clue can reach the cells to the right of the top and bottom + clues in the wire, a property which we call ***safe placement***. We discharge this assumption in Section 3.3.5 by showing all wire gadgets produced by our reduction are safely placed.

Lemma 3.3.1. *Each + in a safely placed wire covers a 2×2 square in the wire.*

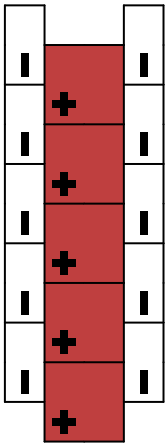
Proof. There is no 3×3 square in the wire that contains a + clue but does not contain any other clue. Thus we cannot cover the + clue by squares larger than 2×2.

Now suppose we cover a + clue by a 1×1 square. Now the cells immediately above and below this clue must be covered. The | clues must be taller than they are long, so they cannot cover these cells. Thus we must cover them by squares containing the + clues above and below. This leaves the cell directly to the right of the 1×1 uncovered. It is easy to see that this cannot be covered by the nearby + clues or |

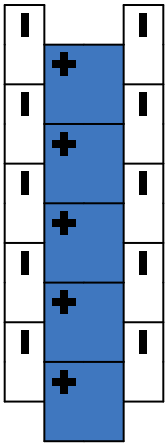
[2] https://github.com/jbosboom/tatamibari-solver



(a) An unsolved wire gadget. Mandatory area is purple and optional areas are brown.

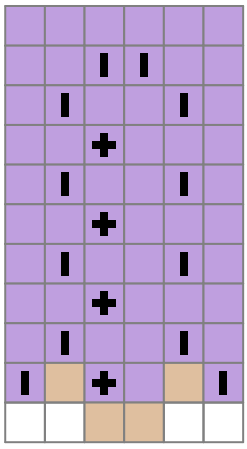


(b) Wire communicating false

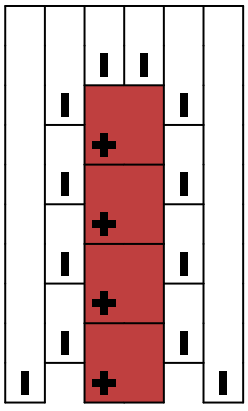


(c) Wire communicating true

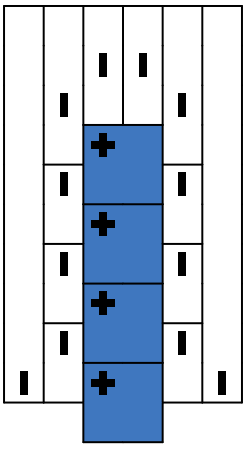
Figure 3-3: Wire gadget and its profile table. The wire can be extended to arbitrary height by repeating rows. Note that between figures (b) and (c), the clues stay in the same place (and the rectangles shift to represent the different values of the wire).



(a) An unsolved terminator gadget. Mandatory area is purple and optional area is brown.



(b) Terminating a false wire



(c) Terminating a true wire

Figure 3-4: Terminator gadget and its profile table.

clues. The cells next to the top and bottom clues cannot be covered by a ▮ clue from outside the gadget by our assumption that the wire is safely placed.

The only remaining possibility is a ▬ clue from outside the gadget extending into the wire gadget. Such a rectangle cannot extend entirely through the wire because the ▮ clues in the sheathing and the ✚ clues inside the wire are in alternating rows. If the external horizontal rectangle enters the wire from the right and covers a cell next to the ✚ clue, that ✚ clue is forced to be a 1×1 rectangle and the cell above it must be covered by the next ✚ clue above. This results in a four-corner violation involving the two ✚ clues and the left sheathing except when the ✚ clue is at the bottom of the wire. In that case, the external horizontal rectangle blocks the bottom-right sheathing clue, making it 1×1 and unsatisfied. □

Corollary 3.3.2. *Satisfied safely placed wires must have all of their 2×2 squares with the ✚ clues in the lower left corner or all in the upper left corner.*

Proof. By Lemma 3.3.1 all ✚ clues must be covered by 2×2 squares. To change whether the ✚ clues are in the lower left or upper left, we will end up leaving a row of two cells between clues blank. By the same arguments in Lemma 3.3.1 these cannot be covered by the nearby ✚ clues or ▮ clues. □

Lemma 3.3.3. *The ▮ clues making up the inner sheathing of satisfied safely placed wires must be covered by 1×2 rectangles of opposite parity to the wire's squares.*

Proof. By Corollary 3.3.2 the wire has one of two parities of squares. If a vertical rectangle ends with the same y coordinate as an adjacent square, then we will have three right angles at a single corner, forcing a four-corner violation or uncovered cell. Because the squares are 2×2, a vertical rectangle of odd height guarantees one of the ends will share a y-coordinate with one of the squares. The ▮ clues occur every other cell, so the vertical rectangles cannot be of length greater than 3. This forces them to be of length 2 and staggered with respect to the squares. □

Theorem 3.3.4. *The safely placed wire gadget's profile table is proper and complete.*

Proof. By Lemma 3.3.1, each optional area must be fully covered or fully uncovered by the neighboring ✚ clue, so the profile table is proper. Corollary 3.3.2 fixes the ✚ clue parity and Lemma 3.3.3 fixes the sheathing parity, so all other profiles are locally impossible, so the profile table is complete. □

We also have a terminator gadget to terminate unused wires regardless of their parity. The terminator gadget is shown in Figure 3-4.

Lemma 3.3.5. *The terminator does not constrain the wire parity.*

Proof. Figures 3-4b and 3-4c show solutions of the terminator with both parities. The same arguments about wire correctness show this gadget does not allow any additional wire solutions nor constrain other gadgets. □

Theorem 3.3.6. *The terminator gadget's profile table is proper and complete.*

Proof. The profile table in Figure 3-4 contains only proper profiles, so the profile table is proper. By the same arguments in Lemma 3.3.1, the two local solutions shown are the only way to cover the wire part of the gadget. A horizontal rectangle from a ➖ outside the gadget could cover part of the top row of the gadget (or the entire top row when terminating a true wire) while leaving the clues in the gadget satisfied and covering the remaining area. We prevent this through the global layout: all clause gadgets (the only gadget containing ➖ clues) appear strictly above (for positive clauses) or strictly below (for negative clauses) all terminator gadgets, so it is not possible for any ➖ rectangles to cover area in the clause gadget. Thus all other profiles are locally impossible, so the profile table is complete. □

3.3.3 Variable Gadgets

The variable gadget is essentially a series of wires placed next to each other with devices we call ***couplers*** in between. Each coupler acts essentially as an "equality" constraint between neighboring wires, thus forcing all the wires connected via a series of couplers to represent the same variable; this collection of wires then forms the ***variable gadget*** of the reduction.

Each coupler takes two columns, and consists of (i) a ➕ clue which interacts with the inner sheathing of the wires to force equality, and (ii) eight ❙ clues (two above and two below the ➕ clue on each column), which prevent the inner sheathings of the neighboring wires from constraining each other (except through the ➕ clue itself). See Figure 3-6 for an example with two wires; additional wires can be added to either side of variable by using more couplers (see Figure 3-7).

First, notice that both wires are constrained to have their squares in one of two parities by the inner sheathing, as in Corollary 3.3.2. It is also important that wires do not constrain each other outside the couplers, either directly (if they happen to be adjacent) or indirectly (through the space in between); we address this in Section 3.3.5.

Now we have to show that two wires separated by the coupler must be in the same configuration. This happens because the wire parity forces the parity of the inner sheathing, which forces the parity of the coupler, which then forces the partiy of the inner sheathing and the wire parity of the next wire over.

Lemma 3.3.7. *The coupler has only two valid coverings of its ➕ clue.*

Proof. The location of the eight ❙ clues around the ➕ clue ensure that it cannot be larger than 2×2. By Corollary 3.3.2 we know that the wire gadgets next to the coupler must have their inner sheathing as 2×1 rectangles in either the up or down position. If the ➕ clue is covered by a 1×1 it will create a four-corner violation with the inner sheathing. Thus it must be one of the two possible positions for a 2×2 square. If both inner sheathings have the same parity, as in Figure 3-6 then the constraints can be locally satisfied. □

Lemma 3.3.8. *All wires in a variable gadget must have the same value (i.e. upwards branches must have the same orientation).*

Proof. We know the coupler has at most two ways to satisfy its constraints, corresponding to a 2×2 square in either the up or down position. Notice that the inner sheathing of both wires must be of different parity from the square or they will cause a four-corner violation. Thus the inner sheathing must have the same parity, ensuring that the wires themselves must have the same parity. If multiple wires are all connected by couplers, then they will all be forced to have the same parity by the same local argument. □

Lemma 3.3.9. *The variable gadget is locally solvable with a given profile if and only if the profile satisfies (i) all upwards branches have the same orientation, (ii) all downwards branches have the same orientation, and (iii) upwards and downwards branches have opposite orientations from each other.*

Proof. The "only if" direction follows from Lemma 3.3.8 and Corollary 3.3.2 (each wire individually must have opposite orientation for upwards and downwards branches due to the couplers, and all wires in the gadget must have the same upward orientation).

The "if" direction follows from Lemma 3.3.5, the individual solvability of each wire and terminator in both orientations (as shown in Figure 3-3b, Figure 3-3c, Figure 3-4b, and Figure 3-4c), and the solvability of the couplers given that adjacent wires have the same orientation (Figure 3-6). Neighboring wires (within the variable gadget) do not conflict with each other (outside of the coupler) because of the "outer sheathing" columns separating them; the meeting points of the two clues in each "outer sheathing" column can be adjusted to avoid four-corner violations with each other, as well as avoiding four-corner violations with the neighboring "inner sheathing". □

Note that this lemma is what we want from a variable gadget: it is locally solvable if and only if its profile corresponds to a specific value for the variable it represents.

3.3.4 Clause Gadgets

The clause gadget, shown in Figure 3-8, interfaces with three wire gadgets representing the three literals of this clause. In the upper-left of the variable gadget is an internal wire, which we call the ***clause verification wire***. The only way to cover the top two cells of that wire is using the wire's top ✚ clue. This is only possible when at least one of the wires is true, allowing a ***variable enforcement line*** (drawn in figures as a purple horizontal bar) to provide a parity shift to the clause verification wire. Otherwise, either those top two cells cannot be covered, or some other cell in the clause will not be covered, or there will be a four-corner violation.

The mandatory areas of the clause include all clues and cells shown in Figure 3-12 and optional areas consisting of the row of cells at the bottom of the gadget, specifically the set of cells under the ▮ clue lines at the bottom of the gadget.

Each of the three wires in this gadget has two intended solutions: true or false. In Figure 3-12, the wire is blue if it represents true and red if it represents false. The leftmost wire behaves somewhat differently from the others because it is closest to the clause verification wire.

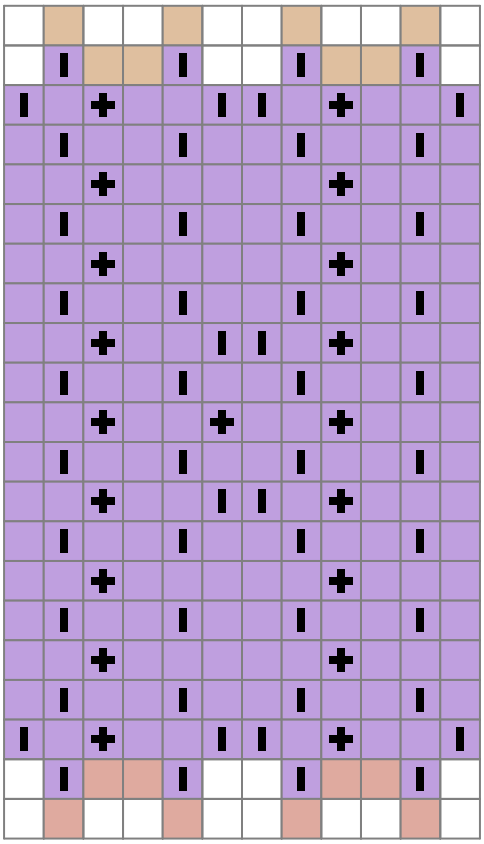


Figure 3-5: The variable gadget. Mandatory area is purple and optional areas are brown.

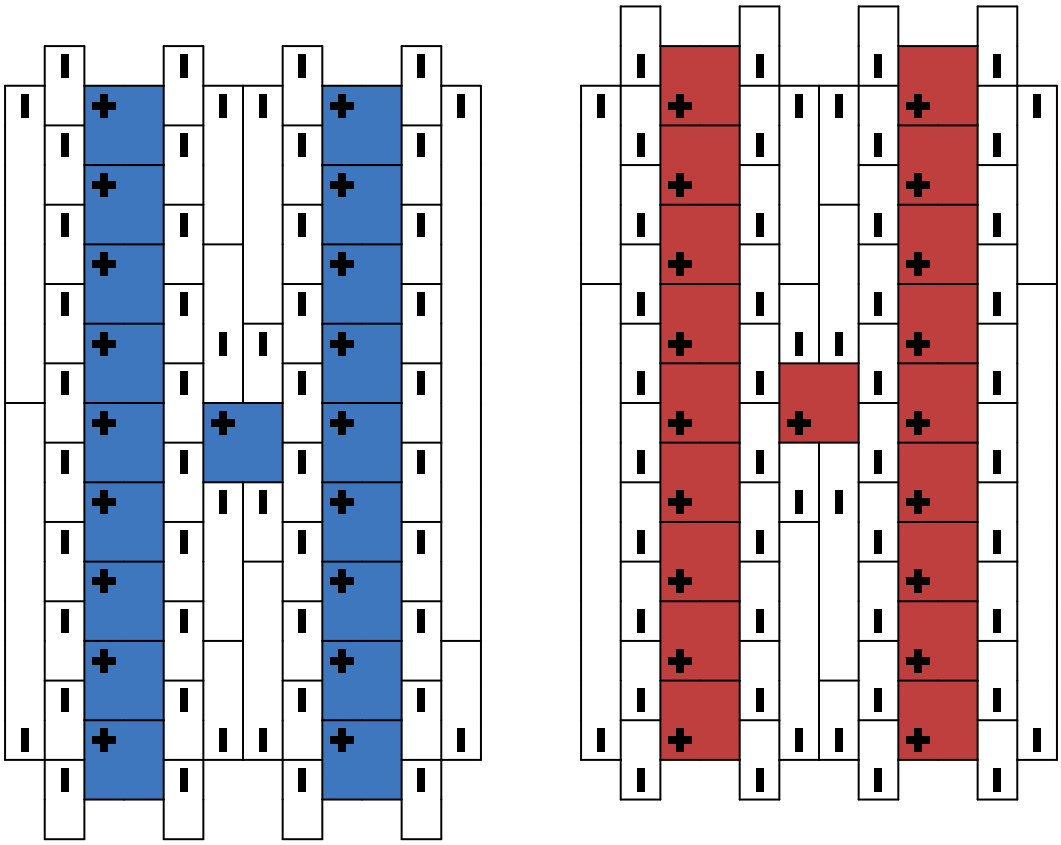


Figure 3-6: The variable gadget's profile table. Left: variable set to true. Right: variable set to false.

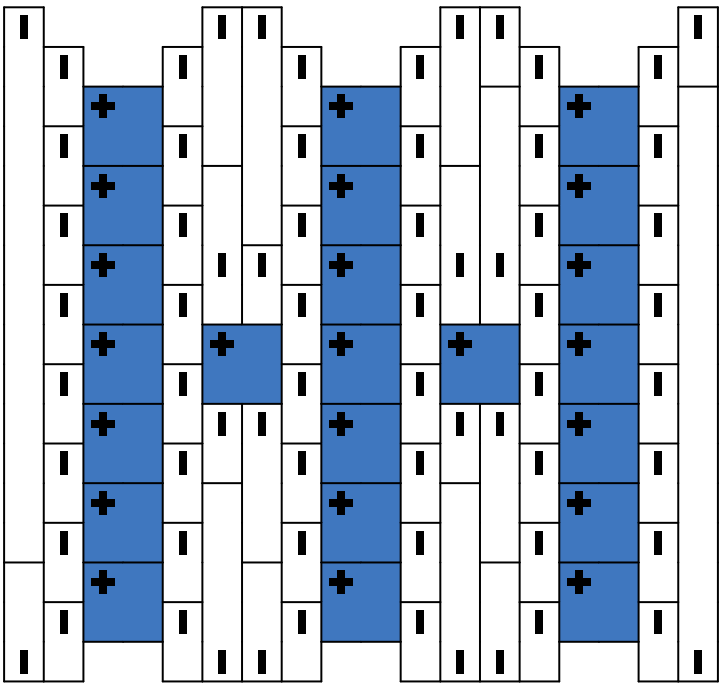


Figure 3-7: A variable gadget widened to provide three wires, shown here set to true.

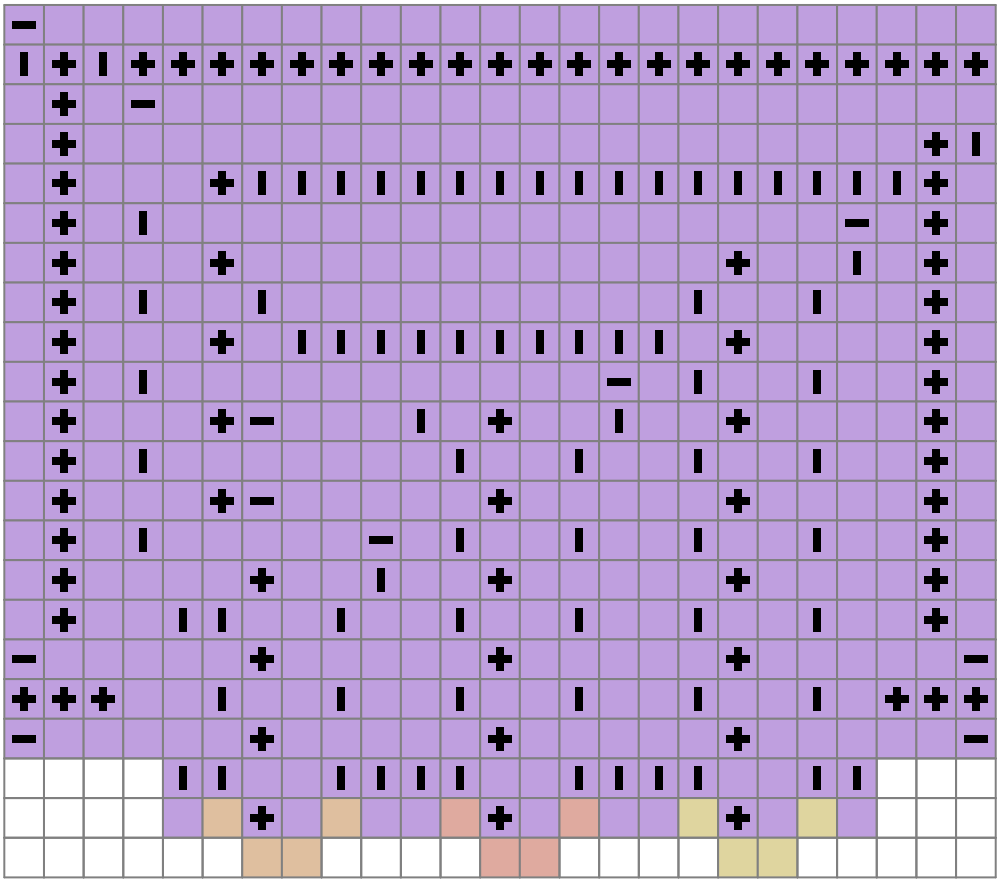


Figure 3-8: An unsolved clause gadget. Mandatory area is purple and optional areas are brown.

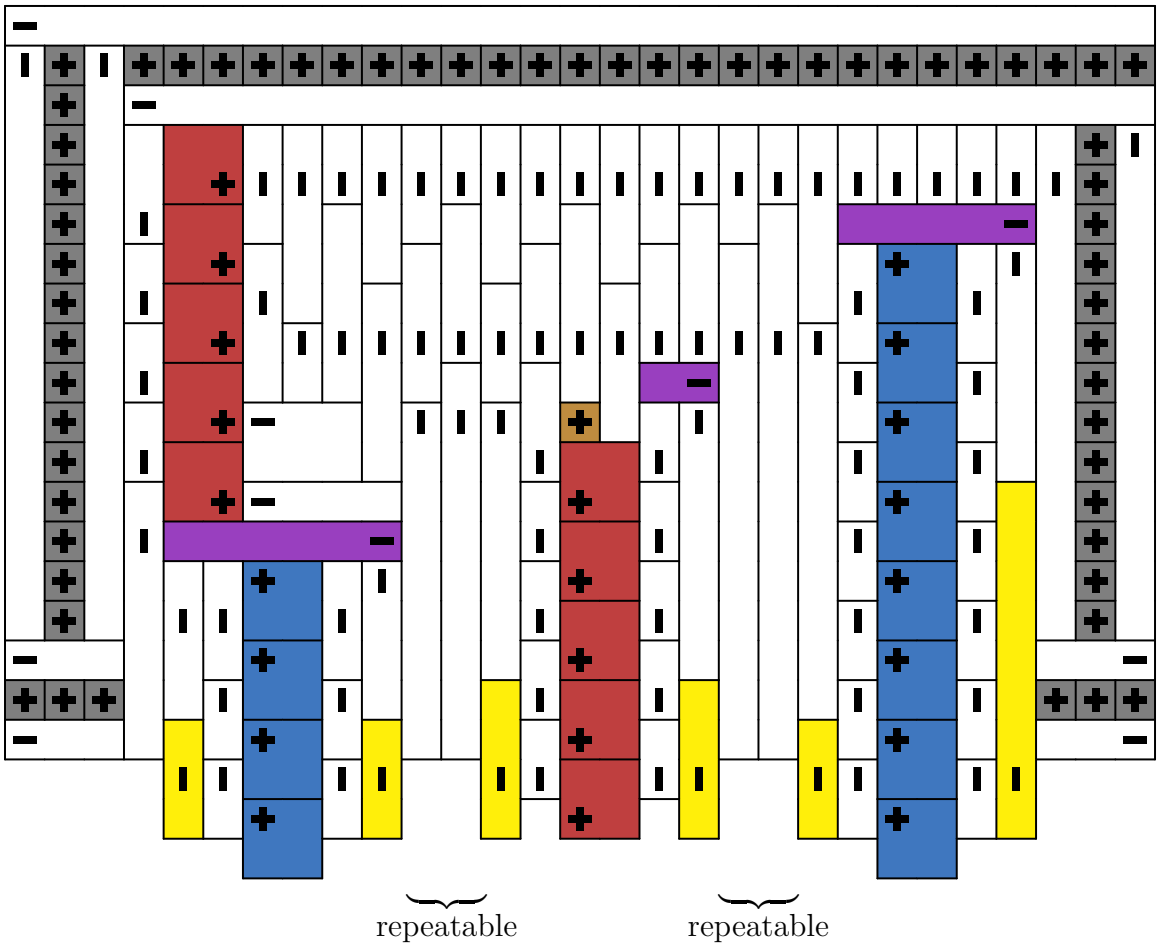


Figure 3-9: Example where the columns in between literal wires in the clause gadget have been expanded. The columns which are able to be repeated an arbitrary number of times have been labeled as "repeatable" in the figure since they can be repeated an arbitrary number of times to make the clause an arbitrary width.

Importantly, the clause gadget can be expanded horizontally such that the variable wires can be spaced an arbitrary amount beyond the width of the base gadget shown in Figure 3-12. The columns between the literal wires in the clause gadget can be expanded an arbitrary number of columns. Such an example expansion is shown in Figure 3-9. In this example, the columns have been expanded such that the entire gadget is wider by 4 columns and the number of columns between each literal in the gadget has been expanded by 2 columns.

Lemma 3.3.10. *If any wire is in the* false *configuration, then the variable enforcement line corresponding to the wire will not be able to go across the gadget.*

Proof. If a wire is in the false configuration, then there exists at two cells on the top of the wire that need to be covered. These two cells can be covered in two different ways. We first prove this lemma for the leftmost wire and then prove the lemma for the other wires since the leftmost wire is different from the others. In this case, the

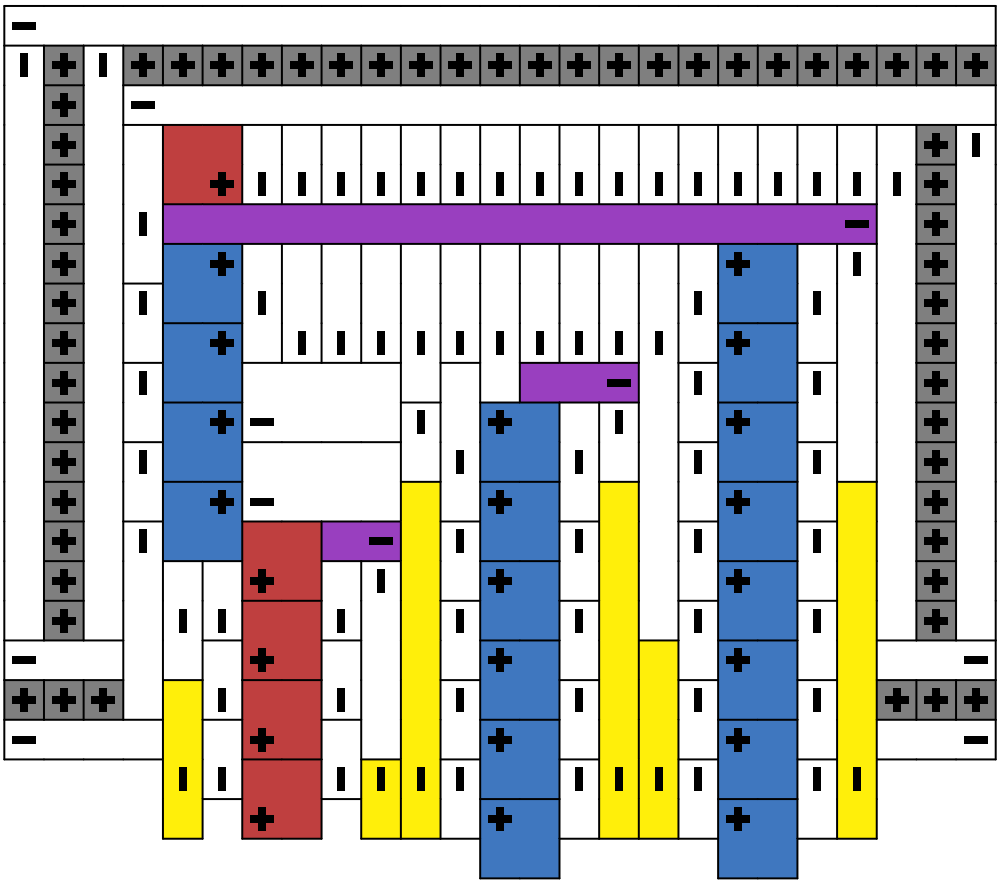


Figure 3-10: When the leftmost wire is set in the false configuration, the only way to cover the top two cells of the wire is with a 2×2 square that blocks the variable enforcement line.

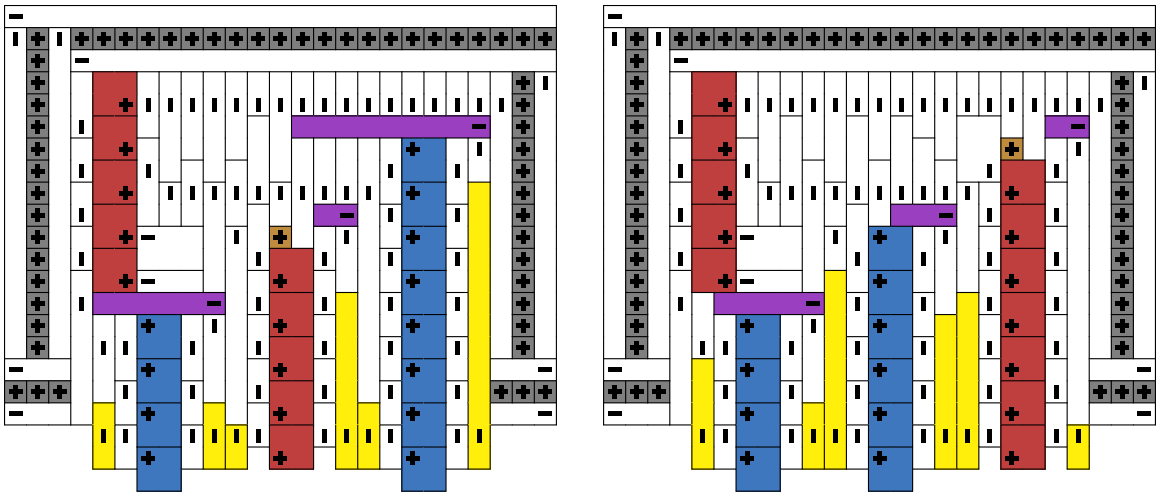
only way to cover the two cells is with a 2×2 square (see Figure 3-10), blocking the variable enforcement line from crossing the top of the wire.

For the other two wires, the top two cells can be covered in only two ways. Either a 1×1 square covers one of the two cells and a vertical line from the top covers the other cell or vice versa (see Figure 3-11).

No other configurations are available that does not violate the four-corner constraint. Thus, this configuration prevents the corresponding variable enforcement lines from going across the gadget. □

Corollary 3.3.11. *When all wires in the gadget are false, the puzzle does not have a solution.*

Proof. By Lemma 3.3.10, no variable enforcement line can go across the gadget if all wires are false. In order to solve the puzzle presented by the gadget, the top two cells of the clause verification line must be covered. These two cells cannot be covered by the horizontal line on top of them nor can they be covered by the vertical lines beside them. Thus, they must be covered by the 2×2 square formed in the clause verification line. However such a square will either leave a cell in the middle



(a) The middle wire is false.

(b) The right wire is false.

Figure 3-11: For the false wires, the only two configurations that will guarantee that the two cells at the top of the wires are covered are the cases where one 1×1 square covers one of the two cells and a long rectangle extending from the top covers the other cell. The 1×1 squares are shown in brown in the figures above.

of the clause verification line uncovered or will leave the bottom two cells of the line uncovered. In this case, no configurations exist in covering these bottom two cells without violating the four-corner rule. See Figure 3-12a. Thus, the gadget is unsatisfiable if all wires into the gadget are false. □

Lemma 3.3.12. *If at least one of the wires entering the clause gadget is in the true configuration, then the clause gadget is locally solvable.*

Proof. In any wire is in the true configuration, then the variable enforcement line corresponding to the gadget will be able to go across the gadget. For the leftmost wire, the clause verification line will be in the configuration that ensures that all cells that need to be covered by the line are covered. Otherwise, the variable enforcement line will be able to cause the clause verification line to cover all the necessary cells. See Figures 3-12b to 3-12h. □

Using the above lemmas, we are able to prove the following properties of the profile table of the clause gadget.

Corollary 3.3.13. *The profile table of the clause gadget is proper.*

Lemma 3.3.14. *The profile table of the clause gadget is complete.*

Proof. The clause gadget's profile table contains all profiles shown in Figure 3-12 except for the all-false configuration shown in Figure 3-12a. By Corollary 3.3.11, the all-false configuration is not locally solvable. It remains to show the all-false configuration is locally impossible.

To do this, we show that no solution to a clue outside of this profile is able to solve any part of the all-false clause profile–essentially that the clause gadget is fully isolated from the rest of the puzzle. By design, no clue above, to the left of, or to the right of the clause can cover any of the cells that are left uncovered by the literals, because the row and columns of single-cell squares blocks any rectangles from reaching the uncovered cells.

We now prove that no clues from the bottom of the gadget can help cover any of these cells. Such clues can only potentially cover the optional areas at the bottom of the gadget. We show that such clues cannot cover parts of the literal gadgets. By Lemma 3.3.7, there are only two possible configurations of the variable gadgets; thus, no other outside fillers can cover any cells in the incoming wires. Hence, no clues adjacent to the bottom of the gadget can help cover any part of the incoming wires.

Thus the all-false profile is locally impossible, so the profile table is complete. □

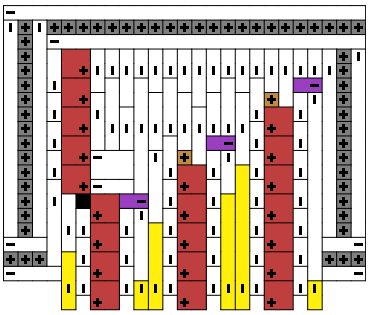
3.3.5 Layout, Sheathing, and Filler

In order to build the full Tatamibari instance corresponding to a planar rectilinear monotone 3SAT instance, we lay out the gadgets as shown in Figure 3-2: variable gadgets are positioned on a central line, while positive and negative clauses are positioned above and below respectively at heights corresponding with how many layers of clauses are nested below them, with wires running vertically from variables to clauses (both variable and clause gadgets can be extended arbitrarily far horizontally). Variable and clause gadgets have rectangular profiles (except for where the wires "plug in" to them). Variables and clauses have a uniform height, and for any two variable or clause gadgets, they are placed on exactly the same set of rows or they share no rows.

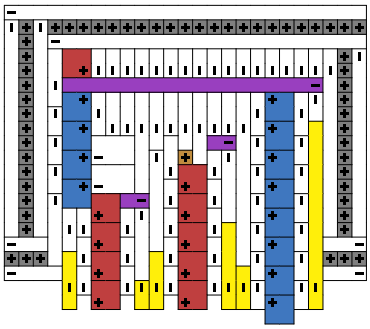
All wire gadgets in the puzzle produced by our reduction are safely placed; that is, no rectangle from a ▯ clue can reach the cells to the right of the top and bottom ✚ clues in the wire. The only ▯ clues in those columns are in clause gadgets. The row of single-cell squares at the top of the clause gadget blocks any rectangles from extending upwards out of the clause gadget. If a rectangle from a ▯ clue in those columns of the clause gadget extends downward past the first ✚ clue in the column to its left, the cell below that ✚ clue cannot be covered by any clue, so rectangles cannot extend downward out of the clause gadget in those columns. Thus ▯ clues from clause gadgets cannot interact with wire gadgets, so the wires are safely placed.

Because we want the solvability of the Tatamibari instance to depend only on solving the gadgets, we need to add ***filler*** clues that are always able to cover the areas outside the gadgets.

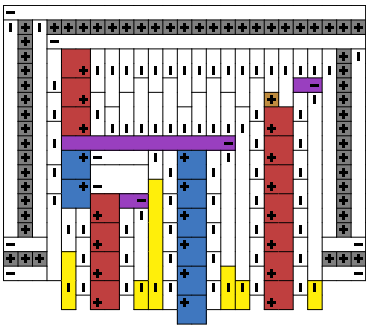
First, we set aside any cells horizontally adjacent to a wire gadget. These cells will be covered by the outer sheathing clues described in described in Section 3.3.2 and Section 3.3.3 and highlighted in yellow in Figure 3-9 and Figure 3-12. In the global solution, the areas assigned to the outer sheathing clues thus extend vertically outside their gadget. For the purposes of the filler algorithm, we consider the cells covered by the outer sheathing to be part of the wire gadget.



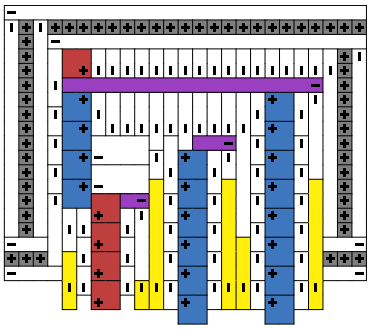
(a) The (false, false, false) configuration.



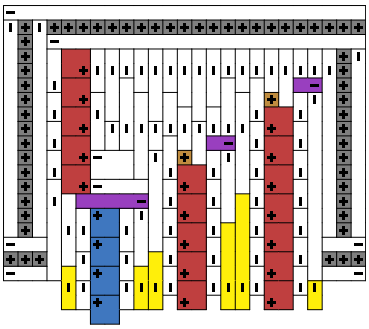
(b) The (false, false, true) configuration.



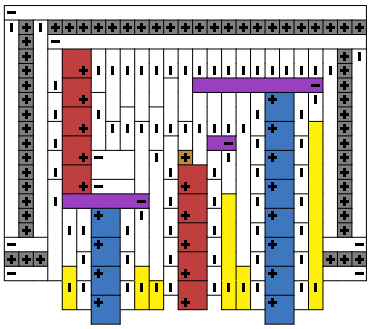
(c) The (false, true, false) configuration.



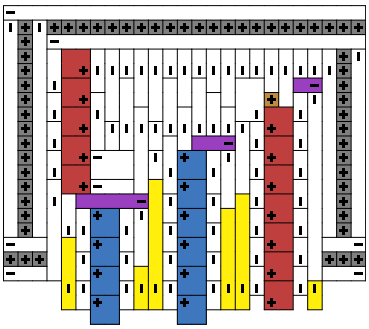
(d) The (false, true, true) configuration.



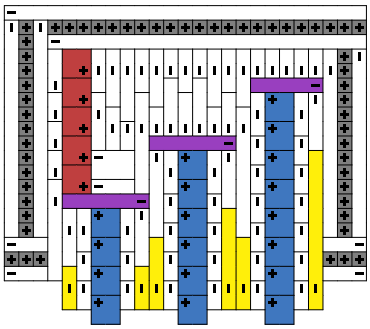
(e) The (true, false, false) configuration.



(f) The (true, false, true) configuration.



(g) The (true, true, false) configuration.



(h) The (true, true, true) configuration.

Figure 3-12: The clause gadget. All configurations shown here except the all-false configuration in Figure 3-12a are in the clause gadget profile table. Clues highlighted in yellow also function as the "outer sheathing" protecting the wires closest to them (see Section 3.3.5)

Each filler clue corresponds to a rectangular area of space between gadgets, formed by breaking each row into maximal horizontally contiguous strips between (and bordered by) the gadgets, then joining vertically contiguous strips into a single rectangle if they have the same width. The filler algorithm places a single clue in each of these rectangles (▮, ▬, or ✚ depending on the rectangle's aspect ratio), placed arbitrarily inside (say, in the upper-right corner). See Figure 3-2 for an example. While it may be possible for the solver to use these clues differently than shown here, we only need to prove that if the solver does assign each rectangular area to its associated clue, it will cover the area.

The only potential problem lies in the possibility of a four-corner violation involving a filler rectangle. This can only happen where either (i) a corner of a filler rectangle meets a gadget and a wire coming from that gadget, or (ii) where two corners of filler rectangles meet along the edge of a gadget. If a corner of a filler section meets an edge of another filler section or the edge of the board there cannot be a four-corner violation.

Remark: There is a potential third problem case, where two wires are directly adjacent with only the outer sheathing (2 columns) between them (see Figure 3-8, which has this property). This can be dealt with in either of two ways: ensuring that no wires are directly adjacent to each other by stretching the instance horizontally, or noting that the meeting points of the outer sheathing of the two adjacent wires can be adjusted to not produce a four-corner violation between them.

Proposition 3.3.15. *If the gadgets can all be satisfied, the filler clues can also be satisfied.*

Proof. Each filler clue will be satisfied by a rectangle covering its entire associated area; the cells horizontally adjacent to wires will be filled by two width-1 vertical rectangles from the outer sheathing clues, one coming from the clause gadget above and the other coming from the variable gadget below. The meeting point between the two outer sheathing rectangles can be adjusted as needed to avoid a four-corner violation. As mentioned, we have only two problem cases: (i) a corner of the filler rectangle meets a gadget and protruding wire; and (ii) corners of two sections meet on the side of a wire. Because both cases involve the side of a wire, we can avoid violations in either case by appropriately adjusting the meeting point of the sheathing clues.

(i) To avoid having a corner where the corner of the filler section meets the wire and gadget, the meeting point of the two sheathing clues can be placed on the edge (not corner) of the filler section, thus avoiding a four-corner violation since the corner of the filler section meets the edge of one of the sheathing rectangles.

(ii) As long as the meeting point of the two sheathing rectangles of the wire is not at the point where the two filler sections meet, there is no four-corner violation. The meeting point can trivially be placed on the side of a filler section (while still respecting the parity of the wire as expressed by the inner sheathing).

Therefore, since the sheathing can always be adjusted to accommodate filler rectangles, the satisfiability of the Tatamibari instance depends only on the gadgets. □

3.3.6 Finale

Now we can show that Tatamibari is NP-hard. Let f be the reduction, which takes an instance Φ of planar rectilinear monotone 3SAT and returns a Tatamibari instance $f(\Phi)$; we want to show:

Proposition 3.3.16. *If Φ has n variables and m clauses, then $f(\Phi)$ has size polynomial in $n+m$, and can be computed in time polynomial in $n+m$.*

Proof. Our construction expands the given planar rectilinear monotone 3SAT instance by a constant factor. Therefore it suffices to prove that planar rectilinear monotone 3SAT is strongly NP-hard when given the coordinates of the rectilinear drawing. Indeed, the height of the drawing is $O(m)$ and the width of the drawing is $O(e)$ if the graph has e edges, which is $O(m+n)$ by planarity. □

Proposition 3.3.17. *If Φ has a solution, then $f(\Phi)$ also has a solution.*

Proof. We begin by taking the solution to Φ and setting the variable gadgets' profiles according to those values; by Lemma 3.3.9, they will all be locally solvable. By Lemma 3.3.12, since each clause gadget is connected to wires representing variables which satisfy the clause, there must be a solution to the clause gadget. Furthermore, by Proposition 3.3.15, if the gadgets are satisfiable then the rest of the space can be filled without contradiction, producing a solution to $f(\Phi)$. □

Proposition 3.3.18. *If Φ has no solution, then $f(\Phi)$ also has no solution.*

Proof. We prove the equivalent statement that if $f(\Phi)$ has a solution, then Φ must also have a solution.

First, we prove that any solution to $f(\Phi)$ must correspond to some setting of the variables $x_1, \ldots, x_n$ of Φ. This is a consequence of Lemma 3.3.8, which shows that all wires in a single variable gadget must carry the same value, which is then taken as the setting for that variable.

Next, we have to show that these settings of the variables x_i are a solution of Φ; to do this, note that by Corollary 3.3.2 each wire ending in a clause gadget must carry its value into this clause gadget; and by Corollary 3.3.11 and Lemma 3.3.12 there is a solution to the clause gadget if and only if the wires represent values which satisfy the clause.

Thus, the values of the variable gadgets must be a solution to Φ. □

The above three propositions imply our desired result:

Theorem 3.3.19. *Tatamibari is (strongly) NP-hard.*

Because a given Tatamibari solution can be trivially checked in polynomial time, this theorem implies that Tatamibari is NP-complete.

3.4 Solving Tatamibari with Z3

In this section we describe our Tatamibari solver[3] based on the SMT (satisfiability modulo theories) solver Z3 [42]. The solver is implemented in Python using Z3's Python interface. We used the solver during the development of our gadgets, especially the clause gadget, to verify the gadgets are solvable exactly when they should be. Section 3.5 presents the solver-guided evolution of the clause gadget.

Variables. The solver creates one Boolean variable for each possible rectangle. A rectangle is possible if it contains exactly one clue and the rectangle's aspect ratio matches the clue. The variable will be assigned true if the rectangle is used in the solution to the puzzle.

Constraints. In standard operation, the solver creates three sets of pseudo-Boolean constraints. The first set of constraints enforces that exactly one rectangle corresponding to each clue is part of the solution. The second set of cosntraints enforces that each cell is covered by exactly one rectangle. The third set of constraints enforces the four-corner constraint. While enumerating the possible rectangles, the solver maintains four maps of cells to lists of rectangles having their upper-left, upper-right, lower-left, and lower-right corners in that cell. Then for each grid point in the interior of the puzzle, the solver looks up the four lists corresponding to the cells incident on that grid point and iterates over all of them (a quadruply nested loop) adding the constraint that at most three of each set of four rectangles meeting at that grid point are part of the solution.

As an aid to gadget development, the solver supports ignoring some of the constraints or making the constraints soft (minimizing the number of violations). When the one-rectangle-per-clue constraints are ignored, allowing rectangle shapes that do not match their contained clue, the solver creates a separate set of covering constraints based on individual cells. The user can request an exact covering (each cell is covered exactly once), an overapproximate covering (each cell is covered at least once, and the solver minimizes multiple covering), an underapproximate covering (each cell is covered at most once, and the solver minimizes uncovered cells), or an incomparable covering (the solver minimizes the total number of multiply-covered or uncovered cells). The four-corner constraint can also be ignored or made soft.

As another aid to gadget development, the solver also supports three-corner constraints at reflex corners of nonrectangular puzzles. Violations of the three-corner constraint become violations of the four-corner constraint when clues from another gadget cover the cell just outside the gadget's reflex corner, so for the gadget to behave as desired, it is required that there are no three-corner violations.

Input and output. The solver reads simple text files in which each character is either a clue symbol, a period indicating an empty cell, or a space indicating a cell not part of the puzzle (for nonrectangular puzzles). The solver outputs a tab-separated

[3] https://github.com/jbosboom/tatamibari-solver

file with the input cell symbols plus integers indicating which rectangle each cell is in. These output files can be visualized by SVG Tiler[4] to produce figures like those in this chapter.

3.5 Solver-Guided Development of Clause Gadget

We used the solver described in Section 3.4 as we developed the gadgets for our hardness proof, checking them as we simplified and improved the gadgets. For example, Figure 3-13 shows the sequence of different clause gadgets we designed, extracted from our Git history for this work. This figure should be read in normal reading order, left to right, top to bottom. Our initial attempt at a clause gadget lacked the rows and columns of single-cell squares that isolate the clause gadget from the rest of the puzzle. We also had not yet adopted inner sheathing for the wires, so the left wire is free to flip its signal. After adding these features, further development of the clause gadget mostly served to regularize clue positions within the gadget, make the gadget narrower and shorter, and adapt the interface with the wires on the bottom of the gadget to avoid four-corner violations. During the development process we used the solver to check the clause is solvable when one of the inputs is true and not solvable when all inputs are false.

[4] https://github.com/edemaine/svgtiler

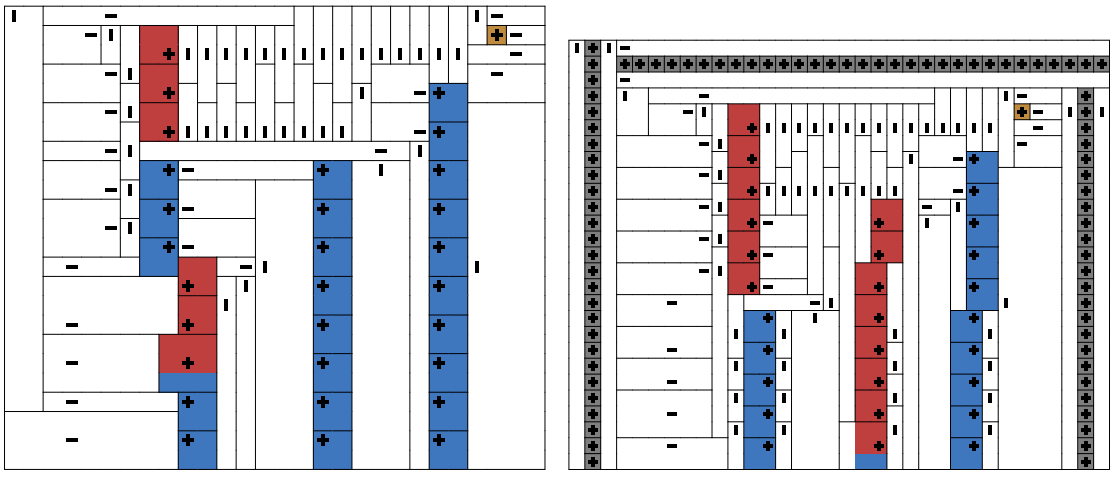


Figure 3-13: Evolution of the clause gadget.

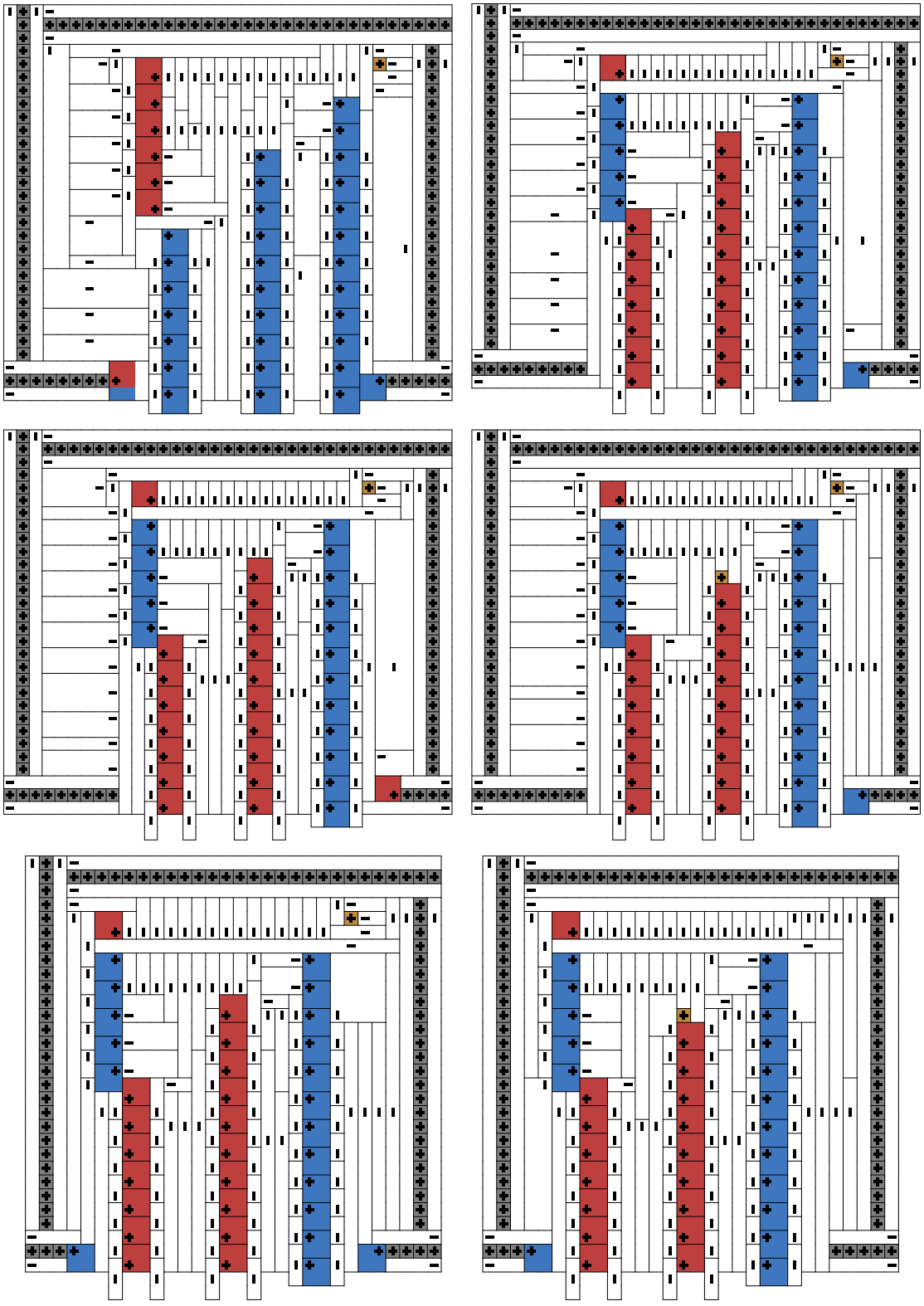


Figure 3-13: Evolution of the clause gadget.

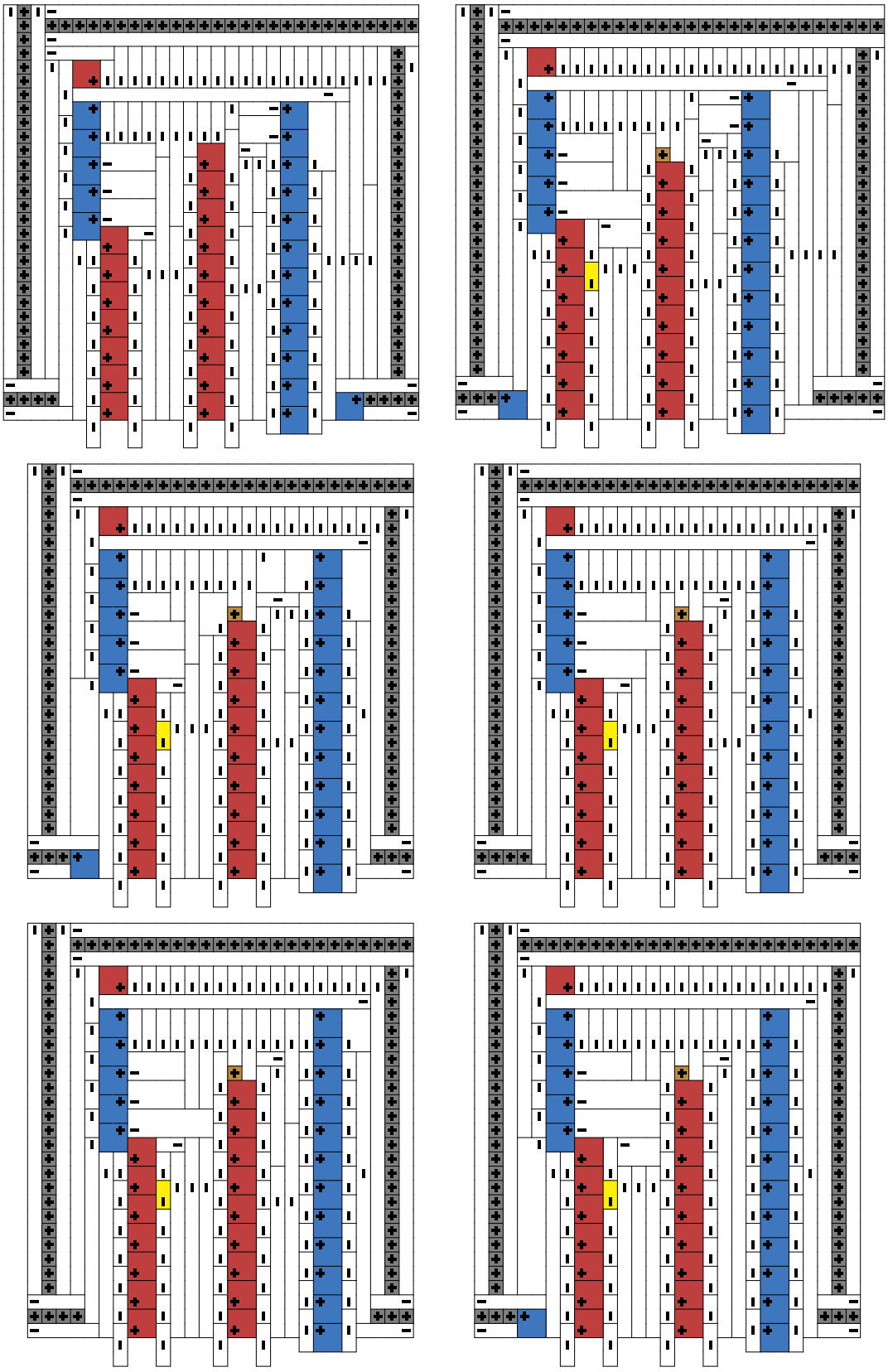


Figure 3-13: Evolution of the clause gadget.

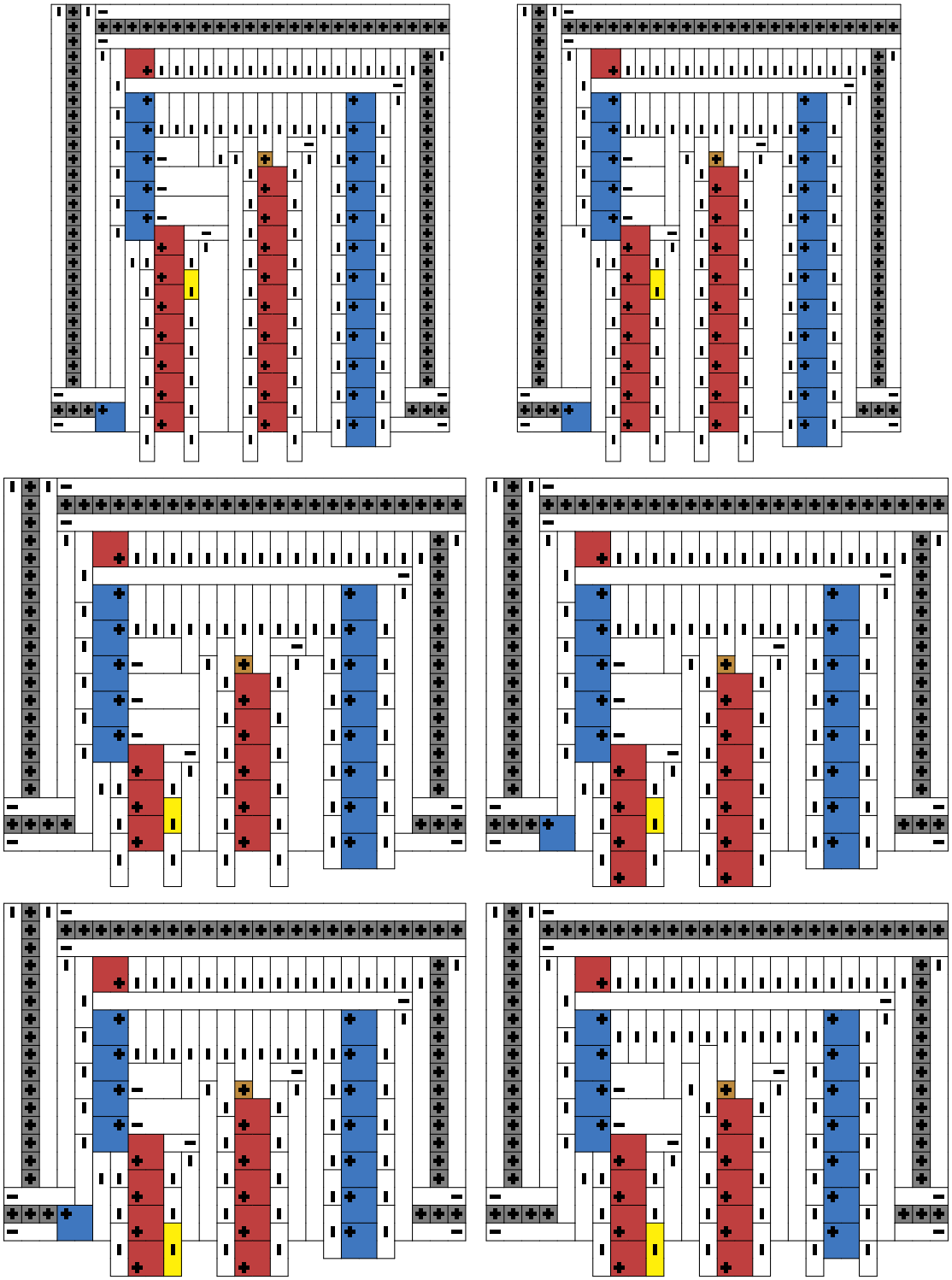


Figure 3-13: Evolution of the clause gadget.

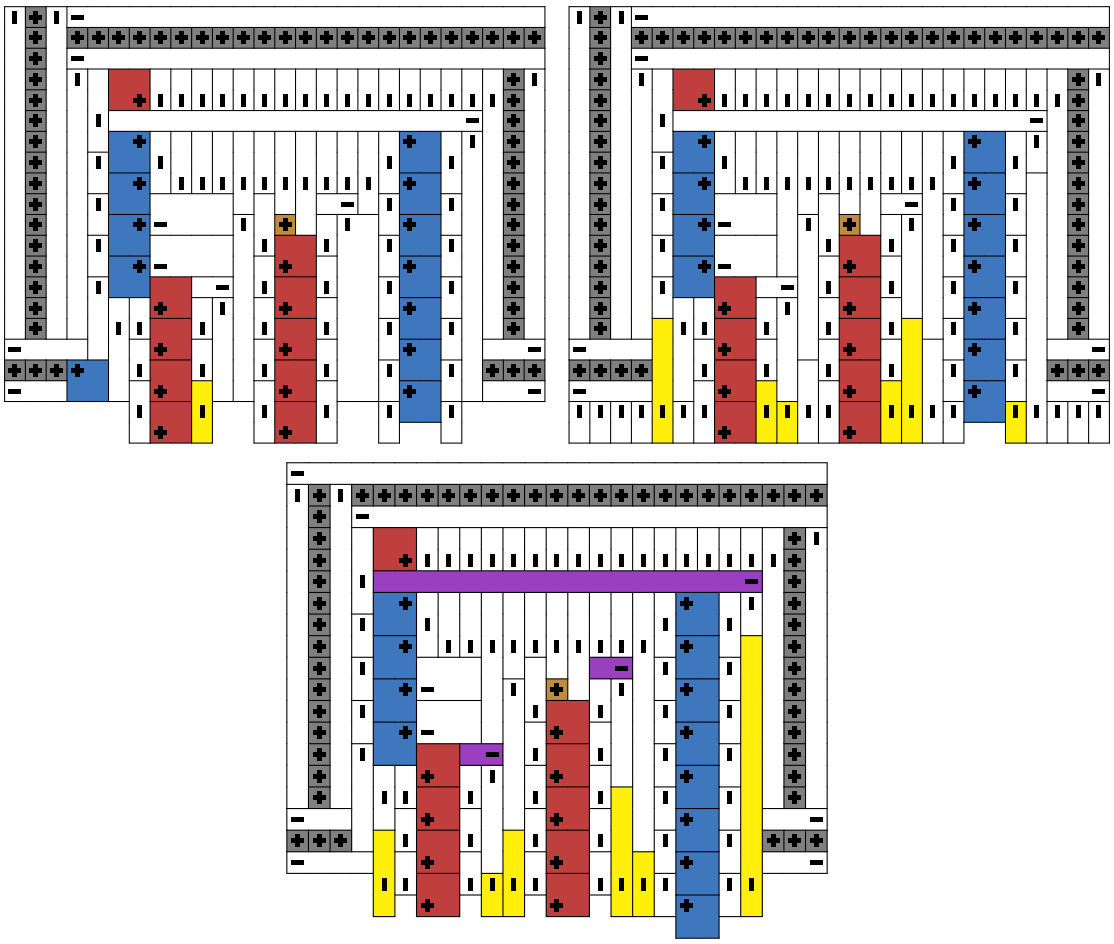


Figure 3-13: Evolution of the clause gadget.

3.6 Font

Figure 3-14 shows a series of twenty-six 10×10 Tatamibari puzzles that we designed, whose unique solutions shown in Figure 3-15 reveal each letter A–Z. To represent a bitmap image in the solution of a Tatamibari puzzle, we introduce two colors for clues, light and dark, and similarly shade the regions corresponding to each clue. As shown in Figure 3-15, the letter is drawn by the dark regions from dark clues. These puzzles were designed by hand, using the solver described in Section 3.4 to iterate until we obtained unique solutions. The font is also available online.[5]

3.7 Open Problems

There remain interesting open questions regarding the computational complexity of Tatamibari. When designing puzzles, it is often desired to have a single unique solution. We suspect that Tatamibari is ASP-hard (NP-hard to determine whether it has another solution, given a solution), and that counting the number of solutions is #P-hard. However, our reduction is far from parsimonious. Some rework of the gadgets, and a unique filler between gadgets, would be required to preserve the number of solutions.

We could ask about restrictions or natural variations of Tatamibari. For example, we are curious whether a Tatamibari puzzle with only ✚ clues, or only ▮ clues, remains hard. We have also wondered about the version of the puzzle without the four-corner constraint. Although initially we thought of the four-corner constraint as a nuisance to be overcome in our reduction, our final proof uses it extensively and centrally.

3.8 Example: Spiral Galaxies

As an example of the gadget area hardness framework, we show how the NP-hardness proof for Spiral Galaxies from [63] can be described using the framework. A Spiral Galaxies puzzle is a rectangular grid with clues in some grid cells or on some grid lines. The goal is to partition the puzzle into areas with a single clue per area such that the area is rotationally symmetric about the clue.

We reduce from planar[6] Boolean circuit satisfiability. We have a wire gadget, a variable gadget, NOT and AND gadgets, a fanout (wire duplicator) gadget, and a vertical shift gadget. We lay out these gadgets to overlap in their optional areas (only), and communicate a truth value in whether the optional area is covered or not.

Wire. The wire gadget consists of repeating pairs of clues three grid units apart. There are two gadget solutions, shown in Figure 3-17: repeating 3×2 rectangles, in

[5] http://erikdemaine.org/fonts/tatamibari/

[6] Friedman's proof [63] provides a crossover gadget, but it is not necessary because AND and NOT build a crossover [68].

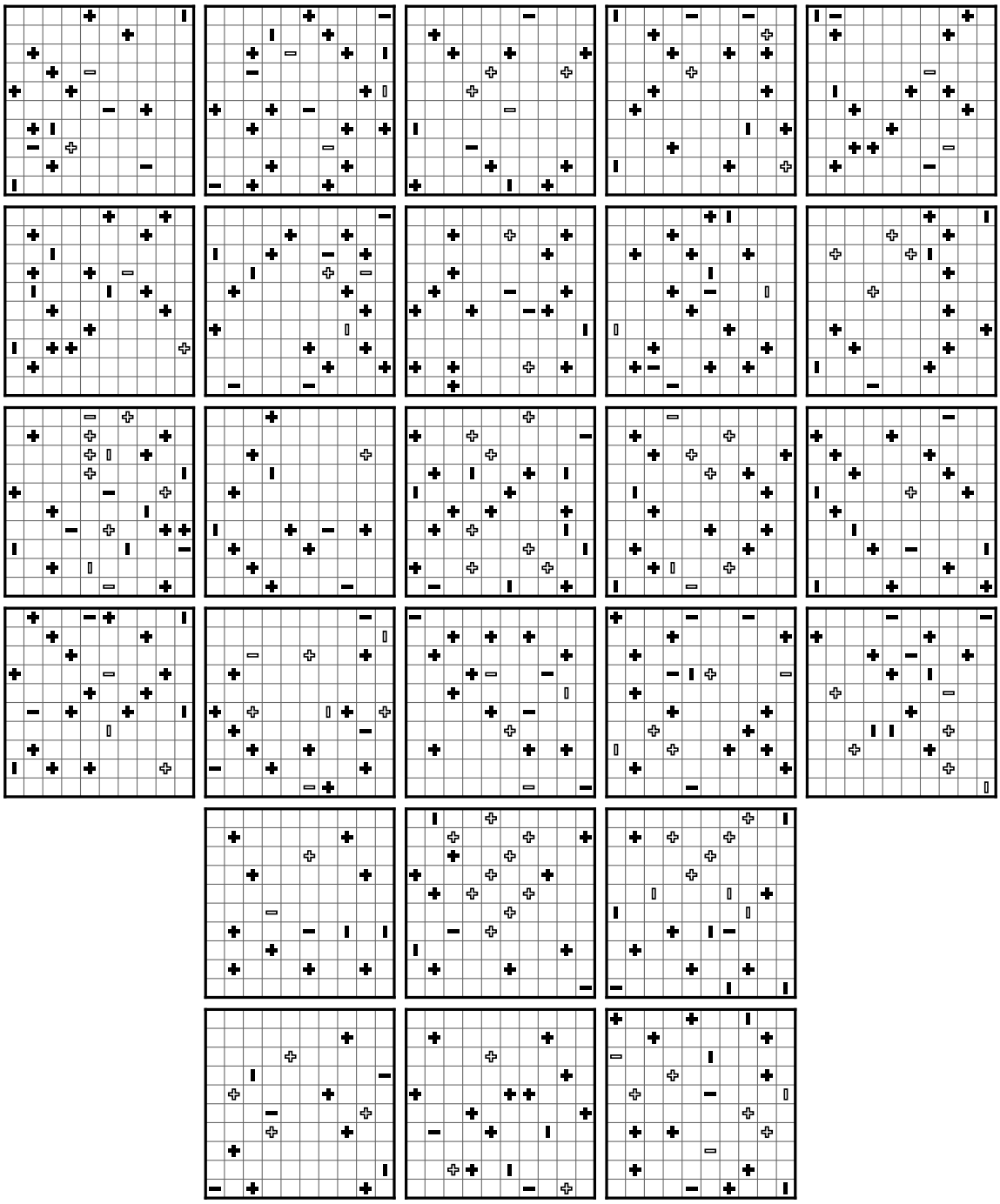


Figure 3-14: Puzzle font: each puzzle has a unique solution whose regions for dark clues (shown in Figure 3-15) form the shape of a letter.

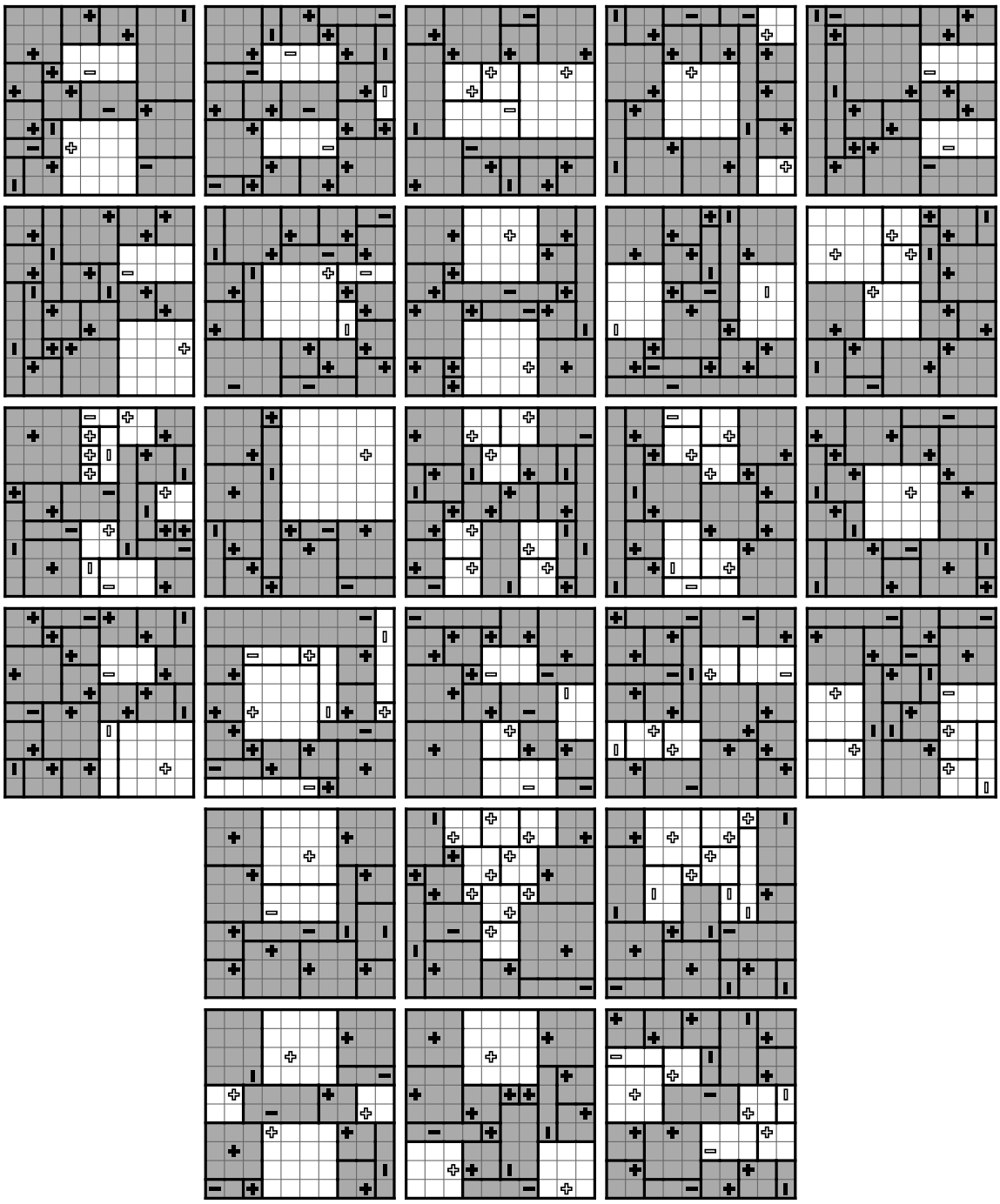


Figure 3-15: Solved font: unique solutions to the puzzles in Figure 3-14.

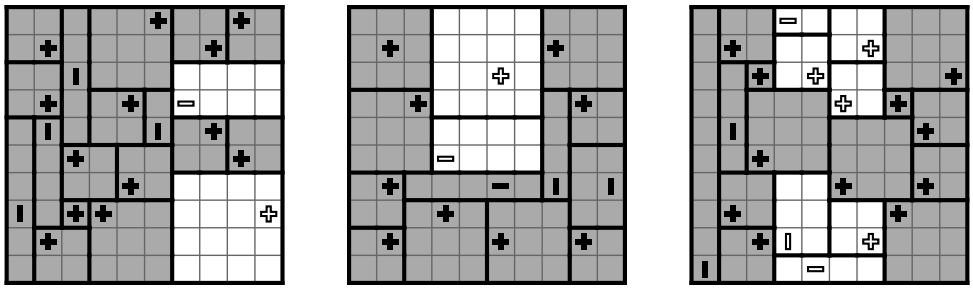
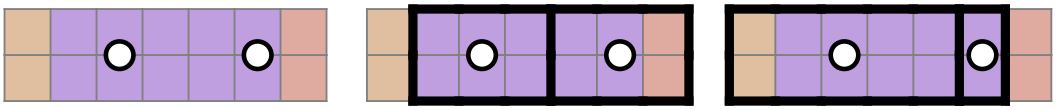


Figure 3-16: Solution to Figure 3-1.



(a) Unsolved wire gadget

(b) Wire carrying true signal: 3×2 rectangles

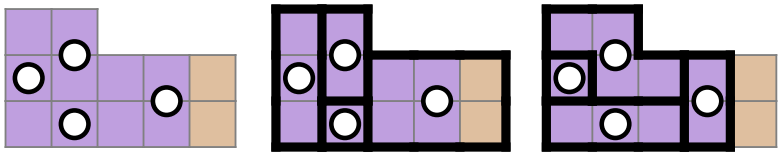
(c) Wire carrying false signal: alternating 1×2 and 5×2 rectangles

Figure 3-17: Spiral Galaxies wire and its profile table (true and false solutions)

which case the wire covers the right optional area, and alternating 1×2 and 5×2 rectangles, in which case the wire covers the left optional area. The wire carries a true signal when it covers the right optional area and false when it covers the left optional area. The wire gadget can be extended to arbitrary length in units of two clues. (The proof in [63] does not explicitly state this parity requirement, but the gate gadgets assume the true signal protrudes into the gadget to cover the optional input area and the false signal does not.)

Boolean circuit satisfiability requires the circuit produce a true output. We can force a wire to be true simply by terminating it. Because the wire has height two, any filler clues to the right of the wire cannot cover area in the wire gadget, so the wire must end in a 3×2 rectangle to cover the right optional area, forcing the rest of the wire to also carry a true signal.

Variable. The variable gadget is shown in Figure 3-18. There are two gadget solutions, one leaving the optional area uncovered (so the adjacent wire is set to true) and the other covering it (so the adjacent wire is set to false). Choosing one solution

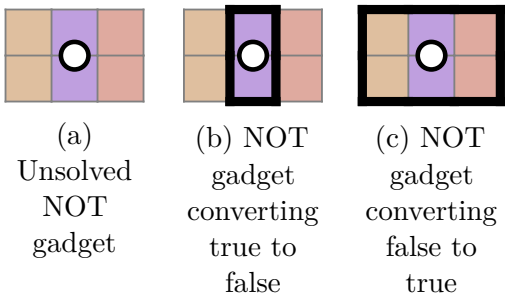


(a) Unsolved variable gadget

(b) Variable set to true

(c) Variable set to false

Figure 3-18: Spiral Galaxies variable and its profile table (true and false solutions)



(a) Unsolved NOT gadget

(b) NOT gadget converting true to false

(c) NOT gadget converting false to true

Figure 3-19: Spiral Galaxies NOT gadget and its profile table

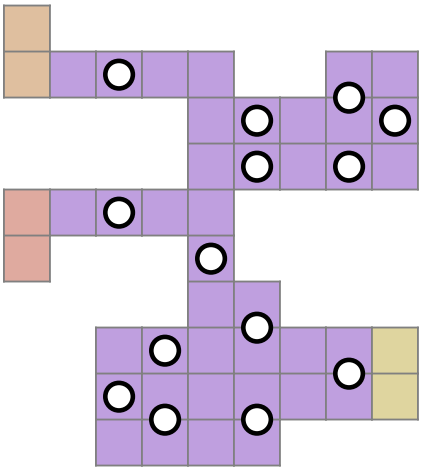
or the other corresponds to assigning true or false to the variable.

NOT. The NOT gadget is shown in Figure 3-19. If the left optional area is covered by the input wire (carrying a true signal), the clue in the NOT gadget must cover a 1×2 rectangle, so the right optional area must be covered by the output wire carrying a false signal. If the left optional area is uncovered (when the input wire is false), the clue in the NOT gadget covers both optional areas, so the output wire must carry a true signal. Thus the NOT gadget inverts the wire's signal.

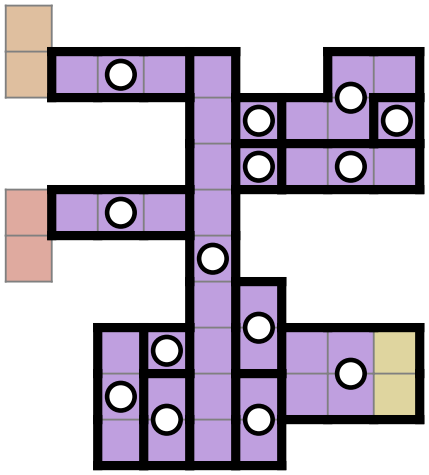
AND. The AND gadget is shown in Figure 3-20. When both inputs are true, both of the left optional areas are covered by the wire, so the clues to the right of the optional area are rectangles and the clue at the center of the gadget is a long vertical rectangle, allowing the right optional area to be covered, propagating a true signal from the gadget. When either or both of the inputs are false, one or both of the left optional areas must be covered by the clue(s) to the right of the areas, blocking the central clue from covering a vertical rectangle, preventing the right optional area from being covered, thus propagating a false signal from the gadget.

Fanout. Like the AND gadget, the fanout gadget (Figure 3-21) is also based around forming or not forming a vertical rectangle. The upper optional area is the input. When it is covered by the input wire (a true signal), the central clue cannot form a vertical rectangle, so the upper-right optional area must be covered by the clue to its left, and because the bottom cell in the central column is covered by the clue to its upper-left, the lower-right optional area must also be covered by the clue to its left. When the upper optional area is not covered by the input wire, it must be covered by the central clue forming a vertical rectangle, so the output optional areas cannot be covered by the clues to their left.

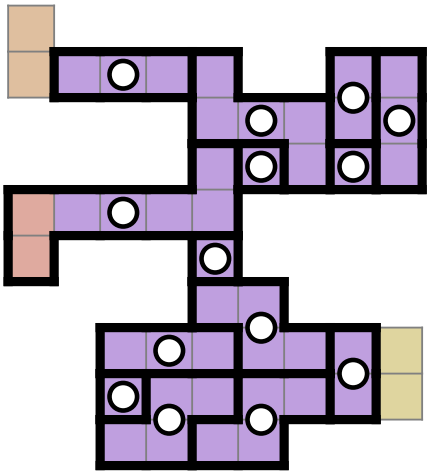
Shift. Because variable and gate outputs are on the right and gate inputs are on the left, we do not need a turn gadget, but we do need to shift wires vertically, which is done using the shift gadget. An upward shift gadget is shown in Figure 3-22; the downward shift gadget is that gadget's reflection across the horizontal axis. When the input wire is true, the input wire covers the left optional area, so the left clue is



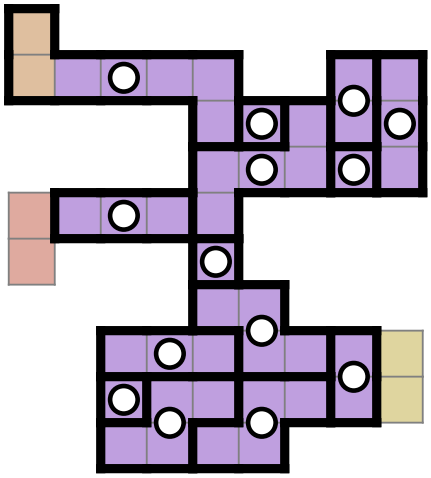
(a) Unsolved AND gadget (inputs at left)



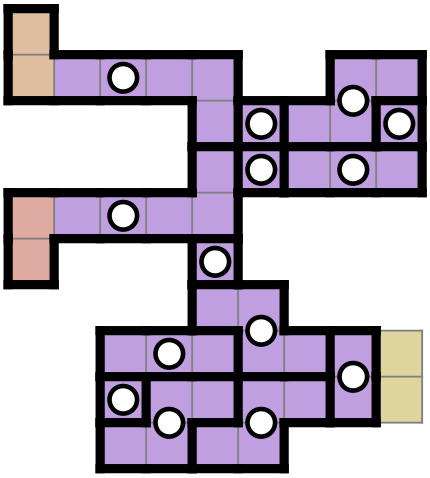
(b) AND with true and true inputs



(c) AND with true and false inputs

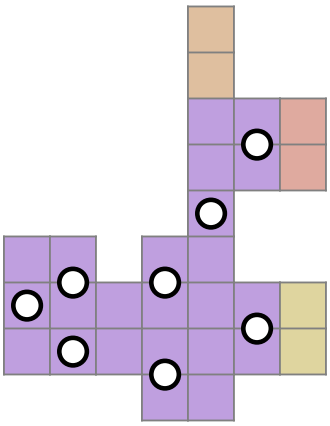


(d) AND with false and true inputs

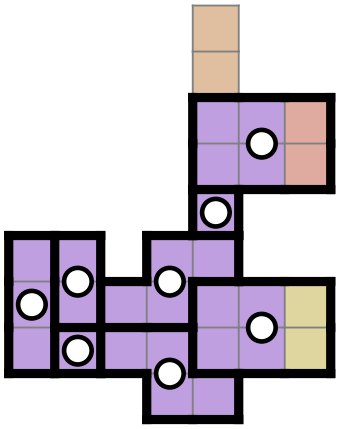


(e) AND with false and false inputs

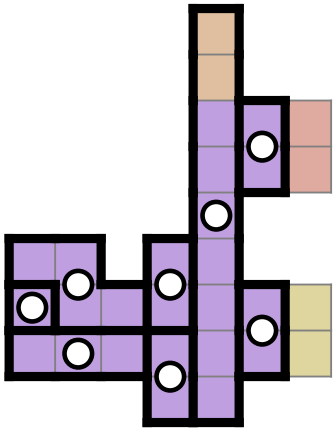
Figure 3-20: Spiral Galaxies AND gadget and its profile table



(a) Unsolved fanout gadget (input at top)

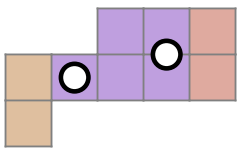


(b) Fanout gadget duplicating true wire

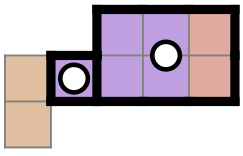


(c) Fanout gadget duplicating false wire

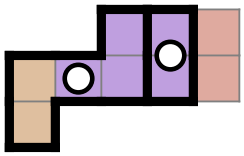
Figure 3-21: Spiral Galaxies fanout gadget and its profile table



(a) Unsolved shift gadget



(b) Shift gadget shifting a true wire



(c) Shift gadget shifting a false wire

Figure 3-22: Spiral Galaxies upward shift gadget and its profile table; the downward shift gadget is this gadget flipped vertically

covered by a single cell and the right clue covers the right optional area, propagating true on the output. When the input wire is false, the left clue covers the left optional area and forces the right clue to be a 1×2 rectangle, leaving the right optional area uncovered, propagating false on the output.

Layout. Friedman's proof in [63] omits discussion of layout, but we sketch a layout algorithm here. We start with a grid embedding of the input planar Boolean circuit. We scale the grid by at least 6 so that our wire gadget fits for unit-length wires, but possibly by a greater factor if the grid embedding has long vertical segments, because our shift gadget consumes horizontal distance to move vertically.

Filling algorithm. The filling algorithm places a clue in the center of every cell that isn't part of a gadget, forcing them to be covered by single-cell areas. Filler clues could only cover area in a gadget if two cells in the gadget area are separated by one filler clue and those cells do not themselves have clues. This is avoided in all gadgets by ensuring all gadget cells that are separated by filler are separated by two or more filler cells, so only local gadget solutions are possible.

Composition algorithm. The local gadget solutions are already consistent with each other, so to form an area assignment for the entire puzzle, the composition algorithm simply takes the local gadget solutions and assigns each filler clue to the cell containing it.

Proper and complete profile tables. The profile tables are proper because they contain only proper profiles. Because the filler clues cannot cover area in the gadgets, we can verify by case analysis that the profile tables are complete (all other profiles are locally impossible). This completes the proof.

Chapter 4

Push Fight[1]

4.1 Introduction

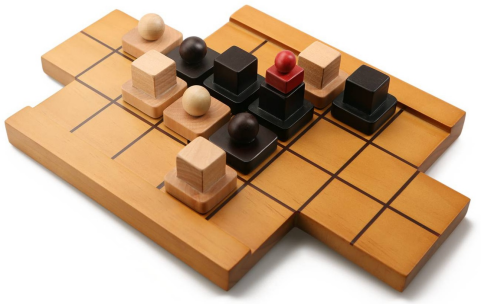


Figure 4-1: A Push Fight game in progress. Photo by Brettco, Inc., used with permission.

Push Fight [115] is a two-player board game, invented by Brett Picotte around 1990, popularized by Penny Arcade in 2012 [93], and briefly published by Penny Arcade in 2015 [77]. Players take turns moving and pushing pieces on a square grid until a piece gets pushed off the board or a player is unable to push on their turn. Figure 4-1 shows a Push Fight game in progress, and Section 4.2 details the rules.

In this chapter, we study the computational complexity of optimal play in Push Fight, generalized to an arbitrary board and number of pieces, from two perspectives:

1. **Who wins?** The typical complexity-of-games problem is to determine which player wins from a given game configuration.

2. **Mate-in-1**: Can the current player win *in a single turn*?

[1]This chapter, except for Section 4.5, is from [32] (also available on arXiv [33]), which is joint work with Erik D. Demaine and Mikhail Rudoy.

	Computational complexity of...	
Moves per turn	**Mate-in-1**	**Who wins?**
≤ 2	P	PSPACE-hard, in EXPTIME
$\leq c$ constant	P	open
$\leq k$ input	NP-complete	open
unlimited	P	open

Table 4.1: Summary of our results.

Table 4.1 summarizes our complexity results. We then describe our efforts to solve Push Fight using computer search. We were not able to solve standard Push Fight, but we determined that Push Fight played on a board with one column (four squares) removed is a draw under perfect play.

Generalized Push Fight is a two-player game played on a polynomially bounded board for a potentially exponential number of moves, so we conjecture the "who wins?" decision problem to be EXPTIME-complete, as with Checkers [119] and Chess [60]. (Certainly the problem is in EXPTIME, by building the game tree.) In Section 4.4, we prove that the problem is at least PSPACE-hard, using a proof patterned after the NP-hardness proof of Push-$*$ [76]. Our proof uses a simple, nearly rectangular board, in the spirit of the original game; in particular, the board we use is hole-free and x-monotone (see Figure 4-9). It remains open whether Push Fight is in PSPACE, EXPTIME-hard, or somewhere in between.

Our mate-in-1 results are perhaps most intriguing, showing a wide variability according to whether and how we generalize the "up to two moves per turn" rule in Push Fight. If we leave the rule as is, or generalize to "up to c moves per turn" where c is a fixed constant (part of the problem definition), then we show that the mate-in-1 problem is in P, i.e., can be solved in polynomial time. However, if we generalize the rule to "up to k moves per turn" where k is part of the input, then we show that the mate-in-1 problem becomes NP-complete. On the other hand, if we remove the limit on the number of moves per turn, then we show that the mate-in-1 problem is in P again. Section 4.3 proves these results.

The mate-in-1 problem has been studied previously for other board games. The earliest result is that mate-in-1 Checkers is in P, even though a single turn can involve a long sequence of jumps [59]. On the other hand, Phutball is a board game also featuring a sequence of jumps in each turn, yet its mate-in-1 problem is NP-complete [46].

After presenting our complexity results, we describe a computer search system aiming to solve Push Fight in Section 4.5. Unlike most computer searches to solve games, which are based on retrograde analysis (working backward by undoing moves), our system is based on repeating a forward search until it reaches a fixed point. We find that Push Fight played on a board with one column removed is a draw under perfect play, and we believe our system extends to standard Push Fight given sufficient computation time.

4.2 Rules

The original Push Fight board is an oddly shaped square grid containing 26 squares; see Figure 4-2. Part of the boundary of this board has *side rails* which prevent pieces from being pushed off across those edges. We generalize Push Fight by considering arbitrary polyomino boards, with each boundary edge possibly having a side rail.

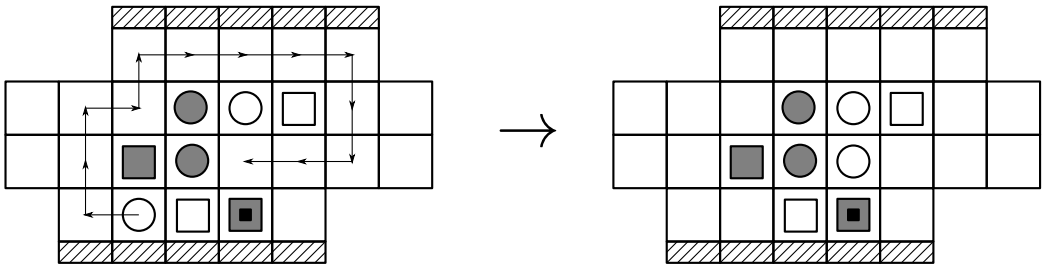


Figure 4-4: An example move.

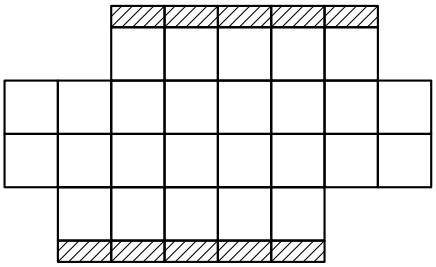


Figure 4-2: The original Push Fight board. The shaded regions represent side rails.

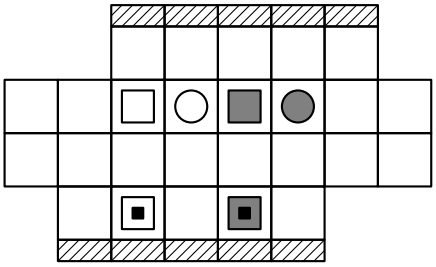


Figure 4-3: Our notation for pieces. From left to right in the middle of the second row: a white king, a white pawn, a black king, and a black pawn. The bottom row shows an anchored king of each color (in an actual game, there is only one anchor).

Push Fight is played with two types of pieces, each of which takes up a square of the board: *pawns* (drawn as circles) and *kings* (drawn as squares). Each piece is colored either black or white, denoting which player the piece belongs to. Standard Push Fight is played with three kings and two pawns per player. Additionally, there is a single *anchor* that is placed on top of a king after it pushes (but is never placed directly on the board). Figure 4-3 shows our notation for the pieces.

Push Fight gameplay consists of the two players alternating *turns*. During a player's turn, the player makes up to two optional *moves* followed by a mandatory *push*.

To make a move, a player moves one of their pieces along a simple path of orthogonally adjacent empty squares; see Figure 4-4.

To push, a player moves one of their kings into an occupied adjacent square. The piece occupying that square is pushed one square in the same direction, and this continues recursively until a piece is pushed into an unoccupied square or off the board. If this process would push a piece through a side rail, or would push the anchored king, the push cannot be made. Pushes always move at least one other piece. When the push is complete, the pushing king is anchored (the anchor is placed on top of that king). Figure 4-5 shows a valid push.

A player loses if any of their pieces are pushed off the board (even by their own push) or if they cannot push on their turn.

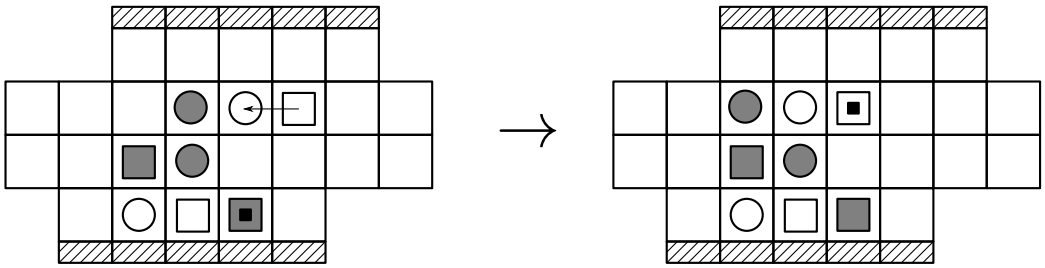


Figure 4-5: An example push.

Push Fight does not have a fixed initial position. Instead, before the first turn is taken, white places their pieces on the half of the board closest to them, then black places their pieces on their half of the board. (In our hardness proofs, our reduction produces a game in progress, so this opening procedure is not relevant.)

Definition 4.2.1. *A* Push Fight game state *is a description of the board's shape, including which board edges have side rails, and for each board square, what type of piece or anchor occupies it (if any).*

Note that the position of the anchor encodes which player's turn it is: if the anchor is on a white king, it is black's turn, and vice versa. If the anchor has not been placed (no turns have been taken), it is white's turn.

4.3 Mate-in-1

We consider three variants of mate-in-1 Push Fight, varying in how the number of moves is specified: as a constant in the problem definition, as part of the input, or without a limit.

4.3.1 c-Move Mate-in-1

Problem 4.3.1. c-Move Push Fight Mate-in-1: *Given a Push Fight game state, can the player whose turn it is win this turn by making up to c moves and one push?*

The standard Push Fight game has $c = 2$.

Theorem 4.3.2. c-Move Push Fight Mate-in-1 *is in P.*

Proof sketch. The number of possible turns is $\leq A^{2c+4}$ on a board of area A. □

Proof. Let A denote the area of the board. There are at most A of the current player's pawns (because every piece occupies a square). On each of the c moves, any of those pawns may move to any of those squares, for a maximum of A^2 possibilities for each move. ("Moving" pawns to their own square represents making fewer than c moves.) Then there are at most A kings of the current player, each of which can potentially push in four directions. Thus there are at most A^{2c+4} possible turns.

Checking that a turn is legal and results in the current player winning requires checking that the moves are all legal and that the push is legal and leads to a win. A move can be verified in polynomial time by finding a path of unoccupied squares between the pawn's start and end positions. A push can be checked in polynomial time by scanning across the board in the direction of the push to see if one of the other player's pieces is pushed off the board, or if the push is invalid because of the anchored king or a side rail. □

4.3.2 k-Move Mate-in-1 is in NP

Problem 4.3.3. k-Move Push Fight Mate-in-1: *Given a Push Fight game state and a positive integer k, can the player whose turn it is win this turn by making up to k moves and one push?*

In this section, we prove the following upper bound on the number of useful moves in a turn:

Theorem 4.3.4. *Given a Push Fight game state on a board having n squares, if the current player can win this turn, they can do so using at most n^6 moves followed by a push.*

Proof sketch. We divide the reachable game states into $\leq n^4$ equivalence classes, and show that two equivalent configurations can be reached via $\leq n^2$ moves within that class. □

Our bound directly implies an NP algorithm for k-Move Push Fight Mate-in-1:

Corollary 4.3.5. k-Move Push Fight Mate-in-1 *is in* NP.

Proof. Use the following NP algorithm. Given a Push Fight game state on a board with n squares, nondeterministically choose an integer m between 0 and $\min\{k, n^6\}$, then nondeterministically choose m moves (a piece and a destination) and one push (a king and a direction). Accept if and only if the chosen moves and push are legal and result in a win for the current player. □

The remainder of this section is devoted to proving Theorem 4.3.4.

A turn consists of making some number of moves followed by a single push. For the purpose of analyzing a single turn, kings other than the single king that pushes are indistinguishable from pawns, so we can assume the current player first chooses a king, then replaces all of their other kings with pawns before making their moves and push. The following definitions are based on this assumption.

Definition 4.3.6. *Given a single-king game state, a* board configuration *is a placement of pieces reachable by the current player making a sequence of moves.*

Definition 4.3.7. *The* pawnspace *of a board configuration is the (possibly disconnected) region of the board consisting of the empty squares and the squares containing the current player's pawns. Equivalently, the pawnspace is the region consisting of all squares not occupied by the current player's king or the other player's pieces.*

Definition 4.3.8. *The* signature *of a board configuration is a list of nonnegative integers, where each integer is a count of the current player's pawns in a connected component of the configuration's pawnspace, ordered according to row-major order on the leftmost topmost square in the corresponding connected component.*

Definition 4.3.9. *Given two board configurations C_1 and C_2 derived from the same game state, we say that $C_1 \equiv C_2$ if and only if*

1. *C_1 and C_2 have the same pawnspace (that is, the current player's only king occupies the same square in C_1 and C_2) and*

2. *C_1 and C_2 have the same signature (that is, each connected component of the pawnspace contains the same number of the current player's pawns in C_1 and C_2).*

Relation $\equiv$ is clearly reflexive, symmetric, and transitive, so it is an equivalence relation inducing a partition of the set of board configurations derived from a given game state into equivalence classes. We need the following two lemmas about $\equiv$ for our proof of Theorem 4.3.4:

Lemma 4.3.10. *For a given game state on a board with n squares, there are at most n^4 equivalence classes of board configurations.*

Proof. Let s be the square occupied by the current player's king. There are at most n choices for s, so it remains to show that, for each s, there are at most n^3 equivalence classes where the current player's king occupies s.

The choice of s, together with the game state (containing the position of the other player's pieces), defines the pawnspace for all board configurations having the current player's king at s. For each connected component R of the pawnspace, s is either adjacent to R or not. In the case where it is not, R is surrounded by the boundary of the board and/or the other player's pieces, so no sequence of moves can change the number of the current player's pawns in R, and all board configurations have the same number of the current player's pawns in R.

Thus the only connected components that may have different numbers of pieces in different board configurations are the connected components bordering s, of which there are at most 4. Each of these components comprises at most $n-1$ squares and so contains between 0 and $n-1$ pieces. The total number of pieces in these components is invariant across all board configurations in each equivalence class, so the count of pawns in one of the components is fully determined by the others (that is, if there are q components, there are $q-1$ degrees of freedom). There are n possible values for each of up to 3 free-to-vary pawn counts, so there are at most n^3 equivalence classes in which the current player's king occupies s. Together with the at most n choices for s, there are at most n^4 equivalence classes of board configurations. □

Lemma 4.3.11. *If $C_1 \equiv C_2$, then C_2 can be reached from C_1 in at most $n^2 - 1$ moves without leaving the equivalence class of C_1.*

Proof. By the assumption that $C_1 \equiv C_2$, the current player's king and the other player's pieces are already in the same positions in C_1 and C_2. Notice that while moving the king may change the connected components of the pawnspace, moving pawns never does, nor can pawns move from one component to another (by the same argument as in Lemma 4.3.10). Thus to ensure we remain in the equivalence class of C_1, it is sufficient to give an algorithm that moves only pawns.

Initialize the current configuration C to be C_1. Call a pawn *misplaced* if it occupies a square in C that is empty in C_2. While there are misplaced pawns, choose one and let s denote the square it occupies. Let R be the connected component in the pawnspace[2] of C containing s and let T be the set of squares in R that contain pawns in C_2. By the assumption that $C_1 \equiv C_2$, there are the same number of pawns in R in C_1 and C_2, so because there is a misplaced pawn, there is at least one square $t \in T$ that is empty (a missing pawn). Let $s_1, s_2, \ldots, s_l$ be the squares containing pawns along a shortest path in R from s to t (with $s_1 = s$). Move the pawn at s_l to t, then move the pawn at s_{l-1} to s_l, and so on, finishing by moving the pawn at s_1 to s_2. The net effect of this sequence of moves is that t now holds a pawn and s no longer holds a pawn (so C now contains one fewer misplaced pawn). Continue with the next iteration of the loop.

During each iteration of the above loop, each pawn moves at most once. The number of misplaced pawns decreases by 1 each iteration, so the number of iterations is at most the number of pawns, of which there are at most $n - 1$. Thus the number of moves used to transform C_1 into C_2 is at most $(n-1)^2 \leq n^2 - 1$. □

We are now ready to prove Theorem 4.3.4:

Theorem 4.3.4. *Given a Push Fight game state on a board having n squares, if the current player can win this turn, they can do so using at most n^6 moves followed by a push.*

Proof. By our assumption that the current player can win this turn, there exists a sequence of moves for the current player after which they can immediately win with a push, corresponding to a sequence of board configurations $C_1, C_2, \ldots, C_l$. Configuration C_1 is obtained from the initial game state by replacing all of the current player's kings, except the one that ends up pushing, with pawns. Each C_{i+1} can be reached from C_i in one move, and C_l is a configuration from which the current player can win with a push.

We now define *simplifying* a sequence of board configurations over an equivalence class E. If the sequence contains no configurations from E, then simplifying the sequence over E leaves it unchanged. Otherwise, let A_i be the first configuration in the sequence in E and A_j be the last configuration in the sequence in E. By Lemma 4.3.11, there exists a sequence of fewer than $n^2 - 1$ moves that transforms A_i into A_j, corresponding to a sequence of board configurations $A_i = D_0, D_1, \ldots, D_u = A_j$ with $u \leq n^2 - 1$. Then simplifying over E consists of replacing all configurations between and including A_i and A_j with the replacement sequence $D_0, D_1, \ldots, D_u$.

[2]Being in the same equivalence class, C_1, C_2 and all values of C have the same pawnspace.

Notice that simplifying a sequence (over any class) never changes the first or last configuration in the sequence, and each configuration in the resulting sequence remains reachable in one move from the previous configuration in the resulting sequence. After simplifying over a class E, the only configurations in the resulting sequence in E are those in the replacement sequence, so the number of configurations in the sequence in E is at most n^2. Furthermore, all configurations in the replacement sequence are in E, so simplifying over E never increases (but may decrease) the number of configurations falling in other classes.

Let $C'_1, C'_2, \ldots, C'_l$ be the result of simplifying $C_1, C_2, \ldots, C_l$ over every equivalence class. By Lemma 4.3.10, there are at most n^4 such classes, and by the above paragraph there are at most n^2 configurations from each class in $C'_1, C'_2, \ldots, C'_l$, so the length of $C'_1, C'_2, \ldots, C'_l$ is at most n^6. Each configuration in $C'_1, C'_2, \ldots, C'_l$ is reachable in one move from the previous configuration, and that sequence of at most n^6 moves leaves the current player in position to win with a push, as desired. □

4.3.3 Unbounded-Move Mate-in-1

Problem 4.3.12. Unbounded-Move Push Fight Mate-in-1: *Given a Push Fight game state, can the player whose turn it is win this turn by making any number of moves and one push?*

Theorem 4.3.13. Unbounded-Move Push Fight Mate-in-1 *is in P.*

We can of course solve Unbounded-Move Push Fight Mate-in-1 by trying all possible sequences of moves to find a board configuration from which the current player can win with a push, but there are exponentially many board configurations, so such an algorithm takes exponential time. Instead, we can use the fact that any two configurations in the same equivalence class are reachable from each other in a polynomial number of moves (from Lemma 4.3.11) to search over equivalence classes of board configurations instead of searching over board configurations. There are at most n^4 equivalence classes (by Lemma 4.3.10), so they can be searched in polynomial time.

We will make use of the following definitions:

Definition 4.3.14. *Two equivalence classes of board configurations C_1 and C_2 are* neighbors *if there exist board configurations $b_1 \in C_1$ and $b_2 \in C_2$ such that b_1 can be reached from b_2 with a king move of exactly one square. The* equivalence class graph *is a graph whose vertices are equivalence classes of board configurations and whose edges connect neighboring equivalence classes.*

An equivalence class of board configurations C is a winning equivalence class *if there exists a board configuration $b \in C$ such that the player whose turn it is can win with a push.*

The key idea for our algorithm is the following:

Lemma 4.3.15. *There exists a path in the equivalence class graph from the equivalence class of the initial board configuration to a winning equivalence class if and only if there exists a winning move sequence.*

Proof. Let b_0 be the initial board configuration, and let C_0 be the equivalence class of b_0.

First, suppose that there exists a path $C_0, C_1, \ldots, C_k$ in the equivalence class graph which ends at some winning equivalence class. Consider any $i \in \{0, 1, \ldots, k-1\}$. Because C_i is adjacent to C_{i+1} in the equivalence class graph, equivalence class C_i neighbors C_{i+1}, and therefore there exist two board configurations $b'_i \in C_i$ and $b_{i+1} \in C_{i+1}$ such that b_{i+1} can be reached from b'_i with a king move of exactly one square. Because C_k is a winning equivalence class, there exists a board configuration b'_k such that the current player can win with a push. Then consider the sequence of board configurations $b_0, b'_0, b_1, b'_1, \ldots, b_k, b'_k$. For each i, b_i and b'_i are both in equivalence class C_i, so by Lemma 4.3.11, there exists a sequence of moves converting board configuration b_i into b'_i. Together with the single-square king moves between b'_i and b_{i+1}, we can use these sequences to form a winning move sequence from board configuration b_0 to board configuration b'_k.

Next, suppose there exists a winning move sequence. Break each king move along a path of more than one square in that sequence into a sequence of king moves of exactly one square. This yields a new winning move sequence whose moves are all king moves of one square or pawn moves. Pawn moves never change the equivalence class of the current board configuration, and single-square king moves always change the equivalence class of the current board configuration to a neighboring equivalence class. Thus, this sequence of moves corresponds to a path in the equivalence class graph. Because this is a winning move sequence, the last board configuration has the property that the player whose turn it is can win with a push, and so the last equivalence class in the path is a winning equivalence class. Thus there exists a path in the equivalence class graph from the equivalence class of the initial board configuration to a winning equivalence class. □

The size of the equivalence class graph is polynomial in n (by Lemma 4.3.10), so provided the graph can be constructed and the winning equivalence classes identified, this type of path in the equivalence class graph, if it exists, can be found in polynomial time.

Recall from Definition 4.3.9 that equivalence classes of board configurations are defined by the pawnspace and signature, and that, for configurations derived from the same game state (i.e., having the other player's pieces in the same positions), the pawnspace is defined by the position of the current player's king. Thus we can uniquely name a class using the king position and signature.

Definition 4.3.16. *The* class descriptor *of an equivalence class of board configurations for a given game state is the ordered pair of the position of the current player's king and the signature defining that class.*

To prove Theorem 4.3.13, we need to give polynomial-time algorithms to compute

the neighbors of an equivalence class and to decide whether a class is a winning equivalence class.

Lemma 4.3.17. *Given an initial game state and a class descriptor for some class C, we can compute in polynomial time the equivalence classes (as class descriptors) neighboring C.*

Proof. Moving the king to an adjacent square changes the pawnspace by adding the previously occupied square to a connected component and removing the newly occupied square from the pawnspace. Moving the king to an adjacent square may also join up to three components adjacent to the king's previous square or split a component containing the king's new square into up to three components. Other components are unaffected, and their corresponding signature elements are not modified.

In all cases, the component in the current configuration containing the king's new position must have area at least one greater than the number of pawns in that component. That is, there must be an empty square in that component. Otherwise, moving the king in that direction is not possible. All neighboring class descriptors specify the new king position, but how the signature is updated varies.

If the king move does not change the number of connected components of the pawnspace, the signature is left unchanged in the neighboring class descriptors.

If the king move joins components, then the neighboring class descriptor is updated by inserting a signature element for the newly joined component with value equal to the sum of the components it was created from, and deleting the signature elements corresponding to the joined components.

If the king move splits components, then there are potentially multiple neighboring class descriptors, one for each possible resulting signature. Suppose the signature value of the component being split is S. There is one neighboring class descriptor for each of the weak compositions[3] of S of length equal to the number of newly split components such that each term in the composition is at most the area of the corresponding newly split component. Each neighboring class descriptor's signature is updated by removing the element corresponding to the component having been split and inserting the terms of the composition in the positions corresponding to the newly split components.

There are up to four directions in which the king can move. Each of the above update procedures takes time polynomial in the number of class descriptors produced. By Lemma 4.3.10, there are only polynomially many classes, so only polynomially many descriptors can be produced, and thus we can compute the neighboring descriptors in polynomial time. □

Lemma 4.3.18. *Given an initial game state and a class descriptor for some class C, we can decide in polynomial time whether C is a winning equivalence class.*

Proof. We wish to decide whether C contains a board configuration such that the current player can win with one push.

[3] A weak composition of an integer is a way of writing that integer as a sum of other positive integers, where terms may be 0 and order is significant.

Consider each possible push direction. The king's position is specified in the class descriptor of C, so let L be the line of squares starting adjacent to the king in the chosen direction and continuing to the boundary of the board. Pushing in this direction results in a win exactly when

1. The square in L furthest from the king contains a piece belonging to the other player.
2. There is no side rail at the boundary edge at the end of L furthest from the king.
3. No square in L contains the anchored king.
4. Every square in L contains a piece.

The first three conditions depend only on the current player's king's position and the positions of the other player's pieces, information specified in the given class descriptor and game state, and so can be checked in polynomial time independent of any specific board configuration in C. Because the current player's pawns can move freely within the connected components of the pawnspace without leaving C, the fourth condition is equivalent to the condition that the intersection of L with each connected component of the pawnspace has area less than or equal to the number of the current player's pawns in that component.

Thus, we can check in polynomial time for each direction whether there exists a board configuration in C such that pushing in the chosen direction results in the current player winning. By repeating this check for all four possible push directions, we can decide whether C is a winning equivalence class in polynomial time. □

We are now ready to prove Theorem 4.3.13:

Theorem 4.3.13. Unbounded-Move Push Fight Mate-in-1 *is in P.*

Proof. First, compute the class descriptor for the equivalence class of the initial board configuration. Then perform a breadth- or depth-first search of the equivalence class graph, using the algorithm given in the proof of Lemma 4.3.17 to compute the neighboring class descriptors and the algorithm given in the proof of Lemma 4.3.18 to decide if the search has found a winning equivalence class. Each of these procedures takes polynomial time. By Lemma 4.3.10, there are only polynomially many equivalence classes, so the search terminates in polynomial time. By Lemma 4.3.15, there exists a winning move sequence if and only if this search finds a path to a winning equivalence class. □

The key idea of the above proof is that, if we do not care how many moves we make inside an equivalence class, then it is sufficient to search the graph of equivalence classes. Thus the above proof does not apply to k-Move Push Fight Mate-in-1, and in the next section, we prove k-Move Push Fight Mate-in-1 is NP-hard.

4.3.4 k-Move Mate-in-1 is NP-hard

To prove k-Move Push Fight Mate-in-1 hard, we reduce from the following problem, proved strongly NP-hard in [64]:

Problem 4.3.19. Integer Rectilinear Steiner Tree: *Given a set of points in $\mathbb{R}^2$ having integer coordinates and a length ℓ, is there a tree of horizontal and vertical line segments of total length at most ℓ containing all of the points?*

Theorem 4.3.20. *k-Move Push Fight Mate-in-1 is strongly NP-hard.*

Proof sketch. The basic idea of our reduction is to create a game state mostly full of the current player's pawns, but with a few empty squares (*holes*). The player must "move" the holes (by moving pawns into them, creating a new hole at the pawn's former square) to free a king that can push one of the other player's pieces off the board. Initially each pawn can only travel one square (into an adjacent hole) per move, but once two holes have been brought together, a pawn can travel two squares per move, and so on. Bringing the holes together optimally amounts to finding a Steiner tree covering the holes' initial positions.

Reduction: Suppose we are given an instance of Integer Rectilinear Steiner Tree consisting of points $p_i = (x_i, y_i)$ with $i = 1, \ldots, n$ and length ℓ. For convenience, and without affecting the answer, we first translate the points so that $\min x_i = 2$ and $\min y_i = 4$ and reorder the points such that $y_1 = 4$.

We then build a Push Fight game state with a rectangular board with a height of $\max y_i$ and a width of $n + \max x_i$, indexed using 1-based coordinates with the origin in the bottom-left square; refer to Figure 4-6. The entire boundary of the board has side rails except the edge adjacent to square $(x_1, 1)$. There is a white king in square $(x_1 + n, 2)$ and a black king with the anchor in square $(x_1 - 1, 2)$. There is a black pawn in square (x, y) if any of the following are true:

1. $y = 3$ and $x \neq x_1$,
2. $y = 2$ and either $x < x_1 - 1$ or $x > x_1 + n$, or
3. $y = 1$.

The squares (x_i, y_i) with $1 \leq i \leq n$ (corresponding to the points in the Integer Rectilinear Steiner Tree instance) are empty. All remaining squares are filled with white pawns. The output of the reduction is this Push Fight board together with $k = \ell + 3$. □

This reduction can clearly be computed in polynomial time. It remains to show that there exists a solution to the Integer Rectilinear Steiner Tree instance if and only if White can win in the corresponding k-Move Push Fight Mate-in-1 instance.

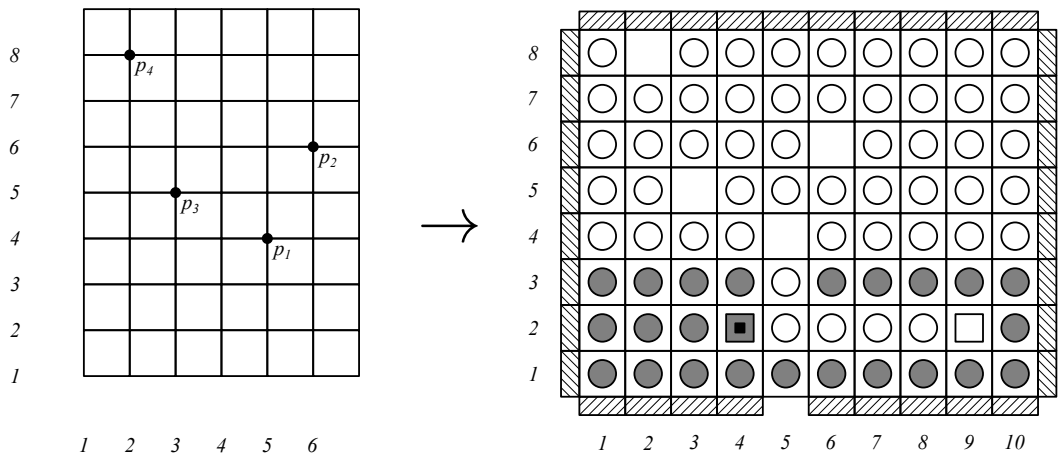


Figure 4-6: A Push Fight board (right) produced during the reduction from the points in an example rectilinear Steiner tree instance (left).

Move sequence $\Longrightarrow$ Steiner tree: To win, White must push a black piece off the board. The only boundary edge without a side rail in the game state produced by the reduction is the south edge of square $(x_1, 1)$, so White must move their king to $(x_1, 2)$ and push downward. Call the n squares directly to the left of the white king the *alley* and all squares with $y > 3$ the *plaza*. Before moving the white king into position to push, White must move the n white pawns in the alley into the plaza (which has exactly n empty squares). For the example reduction output in Figure 4-6, Figure 4-7 shows the state of the board after moving all pawns from the alley into the plaza, then moving the white king through the now-empty alley.

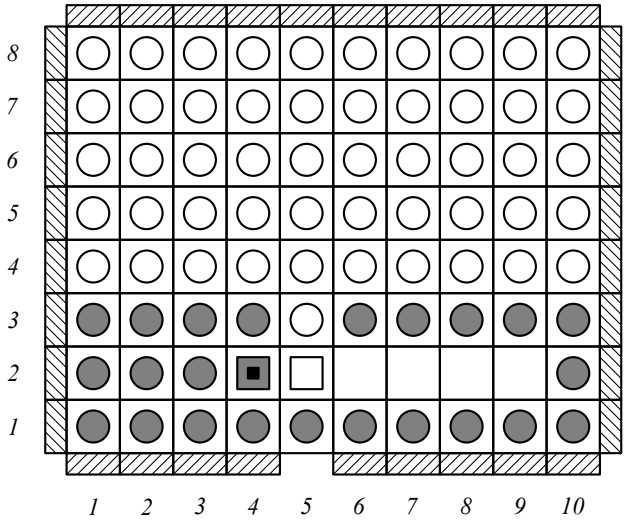


Figure 4-7: The result of moving all alley pawns in Figure 4-6 into the plaza, then moving the white king through the alley. White wins by pushing down.

Suppose White can win with at most k moves and a push. Let S be the set of squares that are ever empty during the winning move sequence and let S' be the

subset of the plaza induced by S. We prove two useful facts about S'.

Lemma 4.3.21. *S' is connected.*

Proof. We can use the move sequence (except for the king move(s)) to build a set of paths $P_1, P_2, \ldots, P_n$ from the empty squares in the plaza to the squares in the alley. Each path essentially traces the path of one hole over the course of the move sequence. As we go through the move sequence building the paths we maintain the invariant that the currently empty squares are the last elements in the paths. Initialize the n different paths to start in the n empty plaza squares. This satisfies the invariant at the start of the move sequence. During each move, a pawn moves from some square s, through some sequence of empty squares $s_1, s_2, \ldots, s_l$, and into a currently empty square t. By our invariant, there exists a path currently ending at t, so we extend that path with the sequence $s_l, \ldots, s_2, s_1, s$. The new endpoint of that path is the square that was just emptied, so the invariant is maintained.

All squares added to paths during this procedure are empty when they are added, so $\bigcup P_i \subseteq S$. In the other direction, all of the initially empty squares are endpoints of paths, the square emptied by each move is always appended to a path, and squares are never removed from paths, so $S \subseteq \bigcup P_i$. Thus $\bigcup P_i = S$.

We know that the alley must be empty for the king to move in position to push, so the final endpoints of the paths are exactly the alley squares. Consider any two squares $s, t \in S$. s occurs in at least one path P_i, and the section of that path starting at s (call it P_i^s) is a path in S from s to an alley square. Similarly, there is a section P_j^t starting at t of some path P_j that also ends in an alley square. Then P_i^s and the reverse of P_j^t, joined by a horizontal path entirely within the alley, is a path in S from s to t. Thus we can construct a path in S between any two elements of S, so S is connected.

Because the plaza is separated from the alley by black pawns except at $(x_1, 3)$, all paths in S containing squares both inside and outside the plaza must contain $(x_1, 3)$. Then we can convert any path in S starting and ending in S' to a path entirely in S' by deleting all squares between and including the first and last instances of $(x_1, 3)$ (if any), so S' is also connected. □

Lemma 4.3.22. $|S'| \leq \ell + 1$.

Proof. The reduction leaves n squares empty. Of the k moves in the move sequence, at least one moves the king; the remaining $k - 1$ moves empty at most one square each, so $|S| \leq n + k - 1$. There are $n + 1$ squares in S that are not in the plaza (the n alley squares and $(x_1, 3)$), so

$$\begin{aligned} |S'| &= |S| - (n+1) \\ &\leq n + k - 1 - n - 1 \\ &= k - 2 \\ &= \ell + 1 \end{aligned}$$

□

Consider the grid graph G induced by vertex set S'. S' is connected (Lemma 4.3.21), so G is also connected. Let T be any spanning tree of G. Each edge in G (and therefore in T) is a unit-length vertical or horizontal segment. S' contains every p_i, so T does also. By Lemma 4.3.22, G has $|S'| \leq \ell + 1$ vertices, and T has one fewer edge than it has vertices, so the total length of T is at most ℓ. Thus T is a solution to the Integer Rectilinear Steiner Tree instance, and such a solution exists if White can win in the k-Move Push Fight Mate-in-1 instance.

Steiner tree $\implies$ move sequence: We are given a Steiner tree with total length at most $\ell = k - 3$. Hanan's Lemma [71, Theorem 4] states that there exists an optimal tree whose edges are entirely contained on vertical and horizontal lines through each p_i, so we assume without loss of generality that the given tree has this form. Each p_i has integer coordinates, so we can subdivide the edges of the given tree into unit-length edges, resulting in a tree with at most $\ell + 1$ vertices, all having integer coordinates. By our assumption that the tree is optimal, each leaf of the tree is one of the p_i (though not all p_i are necessarily leaves).

Let T be the result of augmenting the subdivided given tree with a path through the vertices corresponding to square $(x_1, 3)$ and the squares in the alley, and consider T to be rooted at $(x_1 + n - 1, 2)$ (the alley square adjacent to the white king). We added $n + 1$ vertices, so T has at most $\ell + 1 + n + 1 = (k - 3) + 1 + n + 1 = k + n - 1$ vertices. Observe that each vertex in T corresponds to a square that is either occupied by a white pawn or empty.

We build the move sequence from T by repeatedly choosing a move as follows. Choose any leaf (x, y) of T and walk along the tree towards the root until a vertex (x', y') corresponding to an occupied square is reached. The reverse of that walk is a valid move of the white pawn at square (x', y') through some number of holes to the hole at (x, y). Append that move to the sequence (updating the board state accordingly), then remove the leaf (x, y) from T. Continue this loop until the root of T corresponds to an empty square (such that the preceding procedure would fail to find an occupied square when moving towards the root).

T initially contains vertices corresponding to n holes (the p_i). Each move creates a hole at (x', y'), but fills in a hole at (x, y), so the number of vertices corresponding to holes in T always remains at n. The loop ends when there is a path from a leaf to the root containing only vertices corresponding to holes. All paths from the root of T begin with a specific path of length $n + 1$ (the path we augmented the given tree with), so when the loop terminates, the first n squares corresponding to that path (the alley) must be holes. Each iteration of the loop removes a vertex from T and appends a move to the move sequence. T initially contains $k + n - 1$ vertices and ends containing n vertices, so the generated move sequence contains $k - 1$ moves, to which we append a move of the white king through the now-empty alley into position to win by pushing down. Thus the move sequence is a solution to the k-Move Push Fight Mate-in-1 instance, and such a solution exists if there is a solution to the Integer Rectilinear Steiner Tree instance.

Having proved both directions, we have proved Theorem 4.3.20.

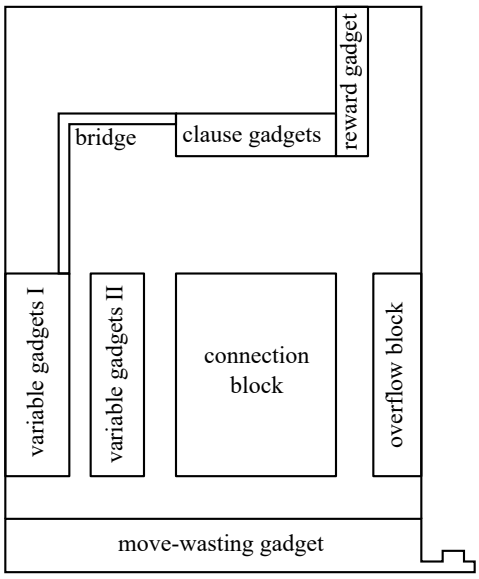


Figure 4-8: An overview of the Push Fight board produced by our reduction.

4.4 Push Fight is PSPACE-hard

In this section, we analyze the problem of deciding the winner of a Push Fight game in progress.

Problem 4.4.1. Push Fight: *Given a Push Fight game state, does the current player have a winning strategy (where players make up to two moves per turn)?*

Theorem 4.4.2. Push Fight *is* PSPACE*-hard.*

To prove PSPACE-hardness, we reduce from Q3SAT, proved PSPACE-complete in [130, 65]:

Problem 4.4.3. Q3SAT: *Given a fully quantified boolean formula in conjunctive normal form with at most three literals per clause, is the formula true?*

Our proof parallels the NP-hardness proof of Push-∗ in [76]. Push-∗ is a motion-planning problem in which a robot (agent) traverses a rectangular grid, some squares of which contain blocks. The robot can push any number of consecutive blocks when moving into a square containing a block, provided no blocks would be pushed over the boundary of the board. The Push-∗ decision problem asks, given a initial placement of blocks and a target location, can the robot reach the target location by some sequence of moves? In our proof, the white king takes the place of the Push-∗ robot[4] and white pawns function as blocks. Our proof has the additional complication that Black sets the universally quantified variables, and that White's moves and Black's push must be forced at all times to keep the other gadgets intact.

Figure 4-8 shows an overview of the reduction. The sole white king begins at the bottom-left of the *variable gadget I* block, setting existentially quantified variables as

[4]The Push-∗ robot can move without pushing blocks, so the correspondence is not exact.

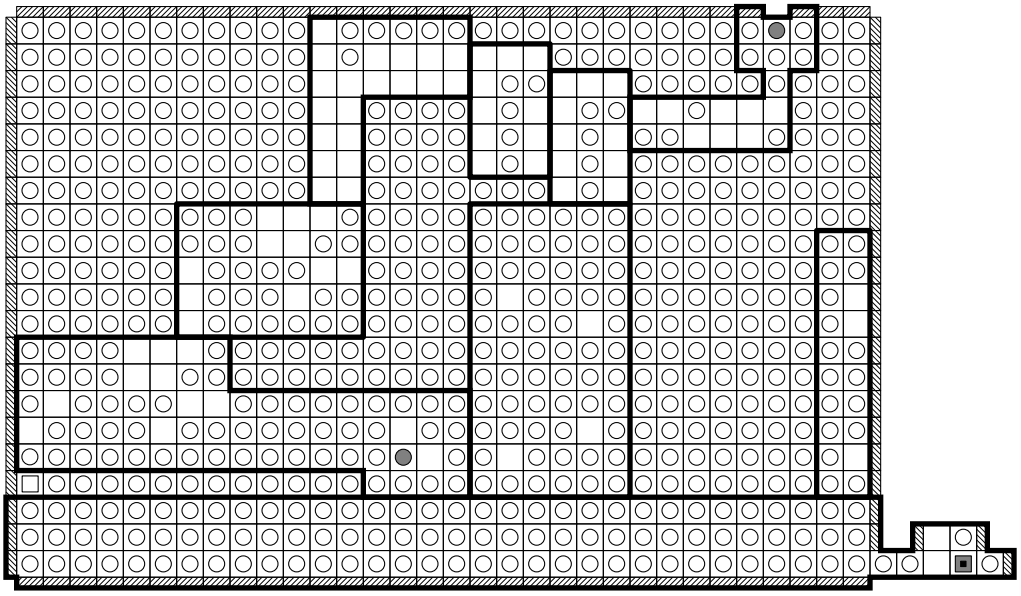


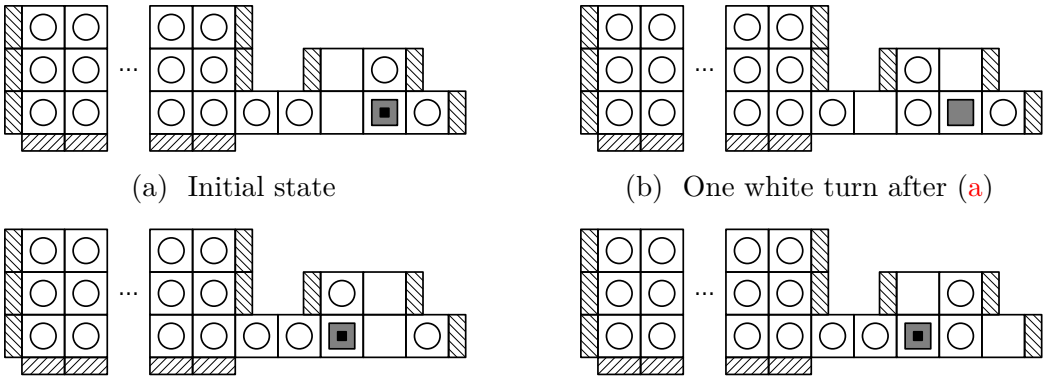
Figure 4-9: The result of performing the reduction on the formula $\forall x \exists y\ (x \vee \neg y) \wedge (\neg x \vee y)$. Gadgets and blocks are outlined.

it pushes up and right. The *variable gadget II* block contains black pawns and holes that allow Black to set the universally quantified variables. After all the variables have been set, the white king traverses the *bridge* to the *clause gadget* block. The variable and clause gadgets interact via a pattern of holes in the *connection block* encoding the literals in each clause. The white king can traverse the clause gadgets only if the variable gadgets were traversed in a way corresponding to a satisfying assignment of the variables. The *reward gadget* contains a boundary square without a side rail, such that the white king can push a black pawn off the board if the white king reaches the reward gadget. The *overflow block* contains empty squares needed by the variable gadgets that were not used in the connection block (for variables appearing in few clauses). The *move-wasting gadget* forces White's moves and Black's push, ensuring the integrity of the other gadgets. Finally, all other squares on the board are filled with white pawns, and the boundary has side rails except at specific locations in the reward and move-wasting gadgets. Figure 4-9 shows an example output of the reduction.

We first prove the behavior of each of the gadgets, then describe how the gadgets are assembled.

4.4.1 Move-Wasting Gadget

The move-wasting gadget requires White to use both moves to prevent Black from winning on the next turn (unless White can win in the current turn). The move-wasting gadget contains the only black king, thus consuming (and allowing) Black's push each turn. When analyzing the other gadgets, we can thus assume White can only push and Black can only move. The move-wasting gadget comprises the entire bottom three



(a) Initial state

(b) One white turn after (a)

(c) One black turn after (b)

(d) One white turn after (c)

Figure 4-10: The move-wasting gadget.

rows of the board, but pieces only move in the far-right portion. Figure 4-10a shows the initial state of the gadget. Throughout this analysis, we assume White cannot win in one turn; Section 4.4.5, which analyzes the reward gadget, describes the position in which White can immediately win in one turn, and can therefore disregard the threat from Black in the move-wasting gadget.

In the initial state, the anchor is on the black king, so it is White's turn. White must move the pawn above the black king to avoid losing next turn. There are only two reachable empty squares, both in the column left of the black king. If the other square in that column remains empty, Black can move the black king into it and push the white pawn in that column off the board. Thus White must fill the other square in that column, and the only way to do so is to move the pawn two columns left of the white king one square right. Figure 4-10b shows the resulting position (after White pushes elsewhere in the board).

Black's only legal push is to the left, resulting in the position shown in Figure 4-10c.

The rightmost four columns in Figure 4-10c are simply the reflection of those columns in Figure 4-10a, so by the same argument White must fill the column to the right of the black king, resulting in Figure 4-10d.

Again, the rightmost four columns of Figures 4-10b and 4-10d are reflections of each other. Black's only legal push is to the right, restoring the gadget to the initial state shown in Figure 4-10a. Thus until White can win in one turn, White must use both moves in the move-wasting gadget, and at all times Black must (and can) push in the move-wasting gadget. In the analysis of the remaining gadgets, if the white king reaches a position from which it cannot push, we conclude that White immediately loses, because if White moves a pawn or the king into position to push, Black can win on the next turn as explained above.

4.4.2 Variable Gadgets

The existential variable gadget forces White to fill all empty squares in one row of the connection block, corresponding to setting the value of that variable. The universal

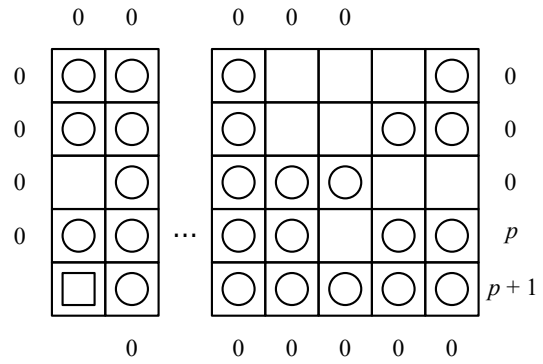


Figure 4-11: The initial configuration of the core gadget together with upper bounds on the number of pushes out of the gadget at each boundary edge. Omitted columns do not have a given upper bound.

variable gadget allows Black to choose the value of the corresponding variable, then forces White to similarly fill a row of empty squares. We first analyze a core gadget; the existential variable gadget is a minor variant of the core gadget and its correctness follows directly, while the universal variable gadget has an additional component to allow Black to choose the variable's value. Throughout our analysis, we take advantage of the board being filled with white pawns to limit the number of pieces that can leave the gadget.

The core gadget occupies a rectangle of width $p+5$ and height 5. When instantiated in the reduction, the gadget lies entirely within the *variable gadget I* block. Integer p is one more than the maximum number of occurrences of a literal in the input formula. The initial state of the core gadget is shown in Figure 4-11. Each number along the boundary of the figure gives the number of empty squares outside the gadget in that direction, and thus an upper bound on the number of pieces that can leave the gadget via that edge.

The following lemma summarizes the constraints we prove about the core gadget.

Lemma 4.4.4. *Starting from the position in Figure 4-11, and assuming the white king does not push down or left from this position,*

(i) the white king leaves in the second-rightmost column, and

(ii) when the white king leaves either

(a) the gadget is as shown in Figure 4-12 and $p+1$ white pawns have been pushed out along the bottom row of the gadget, or

(b) the gadget is as shown in Figure 4-13 and p white pawns have been pushed out along the second-to-bottom row of the gadget,

(iii) and no other pieces have left the gadget.

We will construct the existential and universal variable gadgets from the core gadget such that the assumption holds. Lemma 4.4.4(i) ensures we can chain variable

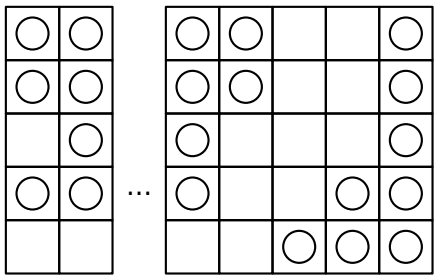


Figure 4-12: The final configuration of the core gadget after setting the variable to true.

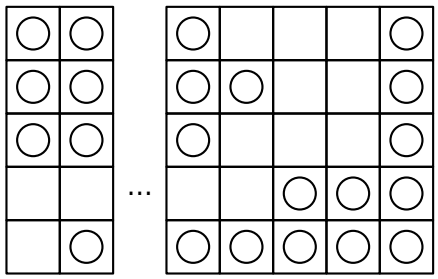


Figure 4-13: The final configuration of the core gadget after setting the variable to false.

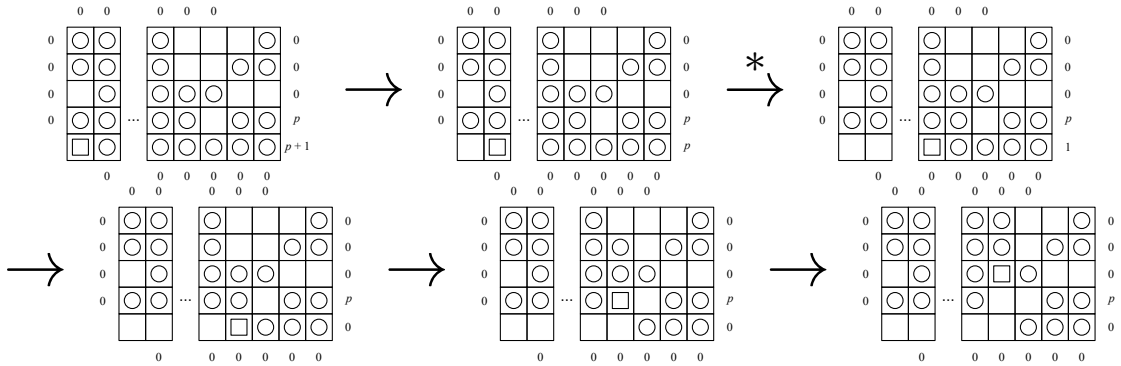


Figure 4-14: One possible push sequence starting from the initial state of the core gadget. The starred arrow elides a series of pushes to the right.

gadgets together in sequence without the white king escaping. The outcomes implied by Lemma 4.4.4(iia) and Lemma 4.4.4(iib) correspond to setting the variable to true or false (respectively) by filling in the empty squares in the connection block that could be used to satisfy a clause gadget for a clause containing the opposite literal; that is, pushing pawns out along the bottom row of a gadget prevents all negative literals from being used to satisfy a clause, and similarly for the second-to-bottom row and positive literals.

Proof. We proceed by case analysis starting from Figure 4-11. The move-wasting gadget consumes White's moves, and there are no black pieces in the core gadget, so we need only analyze the sequence of White's pushes.

Suppose the white king first pushes right. Because of the upper bounds along the top and bottom edges of the gadget, the only legal push in the resulting configuration is to the right, and this remains the case until the white king reaches the fourth column from the right of the gadget. At this point $p + 1$ pawns have been pushed off the right edge along the bottom row of the gadget, so there are no empty squares remaining in that row, so pushing right is no longer possible and the only legal push is up. Then the only legal push is again up because of the constraints on the left edge of the gadget. Figure 4-14 shows the result of this sequence of pushes.

If the white king pushes left from this position, the only possible next push is

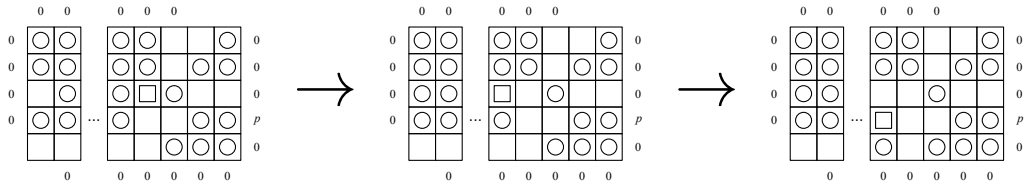


Figure 4-15: The result of pushing left and down from the last position in Figure 4-15. White has no legal pushes in the final position.

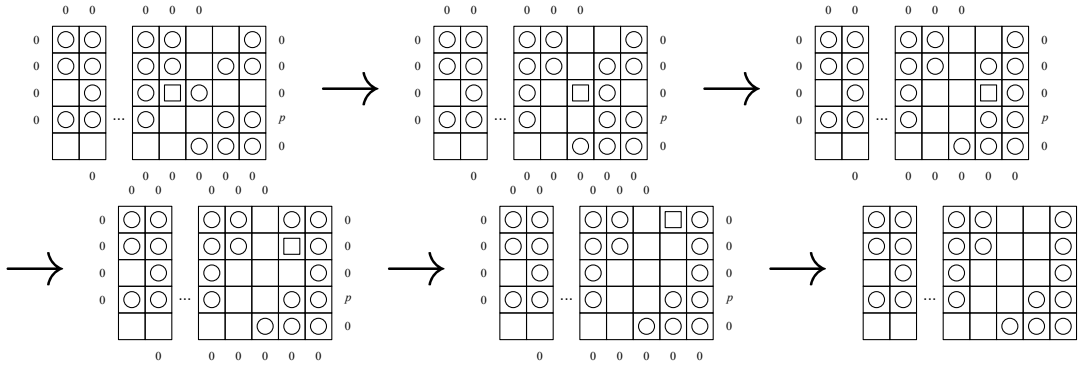


Figure 4-16: The result of pushing right from the last position in Figure 4-14, reaching the position in Figure 4-12.

down, after which there are no legal pushes, resulting in a loss for White. Figure 4-15 shows this sequence of pushes.

The only other legal push from the last position in Figure 4-14 is to the right, after which pushes right, up, up and up again are the only legal pushes. This sequence results in the white king, preceded by a white pawn, exiting the top of the gadget in the second-rightmost column, as desired by Lemma 4.4.4(i). Figure 4-16 shows the positions resulting from this sequence. The final position reached is the position in Figure 4-12, $p+1$ pawns were pushed out of the gadget to the right along the bottom row, as desired by Lemma 4.4.4(iia), and and no other pieces were pushed out of the gadget, as desired by Lemma 4.4.4(iii).

Now suppose that the white king pushes up from the initial configuration. Because of the constraints on the gadget boundary, the only legal push is to the right until the white king reaches the fourth column from the right of the gadget. At this point p pawns have been pushed off the right edge along the second-to-bottom row of the gadget, so there are no empty squares remaining in that row, so pushing right is no longer possible and the only legal push is up. Then the only legal push is again up because of the constraints on the left edge of the gadget. Figure 4-17 shows the result of this sequence of pushes.

If the white king pushes up from this position, there are no legal pushes in the resulting position, resulting in a loss for White. Figure 4-18 shows this push and the resulting losing position.

The only other legal push from the last position in Figure 4-17 is to the right, after which pushes right, up, up and up again are the only legal pushes. This sequence

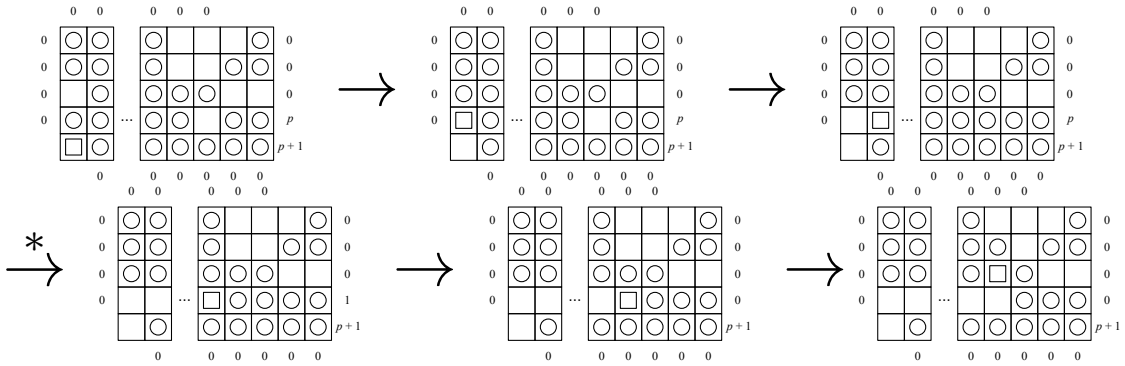


Figure 4-17: The other possible push sequence starting from the initial state of the core gadget. The starred arrow elides a series of pushes to the right.

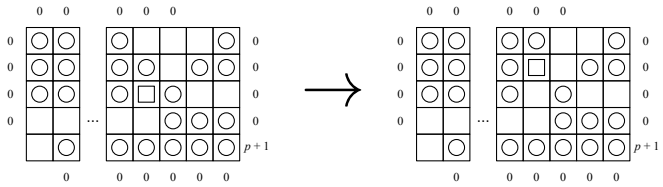


Figure 4-18: The result of pushing up from the last position in Figure 4-17. White has no legal pushes in the final position.

results in the white king, preceded by a white pawn, exiting the top of the gadget in the second-rightmost column, as desired by Lemma 4.4.4(i). Figure 4-19 shows the positions resulting from this sequence. The final position reached is the position in Figure 4-13, and p pawns were pushed out of the gadget to the right along the second-to-bottom row, as desired by Lemma 4.4.4(iib). No other pieces were pushed out of the gadget, as desired by Lemma 4.4.4(iii).

This completes the case analysis. □

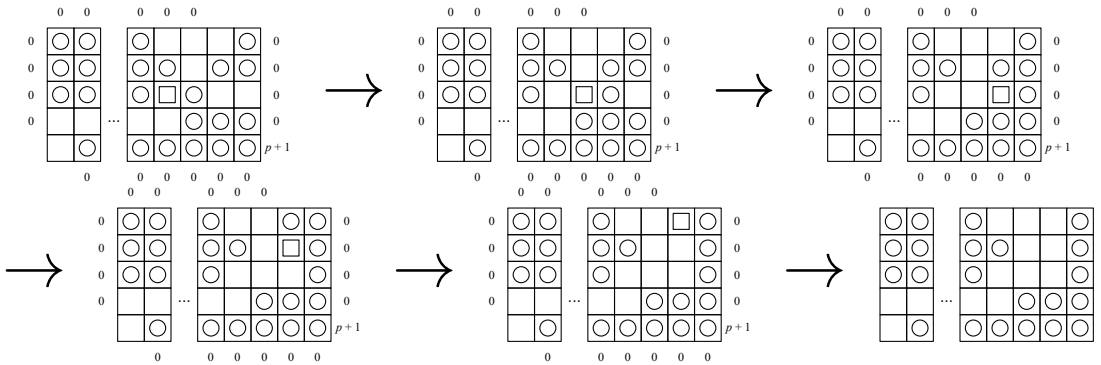


Figure 4-19: The result of pushing right from the last position in Figure 4-17, reaching the position in Figure 4-13.

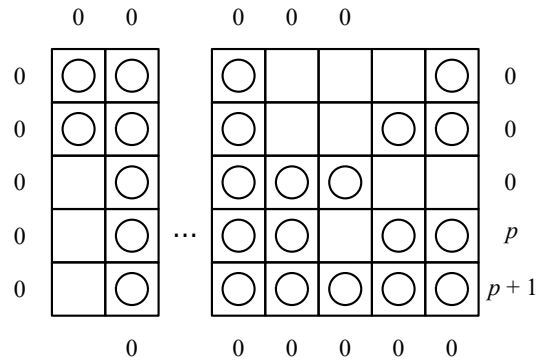


Figure 4-20: The existential variable gadget.

Existential variable gadget: The existential variable gadget, shown in Figure 4-20, is nearly the same as the core gadget, differing only in the bottom of the leftmost column. When instantiated in the reduction, the white king enters the gadget by pushing a white pawn up into the leftmost column, becoming exactly the core gadget. From the position immediately after the white king enters the gadget, the white king cannot push left (because there are no empty spaces in the row to the left) nor down (because it just pushed up, leaving an empty space in its former position), satisfying the assumption in Lemma 4.4.4. Thus by Lemma 4.4.4(i), the white king leaves the existential variable gadget in the second-rightmost column with a white pawn above it, and by either Lemma 4.4.4(iia) or 4.4.4(iib), all empty squares in one of two rows of the connection block are now filled by pawns pushed out of the existential variable gadget.

Universal variable gadget: The universal variable gadget consists of two disconnected regions. The left subregion of the gadget occupies a $(p+6) \times 5$ rectangle in the *variable gadget I* block. As the white king proceeds through the left region of the gadget, a subregion of the gadget reaches the initial state of the core gadget. The right region of the gadget occupies a 4×4 rectangle in the *variable gadget II* block and contains a black pawn to allow Black to control the value of the variable. The bottom of the right region is one row lower than the bottom of the left region. The area between the two regions of the gadget (in the three rows shared by both) is entirely filled by white pawns. Figure 4-21 shows the universal variable gadget, including the pawn-filled area between the regions.

As with the existential variable gadget, when instantiated in the reduction, the white king enters the universal variable gadget by pushing a white pawn up into the leftmost column. Figure 4-22 shows the resulting position. Regardless of Black's move, White's only legal push is to the right. By moving the black pawn, Black can choose between the two positions in Figure 4-23, depending on which of the two rows the black pawn is in when White pushes.

In both of the resulting positions, the black pawn is surrounded, so Black can no longer influence events in this gadget. The left region of the gadget, without the leftmost column, is identical to the initial position of the core gadget. In both positions,

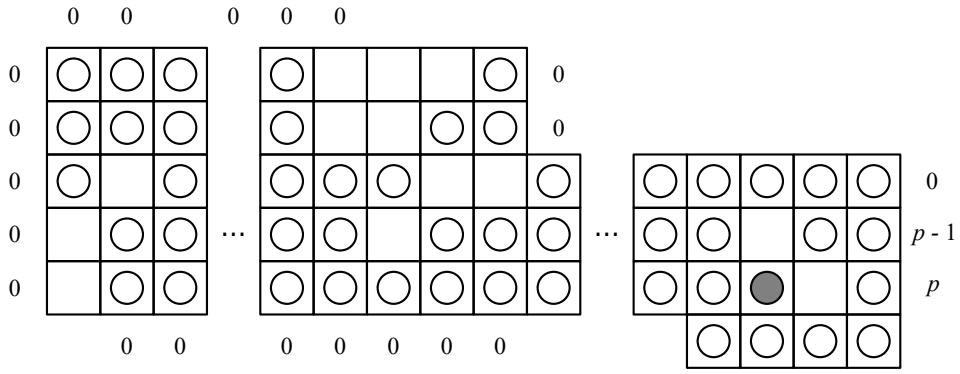


Figure 4-21: The universal variable gadget.

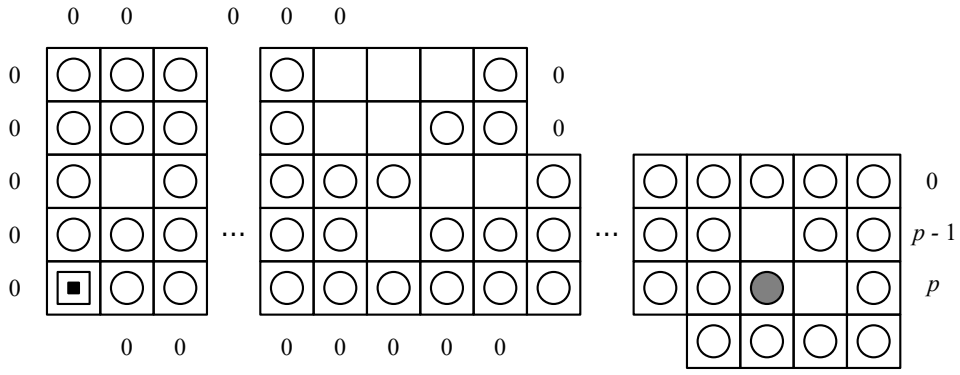
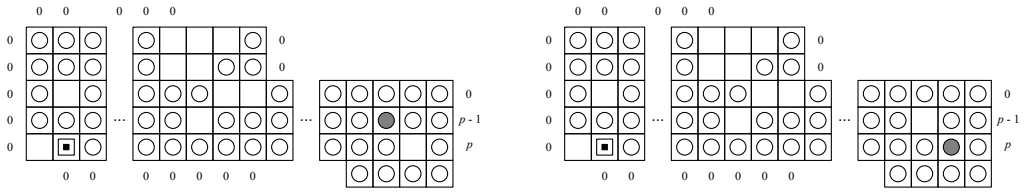
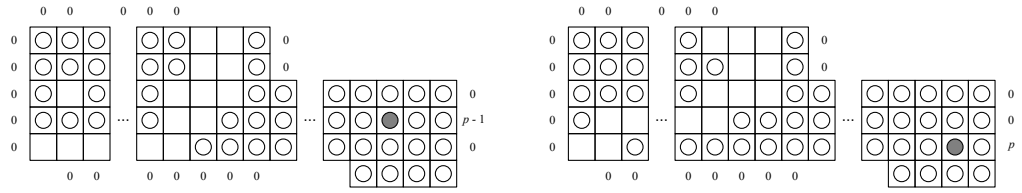


Figure 4-22: The universal variable gadget after the white king enters.



(a) (b)

Figure 4-23: The two possible configurations of the universal variable gadget one white turn after the configuration from Figure 4-22.



(a) (b)

Figure 4-24: The two possible final positions of the universal variable gadget after the white king exits.

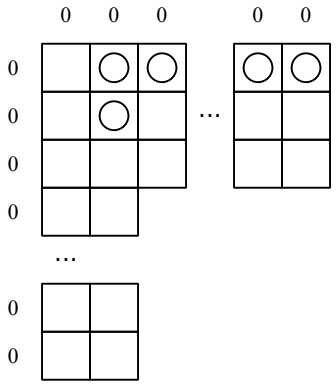


Figure 4-25: The bridge gadget.

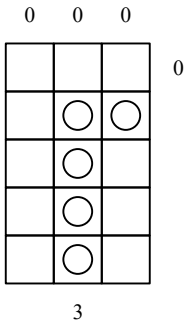


Figure 4-26: The clause gadget.

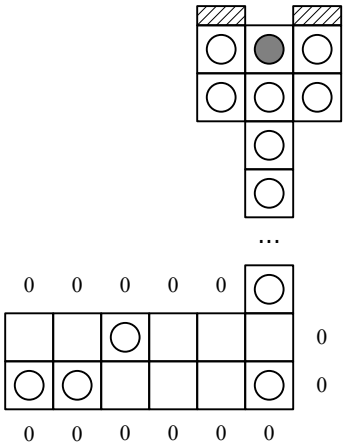


Figure 4-27: The reward gadget.

the white king cannot push left (empty space) or down (no empty spaces down in the column), satisfying the assumption in Lemma 4.4.4. Thus either Lemma 4.4.4(iia) or Lemma 4.4.4(iib) holds. Because of the edge constraints, in Figure 4-23a, only Lemma 4.4.4(iia) is possible, resulting in Figure 4-24a. Similarly, in Figure 4-23b, only Lemma 4.4.4(iib) is possible, resulting in Figure 4-24b. By moving the black pawn to select one of these two cases, Black sets the value of the corresponding variable. Then by Lemma 4.4.4(i), the white king leaves in the second-rightmost column of the left region (in the *variable gadget I* block) of the gadget. In both cases, the black pawn remains surrounded by white pawns in the right region of the gadget.

4.4.3 Bridge Gadget

The bridge gadget, shown in Figure 4-25, brings the white king from the exit of the last variable gadget to the entrance of the first clause gadget. When instantiated in the reduction, the white king enters the bridge gadget from the bottom of the leftmost column, preceded by a white pawn. The white king's traversal of the bridge gadget is entirely forced. The white king leaves the gadget by pushing a white pawn out to the right in the second-to-top row.

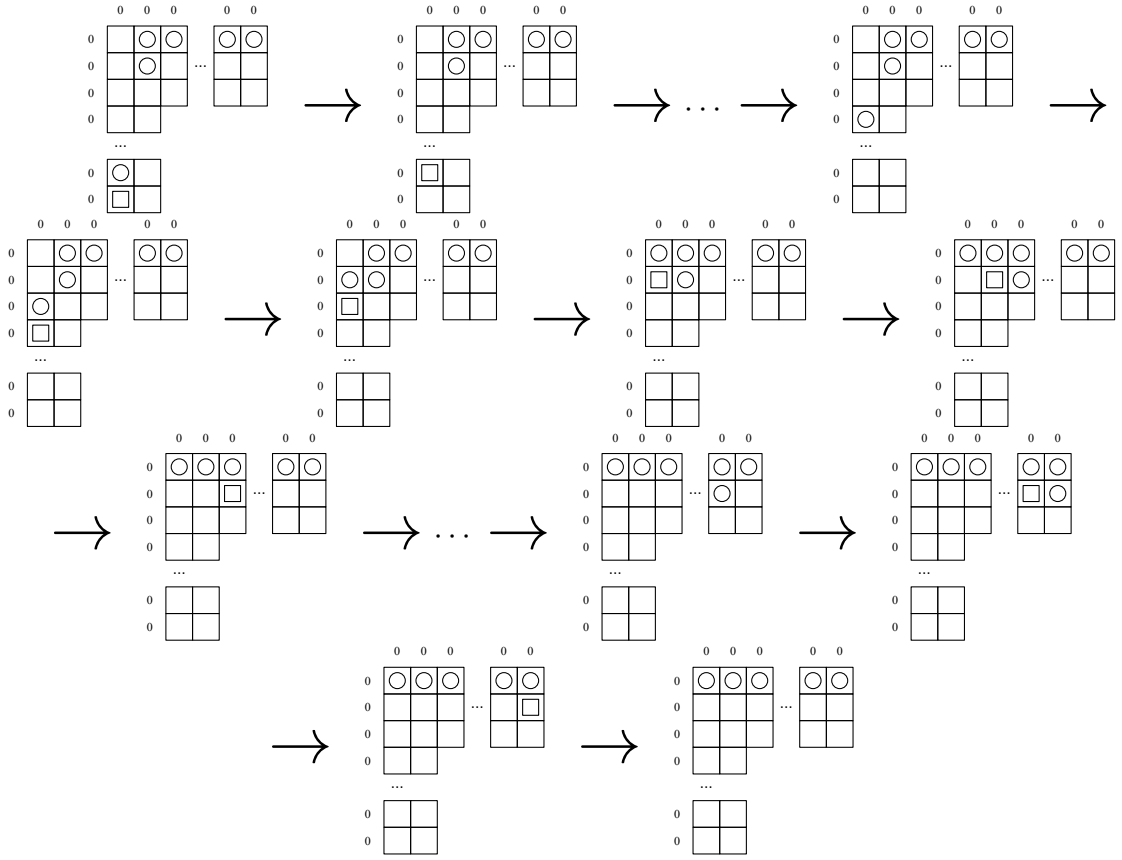


Figure 4-28: The sequence of configurations of the bridge gadget as it is traversed by the white king.

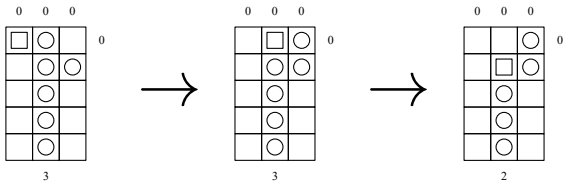


Figure 4-29: The sequence of forced configurations of the clause gadget as it is traversed by the white king.

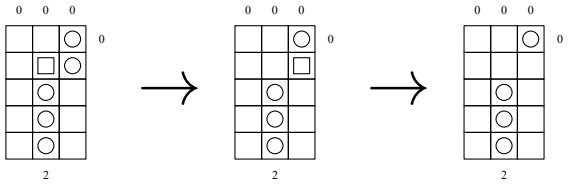


Figure 4-30: Starting from the last position in Figure 4-29, the push sequence by which the white king exits the clause gadget.

4.4.4 Clause Gadget

The clause gadget, shown in Figure 4-26, verifies that a column below the gadget contains at least one empty square. When instantiated in the reduction, the white king enters the gadget from the left in the top row, preceded by a white pawn. The resulting sequence of forced pushes includes a push down in the central column of the gadget; if there are no empty squares below the gadget in that column, the white king has no legal pushes and White loses. If there are more empty squares, White can continue to push down, but (when instantiated in the reduction) there are at most three total empty squares in that column, and once those squares are filled, White cannot push. Thus the white king must push right instead and leave the gadget by pushing a white pawn out to the right in the second-to-top row.

4.4.5 Reward Gadget

The reward gadget, shown in Figure 4-27, allows White to win if the white king reaches the gadget. The black pawn in this gadget cannot move because it is surrounded. When instantiated in the reduction, the white king enters the gadget from the left in the top row, preceded by a white pawn. After pushing right until the white king is in the third column of Figure 4-27, White can win by moving a white pawn and the white king, then pushing upwards to push the black pawn off the board, as shown in Figure 4-32. (Recall that the move-wasting gadget no longer binds White once White can win in one turn; Black loses before Black can win using the move-wasting gadget.)

4.4.6 Layout

Having described the gadgets, it remains to show how to instantiate them in a Push Fight game state for a given quantified 3-CNF formula. We first place gadgets

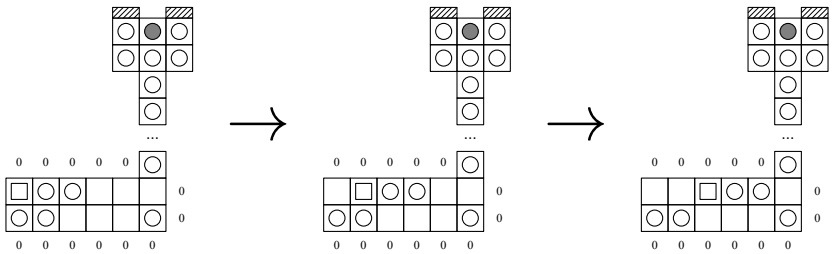


Figure 4-31: The sequence of forced configurations of the reward gadget as it is traversed by the white king.

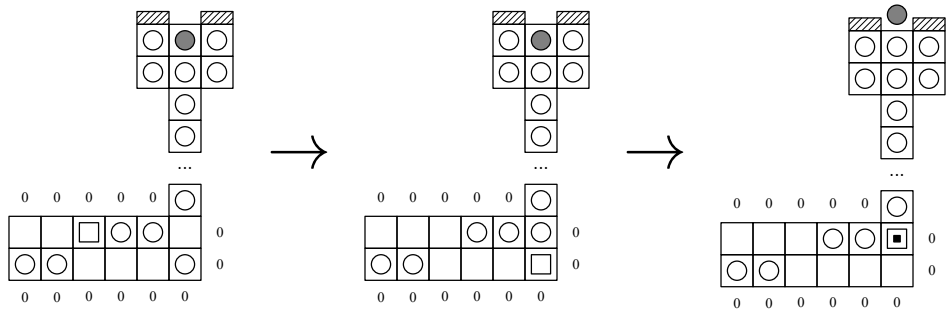


Figure 4-32: Once the White king reaches the third column of the reward gadget, White can win in a single turn.

with respect to each other, remembering which squares should be left empty, then define the board as the bounding box of the gadgets and fill any squares not recorded as empty with white pawns. The resulting board is mostly rectangular with side rails on all boundary edges, with two exceptions: one edge along the top of the rectangle lacks a side rail as part of the reward gadget, and the board is extended in the bottom-right to accomodate the move-wasting gadget along the bottom of the board.

We begin by building the *variable gadget I* block containing the existential variable gadgets and the left portion of the universal variable gadgets. Gadgets are stacked

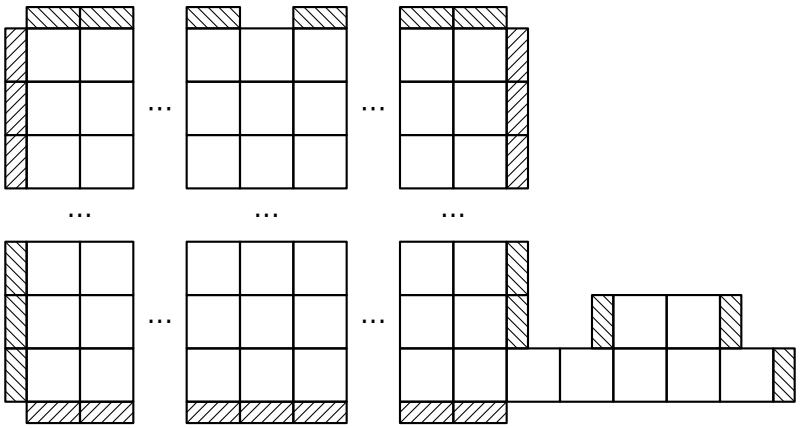


Figure 4-33: The shape of the Push Fight board produced by the reduction.

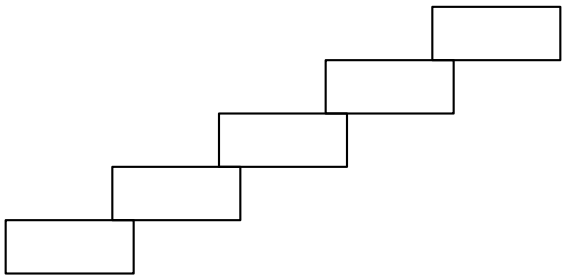


Figure 4-34: The layout of variable gadgets in the *variable gadget I* block.

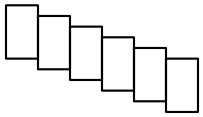


Figure 4-35: The layout of clause gadgets in the clause gadget block.

from bottom to top in the order of the quantifiers in the input formula (using the gadget corresponding to the quantifier), with the leftmost column of each gadget aligned with the second-to-right column of the previous gadget. (Recall that the width of the variable gadgets is defined based on p, one more than the maximum number of occurrences of a literal in the input formula.) This alignment allows (and requires) the white king to traverse the gadgets in sequence as specified by Lemma 4.4.4. Figure 4-34 shows the relative layout of these variable gadgets.

We place the white king one square below the first variable gadget aligned with its leftmost column, and place a white pawn one square above the white king. The white king will push upwards into the first gadget on White's first turn. (If the king was instead placed directly in the variable gadget, if the first variable is universally quantified, Black would not have a move with which to choose the value of the variable before White commits it.)

We then build the *variable gadget II* block by placing the right regions of the universal variable gadgets to the right of the corresponding left regions in a single column (further right than any part of the variable gadget I section).

Next we place one clause gadget for each clause in the input formula. Each clause gadget is directly to the right of and one square lower than the previous clause gadget. The entire clause gadget block is further right of and above the *variable gadget II* block. Figure 4-35 shows the relative layout of the clause gadgets. Then we place a bridge gadget such that the entrance of the bridge gadget aligns with the exit of the last variable gadget and the exit of the bridge gadget aligns with the entrance of the first clause.

We place the reward gadget so that its entrance aligns with the exit of the last clause gadget.

We leave empty squares in the connection block to encode the literals in each clause in the input formula. When traversing each variable gadget, the white king pushes pawns to the right in one of two rows. The lower (upper) row corresponds to setting the variable to true (false), or equivalently, preventing negative (positive)

literals from satisfying clauses. Associate each row with the literal it prevents from satisfying clauses. Each clause gadget enforces that at least one empty square remains below its middle column, corresponding to at least one of its literals not having been ruled out by the truth assignment. To realize this relation, for each literal in a clause, we leave an empty square at the intersection of the column checked by the clause gadget and the row associated with that literal. All other squares in the connection block are filled with white pawns (as are all squares in the board whose contents are not otherwise specified).

The variable gadgets require each row associated with a literal to contain exactly $p-1$, p or $p+1$ empty squares (depending on the type of gadget and whether the row is the upper or lower row). This is at least the number of occurrences of that literal (by the definition of p), but it may be greater. We place any remaining empty squares in each row in columns further right than the reward gadget, forming the overflow block.

The boundary of the board is the bounding box of all the gadgets placed thus far with a move-wasting gadget appended to the bottom of the board. The left column of the move-wasting gadget is aligned with the leftmost column of the first (leftmost) variable gadget and the sixth-from-right column (the rightmost column having height 3) is aligned with the rightmost column of the overflow block. We then fill all squares not part of a gadget nor recorded as empty with white pawns and place side rails on all boundary edges except as described in the move-wasting and reward gadgets. The anchor is on the black king as part of the initial state of the move-wasting gadget.

4.4.7 Analysis

Our analysis of gadget behavior in the preceding sections constrains the white king's pushes under the assumption that there are a specific number of empty spaces (often 0) in a particular row or column on a side of the gadget. We have already discharged the assumptions regarding the rows associated with literals by our layout of the connection and overflow blocks. For every other gadget except the variable gadgets, none of the constrained rows or columns intersects with another gadget, so the constraints on the edges are implied by the dense sea of white pawns outside the gadgets. For the variable gadgets, we assumed that pushing down in the second-to-left column of a variable gadget is not possible, but that column contains the previous variable gadget's rightmost column. We discharge this assumption by noting that in the final state of each variable gadget (after the white king has left the gadget), the rightmost column of that gadget is filled with white pawns, so pushing down in that column is indeed not possible.

Thus the white king must traverse the variable gadgets, setting the value of each variable, then traverse through the bridge gadget to the clause gadgets, where at least one empty space must remain in each checked column for the king to reach the reward gadget. If the choices made while traversing the variable gadgets results in filling all of the empty spaces in a checked column (i.e., the clause is false under the corresponding truth assignment), then White can only push by using a move outside the move-wasting gadget and Black wins on the next turn. If the white king

successfully traverses every clause gadget (i.e., every clause is true under the truth assignment), then White wins when the white king pushes the black pawn off the board in the reward gadget. Thus White has a winning strategy for this Push Fight game state if and only if the input quantified 3-CNF formula is true.

4.5 Solving Push Fight

In this section, we describe our efforts to solve standard, non-generalized Push Fight. We aim to strongly solve[5] the game, that is, to give a strategy for perfect play from any position, not just the initial positions. We find that Push Fight played on a board with one column (four squares) removed is a draw. We believe the standard version of Push Fight is within reach of our current system given sufficient computation time. Then we propose a solver-driven exploration of the design space of Push Fight variants that exploits our solver's ability to generalize to different boards and rule sets, and as a first step in this direction, prove that Push Fight on the same reduced board remains a draw when each player must make exactly one move on their turn.

4.5.1 Overview

Our Push Fight solver is based on repeating a forward search until it reaches a fixed point (unlike most efforts to solve games or create endgame databases, which are based on retrograde analysis). In the first generation, the solver enumerates all positions with an anchored king and takes all possible turns (moves and push) from each position. The position is a win if any turn pushes an opposing piece off the board, and a loss if every turn pushes an allied piece off the board (including when no pushes are possible). These won and lost positions are encoded as integers and stored as level 0 of the win-loss database.

In subsequent generations, the solver enumerates all positions with an anchored king not already identified as a win or a loss and takes all possible turns from each position. The successors are looked up in all previous levels of the win-loss database. The position is a win if at least one successor is a loss, and a loss if all successors are wins. The newly-identified won and lost positions are stored as level i of the win-loss database. The search continues iterating until in some generation no new won or lost positions are identified. Any positions not identified as won or lost at the end of the search are drawn. Drawn positions are stored in the database only implicitly (by not being stored as won or lost).

After the search completes, the win-loss database contains complete information about all positions containing an anchored king. The database can be directly queried for the win-loss-draw value of a position by encoding it as an integer and searching all levels of the database, but the turn to take to realize that value is not stored explicitly in the database. To find the turn, that position's successors must be generated and looked up in the database. For a won position, turns leading to lost positions in the earliest level of the database provide the quickest win; for a lost position, turns leading

[5]This terminology is from Allis [9].

to won positions in the latest level of the database provide the longest loss; for a drawn position, any turn leading to another drawn position preserves the draw.

A Push Fight game begins with the first player placing their pieces on their half of the board, then the second player doing the same. The resulting initial position does not have an anchored piece, so determining the winner requires taking all turns from that initial position and, if they are not immediately winning or losing, looking them up in the win-loss database and inverting the result (as in the looked-up position it is the second player's turn). After computing which initial positions win and lose for the first player, the overall game is a win for the first player if, for some placement of the first player's pieces, all placements of the second player's pieces result in a winning opening; the game is a loss for the first player if each placement of the first player's pieces has a placement of the second player's pieces resulting in a losing opening; and the game is a draw otherwise.

4.5.2 Related Work

Retrograde analysis. Retrograde analysis [131, 142] is a method for determining whether a class of game positions is a win, loss or draw (or unknown) under perfect play by working backwards. Starting from lost positions (e.g., checkmated positions in Chess), we generate all predecessor positions and record that they are known to be wins for the current player. We then generate all predecessors of those positions, but we do not immediately know they are losses. Instead, we must either do a 1-ply forward search to see whether all successors are known to be wins, or store a count of "outs" (possibly losing successors) and detect when it reaches zero. (See outcounting, below.) We then continue until no new positions are generated (within the class we are solving; e.g., 8-piece positions in Chess), at which point all positions not yet recorded as wins or losses are draws.

Retrograde analysis has been used to strongly solve Awari [120] and Pentago [81] and to weakly solve Nine Men's Morris in combination with a forward search covering the placement phase of the game [66]. Retrograde analysis has also been used to create endgame databases for Chess [134, 106], Chinese Chess [58, 151], Checkers [122] (contributing to the weak solution of Checkers by proof-number search [123]), Kriegspiel [39], and the backgammon variant Plakoto [113].

Outcounting. There is a time-memory tradeoff revolving around how lost positions are detected in retrograde analysis [24]. Each predecessor of a lost position is a win, but predecessors of wins are not necessarily lost. The obvious strategy is to perform a 1-ply forward search to check whether all the successors of the position in question are known to be lost. This forward search must be repeated each time the position is generated as the predecessor of a win until it is known to be lost (or won), introducing some repeated computation, and in distributed implementations, repeated communication to check the status of positions owned by other machines. The other strategy is to store the count of unknown successors of each state (the *out count*) and decrement this count each time the position is generated as a predecessor of a win. When the count reaches zero, the position is known to be lost. If the count is nonzero

at the end of the analysis (and the position isn't known to be a win), then it is part of a drawing line of play for that many positions.

Our Push Fight solver does not use retrograde analysis, but it uses a variant of the outcounting idea to exploit locality in win-loss database queries.

Design space exploration. Sturtevant [132] uses a combination of retrograde analysis and forward search to explore how modifying puzzles in the single-player puzzle game Fling! affects their solvability and difficulty. Work in procedural level generation and evolutionary game design (see Section 8.3) often uses search agents of varying sophistication as part of fitness functions, approximating difficulty or human interest.

4.5.3 Implementation

This section describes implementation details of our solver, including referencing function and class names, so it may be useful to read this section along with the code. The code is available on GitHub.[6]

Board representation. Because the Push Fight board is irregularly shaped, we identify squares by index rather than by their coordinates. We assign indices to squares arbitrarily without reference to their coordinates, except that we assign lower indices to anchorable squares. Because every push must push at least one other piece, and pushing a piece off the board ends the game, there are only 18 anchorable squares on the standard Push Fight board. For a square to be anchorable, it must have squares either above and below it or to its left and right. Assigning lower indices to anchorable squares makes our encoding of positions as integers more compact.

The topology of the board is represented as an array of integers, four per square, indicating the left, above, right, and below neighbors of that square. Each integer is either the index of the neighboring square in that direction or the special values `RAIL` (indicating there is a side rail in that direction) or `VOID` (indicating that direction is off the edge of the board). This topology specification is general enough to support holes in the interior of the board, rails between squares in the middle of the board, including rails that can be moved through in one direction but not the reverse, and cylindrical boards.

While not, strictly speaking, part of the board, the board object also specifies the number of kings and pawns, the number of allowed moves per turn (which need not be a range, e.g., exactly 1 or 3 moves), and the areas for placement of pieces during the opening procedure (as lists of square indices).

Board compiler. To facilitate experimentation with different boards and rule sets, boards are specified using a small language defining areas and rails, along with the number of kings and pawns and the allowed numbers of moves per turn. Board definitions are compiled into C++ code by the board compiler, a Python

[6] https://github.com/jbosboom/pushfight-solver

```
- name: traditional
  topology:
    - add rectangle (0, 0) 4 8
    - remove square (0, 0)
    - remove square (0, 1)
    - remove square (0, 7)
    - remove square (3, 0)
    - remove square (3, 6)
    - remove square (3, 7)
    - rail up line (0, 2) (0, 6)
    - rail down line (3, 1) (3, 5)
  pawns: 2
  pushers: 3
  placement:
    -
      - add rectangle (0, 0) 4 4
      - remove square (0, 0)
      - remove square (0, 1)
      - remove square (3, 0)
    -
      - add rectangle (0, 4) 4 4
      - remove square (0, 7)
      - remove square (3, 6)
      - remove square (3, 7)
  moves: [0, 1, 2]
```

Listing 4.1: Definition of the standard Push Fight board and rule set.

script. Listing 4.1 shows the definition for the standard Push Fight board (here, and throughout the solver, kings are called pushers).

Position representation. Positions are represented as instances of `struct State`, a group of five bitsets: four bitsets storing the indices of the allied kings, allied pawns, opposing kings, and opposing pawns, and one additional bitset storing the index of the anchored king, if any. The allied pieces belong to the player whose turn it is and the opposing pieces belong to the opponent.

Encoding positions as integers. The `rank` function, shown in Listing 4.2, encodes a position as an integer for storage in the win-loss database. The basic idea of the encoding is to convert a position to a tuple containing the square indices of (in order) the anchored piece (assumed to be an opposing king), the remaining opposing king, the opposing pawns, the allied kings and the allied pawns, then convert this tuple to an integer using a variable base. Each tuple item is multiplied by the number of squares that have not been used by an encoded piece. That is, for the standard Push Fight board with 26 squares, the first tuple item is multiplied by 26, then the next by 25, and so on.

The square indices are recovered from the position representation using the trailing zero count instruction (exposed by the C++ standard library as `std::countr_zero`).

```
unsigned long rank(State state, const Board& board) {
  unsigned long result = 0;
  unsigned int pext_mask = (1 << board.squares()) - 1;
  unsigned int squares = board.squares();

  int anchored_pusher_idx = std::countr_zero(state.anchored_pieces);
  result += anchored_pusher_idx;
  pext_mask &= ~state.anchored_pieces;
  --squares;

  auto enemy_pushers = state.enemy_pushers & ~state.anchored_pieces;
  enemy_pushers = pext(enemy_pushers, pext_mask);
  pext_mask &= ~state.enemy_pushers;
  while (enemy_pushers) {
    int low_bit = std::countr_zero(enemy_pushers);
    result *= squares;
    result += low_bit;
    --squares;
    enemy_pushers >>= low_bit + 1;
  }

  auto enemy_pawns = pext(state.enemy_pawns, pext_mask);
  pext_mask &= ~state.enemy_pawns;
  while (enemy_pawns) {
    int low_bit = std::countr_zero(enemy_pawns);
    result *= squares;
    result += low_bit;
    --squares;
    enemy_pawns >>= low_bit + 1;
  }

  auto allied_pushers = pext(state.allied_pushers, pext_mask);
  pext_mask &= ~state.allied_pushers;
  while (allied_pushers) {
    int low_bit = std::countr_zero(allied_pushers);
    result *= squares;
    result += low_bit;
    --squares;
    allied_pushers >>= low_bit + 1;
  }

  auto allied_pawns = pext(state.allied_pawns, pext_mask);
  pext_mask &= ~state.allied_pawns;
  while (allied_pawns) {
    int low_bit = std::countr_zero(allied_pawns);
    result *= squares;
    result += low_bit;
    --squares;
    allied_pawns >>= low_bit + 1;
  }
  return result;
}
```

Listing 4.2: Definition of the `rank` function for encoding a position as an integer.

Before executing the trailing zero count, however, we use the parallel bit extract instruction (the calls to `pext` in Listing 4.2) to compress the bitsets from the position representation by skipping over any bits from bitsets already processed. This makes the encoding more compact by making the tuple items smaller.

This encoding is compact: it needs 44 bits to encode all 278,008,038,000 anchored positions on the standard Push Fight board, which require a minimum of just over 38 bits. On the other hand, there are gaps in the encoding (some integers do not correspond to positions) and the encoding is not easily reversible, so when the solver enumerates all anchored positions not in the win-loss database, it must actually enumerate all positions and look up in the database rather than scan the database and convert the missing integers back into positions.

Win-loss database. Each level of the win-loss database consists of four files, a pair each for wins and losses. Contiguous runs of encoded positions are collected into intervals; one file stores the interval start positions in sorted order, while the other stores interval lengths. The win-loss database is queried by mapping all levels into memory, binary searching in each one for the largest interval start less than the query point, getting the corresponding length and checking whether the query point is within the interval. (We tried using interpolation search, but it turned out to be slower.) The `WinLossUnknownDatabase` class implements database queries and the `write_intervals` function implements writing new levels of the database.

Position enumerator. The position enumerator (the `enumerate_anchored_states` function) enumerates all anchored positions. First the anchored opposing king is placed on one of the anchorable squares. Then the remaining opposing kings, opposing pawns, allied kings and allied pawns are placed on unoccupied squares. To be generic over variable numbers of kings and pawns, instead of using one nested loop per piece, all pieces of the same type are placed simultaneously based on precalculated bitsets of the possible placements of $1 <= i <= 3$ pieces on the board. The position enumerator calls the turn generator on each position.

Turn generator. The turn generator has two parts, the move generator (the `next_states` function) and the push generator (`do_all_pushes`). The push generator is invoked before each move if the number of moves made this turn is in the board's allowable number of moves (for the standard game, this is 0, 1 or 2 moves). Then if the number of moves made this turn is less than the maximum number of allowed moves, the move generator is invoked to make a move and calls the turn generator recursively.

The move generator iterates over each allied piece and calls a bitset-based flood fill algorithm to find the connected component of empty space (if any) adjacent to that piece. The move generator then moves the piece by moving the bit in the corresponding position bitset, then calls the turn generator recursively.

The push generator iterates over each allied king and each direction, building a sequence of pieces next to the king in that direction using the board's topology array.

If the piece sequence ends in `VOID`, the last piece in the chain is removed from the position; if the sequence ends in `RAIL`, the push is invalid and the push generator continues with the next direction. If a valid push is made, each piece in the sequence is moved to the square of the next piece in the sequence, the anchored piece bitset is updated to anchor the king that just pushed on its new square, then the allied and opposing piece bitsets in the position are swapped because it is now the other player's turn.

Position evaluator. The turn generator passes each resulting position and the piece pushed off (if any) to a position evaluator's `accept` method. Position evaluators also expose `begin` and `end` methods that the turn generator calls with the predecessor position before and after generating successors. `begin` and `accept` can return false to cause the turn generator to skip generating all or the rest of the successor positions.

The first generation uses the inherent-value position evaluator (`InherentValueVisitor`), which implements the rules for the first generation: a position is a win if any turn results in an opposing piece being pushed off the board (in which case turn generation stops early), and a loss if all turns result in an allied piece being pushed off the board (including when no pushes are possible).

Subsequent generations use a more complicated position evaluator based on outcounting to exploit locality in win-loss database queries. The rules for subsequent generations are simple: a position is a win if any successor is a loss, and is a loss if all successors are wins. We started with a straightforward implementation of these rules (`CompositeValueVisitor`). Through instrumentation, we discovered the win-loss database was being queried repeatedly for the same positions. The nature of Push Fight turns, particularly moves, means that similar positions tend to have overlap in their successors. We exploit this locality using outcounting. `OutcountingVisitor`'s `accept` method merely encodes the successor position as an integer and stores it in a set. The `end` method records the size of the successor set as the predecessor position's out count and appends successor-predecessor pairs to a buffer. When this buffer fills up, we sort it by successor using a radix sort [128], grouping the predecessors by successor (like the reduce step in a MapReduce computation). We look up each successor in the win-loss database. If it is a loss, all associated predecessors are recorded as wins. If it is a win, the out count of the predecessors is decremented. Then after processing all successors, any predecessors whose out count is zero are recorded to be losses (because all their successors are wins). Outcounting cuts the number of win-loss database lookups by about 1000 for the standard Push Fight board, so it is worth the extra complexity.

Opening evaluation uses a position evaluator (`OpeningProcedureVisitor`) implementing both the first generation's rules about the pushed-off piece and a non-outcounting version of the subsequent generation's rules about successor positions. We can get away without the complexity of outcounting here because there are relatively few openings. Because the forward search has completed, this position evaluator has full information about anchored positions, so openings that are not won or lost can be conclusively recorded as drawn.

Opening enumerator. The opening enumerator enumerates openings, non-anchored states in which each player's pieces are in their respective placement areas (their half of the board). The opening enumerator first computes half-positions for the allied and opposing pieces, then uses a nested loop to iterate over each pair of half-positions and invoke the turn generator on the merged position.

Opening database. Won, lost, and drawn openings are recorded in an opening database written at the end of the search. There are few enough openings that this database can consist of human-readable plain text files for each allied half-position storing the won, lost, and drawn enemy half-positions. The game is an overall win for the first player if there exists an allied half-position that combines with all opposing half-positions to be a winning opening, an overall loss for the first player if each allied position combines with at least one opposing half-position to be a lost opening, and is a draw otherwise.

Parallelism and checkpointing. To scale effectively to large boards, the solver must be parallel. To make progress on hardware with less-than-perfect uptime, the solver must split generations into chunks to provide resumption points. We implemented two position generators with differing task granularity, defined in terms of slices and subslices. A slice is a set of positions having their anchored king on a particular square (so there is one slice per anchorable square); a subslice is a set of positions having their non-anchored opposing kings on particular squares. For the standard Push Fight board, there are 18 slices each with 325 subslices (25 of which contain no positions because the anchored and non-anchored kings conflict).

The coarse-grained position enumerator splits the generation by slice and parallelizes across subslices. Thus the solver process runs 18 times per generation (sequentially), using all hardware threads on the machine to enumerate subslices of that slice. The fine-grained position enumerator splits the generation by slice and subslice, doing no parallelism of its own; instead, multiple solver processes can be run in parallel using a tool like GNU `parallel` [133]. In both cases, the solver writes chunks of the current level of the win-loss database before exiting that need to be concatenated before the next generation begins.

The fine-grained position enumerator offers better checkpointing because each subslice is computed independently. It also offers theoretically better CPU utilization because subslices from different slices can proceed in parallel, while the coarse-grained enumerator must wait for the longest subslice to finish before proceeding to the next slice. However, in practice the fine-grained enumerator has lower throughput because we use memory-mapped files to read the win-loss database and each page must be faulted into each process separately, resulting in more time being spent handling page faults instead of doing work. For the first slice of the standard Push Fight board, we take 2.1 times as many page faults using the fine-grained enumerator as we do using the coarse-grained enumerator, a major contributor to the computation requiring 87.05 hours with the fine-grained enumerator compared to 42.75 hours with the coarse-grained enumerator.

Symmetries. For our purposes, only distinguishing the current player's turn is relevant, not the colors of the pieces, hence our use of allied and opposing pieces instead of white and black pieces. The standard Push Fight board also offers 180-degree rotational symmetry. Our solver implements this symmetry by canonicalizing the position so the anchor is in player 1's piece placement area. If we enumerate only canonical slices, we only need to do this canonicalization in the turn generator just before calling the position evaluator. This symmetry is currently hard-coded in the solver, not deduced from the board specification by the board compiler.

4.5.4 Results

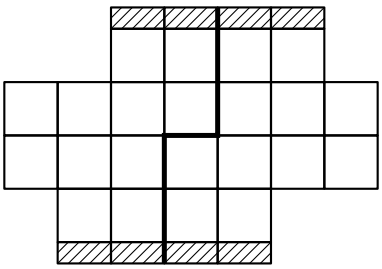


Figure 4-36: Modified Push Fight board with one column removed (compare to Figure 4-2). The bold line indicates how the board is divided at the beginning of the game.

We find that Push Fight played on a board with one column (four squares) removed and the new middle column split between the players for placement purposes (see Figure 4-36) is a draw under perfect play. We ran our solver on an AMD Ryzen Threadripper 2990WX with 64 hardware threads and 128 GiB of RAM running Arch Linux kernel version 5.5.3. Our code was compiled with GCC 9.2.1. The computation took six generations. The initial generation took 3.25 hours and resolved 89% of the positions as wins or losses, while each subsequent generation took between 10.5 and 11.5 hours and resolved positions as shown in Table 4.2, for a total of 59.2 hours. The win-loss database is 33.9 GB in size. Evaluating the opening positions took only 17 seconds; of the 21344400 opening positions, 19847380 are a win for the first player, 1497020 are drawn, and none are losses for the first player. For each placement of the first player's pieces, there is at least one placement of the second player's pieces resulting in a drawn opening, so the overall game is a draw.

The results of the initial generation show that most positions in this variant are immediate wins for the player to move. This is surprising from the perspective of most combinatorial board games, but makes sense for Push Fight because most positions have at least one opposing piece on the edge of the board and an allied pusher can usually move into position to push it off unless the other pieces form a wall. The results of the opening evaluation show that most openings are wins and none are losses. Together, these results show the first move is a significant advantage in this Push Fight variant.

gen	wins	T%	R%	losses	T%	R%	CPU time	real time	util
0	27719245608	89.102	89.102	48	0.000	0.000	145h10m41s	3h14m58s	44.7
1	0	0.000	0.000	4752534	0.015	0.140	516h28m13s	11h30m23s	44.9
2	15414038	0.050	0.455	0	0.000	0.000	515h14m31s	11h31m01s	44.7
3	0	0.000	0.000	38860	0.000	0.001	510h41m55s	11h20m29s	45.0
4	404724	0.001	0.012	0	0.000	0.000	482h26m50s	10h56m49s	44.1
5	0	0.000	0.000	0	0.000	0.000	462h57m28s	10h38m53s	43.5

Table 4.2: Summary of solver generations for the one-column-removed Push Fight board. For each generation, this table gives the number of winning and losing positions found as a raw number and as percentages of the total positions and of the remaining positions (not yet known to be won or lost), plus the total CPU time, real time, and utilization on our 64-thread machine.

We believe standard Push Fight is within reach of our current system given sufficient computation time. We have computed the initial generation for standard Push Fight, taking 2780.3 CPU-hours over 53.5 real-time hours and using 113.8 GB of storage, and the first of 9 slices (using the rotational symmetry of the board) of the second generation, taking 1395 CPU-hours. Assuming perfect utilization of a 64-thread machine and the search reaching a fixed point after five generations, we would need 40.9 days to compute all of the non-initial generations. With the coarse-grained position enumerator we only get about half utilization (32.6) because positions in different subslices have verying numbers of successors; this utilization is worse than on the one-column-removed board because the variance in subslice computation time increases with board size. We believe a computation time of 82 days is still feasible. Unfortunately, our machine is not reliable enough to compute entire slices at once,[7] necessitating the use of the fine-grained enumerator for better checkpointing. Using the fine-grained enumerator halves our utilization again due to increased page faults, increasing the required time to about 160 days.

4.5.5 Future Work: Design Space Exploration

After using our solver to solve standard Push Fight, we propose a solver-driven exploration of the design space of Push Fight variants. These are some of the questions we could answer by solving variants:

- **Board topology.** The standard Push Fight board has side rails on two sides and some corner squares removed. What if we add those corner squares back, so that the board is a complete rectangle? What if, instead of pushing over the side rails being illegal, the pushed pieces "wrap around" to the opposite side of the board, as if the board is on a cylinder? What if there are thin walls (blocking piece movement/pushing, but not occupying squares) in the middle of the board?

[7] Our machine hangs under sustained loads, requiring a power cycle. Our ability to troubleshoot the machine is limited due to the COVID-19 pandemic.

- **Piece count.** Standard Push Fight uses three kings and two pawns per player. What if each player has three pawns and two kings instead, or some other combination? What if the second player has additional pieces as compensation for going second? Is there a piece density (ratio of pieces to squares) after which no draws are possible?

- **Initial piece placement.** To start a standard Push Fight game, the first player places their pieces on their half of the board, then the second player places their pieces. What if players alternated placing single pieces (as in Nine Men's Morris)? What if players can place pieces anywhere on the board instead of only in their half? What if the first player places all pieces for both players, then the second player chooses which set to play?

- **Move count.** In standard Push Fight, a player may make up to two moves each turn. What if players could make up to one move, or up to three? What if players *must* make some number of moves, possibly introducing zugzwang?

- **Win condition.** In standard Push Fight, a player loses when one of their pieces is pushed off the board, regardless of which player is pushing. What if a player can continue playing after pushing their own piece off the board? (Having fewer pieces is not obviously a disadvantage.)

Our solver is generic enough to handle some of these questions already and can be extended to handle others. We already support different board shapes through the board compiler, including thin walls (by specifying rails in the interior of the board), and even one-way walls. It is not currently possible to specify a board is on a cylinder, but our board representation can encode cylindrical boards in its topology array. We also already support games with other numbers of pieces, and the solver (but not the board compiler) does not care if the players have different numbers of pieces. The solver supports other allowed move counts, but does not currently prevent a piece from being moved more than once. Different placement areas are already supported, but different opening procedures (e.g., players alternate placing single pieces) require writing new opening enumerators. Finally, allowing the game to continue after a piece is pushed off the board would require several changes, including to the `rank` function used to encode positions as integers, as it will not be one-to-one if some encoded positions have fewer pieces than others.

As a first step in design space exploration, we computed that Push Fight played on the same one-column-smaller board where each player must make exactly one move on their turn remains a draw. The computation took 13 generations, indicating that this variant has longer forced wins compared to the variant with just the modified board. There are also substantially more lost positions in this variant.

gen	wins	T%	R%	losses	T%	R%	CPU time	real time	util
0	26637037392	85.623	85.623	239796	0.001	0.001	10h10m36s	0h11m14s	54.4
1	2380284	0.008	0.053	15036188	0.048	0.336	32h19m15s	1h56m24s	16.7
2	14306410	0.046	0.321	23888	0.000	0.001	32h21m11s	1h58m39s	16.4
3	34456	0.000	0.001	421018	0.001	0.009	32h05m43s	1h59m45s	16.1
4	531672	0.002	0.012	798	0.000	0.000	32h15m11s	2h01m15s	16.0
5	4024	0.000	0.000	2358	0.000	0.000	32h18m14s	2h02m43s	15.8
6	13474	0.000	0.000	40	0.000	0.000	32h21m19s	2h03m56s	15.7
7	0	0.000	0.000	204	0.000	0.000	32h08m40s	2h04m36s	15.5
8	1176	0.000	0.000	0	0.000	0.000	32h32m47s	2h04m58s	15.6
9	0	0.000	0.000	9	0.000	0.000	27h54m46s	1h47m35s	15.6
10	10	0.000	0.000	5	0.000	0.000	32h13m21s	2h03m17s	15.7
11	10	0.000	0.000	0	0.000	0.000	30h18m37s	1h56m08s	15.7
12	0	0.000	0.000	0	0.000	0.000	32h06m46s	2h03m42s	15.6

Table 4.3: Summary of solver generations for the one-column-removed Push Fight board when each player must make exactly one move on their turn. Columns are the same as Table 4.2.

Chapter 5

Path Puzzles[1]

5.1 Introduction

Path puzzles are a type of pencil-and-paper logic puzzle introduced in Roderick Kimball's 2013 book [86] and featured in *The New York Times*'s Wordplay blog [19]. Figure 5-1 gives a small example. A puzzle consists of a (rectangular) grid of cells with two exits (or "doors") on the boundary and numerical constraints on some subset of the rows and columns. A solution consists of a single non-intersecting path which starts and ends at two boundary doors and which passes through a number of cells in each constrained row and column equal to the given numerical clue. Many variations of path puzzles are given in [86] and elsewhere, for example using non-rectangular grids, grid-internal constraints, and additional candidate doors, but these generalizations make the problem only harder.

Path puzzles are closely related to *discrete tomography* [75], in particular the 2D orthogonal form: given the number of black pixels in each row and column, reconstruct

[1]This chapter, except for Section 5.4, is from [31] (also available on arXiv [30]), which is joint work with Erik D. Demaine, Martin L. Demaine, Adam Hesterberg, Roderick Kimball and Justin Kopinsky.

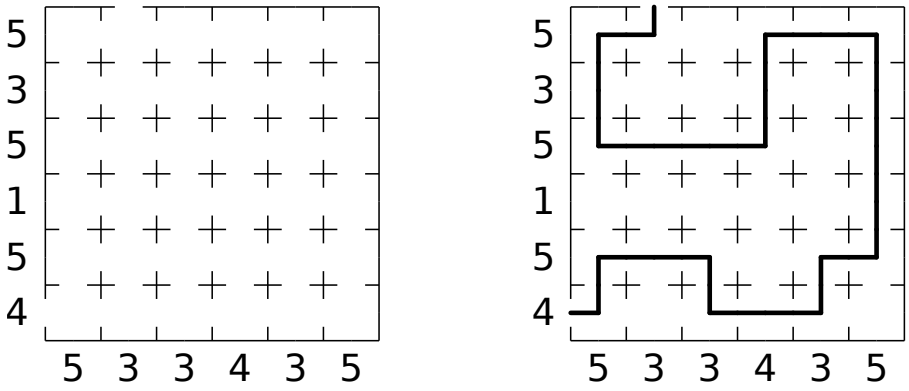


Figure 5-1: A Path Puzzle with complete row/column information (left) and its solution (right).

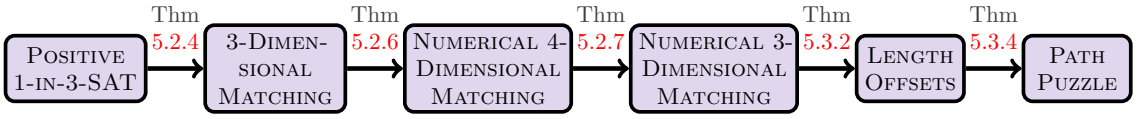


Figure 5-2: The chain of reductions used in our proof.

a black-and-white image. This problem arises naturally in reconstruction of shapes via x-ray images (which measure density). Vanilla 2-dimensional discrete tomography is known to have efficient (polynomial-time) algorithms [75], though it becomes hard under certain connectivity constraints on the output image [43].[2] A path puzzle is essentially a 2-dimensional discrete tomography problem with partial information (not all row and column counts) and an additional Hamiltonicity (single-path) constraint on the output image.

Our results. Unlike 2-dimensional discrete tomography, we show that path puzzles are NP-complete, even with perfect information (i.e., with all row and column counts specified). In other words, 2-dimensional discrete tomography becomes NP-complete with an added Hamiltonicity constraint. In fact, we prove the stronger results that perfect-information path puzzles are Another Solution Problem (ASP) hard and (to count solutions) #P-complete.

Figure 5-2 shows the chain of reductions we use to prove hardness of Path Puzzle. To preserve hardness for the ASP and #P classes, our reductions are *parsimonious*; that is, they preserve the number of solutions between the source and target problem instances, generally by showing a one-to-one correspondence thereof. We start from the source problem of Positive 1-in-3-SAT which is known to be ASP-hard [127, 80] and (to count solutions) #P-complete [80]. Along the way, we newly establish strong ASP-hardness and #P-completeness for 3-Dimensional Matching, Numerical 4-Dimensional Matching, Numerical 3-Dimensional Matching, and a new problem Length Offsets, in addition to Path Puzzle.

Solver. We also describe a path puzzles solver based on depth-first search and evaluate its performance on the most difficult puzzles from the path puzzles book [86]. Our solver is available on GitHub.[3]

Fonts. To further communicate the challenge of path puzzles to the general public, we designed a mathematical puzzle typeface (as part of a series[4]). Figure 5-3 gives the puzzle font, which has one path puzzle for each letter of the alphabet. Their solutions are designed to look like the 26 letters of the alphabet, and are verified unique using our solver. Look ahead to the solved font in Figure 5-8 in Section 5.6 when you no longer want to solve the puzzles.

[2] Most sets of row and column constraints are ambiguous; constraining the output image makes the problem harder by preventing an easy image from being found instead.

[3] https://github.com/jbosboom/pathpuzzle-solver

[4] See http://erikdemaine.org/fonts

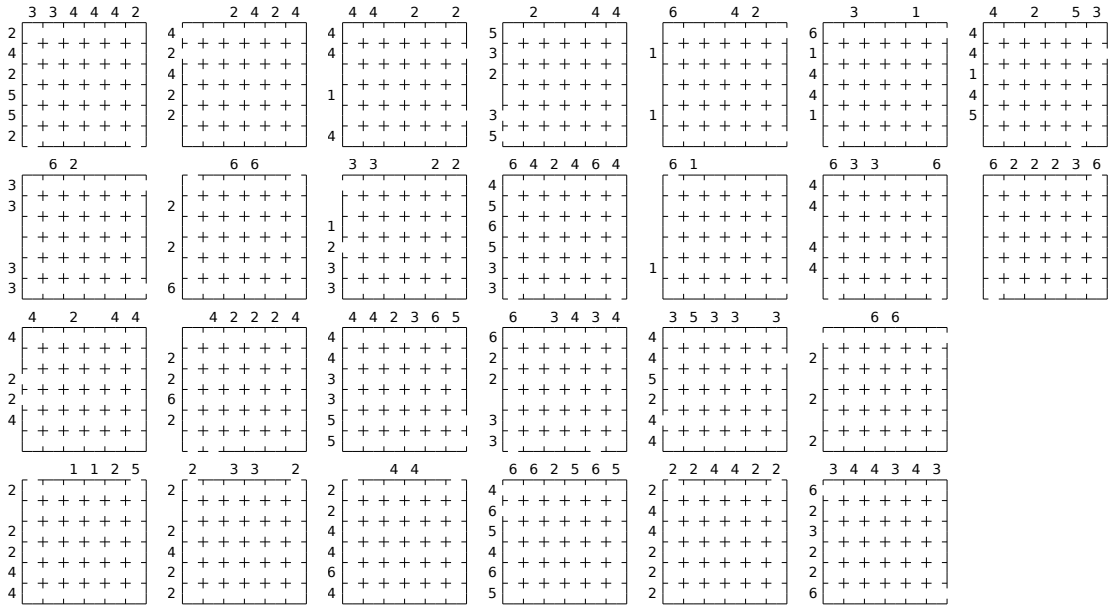


Figure 5-3: Puzzle font

5.2 Numerical 3DM is ASP-complete and #P-complete

The goal of this section is to prove that Numerical 3-Dimensional Matching is *strongly* ASP- and #P-complete, i.e., ASP- and #P-complete even when the n numbers are bounded by a polynomial in n. We follow a similar chain of reductions by Garey and Johnson [65], namely 3SAT → 3-Dimensional Matching → 4-Partition → 3-Partition, but replacing k-Partition with Numerical k-Dimensional Matching and starting from a different version of SAT:

Problem 5.2.1 (Positive 1-in-3-SAT)**.** *Given a 3CNF formula C with only positive literals, is there an assignment of variables such that exactly one literal in each clause of C is true?*

Lemma 5.2.2. Positive 1-in-3-SAT *is ASP-hard and #P-hard.*

Proof. 3SAT is shown to be #P-hard in [141]. Section 3.2.1 of [127] shows that 3SAT is ASP-hard.[5] Theorem 3.8 of [80] gives a parsimonious reduction from 3SAT to Positive 1-in-3-SAT.[6] Combining these results gives the claim. □

Problem 5.2.3 (3-Dimensional Matching)**.** *Given three sets X, Y, Z of equal cardinality and a set T of triples (x, y, z) where $x \in X, y \in Y, z \in Z$, is there a set $S \subseteq T$ such that each element of X, Y, Z appears in exactly one triple in S?*

Theorem 5.2.4. 3-Dimensional Matching *is ASP-hard and #P-hard, even when T is constrained not to contain any two triples agreeing on more than one coordinate.*

[5] Section 3.2.4 of [127] proves that 1-in-3-SAT is ASP-hard. Unfortunately, their problem definition allows negative clauses, while we need Positive 1-in-3-SAT.

[6] In [80], Positive 1-in-3-SAT is called "1-Ex3MonoSat".

Proof. We give a parsimonious reduction from Positive 1-in-3-SAT, using the variable gadget from Garey and Johnson's reduction from 3-SAT to 3-Dimensional Matching [65, Thm. 3.2, p. 50]. Given a Positive 1-in-3-SAT instance with a set V of variables and C of clauses, we construct the corresponding 3-Dimensional Matching instance as follows. We will represent the 3-Dimensional Matching instance as a hypergraph that is tripartite and 3-uniform, i.e., in which each edge connects exactly three vertices of different colors according to a 3-coloring of the vertices. We will say that vertices colored 0 belong to X, vertices colored 1 belong to Y, and vertices colored 2 belong to Z.

Clause triplication. First we triplicate each clause, producing the multiset $C' = C \sqcup C \sqcup C$ (the disjoint union of three copies of C). As a result, the number n_x of occurrences of each variable $x \in V$ in clauses in C' (multiply counting if x occurs multiple times in the same clause) is divisible by 3. A truth assignment for V satisfies C' if and only if it satisfies C, so this triplication does not affect correctness, but it will help us obtain a 3-coloring.

Variable gadget. Next, for each variable $x \in V$, we create a *variable gadget* consisting of $4n_x$ vertices associated with x; refer to Figure 5-4. We call n_x of the vertices *positive* x vertices, denoted $x_0, x_1, \ldots, x_{n_x-1}$ (one for each occurrence of x in C'); we call n_x of the vertices *negative* x vertices, denoted $\bar{x}_0, \bar{x}_1, \ldots, \bar{x}_{n_x-1}$; and we call $2n_x$ of them *auxiliary* vertices, denoted $x'_0, x'_1, \ldots, x'_{2n_x-1}$. The edges covering the auxiliary vertices are as follows: for each $i \in \{0, 1, \ldots, n_x - 1\}$, we add the "positive" edge $(x_i, x'_{2i}, x'_{2i+1})$ and the "negative" edge $(\bar{x}_i, x'_{2i+1}, x'_{(2i+2) \bmod 2n_x})$. No other edges cover the auxiliary vertices, so there are only two ways to cover them: choose all the positive edges, thereby covering all the positive vertices and none of the negative vertices, or choose all the negative edges, thereby covering all the negative vertices and none of the positive vertices. The former choice represents assigning x to be TRUE, while the latter choice represents assigning x to be FALSE.

Because n_x is divisible by 3, we can 3-color the vertices by assigning colors $0, 1, 2, 0, 1, 2, \ldots$ to the auxiliary vertices $x'_0, x'_1, x'_2, \ldots$, then coloring each positive and negative vertex with the one color not used in its edge, as shown in Figure 5-4.

The final edges of the variable gadget serve to collect "garbage" negative vertices. For each $i \in \{0, 1, \ldots, n_x/3 - 1\}$ (using that n_x is divisible by 3), we add another positive edge $(\bar{x}_{3i}, \bar{x}_{3i+1}, \bar{x}_{3i+2})$. These positive edges overlap the negative edges, so cannot be chosen in the FALSE assignment, but do not overlap the positive edges, and then they cover all the negative vertices. No other edges cover the negative vertices, so again we must choose all the positive edges or all the negative edges.

Therefore, local to the variable gadget, we cover all the auxiliary and negative x vertices, and either all or none of the positive x vertices. Only the positive x vertices will interact with other gadgets, through clause gadgets.

Clause gadget. Finally, for each clause $c = \langle x, y, z \rangle \in C'$, we *identify* one positive x vertex, one positive y vertex, and one positive z vertex all of the same color, resulting

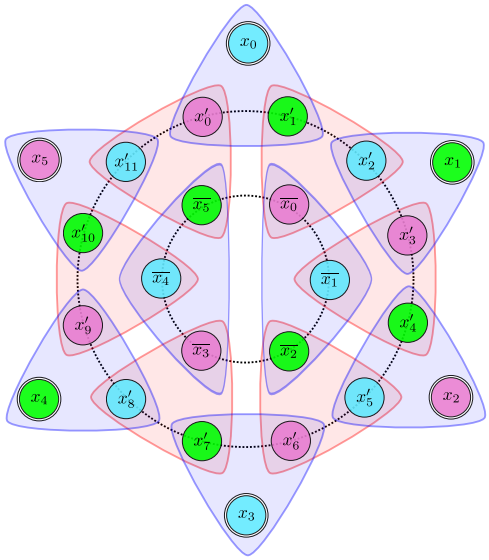


Figure 5-4: 3DM variable gadget for variable x occurring in $n_x = 6$ clauses in C' (two clauses in C). Hyperedges are drawn as shaded triangles; any solution must include all the positive (blue) or all the negative (red) hyperedges. Vertex colors $0, 1, 2$ are drawn as magenta, green, and cyan. Only the positive vertices (drawn with doubled outlines) are attached to other gadgets.

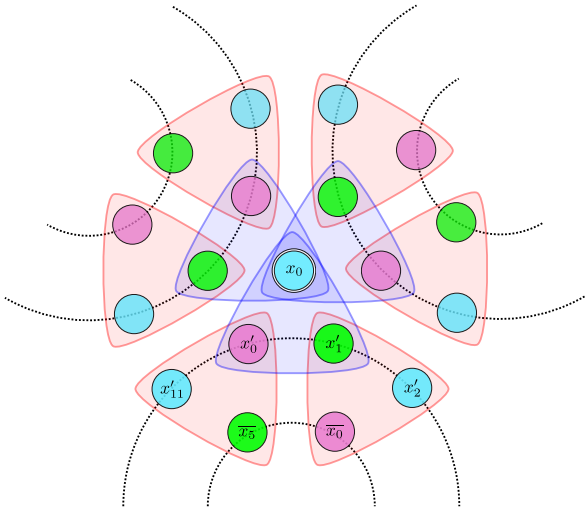


Figure 5-5: 3DM clause gadget, identifying cyan positive vertices from three variable gadgets (Figure 5-4). Although here we draw the three variables as distinct, we may also identify positive vertices from the same variable gadget (when the same variable appears twice in the same clause).

in a single vertex covered by one positive edge from each of the three corresponding vertex gadgets; refer to Figure 5-5. The three identified vertices are chosen to be unique to this clause gadget, so they will not be identified again, and thus will be covered exactly once if and only if exactly one of the three variables is assigned TRUE.

For each of the three copies of a clause in C, we choose the identified vertices to be a different color among $\{0, 1, 2\}$, so that each clause in C consumes exactly one positive vertex of each color from each of the three variable gadgets. (When a variable appears twice in the same clause, two of these variable gadgets will actually be the same, and we will end up consuming two positive vertices of each color, but the accounting remains the same.) Thus we will be able to use each positive vertex in each variable gadget exactly once, without running out of any particular color.

Equivalence. Because the identified (x, y, z) vertex in a clause gadget must be covered by exactly one edge in the 3-DIMENSIONAL MATCHING problem, exactly one of x, y, z must have an assignment of TRUE, which is the POSITIVE 1-IN-3-SAT constraint. Thus, given a solved instance of 3-DIMENSIONAL MATCHING, we can extract exactly one solution to the original POSITIVE 1-IN-3-SAT instance. Furthermore, given a solution to the POSITIVE 1-IN-3-SAT instance, we can produce exactly one solution to the 3-DIMENSIONAL MATCHING instance by choosing all the positive x edges if x is set to TRUE and all the negative edges if x is set to FALSE. Thus the reduction is parsimonious.

Examining Figures 5-4 and 5-5, we also see that no hyperedge shares more than one vertex, as claimed. □

Problem 5.2.5 (NUMERICAL k-DIMENSIONAL MATCHING)**.** *Given k multisets of positive integers $X_1, \ldots, X_k$ and a positive integer target sum t, does there exist a set $S \subseteq X_1 \times \cdots \times X_k$ of k-tuples such that, for each $(x_1, \ldots, x_k) \in S$, $x_1 + \cdots + x_k = t$, and each element of each X_i appears as the ith coordinate in exactly one element of S? (Thus $|X_1| = \cdots = |X_k| = |S|$, and we denote this common size by n.)*

We will consider specially the cases $k = 3$ and $k = 4$ for which we label the sets X, Y, Z and W, X, Y, Z respectively.

Theorem 5.2.6. NUMERICAL 4-DIMENSIONAL MATCHING *is strongly ASP-hard and #P-hard, even if $Y \cup (Y + Z)$ (where $Y + Z = \{y + z : y \in Y, z \in Z\}$) is guaranteed to be a set (not a multiset).*

Proof. We give a parsimonious reduction from 3-DIMENSIONAL MATCHING where no two triples in T agree on more than one coordinate, as guaranteed by Theorem 5.2.4. Our reduction loosely follows Garey and Johnson's original reduction [65, Thm. 4.3, p. 97] with extra care to ensure parsimony.

We are given a 3-DIMENSIONAL MATCHING instance with elements partitioned into sets

$$X = \{x_1, \ldots, x_n\}, Y = \{y_1, \ldots, y_n\}, Z = \{z_1, \ldots, z_n\}$$

and a set of triples $T \subseteq X \times Y \times Z$. Let $m_T(x_i)$ be the multiplicity of x_i in T, that is, the number of triples of T where x_i is the first coordinate, and similarly define $m_T(y_j)$ and $m_T(z_k)$.

	$x_i \in X$	$y_j \in Y$	$z_k \in Z$	$(x_i, y_j, z_k) \in T$
W'	$(10, i, 0, 0, 0, 0)_B$ $\mathbf{(10, i, -i, 0, 0, 0)_B}$	$(12, 0, 0, 0, -j, 0)_B$		
X'		$(10, 0, 0, j, 0, 0)_B$ $\mathbf{(10, 0, 0, j, -j, 0)_B}$	$(7, 0, 0, 0, 0, -k)_B$	
Y'			$(10, 0, 0, 0, 0, k)_B$	$(10, 0, i, 0, j, k)_B$
Z'	$(11, 0, -i, 0, 0, 0)_B$			$(10, -i, 0, -j, 0, -k)_B$

Table 5.1: The constructions of W', X', Y', Z'. Each column represents the source of the constructed elements from the original 3-Dimensional Matching instance. Most elements have multiplicity 1; bold elements have multiplicity one fewer than the corresponding 3-Dimensional Matching source item.

First we pick a large base $B = 100n$. We use the notation $(d_5, d_4, d_3, d_2, d_1, d_0)_B$ to represent the base-B number equal to $\sum_{i=0}^{5} d_i B^i = d_5B^5 + d_4B^4 + d_3B^3 + d_2B^2 + d_1B + d_0$. In the discussion that follows, we use that B is large enough that addition with a digit of the base-B representation of the numbers in question will never carry over to another digit, so $(d_5, d_4, d_3, d_2, d_1, d_0)_B + (d'_5, d'_4, d'_3, d'_2, d'_1, d'_0)_B = (d_5 + d'_5, d_4 + d'_4, d_3 + d'_3, d_2 + d'_2, d_1 + d'_1, d_0 + d'_0)_B$.

We construct a Numerical 4-Dimensional Matching instance with target $t = (40, 0, 0, 0, 0, 0)_B$ and multisets W', X', Y', Z' having the following elements (also refer to Table 5.1):

(a) For each $x_i \in X$, place $(10, i, 0, 0, 0, 0)_B$ in W', $(11, 0, -i, 0, 0, 0)_B$ in Z', and $m_T(x_i) - 1$ copies of $(10, i, -i, 0, 0, 0)_B$ in W'.

(b) For each $y_j \in Y$, place $(10, 0, 0, j, 0, 0)_B$ in X', $(12, 0, 0, 0, -j, 0)_B$ in W', and $m_T(y_j) - 1$ copies of $(10, 0, 0, j, -j, 0)_B$ in X'.

(c) For each $z_k \in Z$, place $(10, 0, 0, 0, 0, k)_B$ in Y' and $(7, 0, 0, 0, 0, -k)_B$ in X'.

(d) For each $(x_i, y_j, z_k) \in T$, place $(10, -i, 0, -j, 0, -k)_B$ in Z' and $(10, 0, i, 0, j, k)_B$ in Y'.

Importantly, the x_i, y_j, z_k above are indexed starting at 1, not 0. One can verify that the only quadruples summing to t are the following (given in W', X', Y', Z' order):

Type 1. For $(x_i, y_j, z_k) \in T$:

$$\begin{array}{rrrrrr} (10, & i, 0, & 0, 0, & 0)_B \\ +(10, & 0, 0, & j, 0, & 0)_B \\ +(10, & 0, 0, & 0, 0, & k)_B \\ +(10, & -i, 0, & -j, 0, & -k)_B \\ =(40, & 0, 0, & 0, 0, & 0)_B \end{array}$$

Type 2. For $(x_i, y_j, z_k) \in T$:

$$\begin{array}{rrrrrr} (12, 0, & 0, 0, & -j, & 0)_B \\ +(\ 7, 0, & 0, 0, & 0, & -k)_B \\ +(10, 0, & i, 0, & j, & k)_B \\ +(11, 0, & -i, 0, & 0, & 0)_B \\ =(40, 0, & 0, 0, & 0, & 0)_B \end{array}$$

Type 3. For $(x_i, y_j, z_k) \in T$:

$$\begin{array}{rrrrrr} (10, & i, -i, & 0, & 0, & 0)_B \\ +(10, & 0, & 0, & j, -j, & 0)_B \\ +(10, & 0, & i, & 0, & j, & k)_B \\ +(10, & -i, & 0, & -j, & 0, & -k)_B \\ =(40, & 0, & 0, & 0, & 0, & 0)_B \end{array}$$

Furthermore, there is a one-to-one correspondence between solutions to the source instance and solutions to the constructed instance by choosing a triple to be in the

solution $S \subseteq T$ of the 3-Dimensional Matching instance if and only if both the corresponding triples of the types 1 and 2 above are included in the solution S' of the constructed Numerical 4-Dimensional Matching instance. If a type-1 quadruple is included in the solution for some $(x_i, y_j, z_k) \in T$, then the corresponding type-2 quadruple must also be included because there is no other way to cover the element $(10, 0, i, 0, j, k)_B \in Y'$, and similarly for the reverse. To confirm that the rest of the elements can be covered by the type-3 quadruples, notice that there are exactly the correct number of $(10, i, -i, 0, 0, 0)_B \in W'$ and $(10, 0, 0, j, -j, 0) \in X'$ elements. In particular, for every i, there are $m_T(x_i)$ triples of the form $(10, -i, 0, *, 0, *)_B \in Z'$ and exactly one of these is covered by a type-1 quadruple, so the remaining ones can be matched with the $m_T(x_i) - 1$ elements of the form $(10, i, -i, 0, 0, 0)_B \in W'$, and similarly for the y_j. Thus we have a parsimonious reduction from 3-Dimensional Matching to Numerical 4-Dimensional Matching.

We can verify the claim that $Y' \cup (Y' + Z')$ is a set (not a multiset) using the initial assumption that no two triples in T agree on more than one coordinate. First, Y' is a set because, for each z_k, there is exactly one element $(10, 0, 0, 0, 0, k)_B \in Y'$, and for each triple $(x_i, y_j, z_k) \in T$, there is exactly one element $(10, 0, i, 0, j, k)_B \in Y'$; these two types of elements are disjoint because the third and fifth digits are always zero in the former but nonzero in the latter. Similarly, Z' is a set (a fact we will need later). Also, Y' and $Y' + Z'$ are disjoint because the first digit of any element of Y' is 10 while the first digit of any element of $Y' + Z'$ is at least 20.

To see that $Y' + Z'$ is a set, consider two equal sums $s_1 = y'_1 + z'_1$ and $s_2 = y'_2 + z'_2$ for $y'_1, y'_2 \in Y'$ and $z'_1, z'_2 \in Z'$. From $s_1 = s_2$, it follows that $y'_2 - y'_1 = z'_2 - z'_1$. We claim that, if $s_1 = s_2$, then $z'_1 = z'_2$ and thus $y'_1 = y'_2$, which suffices because we argued that Y' and Z' are sets. To prove the claim, we have two cases, one for each type of element of Z':

Case 1: If $z'_1 = (11, 0, -i_1, 0, 0, 0)_B$, then $z'_2 = (11, 0, -i_2, 0, 0, 0)$ or else s_1 and s_2 would differ in the first digit. Thus $y'_2 - y'_1 = z'_2 - z'_1 = (0, 0, i_1 - i_2, 0, 0, 0)_B$, but by the assumption that there are no two distinct triples of T sharing both y_j and z_k, there are no two distinct elements of Y' whose last two digits are equal but whose third digits are not, so this difference is impossible unless $y'_1 = y'_2$ and $z'_1 = z'_2$.

Case 2: If $z'_1 = (10, -i_1, 0, -j_1, 0, -k_1)_B$, then $z'_2 = (10, -i_2, 0, -j_2, 0, -k_2)_B$ or else s_1 and s_2 would differ in the first digit. Thus $y'_2 - y'_1 = z'_2 - z'_1 = (0, i_1 - i_2, 0, j_1 - j_2, 0, k_1 - k_2)_B$, but no elements of Y' have nonzero second or fourth digits, so it must be that $i_1 = i_2$ and $j_1 = j_2$. By the assumption that no distinct triples of T share both x_i and y_j, it must be that $k_1 = k_2$ as well, so $z'_1 = z'_2$ as claimed. □

Theorem 5.2.7. Numerical 3-Dimensional Matching *is strongly ASP-hard and #P-hard, even if X is required to be a set (not a multiset).*

Proof. We give a parsimonious reduction from Numerical 4-Dimensional Matching where $W \cup (W + X)$ is a set, as guaranteed by Theorem 5.2.6 (relabelling Y, Z to W, X). Our reduction is essentially Garey and Johnson's reduction from 4-Partition

to 3-Partition [65, Thm. 4.4, p. 99], with some extra care regarding identical elements and splitting elements into separate sets X', Y', Z'.

Following the reduction in [65], given a Numerical 4-Dimensional Matching instance $W = \{w_1, \ldots, w_n\}$, $X = \{x_1, \ldots, x_n\}$, $Y = \{y_1, \ldots, y_n\}$, $Z = \{z_1, \ldots, z_n\}$ with target t, we will construct a Numerical 3-Dimensional Matching instance X', Y', Z' with target sum $t' = (64, 4)_B$ in base $B = t$. We assume without loss of generality that every element of W, X, Y, Z is strictly between $t/5$ and $t/3$.[7] First we define the elements that will appear in $X' \cup Y' \cup Z'$:

$$\begin{aligned} w'_i &= (21, 4w_i + 1)_B, \\ x'_j &= (19, 4x_j + 1)_B, \\ y'_k &= (19, 4y_k + 1)_B, \\ z'_\ell &= (21, 4z_\ell + 1)_B, \\ u[w_i, x_j] &= (24, -4(w_i + x_j) + 2)_B, \\ \bar{u}[w_i, x_j] &= (20, \quad 4(w_i + x_j) + 2)_B, \\ C &= (20, 0)_B. \end{aligned}$$

Now we can construct the desired Numerical 3-Dimensional Matching instance, splitting these elements into three multisets X', Y', Z':

$$\begin{aligned} X' &= \{w'_i : 1 \le i \le n\} \cup \{\bar{u}[w_i, x_j] : 1 \le i, j \le n\}, \\ Y' &= \{x'_j : 1 \le j \le n\} \cup \{z'_\ell : 1 \le \ell \le n\} \cup \{n^2 - n \text{ copies of } C\}, \\ Z' &= \{u[w_i, x_j] : 1 \le i, j \le n\} \cup \{y'_k : 1 \le k \le n\}. \end{aligned}$$

There are $2 \times 3 \times 2 = 12$ possible forms of triples, shown below grouped by the equivalence classes modulo 4 of the second coordinate of their sum (with shaded boxes to indicate the only triples that will turn out to be valid):

0 (mod 4)	**1 (mod 4)**	**2 (mod 4)**	**3 (mod 4)**
$(w'_{i'},\ x'_{j'},\ u[w_i, x_j])$	$(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}],\ x'_{j'},\ u[w_i, x_j])$	$(w'_{i'},\ C,\ y'_{k'})$	$(w'_{i'},\ x'_{j'},\ y'_{k'})$
$(w'_{i'},\ z'_{\ell'},\ u[w_i, x_j])$	$(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}],\ z'_{\ell'},\ u[w_i, x_j])$		$(w'_{i'},\ z'_{\ell'},\ y'_{k'})$
$(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}],\ x'_{j'},\ y'_{k'})$			$(w'_{i'},\ C,\ u[w_i, x_j])$
$(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}],\ z'_{\ell'},\ y'_{k'})$			$(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}],\ C,\ y'_{k'})$
$(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}],\ C,\ u[w_i, x_j])$			

The second coordinate of t' is congruent to 0 (mod 4), so triples in the second, third, and fourth columns never sum to t'.

Of the triples in the first column, two of them cannot actually sum to t'. Triples of the form $(w'_{i'}, z'_{\ell'}, u[w_i, x_j])$ sum to $(66, 4(w_{i'} + z_{\ell'} - w_i - x_j) + 4)_B$. For this to equal t', the second coordinate must equal $-2t + 4$, so $w_{i'} + z_{\ell'} - w_i - x_j$ must equal $-t/2$.

[7] If a Numerical 4-Dimensional Matching instance has any elements $\ge t$, it trivially has no solutions (as all elements are positive). Otherwise, we can convert it to an instance with this property by adding $2t$ to each element in W, X, Y, Z and changing the target sum from t to $\hat{t} = 9t$. Then every element is strictly between $2t$ and $3t$, and thus strictly between $\hat{t}/5 = 9t/5$ and $\hat{t}/3 = 9t/3$.

But by the assumption that every element of W, X, Y, Z is strictly between $t/5$ and $t/3$, the smallest possible value for $w_{i'} + z_{\ell'} - w_i - x_j$ is greater than $-4t/15$, so triples of this form never sum to t'. Similarly, triples of the form $(\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}], x'_{j'}, y'_{k'})$ sum to $(58, 4(w_{\bar{\imath}} + x_{\bar{\jmath}} + x_{j'} + y_{k'}) + 4)_B$. For this to equal t', the second coordinate must equal $6t + 4$, so $w_{\bar{\imath}} + x_{\bar{\jmath}} + x_{j'} + y_{k'}$ must sum to $3t/2$, but the largest possible value for that expression is less than $4t/3$, so triples of this form never sum to t'.

This leaves three forms of triples that can sum to $t' = (64, 4)_B$:

$$(w'_{i'}, x'_{j'}, u[w_i, x_j]), \quad (\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}], z'_{\ell'}, y'_{k'}), \quad \text{and} \quad (\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}], C, u[w_i, x_j]).$$

A triple of the second form encodes a quadruple in the input Numerical 4-Dimensional Matching instance; the triple sums to t' (after a carry $(60, 4t+4)_B = (64, 4)_B$) exactly when $w_{\bar{\imath}} + x_{\bar{\jmath}} + z_{\ell'} + y_{k'} = t$. The map $w_{\bar{\imath}} + x_{\bar{\jmath}} \mapsto \bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}]$ is one-to-one, so from our assumption that $W + X$ is a set, $\{\bar{u}[w_i, x_j]\}$ is also a set, and so this encoding is unique. A triple of the first form sums to t' exactly when $w_{i'} + x_{j'} - w_i - x_j = 0$. Because $W + X$ is a set and the map $w_i + x_j \mapsto u[w_i, x_j]$ is one-to-one, we must have $i' = i$ and $j' = j$ in valid triples of the first form, uniquely collecting the $u[w_i, x_j]$ elements corresponding to $\bar{u}[w_{\bar{\imath}}, x_{\bar{\jmath}}]$ elements used in triples of the second form. Similarly, $\bar{\imath} = i$ and $\bar{\jmath} = j$ in valid triples of the third form, uniquely collecting the unused $u[w_i, x_j]$ and $\bar{u}[w_i, x_j]$ elements using all $n^2 - n$ copies of C. Thus there is a one-to-one correspondence between solutions to the input Numerical 4-Dimensional Matching instance and the constructed Numerical 3-Dimensional Matching instance, so the reduction is parsimonious and Numerical 3-Dimensional Matching is ASP- and #P-hard.

It remains to verify that $X' = \{w'_i\} \cup \{\bar{u}[w_i, x_j]\}$ is a set (not a multiset). We argued above that $\{\bar{u}[w_i, x_j]\}$ is a set (using that $W + X$ is a set), and $\{w'_i\}$ is a set because we assumed W is a set and the map $w_i \mapsto w'_i = (20, 4w_i + 1)_B$ is one-to-one. It remains to show that $\{w'_i\}$ is disjoint from $\{\bar{u}[w_i, x_j]\}$, which follows because $w'_i \equiv 1 \pmod 4$ and $\bar{u}[w_i, x_j] \equiv 2 \pmod 4$. Therefore X' is a set. □

5.3 Parsimonious Reductions from Numerical 3DM to Path Puzzles

The goal of this section is to parsimoniously reduce Numerical 3-Dimensional Matching (as analyzed in Section 5.2) to Path Puzzle, thereby proving the latter strongly NP-, ASP-, and #P-hard. We first introduce a more geometric view of Numerical 3-Dimensional Matching, called Length Offsets, and prove its equivalence. It will then be relatively easy to represent Length Offsets as a Path Puzzle.

Problem 5.3.1 (Length Offsets)**.** *Given a set (not a multiset) of positive integer lengths $a_1, a_2, \ldots, a_n$, and given m nonnegative integer target densities $t_0, t_1, \ldots, t_{m-1}$, can we place n intervals with integer endpoints within $[0, m]$ and lengths $a_1, a_2, \ldots, a_n$, respectively, such that the number of intervals overlapping $(i, i+1)$ is exactly the target density t_i? In other words, can we choose nonnegative integer offsets $b_1, b_2, \ldots, b_n$*

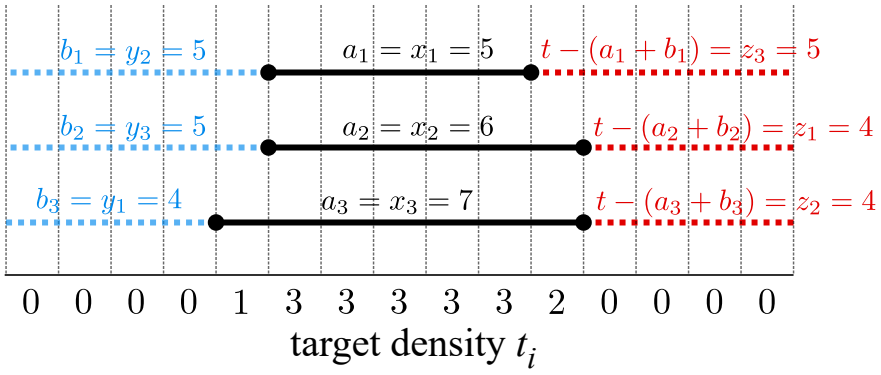


Figure 5-6: Length Offsets instance obtained by reducing from Numerical 3-Dimensional Matching where $X = \{5, 6, 7\}, Y = \{4, 5, 5\}, Z = \{4, 4, 5\}, t = 15$; and its solution corresponding to the Numerical 3-Dimensional Matching solution of $(5, 5, 5)$, $(5, 6, 4)$, $(4, 7, 4)$.

such that $a_j + b_j \leq m$ for each j $(1 \leq j \leq n)$; and, for each i $(0 \leq i < m)$, there are exactly t_i indices j such that $b_j \leq i < a_j + b_j$?

Theorem 5.3.2. Length Offsets *is parsimoniously reducible from* Numerical 3-Dimensional Matching *in which at least one of the three multisets is actually a set.*

Proof. We give a parsimonious reduction from Numerical 3-Dimensional Matching where X is a set, as guaranteed by Theorem 5.2.7. Specifically, consider a Numerical 3-Dimensional Matching instance with set $X = \{x_1, \ldots, x_n\}$, multisets Y and Z, and a target sum t. Assume without loss of generality that every element of X, Y, Z is strictly between $t/4$ and $t/2$.[8] We construct a Length Offsets instance that we claim has the same number of solutions: the n lengths are given simply by $a_i = x_i$, and the target densities are given by $t_i = n - |\{y \in Y : y > i\}| - |\{z \in Z : t - z \leq i\}|$, where $0 \leq i < m$ and $m = t$. See Figure 5-6 for an example. The intuition is that we place intervals for X, Y, Z, left-align the intervals for Y, right-align the intervals for Z, and count the remaining density for X intervals.

It remains to show that every solution to the original Numerical 3-Dimensional Matching instance corresponds to a solution to the constructed Length Offsets instance, and different Numerical 3-Dimensional Matching solutions correspond to different Length Offsets solutions. Equivalently, we will provide an injective map from Numerical 3-Dimensional Matching solutions to Length Offsets solutions, and an injective map from Length Offsets solutions to Numerical 3-Dimensional Matching solutions.

[8] If a Numerical 3-Dimensional Matching instance has any elements $\geq t$, it trivially has no solutions (as all elements are positive). Otherwise, we can convert it to an instance with this property by adding t to each element in X, Y, Z and changing the target sum from t to $\hat{t} = 4t$. Then every element is strictly between t and $2t$, and thus strictly between $\hat{t}/4 = 4t/4$ and $\hat{t}/2 = 4t/2$.

N3DM to Length Offsets. To convert a Numerical 3-Dimensional Matching solution into a Length Offsets solution, we assign $b_j = y_k$ for each solution triple (x_j, y_k, z_ℓ). Figure 5-6 shows this solution for the example.

For each i and each triple (x_j, y_k, z_ℓ), $b_j \leq i < a_j + b_j$ if and only if $y_k \leq i$ and $i < x_j + y_k = t - z_\ell$, so either

1. $y_k > i$; or
2. $t - z_\ell \leq i$; or
3. $b_j \leq i < a_j + b_j$.

The first case applies $|\{y \in Y : y > i\}|$ times, and the second case applies $|\{z \in Z : t - z \leq i\}|$ times, so the third case applies $n - |\{y \in Y : y > i\}| - |\{z \in Z : t - z \leq i\}| = t_i$ times. Thus our choice of the offsets is a valid solution to the Length Offsets instance.

If two Numerical 3-Dimensional Matching solutions differ, then (using that X is a set) some x is matched with a different y in each solution, so when converting those solutions to Length Offsets solutions, we assign the corresponding length different offsets.

Length Offsets to N3DM. To convert a Length Offsets solution into a Numerical 3-Dimensional Matching solution, for each length–offset pair (a_i, b_i), we match the triple $(a_i, b_i, t - a_i - b_i)$. These triples obviously sum to t and are therefore legal, but we need to show that their elements exist and cover X, Y, and Z, respectively.

1. Every a_i is an x_i and vice versa, so X is covered by $\{a_i\}$.
2. For each i, we have

$$\begin{aligned} t_i - t_{i-1} &= (n - |\{y \in Y : y > i\}| - |\{z \in Z : t - z \leq i\}|) \\ &\quad - (n - |\{y \in Y : y > i - 1\}| - |\{z \in Z : t - z \leq i - 1\}|) \\ &= |\{y \in Y : y = i\}| - |\{z \in Z : t - z = i\}| \end{aligned}$$

   For $i < t/2$, the second term is 0 (because $z < t/2$ by assumption), so $t_i - t_{i-1}$ is precisely the number of elements of Y that equal i. On the other hand, in a Length Offsets instance, when $i < t/2$, $t_i - t_{i-1}$ is the number of segments which pass through i but not $i - 1$, i.e., the number of segments which begin at i and therefore have offset $b_j = i$. Therefore, Y is covered by $\{b_j\}$.

3. Following the same argument as above, but for $i > t/2$, we have that $t_i - t_{i-1} = -|\{z \in Z : t - z = i\}|$. On the other hand, in the Length Offsets problem, for $i > t/2$, $t_i - t_{i-1}$ is the negative of the number of segments which *end* at $i - 1$; i.e., it is the negative of the number of indices j such that $a_j + b_j = i$, or equivalently, $t - (t - (a_j + b_j)) = i$. Thus, Z is covered by $\{t - (a_j + b_j)\}$ as claimed.

Different solutions to the Length Offsets instance correspond to different solutions to the Numerical 3-Dimensional Matching instance because, if two Length Offsets solutions differ, then some length a_j gets different offsets, so the corresponding x_j is matched to different elements of Y in the two Numerical 3-Dimensional Matching solutions.

We have shown two injective maps between solutions of the Length Offsets instance and solutions of the Numerical 3-Dimensional Matching instance, so our reduction is parsimonious. □

We now make a brief observation that we make use of later.

Lemma 5.3.3. *In every solution to* Length Offsets *instances produced by the reduction in the proof of Theorem 5.3.2, no line segment shares its left endpoint with the right endpoint of another.*

Proof. By assumption, all elements of the Numerical 3-Dimensional Matching instance lie in the exclusive interval $(t/4, t/2)$. Our reduction sets $a_i = x_i$, so $t/4 < a_i < t/2$. Our mapping between solutions assigns $b_i = y_k$, so $t/4 < b_i < t/2$. Adding these inequalities yields $t/2 < a_i + b_i < t$ for all i. Then $b_i < t/2 < a_j + b_j$ for all i and j, so the sets of left and right endpoints are disjoint. □

We are now ready to prove our main theorem.

Theorem 5.3.4. Path Puzzle *is NP-, #P- and ASP-hard.*

Proof. We give a parsimonious reduction from Length Offsets, as produced by Theorem 5.3.2 so that Lemma 5.3.3 applies. Given a Length Offsets problem with lengths $a_1, \ldots, a_n$ and target densities $t_0, \ldots, t_{m-1}$, we construct an equivalent Path Puzzle instance as follows. Figure 5-7 shows our construction instantiated for the same Length Offsets instance from Figure 5-6.

Dimensions: The grid has $2m + 3$ rows and $(12n + 6)n - 1$ columns.

We group the columns into n *blocks* $B_1, \ldots, B_n$ of $12n + 5$ columns each, interspersed with $n - 1$ *lone columns*. Thus block B_i $(1 \le i \le n)$ consists of columns $(12n + 6)i - (12n + 5), \ldots, (12n + 6)i - 1$ and the ith lone column $(1 \le i < n)$ is column $(12n + 6)i$.

Doors: We place doors at the left and right ends of the top row.

Row labels: Counting up from the bottom, the $(2i + 2)$nd row has a label of $4n + t_i$, for each i $(0 \le i \le m)$; the $(2m + 2)$nd and $(2m + 3)$rd (topmost) rows have labels $4n$ and $5n - 1$, respectively; and all other row labels are blank.

Column labels: Each lone column has a column label of 1. Each block B_j $(1 \le j \le n)$ has labels of $2m + 3$ on its first two and last two columns; a label of $2a_j + 1$ on its middle column ($6n + 3$th column); and labels of 1 on all other columns (which split into two sections of $6n$ consecutive columns).

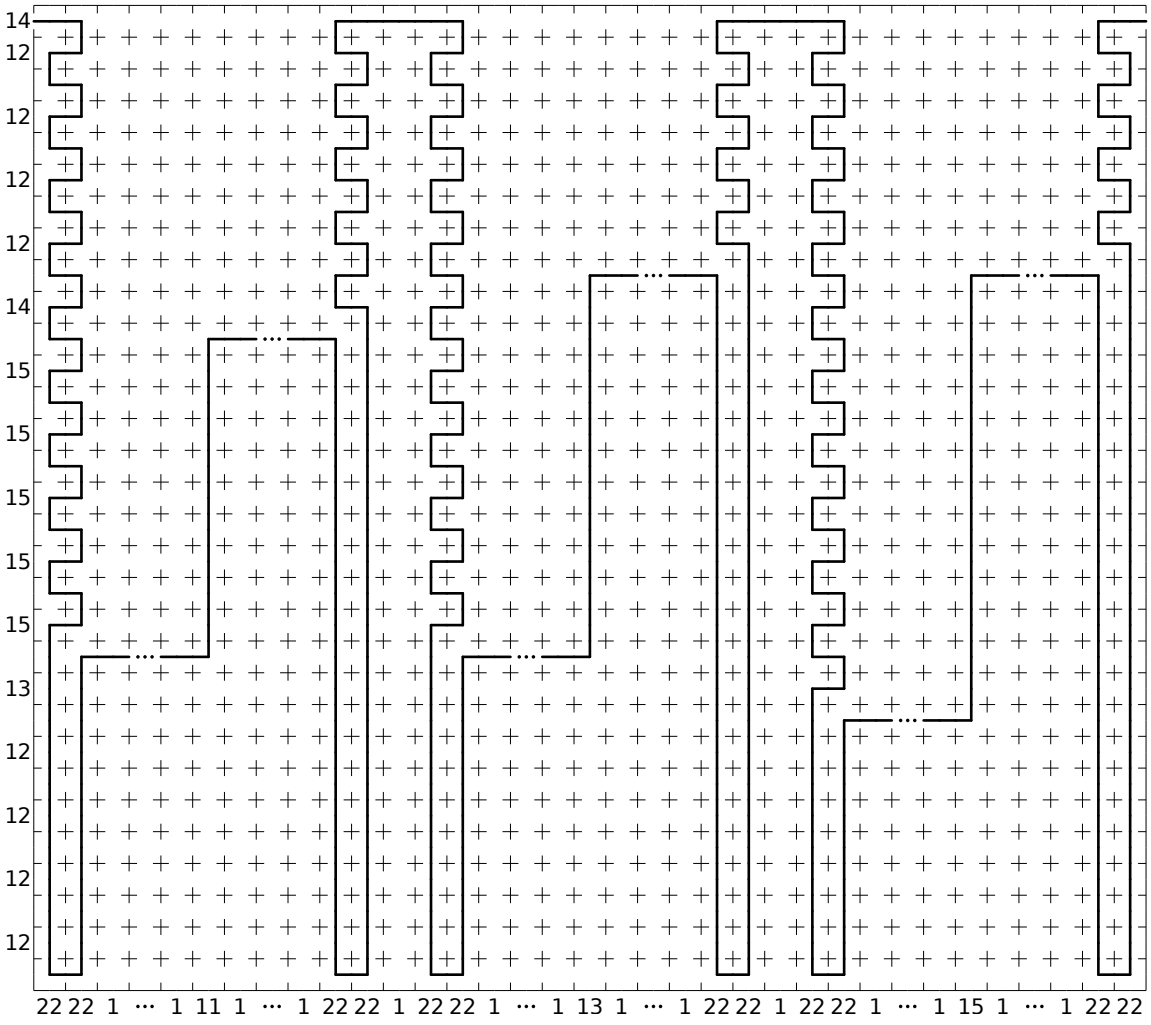


Figure 5-7: Solution to a Path Puzzle instance, reduced from Length Offsets from Figure 5-6 where $n = 3, m = 15, a_i = (5, 6, 7), t_i = (0, 0, 0, 0, 1, 3, 3, 3, 3, 3, 2, 0, 0, 0, 0)$. Ellipses elide sections of $6n = 18$ columns each labeled 1.

Any solution to this path puzzle has the following properties:

1. Every square in the first two and last two columns of each block is visited, by the column labels of $2m + 3$.

2. Every section of $6n$ consecutive columns labeled 1 (within a block) corresponds to a single horizontal path, which must be in one of the blank rows because all row labels are less than $6n$.

3. Every column labeled $2a_i + 1$ is a single vertical line segment, because both neighboring columns are labeled 1, just enough to enter and exit the column once.

4. No top square of a column labeled $2a_i + 1$ is visited, because if one were, the second square from the top of such a column would also be visited (by Property 3 and because $2a_i + 1 > 1$), but then that row would have more than $4n$ visited squares (by Property 1).

5. In each line column, the top square (and only that square) is visited, because those are the remaining squares on the top row that can be visited, and are just enough (with Property 1) to account for a total of $5n$.

6. The vertical line segment in the (unique) column labeled $2a_i+1$ (from Property 3) visits a_i rows with labels of $4n + t_j$ for various j. Of that $4n + t_j$, $4n$ visits are accounted for by the full columns of Property 1, so the positions of those line segments are a solution to the Length Offsets problem.

7. Given placements of the vertical line segments corresponding to a valid solution to the Length Offsets problem, we claim that the rest of the path is uniquely determined. The set of visited squares in each gadget is uniquely determined by the previous properties. In each pair of columns labeled $2m + 3$:

   (a) The bottom two squares each have only two visited neighbors, each other and the square above them, so each of them connects by the path to those two squares.

   (b) For squares in the pair of columns below the entry point of the length $6n$ horizontal path, the long U-shaped path shown in Figure 5-7 is forced. For each horizontal pair of squares except the bottom pair, the squares below connect to them. If the pair squares connect to each other, they form a closed loop, so they must connect to the squares above them instead.

   (c) For squares above the entry point of the length $6n$ horizontal path, the zig-zag path is forced. The pair of columns divides evenly into 2×2 chunks because the entry point of the length $6n$ path is in a row with no label, and the only such rows are at even height. Let *inside* and *outside* be relative to the center of the block. In each chunk, the bottom inside square can't connect to the square below (because that square is already known to connect down and to the inside), so it connects to the outside and up.

Similarly, the top outside square can't connect to the square below (because that square is already known to connect down and to the bottom inside square), so it connects to the inside and up (except that in the very top 2×2 chunk, the top outside square can't connect down and can and must connect to the outside to satisfy the top row).

Thus, each solution to the path puzzle determines a solution to the Length Offsets problem, and that solution is uniquely determined, so the number of solutions to the Length Offsets problem is the same as the number of solutions of the path puzzle, and the reduction is parsimonious as desired. Note that we are relying on the uniqueness of the lengths a_i from the Length Offsets problem definition; otherwise, permuting which copy of a duplicated length gets which offset in the path puzzle would generate multiple solutions to Path Puzzle from each solution of Length Offsets. □

In fact, our reduction can be converted into one giving complete information (i.e., all row and column labels), demonstrating that partial information is not the source of Path Puzzle's hardness.

Theorem 5.3.5. *Perfect-information* Path Puzzle *(with all row and column labels given as labels) is NP-, #P- and ASP-hard.*

Proof. Recall that the reduction from the proof of Theorem 5.3.4 (referring to Figure 5-7) already provides all column sums and about half of the row sums. We show how to provide the remaining row sums without giving away information about the solution to the original Length Offsets instance.

The rows with missing labels are the $(2i-1)$st rows for $i = 1, 2, \ldots, m$. In each such row, the solution path must visit $(6n+2)r_i$ cells, where r_i is the number of segments in the Length Offsets solution which have an endpoint at i. Recall from Lemma 5.3.3 that no line segment shares its left endpoint with the right endpoint of another. Thus there is only one type of endpoint at each coordinate i, and we can compute $r_i = |t_{i+1} - t_i|$. The value of r_i depends only on the Length Offsets instance, not its solutions, so we can modify our reduction to specify a label of $(6n+2)r_i$ for row $2i-1$, producing a perfect-information instance of Path Puzzle. □

5.4 Solving Path Puzzles Using Depth-First Search

We implemented a path puzzles solver in C++ based on depth-first search. We used this solver to verify that our path puzzle font has unique solutions. In this section, we describe the solver and investigate its performance on the most difficult puzzles from the path puzzles book [86].

The basic idea of the solver is to explore possible paths in a depth-first manner while tracking the count of cells remaining to be used for each clue. When the solver considers extending the path into a neighbor of the last cell in the path, the counters for each clue covering that neighbor are checked. If all those counters are greater than zero, the solver decrements all the counters and extends the path; if any of those

counters is 0, moving into that cell would violate a clue. After trying all the neighbors of a cell, the solver backtracks, incrementing counters for the cell being removed from the path. When the path reaches an exit, if all the counters are 0, all the clues are satisfied and the path is reported as a solution. The solver continues the search after reporting a solution to check that the solution is unique.

Details. The core of the solver (excluding puzzle input and solution output) is small enough to be presented in Listing 5.1. Each cell in the puzzle is represented by an instance of `struct Cell` in the `cells` vector and each clue is represented by an `int` in the `counters` vector. To simplify handling of non-rectangular puzzles, we identify cells and clues by index instead of their coordinates in the puzzle. The `terminals` vector stores the indices of door cells in sorted order.

Each cell stores its index, its row and column in the puzzle (for use when printing solution paths), the indices of its four neighbor cells (padded with `-1` if a cell has fewer than four neighbors), and the indices of four counters corresponding to the clues covering the cell.

Each counter in the `counters` vector that corresponds to a clue is initialized to the clue's number. Additionally, there is one dummy counter initialized to `-1` that does not correspond to a clue. Cells covered by fewer than four clues use this dummy counter (possibly multiple times) so that every cell references four counters, simplifying the loops that check, decrement and increment counters.

Most of the solver's logic is in the `solve_recurse` function. The current path is stored in a `tsl::ordered_set`,[9] a hash set that maintains the order of insertion. First, the solver checks whether the current path ends at a door cell (lines 14-15) and all the counters are less than or equal to zero (line 16); if so, the current path is a solution and the solver saves it in the `results` vector. The condition on line 13 that the last cell in the path has a greater index than the first cell ensures each path is found only once; without it, the solver would find each path twice (once reversed).

After checking for solutions, the solver attempts to extend the path into a neighbor of the last cell in the path. This is not possible if the neighbor index is `-1` (line 21) or if the neighbor is already in the path when the solver tries to insert it (line 27). The solver also does not extend the path if any of the candidate cell's counters are 0 (lines 22-26); such counters represent clues that would be violated if the path was extended. The dummy counter is initialized to `-1` and no counter is ever incremented above its initial value, so the dummy counter cannot cause a candidate to be rejected. If these checks all pass, the newly-added cell's counters are decremented and the solver recurses. After the recursive call returns, the cell's counters are incremented and the newly-added cell is removed from the path.

In the `solve` function, the solver iterates over each terminal except the last, adding it as the first cell in the path, decrementing the terminal's counters and calling `solve_recurse` to begin the depth-first search. After the search terminates, the terminal's counters are incremented and the solver continues with the next terminal. The solver does not search from the last terminal in the `terminals` vector because the

[9] https://github.com/Tessil/ordered-map

```
struct Cell {
    unsigned int n, r, c;
    std::array<int, 4> neighbors;
    std::array<unsigned int, 4> counters;
};

std::vector<Cell> cells;
std::vector<unsigned int> terminals;
std::vector<int> counters;

void solve_recurse(tsl::ordered_set<unsigned int>& path,
        std::vector<std::vector<unsigned int>>& results) {
    if (path.back() > path.front() &&
            std::find(terminals.begin(), terminals.end(),
                    path.back()) != terminals.end() &&
            std::all_of(counters.begin(), counters.end(), le_zero))
        results.push_back(std::vector<unsigned int>(path.begin(),
                path.end()));

    for (int n : cells[path.back()].neighbors) {
        if (n < 0) continue;
        bool counters_continue = false;
        for (int ci : cells[n].counters)
            if (counters[ci] == 0)
                counters_continue = true;
        if (counters_continue) continue;
        if (!path.insert(n).second) continue;

        for (int ci : cells[n].counters)
            --counters[ci];
        solve_recurse(path, results);
        for (int ci : cells[n].counters)
            ++counters[ci];
        path.pop_back();
    }
}

std::vector<std::vector<unsigned int>> solve() {
    std::vector<std::vector<unsigned int>> results;
    tsl::ordered_set<unsigned int> path;
    for (unsigned int ti = 0; ti + 1 < terminals.size(); ++ti) {
        path.clear();
        path.insert(terminals[ti]);
        for (int ci : cells[path.back()].counters)
            --counters[ci];
        solve_recurse(path, results);
        for (int ci : cells[path.back()].counters)
            ++counters[ci];
    }
    return results;
}
```

Listing 5.1: Core of the path puzzle solver.

puzzle	cells	clues	doors	runtime (s)
p. 109	56	17	2	0.1
p. 110 top	69	20	2	0.1
p. 110 bottom	88	18	2	5.1
p. 112 top	64	16	2	0.0
p. 113	90	22	2	41.4
p. 114 top	57	16	4	0.1
p. 114 bottom	100	20	2	0.2
p. 116 bottom	49	14	2	0.0
p. 117 top	57	14	2	0.0
p. 117 bottom	84	12	2	1.2
p. 119	63	15	2	0.1
p. 120 top	62	19	7	0.0
p. 120 bottom	83	21	2	0.3
p. 122	61	14	2	3.5
p. 124	194	56	2	–

Table 5.2: Running times of our solver on the puzzles from the last chapter of the path puzzles book [86]. "–" indicates the solver exceeded the 10-minute time limit.

symmetry-breaking check on line 13 implies searches starting at a particular terminal will only find paths to terminals coming later in the sorted `terminals` vector.

The solver takes as input a text file containing cell data, counters and their initial values, and terminal cell indices. A separate Python script converts tab-separated puzzle files (such as those exported from a spreadsheet program) into the solver's input format. The solver prints solution paths to standard output using the row and column stored for each cell.

Evaluation. We evaluated our solver on all 15 puzzles with numeric clues[10] from the last chapter of the path puzzles book [86]. We ran the solver on an AMD Ryzen 2200G with a base clock of 3.5 GHz and a boost clock of 3.7 GHz with 6 GiB of DDR4-3200 memory running Arch Linux kernel version 5.6.14. Table 5.2 shows the results.

Overall, our solver performed well, solving all but one puzzle and solving 10 of the 15 puzzles in less than a second. Two puzzles stand out as taking a long time to solve, the puzzles on page 113 and page 124. It is not just that these puzzles are large — note that the bottom puzzle on page 114 is larger than the puzzle on page 113, but is solved in less than a second — but that they have a large amount of area far away from the doors. If the solver makes a mistake near the doors, it will spend a long time enumerating paths in the area far from the doors before being able to correct its mistake. Solving these puzzles quickly probably requires a solver that tries to make human-like deductions about the puzzle instead of just checking paths.

[10]Some puzzles in the book have "encrypted" clues, where the numbers are replaced one-to-one with letters.

5.5 Open Problems

One interesting open problem is whether Planar 3DM, where the bipartite graph of elements and triples is planar, is also ASP-hard and #P-hard. This problem is known to be NP-hard [53], and the variable–clause gadget structure in the proof of Theorem 5.2.4 is close to preserving planarity. Unfortunately, the initial clause tripling destroys any planarity in the input, and seems difficult to avoid.

Another intriguing open problem is whether discrete tomography with partial information, but no Hamiltonian path constraint, is NP-hard. If true, this would be another aspect of path puzzles which make them hard.

5.6 Solution to the Font Puzzles

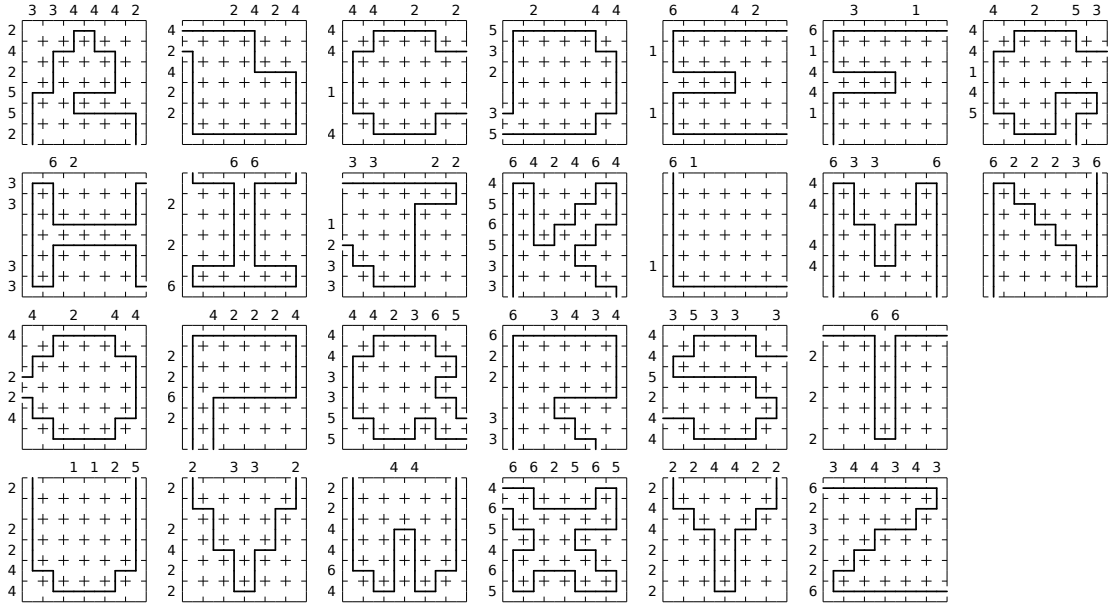


Figure 5-8: Solved font

Chapter 6

Solving Nonograms with Automata

6.1 Introduction

Nonograms (also known as Paint-by-Numbers, Griddlers, and Picross) are a type of logic puzzle involving coloring the faces of a square grid either black (present) or white (absent). Each row and column of the puzzle (collectively called *lines*) may have a sequence of integer clues indicating the sequence of lengths of black cells in that line, each of which is separated by at least one white cell, possibly with white cells before the first black cell or after the last black cell. Lines without clues are unconstrained (may have any pattern of white and black cells). See Figure 6-1 for an example nonogram puzzle and its solution.

In this chapter, we present experiments in using automata to solve nonograms. Automata are an appealing way to solve nonograms because the constraints have a reasonable representation as regular expressions and thus solutions can be straightforwardly implemented on top of existing libraries providing basic automata and

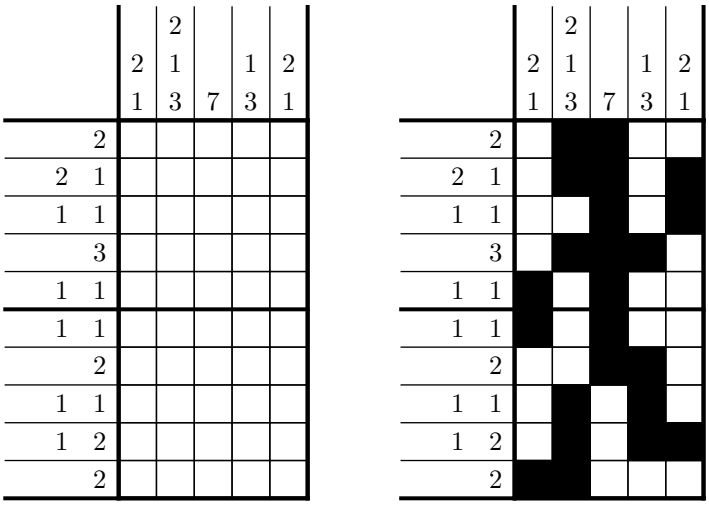


Figure 6-1: "Dancer" by Jan Wolter and its solution. Per the survey [149] (of which Jan Wolter is the author), permission for redistribution of this puzzle has been granted.

regular expression operations. Unfortunately, we find that the performance of our automata-based solver is not competitive with state-of-the-art solvers [149].

We find that the order in which automaton intersections are carried out strongly affects the space required to solve nonograms. Our experience matches related work on solving all-solutions Boolean satisfiability with automata, which found strong performance effects from automata intersection ordering (clause ordering) [37]. We prove that finding an optimal intersection ordering is strongly NP-complete or weakly PSPACE-hard depending on the magnitude of the upper bound on intermediate automaton size, and hard to approximate in both cases. These results improve upon prior work [28] that proved NP-completeness for intersection optimization with polynomially bounded intermediate automaton size, and did not consider inapproximability.

6.2 Related Work

Nonograms. Nonograms are NP-complete and ASP-complete [139]. Nonetheless, there are many published algorithms for solving nonograms [155, 23, 150, 26]. Wolter [149] surveys many solvers, including both solvers specialized for nonograms and frontends for generic constraint solvers, and benchmarks them on a set of puzzles taken from Wolter's nonogram website, `webpbn.com`. We use Wolter's benchmark set in our experiments. Of the solvers surveyed, some use regular expressions as an input format for a generic constraint solver, and some use regular expression engines for solving single lines, but none use automata directly to solve the whole puzzle.

Automata for specification and solving. Automata and regular expressions are commonly used to specify constraints in constraint satisfaction problems, to the point of being included in the MiniZinc standard constraint modeling language [104]. For example, the Gecode constraint solving system supports both automata and regular expressions [126, Section 4.4.13, Section 7.4] and uses them in the example solvers for nonograms. Lagerkvist and Pesant [94] describe regular expression encodings of pentomino puzzles and solitaire battleship puzzles and solve them using Gecode. However most constraint solvers do not use automata operations to solve the system, instead translating them into a generic representation used for all supported constraint types. There has also been work to use automata as input for SAT solvers [117].

Automata have also been used to solve constraint problems, both for generic problems [143, 12] and specifically problems over strings [67]. They have also been used to solve all-solutions Boolean satisfiability [37], with explicit mention of the impact of automata intersection ordering (clause ordering) on performance.

6.3 Our Nonogram Solver

Our nonogram solver works by computing an automaton whose language is the set of solutions to the nonogram (using an alphabet of `0` for white cells and `1` for black cells). For each row and each column having clues, our solver creates an extended regular

expression describing exactly the set of solutions to that line while not constraining other lines and compiles it into a finite automaton. The resulting automata are sorted according to an intersection ordering heuristic, then the intersection of the first two automata is computed and the result is minimzed to form the "accumulator" automaton. For each remaining line automaton, the intersection of the accumulator and that line automaton is computed and minimized to form the new accumulator automaton. The solution(s) to the puzzle can then be read off the accepting paths of the final accumulator automaton.

Note that our solver always finds all solutions to the puzzle. This allows the solver to determine whether the solution is unique. On the other hand, for puzzles with many solutions, always tracking them all can be a disadvantage if all solutions are not required.

Creating a regular expression for a row is straightforward. For each clue n, we create the expression 1^n, separated by 0^+ and padded on each end with 0^*. That expression is intersected with a regular expression matching any string of the row's length, and the result is padded on either side by expressions matching the prior and subsequent rows. For example, the second row of the puzzle in Figure 6-1 has clues 2 1, for which our solver would build the regular expression $.^5((0^*1^20^+10^*)\&.^5).^{40}$, where dot matches any symbol and & represents intersection. (The regular expressions for each row could all be combined into a single regular expression, but we leave them separate so the ordering heuristics can reorder them.)

Creating a regular expression for a column is more complicated because the symbols for a column are not adjacent in the string as they are for rows. If there are a columns before and b columns after the current column, for each clue n, we create the expression $(.^a1.^b)^n$, separated by $(.^a0.^b)^+$ and padded on each end with $(.^a0.^b)^*$. That expression is intersected with a regular expression matching any string of length equal to the size of the puzzle. For example, the second column of the puzzle in Figure 6-1 has clues 2 1 3, for which our solver would build the regular expression $((.^10.^3)^*(.^11.^3)^2(.^10.^3)^+(.^11.^3)(.^10.^3)^+(.^11.^3)^3(.^10.^3)^*)\&.^{50}$. Because column regular expressions are more complex than row regular expressions, column automata are larger than row automata and vary more in size with the number of clues.

After the regular expressions are compiled into automata, the automata are sorted according to one of the following intersection ordering heuristics:

- Rows: row automata in order of the corresponding rows, then column automata in order of the corresponding columns. (This is the order the automata are generated in, so this heuristic is a no-op.)

- Cols: column automata in order of the corresponding columns, then row automata in order of the corresponding rows.

- Interleave: alternating row and column automata in order of the corresponding rows and columns, then any remaining automata in order. (For example, for the puzzle in Figure 6-1, the order would alternate row 1, then column 1, then row 2, then column 2, and so on, until after column 5 when the remaining rows appear in order.)

- Fewest (most) solutions: automata sorted in ascending (descending) order by the number of solutions to the line.

- Fewest (most) states: automata sorted in ascending (descending) order by their number of states.

Experimental setup. We conducted our experiments on an AMD Ryzen Threadripper 2990WX with a base clock of 3.0 GHz and a boost clock of 4.2 GHz and 128 GiB of DDR4-2933 RAM running Arch Linux kernel version 5.5.3. Our code was compiled with GCC 9.2.1. We ran our solver on each puzzle with each heuristic, with a time limit of 30 minutes (using `timeout` from GNU coreutils) and a memory limit of 16 GiB (via `ulimit -v`[1]). (This memory limit is twice that used in the survey.) Running times and memory high-water marks were recorded using `time -f "%e %M"` from GNU coreutils. GNU `parallel` [133] was used for orchestration. For puzzles with multiple solutions, we include the time to count the solutions (the number of accepting paths in the final automaton), but not the time to enumerate them.

Results. Table 6.1 shows the running time for every puzzle and heuristic combination. No trial exceeded the time limit, but many trials were killed to due exceeding the memory limit (indicated in the table by a dash). The solver performed best using the rows-first heuristic, solving 13 of the 28 puzzles in the benchmark set. The solver performed worse overall using the fewest-states heuristic, but solved one puzzle (Flag) that it did not solve with the rows-first heuristic, so rows-first is not strictly better. Because column automata are larger and more varied in size than row automata, the difference between these two heuristics is mostly in the ordering of the column automata.

As a check on the quality of our heuristics, for puzzles that finished within one minute under any heuristic, we also ran 20 trials with a random intersection order and an additional 20 trials where only the column automata order was randomized, both with a 1 minute timeout. Of these 480 random trials, 180 finished within the time limit. None of the times were better than the time of the best heuristic for that puzzle. While this is a small sample, it suggests our heuristics are reasonable.

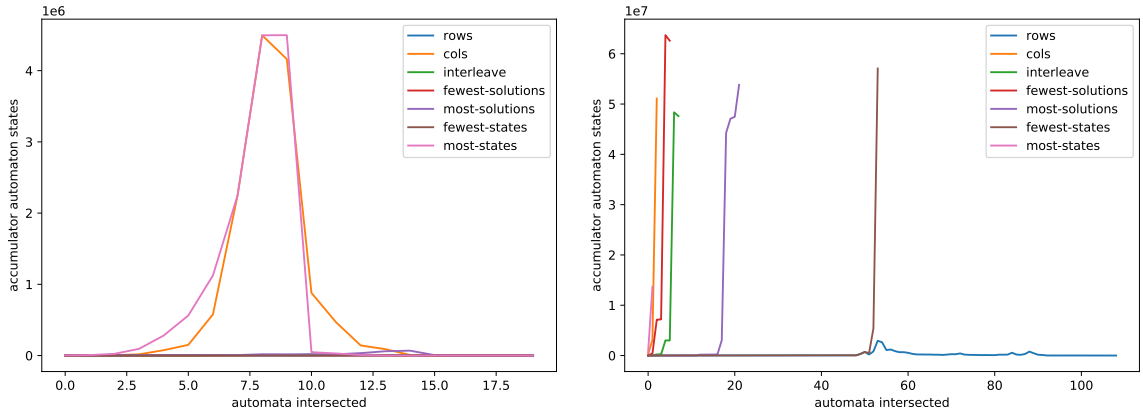
In comparison to solvers from the survey, our solver's performance is poor, as most solvers from the survey solved more puzzles (including one that solved all 28) faster while running on a much weaker machine (an AMD Phenom II X4 810) with a lower memory limit (8 GiB).

The solver failed with most puzzle-heuristic combinations due to exceeding the memory limit. The solver's memory consumption is at its maximum during intersection, when it has to store the two input automata, the output automaton, and a hash table mapping pairs of states in the input automata to states in the output automaton. The intersection order impacts the peak memory consumption. Figure 6-2 shows two graphs of accumulator automaton size for each heuristic as the solver solves two

[1]This is a virtual address space limit, but our solver touches all memory it maps and our system has no swap, so this is effectively a resident set size limit.

puzzle	rows	cols	interl.	few. sol.	most sol.	few. st.	most st.
#1: Dancer	0.0	0.1	0.0	0.0	0.0	0.0	0.1
#6: Cat	0.1	–	–	–	11.3	0.2	–
#21: Skid	0.1	–	52.5	0.4	–	0.4	–
#27: Bucks	0.4	–	–	–	–	19.5	–
#23: Edge	0.0	42.2	0.1	0.0	0.2	0.0	26.4
#2413: Smoke	0.2	–	–	82.8	–	0.2	–
#16: Knot	–	–	–	–	–	–	–
#529: Swing	26.5	–	–	–	–	275.4	–
#65: Mum	139.6	–	–	–	–	–	–
#7604: DiCap	48.4	–	–	–	–	–	–
#1694: Tragic	–	–	–	–	–	–	–
#1611: Merka	–	–	–	–	–	–	–
#436: Petro	5.4	–	–	–	–	–	–
#4645: M&M	–	–	–	–	–	–	–
#3541: Signed	22.8	–	–	–	–	–	–
#803: Light	–	–	–	–	–	–	–
#6574: Forever	48.1	–	–	–	–	–	–
#10810: Center	–	–	–	–	–	–	–
#2040: Hot	–	–	–	–	–	–	–
#6739: Karate	–	–	–	–	–	–	–
#8098: 9-Dom	0.9	–	–	19.1	–	0.6	–
#2556: Flag	–	–	–	–	–	71.9	–
#2712: Lion	–	–	–	–	–	–	–
#10088: Marley	–	–	–	–	–	–	–
#18297: Thing	–	–	–	–	–	–	–
#9892: Nature	–	–	–	–	–	–	–
#12548: Sierp	–	–	–	–	–	–	–
#22336: Gettys	–	–	–	–	–	–	–

Table 6.1: Running times (in seconds) of our automata-based nonogram solver on the benchmark puzzles from the survey [149]. "–" indicates the solver exceeded the 16 GiB memory limit. Puzzles are listed in the same order as in the survey, roughly sorted by increasing size and difficulty.



(a) #23: Edge

(b) #3541: Signed

Figure 6-2: Graphs of the size of the accumulator automaton after each intersection (before minimizing) with each heuristic for two puzzles. (Note the y-axis scale is in millions in the left graph and tens of millions in the right graph.)

puzzles. For the Edge puzzle (which was solved under every heuristic), we can see in Figure 6-2a that the cols-first and most-states heuristics result in accumulator automata that get large, then simplify as more line automata are intersected. These two heuristics also took much longer to solve the puzzle than the other heuristics. For the Signed puzzle, which was only solved under the rows-first heuristic, we can see the most of the other heuristics' accumulator size spike early in the process before hitting the memory limit, while the fewest-states heuristic spikes about halfway through when it begins processing column automata (this puzzle is nearly square).

6.3.1 Scalability Analysis on the n-Dom Puzzles

The 9-Dom puzzle from the survey's benchmark set, shown in Figure 6-3, is one instance of a family of puzzles of varying varying size that are the subject of a solver scalability analysis [148] as a part of Wolter's survey. As explained in [148], these puzzles share the same solution logic regardless of size: the black cells in the upper-right of the puzzle can be deduced, after which the next block of black cells in the upper-right of the remaining area can be deduced, and so on. Effectively, making a deduction in n-Dom leaves $(n-1)$-Dom in the remaining area, to be solved the same way. For humans, all n-Dom puzzles are of the same difficulty.

Most computer solvers are not capable of this logic and instead exhibit exponentially increasing runtime as n increases; of the solvers examined in the survey's scalability analysis, the Kjellerstrand solver running on the lazyfd constraint engine clearly performs better than the other solvers, but still exhibits exponential growth (just more slowly). As shown in Tables 6.2 and 6.3 and Figure 6-4, our automata-based solver is no different, with both running time and memory usage growing exponentially with n.

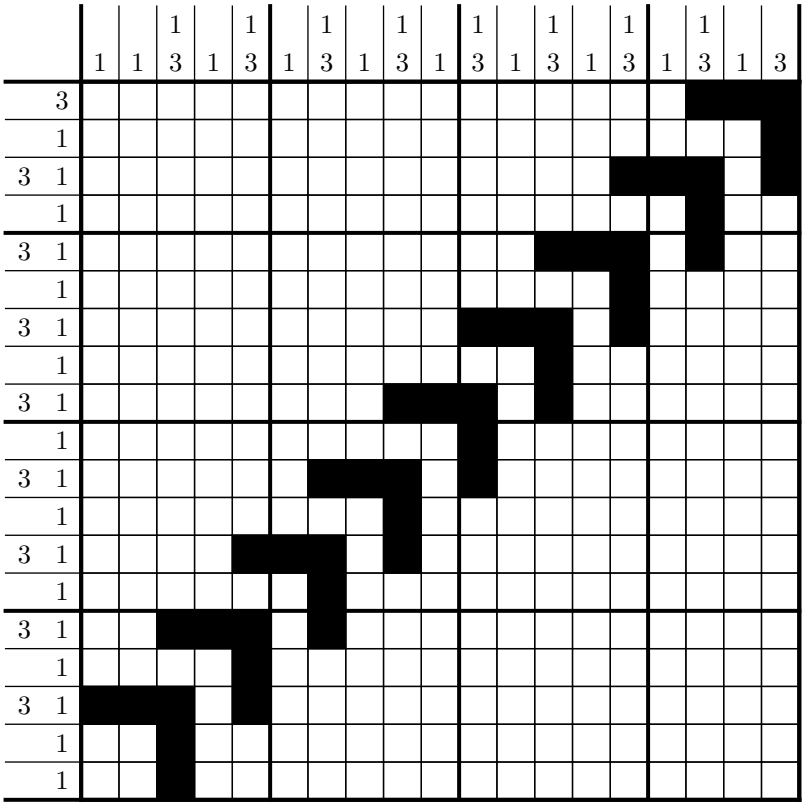


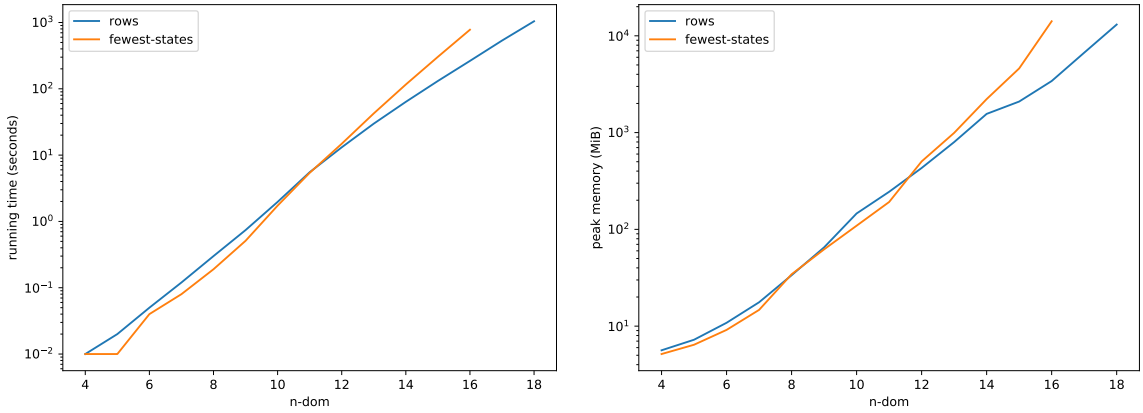
Figure 6-3: "Domino Logic III" (9-Dom) by Josh Greifer. Per the survey [149], permission for redistribution of this puzzle has been granted.

puzzle	rows	fewest states
4-Dom	0.0	0.0
5-Dom	0.0	0.0
6-Dom	0.1	0.0
7-Dom	0.1	0.1
8-Dom	0.3	0.2
9-Dom	0.7	0.5
10-Dom	2.0	1.7
11-Dom	5.5	5.4
12-Dom	13.1	14.8
13-Dom	29.8	42.6
14-Dom	63.6	116.2
15-Dom	131.3	303.7
16-Dom	263.6	778.1
17-Dom	533.8	–
18-Dom	1043.7	–
19-Dom	–	–

Table 6.2: Running times (in seconds) of our automata-based nonogram solver on the n-Dom puzzles. – indicates the solver exceeded the 16 GiB memory limit.

puzzle	rows	fewest states
4-Dom	5.6	5.1
5-Dom	7.2	6.4
6-Dom	10.8	9.1
7-Dom	17.6	14.7
8-Dom	33.6	34.4
9-Dom	65.2	62.1
10-Dom	146.0	109.1
11-Dom	244.4	191.7
12-Dom	430.7	503.4
13-Dom	797.6	993.0
14-Dom	1560.2	2218.5
15-Dom	2092.6	4600.5
16-Dom	3409.0	14134.7
17-Dom	6689.7	–
18-Dom	13071.0	–
19-Dom	–	–

Table 6.3: Peak memory consumption (in MiB) of our automata-based nonogram solver on the n-Dom puzzles. "–" indicates the solver exceeded the 16 GiB memory limit.



(a) Running time.

(b) Peak memory consumption.

Figure 6-4: Graphs of the data in Table 6.2 and Table 6.3, with a logarithmic scale on the y-axis.

6.4 Optimal Automaton Intersection is Hard[2]

In the previous section, we found that automata are not a competitive way to solve nonograms. The order in which our solver carried out automata intersections strongly affected our solver's ability to finish solving puzzles without running out of memory. Work on solving all-solutions Boolean satisfiability by automaton intersection [37] also found that performance depended on intersection ordering. This evidence naturally raises the problem of finding an optimal intersection ordering, that is, an ordering that minimizes the maximum size of the intermediate automata. We formalize this as the following decision problem.

Problem 6.4.1 (Optimal Automaton Intersection)**.** *Given k automata $A_1, \ldots, A_k$ and an integer M, is there a expression tree of intersection operations such that, for each tree node, the smallest automaton recognizing the intersection of the languages of descendants of that node has at most M states?*

This problem statement is phrased in terms of an expression tree and languages to permit implementations more advanced than the linear-order intersect-then-minimize implementation we used for solving nonograms. In particular, an algorithm that can analyze the input automata can directly construct automata for the intersection languages instead of using the generic intersection algorithm based on taking the cross product of the states. Even with this freedom, this problem is hard, even to approximate.

Prior work studied a similar problem in which intersections are constrained to occur in some linear order instead of a general expression tree, proving NP-completeness when the limit on intermediate automaton size is polynomially bounded by reduction from the travelling salesman problem [28]. We give a proof of NP-completeness for our problem (allowing a general expression tree) by reduction from Boolean satisfiability, further obtaining an inapproximability result. Then we study the case when the limit on intermediate automaton size is exponential and prove it PSPACE-hard and hard to approximate; it remains open whether it is in PSPACE or EXPTIME-complete.

Both of our reductions are based on the idea that when the input problem is a YES instance there is a linear intersection order resulting in the empty language where all the intermediate languages have automata of size less than M, after which all the remaining automata can be intersected with no size constraint; when the input problem is a NO instance, any intersection order results in a language containing strings of length x mod P where $P > M$ is a product of primes, which requires at least P states to recognize by repeated application of [116, Theorem 5].

Theorem 6.4.2. *When M is polynomially bounded, Optimal Automaton Intersection is strongly NP-complete and NP-hard to approximate within a factor of $n^{2/3-\varepsilon}$.*

Proof. This problem is in NP: we can guess an expression tree of intersection operations and evaluate each node by computing the intersection and minimizing the resulting

[2]This section is joint work with Josh Brunner, Michael Coulombe, Erik D. Demaine, and Jayson Lynch.

automaton. The largest intersection we might evaluate is between two automata with M states, which may result in minimizing an automaton with at most M^2 states, so we can verify in polynomial time.

To prove NP-hardness, we reduce from EU3SAT-4, a SAT variant where each clause contains three literals, each variable appears in at most four clauses, and each variable appears at most once in each clause, which is NP-complete [136]. Given an EU3SAT-4 formula, for each variable x_i, we add the clause $x_i \vee \neg x_i$, breaking the EU3 property and adding two variable occurrences, but preserving uniqueness of literals in each clause. The resulting formula has n variables and m clauses. We then set $M = n^{3c}$ for some constant c and choose the three smallest primes $p_1, p_2, p_3 > n^c$ (i.e., greater than $\sqrt[3]{M}$) and the two smallest primes $q_1, q_2 > n^{3c/2}$ (i.e., greater than $\sqrt{M}$). The automata we generate are over an alphabet containing a symbol x_i for each variable, a symbol c_i for each clause (including the clauses we added), and an extra symbol 0.

For each variable, we generate two automata A_i and $\neg A_i$ that correspond to setting that variable true or false. Both A_i and $\neg A_i$ accept strings that begin with 0 or any variable symbol except x_i, followed by a large alternation over clause symbols, where branches of the alternation representing literals that conflict with the truth assignment for x_i give rise to large loops in the automaton. For A_i, if $\neg x_i$ appears as the kth literal in clause c_i, this alternation contains $({c_i}^{p_k})^*$ if c_i contains three variables and $({c_i}^{q_k})^*$ if c_i contains two variables; otherwise, this alternation contains ${c_i}^*$. Automaton $\neg A_i$'s alternation is similar except defined with respect to clauses containing positive x_i literals. Finally, for both automata, the alternation always contains $(0^{q_1})^*$. For example,

$$A_i = V_i({c_1}^*|({c_2}^{p_2})^*|{c_3}^*|({c_4}^{q_1})^*|(0^{q_1})^*)$$

where V_i denotes 0 or any variable symbol except x_i, is one possible variable automaton. Each variable has conflicting literals in up to four clauses in the input formula and in one binary clause we added, necessitating at most $4p_3 + q_2$ states for those branches of the alternation, plus another q_1 states for the 0 branch and up to m states in the branches for other clause symbols. Then each variable automaton takes at most $2q_2 + 4p_3 + m + 1 = O(\sqrt{M})$ states.

We also create a finalizer automaton F that accepts strings beginning with any variable symbol followed by an alternation over ${c_i}^*$ for each clause symbol c_i and $(0^{q_2})^*$. F also has $O(\sqrt{M})$ states.

If the input formula has a satisfying assignment, intersecting the corresponding variable automata results in an automaton of size less than M. The intersection automaton accepts strings starting with 0 followed by an alternation over clause symbols and 0 (the same structure as the individual variable automata). Because the assignment is satisfying, at least one literal in each clause is satisfied, so each branch of the alternation is a loop of size at most p_2p_3 (or q_1 or q_2, which are smaller), so the overall automaton size is less than $(m+1)p_2p_3+2 < (5n+1)p_2p_3+2 = O(n^{2c+1})$. Then intersecting this automaton with the finalizer automaton gives the empty language, after which all unused variable automata can be intersected.

If there is no satisfying assignment, you must eventually intersect either a group of

two or three variable automata that falsify a clause, or the finalizer automaton with any automaton accepting $x_i(0^{q_1})^*$. In the former case, the resulting automaton must contain a loop of size $p_1p_2p_3 > M$ or $q_1q_2 > M$ to count the number of occurrences of the clause symbol c_i corresponding to the falsified clause. In the latter case, the resulting automaton must contain a loop of size $q_1q_2 > M$ to count ocurrences of 0.

In fact, we have proved that it is NP-hard to distinguish between instances having intermediate automaton size less than $O(n^{2c+1})$ and those with size greater than n^3, for a gap ratio of n^{c-1}. Our instances contain $2n+1$ automata of size $O(\sqrt{M})$ each for a total problem size of $N = O(n^{3c/2+1})$, so this problem is NP-hard to approximate within a factor of $N^{2/3-\varepsilon}$, where we can make ε as close to 0 as desired by increasing c. □

Before we can prove PSPACE-hardness, we need two lemmas.

Lemma 6.4.3. *Given an automaton accepting a language L, we can modify it to accept $\{w\text{␣}^* \mid w \in L\}$ by adding two states.*

Proof. Introduce a new symbol ␣ not already in the automaton's alphabet. Add a transition on ␣ from every accepting state to a single new accept state. From this new accept state, add a transition on ␣ to itself and a transition on all other symbols to a new rejecting state. Then add a transition on ␣ from all rejecting states to the new reject state (including itself). After these modifications, if the original automaton accepted a string w, the modified automaton accepts $w\text{␣}^*$. □

Lemma 6.4.4. *Given a minimal deterministic m-state automaton A accepting a language L over an alphabet Σ, there is a minimal deterministic automaton $A + \Gamma$ accepting $L + \Gamma = \{(x_1, g_1)(x_2, g_2) \cdots (x_n, g_n) \mid x_1x_2 \cdots x_n \in L, g_1g_2 \cdots g_n \in \Gamma^*\}$ over alphabet $\Sigma \times \Gamma$ with exactly m states.*

Proof. First, we can build an automaton $A + \Gamma$ accepting $L + \Gamma$ by expanding each transition $a \underset{x}{\to} b$ into transitions $a \underset{(x,g)}{\to} b$ for all $g \in \Gamma$. The number of states is unchanged, and the automaton remains deterministic.

Now we prove that $A + \Gamma$ is minimal. By the Myhill–Nerode Theorem [107], there are m equivalence classes $C_1, C_2, \ldots, C_m$ partitioning L where no two strings in a common class have a distinguishing extension, and any two strings from different classes have a distinguishing extension. We claim that the equivalence classes for $L + \Gamma$ are $C_1 + \Gamma, C_2 + \Gamma, \ldots, C_m + \Gamma$. Consider two strings $x' \in C_i + \Gamma$ and $y' \in C_j + \Gamma$ with a distinguishing extension z'; assume by symmetry that $x'z' \in L + \Gamma$ and $y'z' \notin L + \Gamma$. Let x, y, z be the corresponding strings over Σ from dropping the second coordinate of x', y', z' respectively. By the definition of $+$, the string $xz \in L$ and $yz \notin L$, which implies that $i \neq j$. □

Theorem 6.4.5. *Optimal Automaton Intersection is weakly PSPACE-hard and PSPACE-hard to approximate to within a factor of $2^{\Theta(\sqrt{n \log n})}$.*

Proof. We reduce from the problem of deciding whether the intersection of k automata $A_1, \ldots, A_k$ over a constant-size alphabet Σ is nonempty, which is known to be PSPACE-hard [92, Lemma 3.2.3]. We modify these automata using Lemma 6.4.3.

We then create m auxiliary unary automata $B_1, \ldots, B_m$ (m to be defined later) over $\Gamma = \{x\}$ where $L(B_i) = \{x^j \mid j \equiv 0 \pmod{p_i}\}$ where p_i is the ith prime. These automata have p_i states and their intersection has $M = \prod_i p_i$ states by repeated application of [116, Theorem 5]. We modify these automata using Lemma 6.4.3, so each automaton has $p_i + 2$ states and their intersection has $M + 2$ states.

Now create automata $A'_i = A_i + \Gamma$ and $B'_i = \Sigma + B_i$ by modifying all automata A_i, B_i to work over the alphabet $\Sigma \times \Gamma$, where the A'_i ignore the Γ part and the B'_i ignore the Σ part. The intersection of the B'_i is equivalent to $\Sigma +$ (the intersection of the B_i), which by Lemma 6.4.4 has at least M states like the intersection of the B_i. Our produced problem instance consists of the A'_i and B'_i.

We need M to be larger than the number of states in any intermediate intersection $\bigcap_{i \in \mathcal{I}} A'_i$, which is at most

$$\prod_i |A'_i| \leq \left(\max_i |A'_i|\right)^k \leq \left(2 + \max_i |A_i|\right)^k.$$

So we set $m > k \log(2 + \max_i |A_i|)$.

If $\bigcap_i L(A'_i) = \emptyset$, then the maximum number of states of any intersection in the linear intersection order $A_1 \cap \cdots \cap A_k \cap B_1 \cap \cdots \cap B_m$ will be less than M.

Otherwise, let w be the shortest word in $\bigcap_i L(A'_i)$. We claim that

$$\left[\bigcap_i L(A'_i) \cap \bigcap_i L(B'_i)\right] \text{ projected to second coordinate} = \{x^j \sqcup^l \mid j + l \geq |w|, j \equiv 0 \pmod{M}\}$$

which requires at least M states. For the $\subseteq$ direction, $j + l \geq |w|$ follows from $\bigcap_i L(A'_i)$ and $j \equiv 0 \pmod{M}$ follows from $\bigcap_i L(B'_i)$. For the $\supseteq$ direction, note that the string $(w_1, x) \ldots (w_{|w|}, x)(\sqcup, x)^j (\sqcup, \sqcup)^l \in \bigcap_i L(A'_i) \cap \bigcap_i L(B'_i)$ whenever $|w| + j \equiv 0 \pmod{M}$.

In fact, we have proved it is PSPACE-hard to distinguish between instances having intermediate automaton size at most $(2 + \max_i |A_i|)^k$ and instances having intermediate automaton size at least $(2 + \max_i |A_i|)^k + \prod_i p_i$. As we increase m, the problem size N grows as $\Theta(m^2 \log m)$ (the sum of the first m primes), but the gap between YES and NO instances grows as $2^{\Theta(m \log m)}$ (the product of the first m primes), so this problem is PSPACE-hard to approximate within a factor of $2^{\Theta(\sqrt{N \log N})}$. □

Chapter 7

The Witness[1]

7.1 Introduction

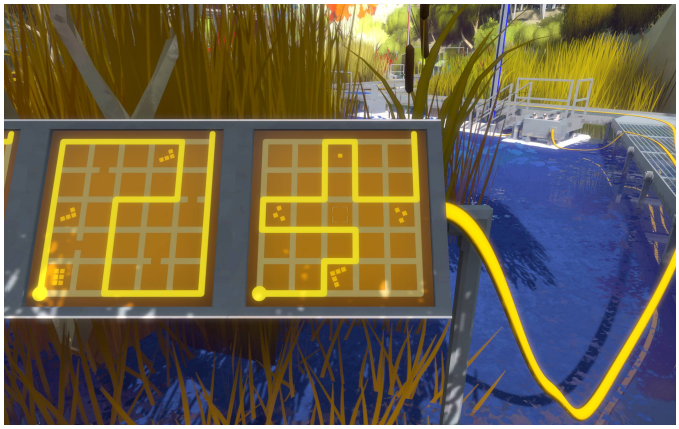


Figure 7-1: A screenshot from The Witness, featuring 2D puzzles in a 3D world.

The Witness[2] [145] is an acclaimed 2016 puzzle video game designed by Jonathan Blow (who originally became famous for designing the 2008 platform puzzle game Braid, which is undecidable [69]). The Witness is a first-person adventure game, but the main mechanic of the game is solving 2D puzzles presented on flat panels (sometimes CRT monitors) within the game; see Figure 7-1. The 2D puzzles are in a style similar to pencil-and-paper puzzles, such as Nikoli puzzles. Indeed, one clue type in The Witness (triangles) is very similar to the Nikoli puzzle *Slitherlink* (which is NP-complete [152]).

In this chapter, we perform a systematic study of the computational complexity of all single-panel puzzle types in The Witness, as well as some of the 3D "metapuzzles" embedded in the environment itself. Table 7.1 summarizes our single-panel results, which range from polynomial-time algorithms (as well as membership in L) to completeness in two complexity classes, NP (i.e., Σ_1) and the next level of the polynomial hierarchy, Σ_2. Table 7.3 summarizes our metapuzzle results, where PSPACE-completeness typically follows immediately.

[1]This chapter is from [2] (also available on arXiv [1]), which is joint work with Zachary Abel, Michael Coulombe, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, Mikhail Rudoy and Clemens Thielen.

[2]The Witness is a trademark owned by Jonathan Blow. Screenshots and game elements are included under Fair Use for the educational purposes of describing video games and illustrating theorems.

broken edge	hexagon	square	star	triangle	polyomino	antipolyomino	antibody	complexity	ref
✓								$\in$ L	Obs 7.3.2
✓	✓ vertices							NP-complete	Obs 7.3.3
	✓ vertices							*OPEN*	Prob 7.3.1
	✓ edges							NP-complete	Thm 7.3.5
✓	✓ on boundary							$\in$ P	Thm 7.3.7
		✓ 1 color						always YES	Obs 7.4.1
		✓ 2 colors						NP-complete	Thm 7.4.2 [90]
			✓ 1 color					*OPEN*	Prob 7.5.2
			✓ n colors					NP-complete	Thm 7.5.1
✓				✓ any				NP-complete	[152]
				✓ ▲				NP-complete	Thm 7.6.1
				✓ ▲▲				NP-complete	Thm 7.6.2
				✓ ▲▲▲				NP-complete	Thm 7.6.3
					✓ ■			$\in$ P	[90]
✓					✓ ■			$\in$ P	Thm 7.7.1
					✓ ■	✓ ▫		NP-complete	Thm 7.7.3
					✓ ▮			NP-complete	Thm 7.7.2
					✓ ▮			NP-complete	Thm 7.7.4
✓	✓	✓	✓	✓	✓	✓		$\in$ NP	Obs 7.8.2
✓	✓	✓	✓	✓			✓ n	$\in$ NP	Thm 7.8.3
					✓		✓ 2	Σ_2-complete	Thm 7.8.8
✓	✓	✓	✓	✓	✓		✓ 1	$\in$ NP	Thm 7.8.4
					✓	✓	✓ 1	Σ_2-complete	Thm 7.8.9
✓	✓	✓	✓	✓	✓	✓	✓ n	$\in \Sigma_2$	Thm 7.8.5

Table 7.1: Our results for one-panel puzzles in The Witness: computational complexity with various sets of allowed clue types (marked by ✓). Allowed polyomino clues are either arbitrary (✓), or restricted to be monominoes (✓■), vertical dominoes (✓▮), or rotatable dominoes (✓▮).

Witness puzzles. Single-panel puzzles in The Witness (which we refer to henceforth as *Witness puzzles*) consist of an $x \times y$ full rectangular grid;[3] one or more *start circles* (drawn as a large dot, ●); one or more *end caps* (drawn as half-edges leaving the rectangle boundary); and zero or more *clues* (detailed below) each drawn on a vertex, edge, or cell[4] of the rectangular grid. Figure 7-2 shows a small example and its solution. The goal of the puzzle is to find a simple path that starts at one of the start circles, ends at one of the end caps, and satisfies all the constraints imposed by the clues (again, detailed below). We generally focus on the case of a single start circle and single end cap, which makes our hardness proofs the most challenging.

We now describe the clue types and their corresponding constraints. Table 7.2

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

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

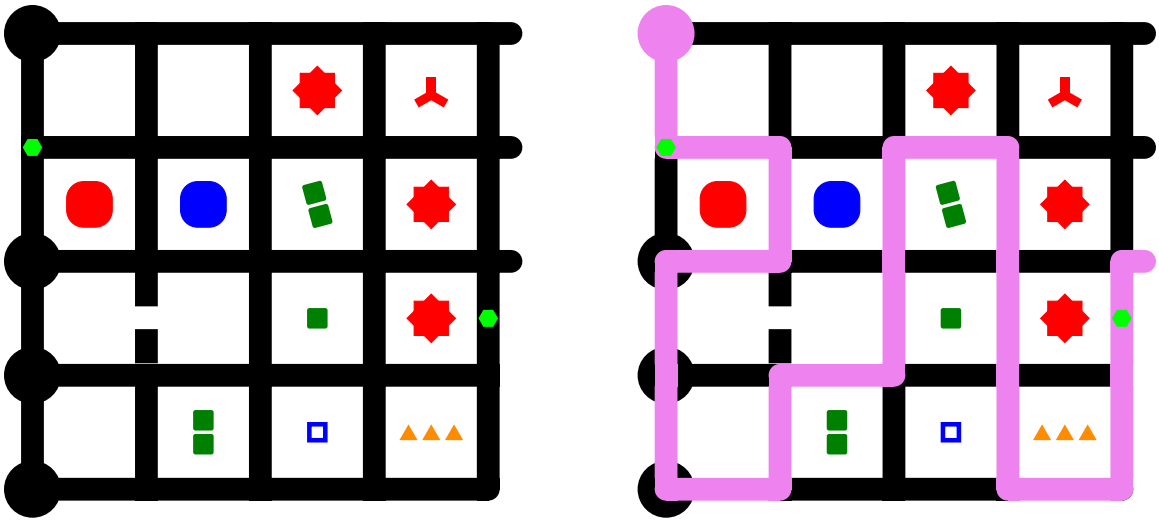


Figure 7-2: A small Witness puzzle featuring all clue types (left) and its solution (right). (Not from the actual video game.)

lists the clues by what they are drawn on — grid edge, vertex, or cell — which we refer to as "this" edge, vertex, or cell. While the last five clue types are drawn on a cell, their constraint applies to the *region* that contains that cell (referred to as "this region"), where we consider the regions of cells in the rectangle as decomposed by the (hypothetical) solution path and the rectangle boundary.

The solution path must satisfy *all* the constraints given by all the clues. (The meaning of this statement in the presence of antibodies is complicated; see Section 7.8.) Note, however, that if a region has no clue constraining it in a particular way, then it is free of any such constraints. For example, a region without polyomino or antipolyomino clues has no packing constraint.

As summarized in Table 7.1, we prove that most clue types *by themselves* are enough to obtain NP-hardness. The exceptions are broken edges, which alone just define a graph search problem; and vertex hexagons, which are related to Hamiltonian path in rectangular grid graphs as solved in [83] but remain open. But vertex hexagons are NP-hard when we also add broken edges. On the other hand, vertex and/or edge hexagons restricted to the boundary of the puzzle (even for nonrectangular boundaries) are polynomial; this result more generally solves "subset Hamiltonian path" (find a simple path visiting a specified subset of vertices and/or edges) when the subset is on the outside face of a planar graph. For squares, we determine that exactly two colors are needed for hardness. For stars, we do not know whether one or any constant number of colors are hard. For triangles, each single kind of triangle clue alone suffices for hardness, though the proofs differ substantially between kinds. For polyominoes, monominoes alone are easy to solve, but monominoes plus antimonominoes are hard, as are rotatable dominoes by themselves and vertical nonrotatable dominoes by themselves. All problems without antibodies or without (anti)polyominoes are in NP. Antibodies combined with (anti)polyominoes push the complexity up to Σ_2-completeness, but no further.

clue	drawn on	symbol	constraint
broken edge	edge		The solution path cannot include this edge.
hexagon	edge		The solution path must include this edge.
hexagon	vertex		The solution path must visit this vertex.
triangle	cell		There are three kinds of triangle clues (, , ,). For a clue with i triangles, the path must include exactly i of the four edges surrounding this cell.
square	cell		A square clue has a color. This region must not have any squares of a color different from this clue.
star	cell		A star clue has a color. This region must have exactly one other star, exactly one square, or exactly one antibody of the same color as this clue.
polyomino	cell		A polyomino clue has a specified polyomino shape, and is either nonrotatable (if drawn orthogonally, like) or rotatable by any multiple of 90° (if drawn at 15°, like). Assuming no antipolyominoes, this region must be perfectly packable by the polyomino clues within this region, each used exactly once.
antipolyomino	cell		Like polyomino clues, an antipolyomino clue has a specified polyomino shape and is either rotatable or not. For some $i \in \{0, 1\}$, each cell in this region must be coverable by exactly i layers, where polyominoes count as $+1$ layer and antipolyominoes count as -1 layer (and thus must overlap), with no positive or negative layers of coverage spilling outside this region, and each polyomino/antipolyomino clue used exactly once.
antibody	cell		Effectively "erases" itself and another clue in this region. This clue also must be necessary, meaning that the solution path should not otherwise satisfy all the other clues. See Section 7.8 for details.

Table 7.2: Witness puzzle clue types and the definitions of their constraints.

features	complexity	ref
sliding bridges	PSPACE-complete	Thm 7.10.1
elevators and ramps	PSPACE-complete	Thm 7.10.2
power cables and doors	PSPACE-complete	Thm 7.10.3
light bridges	OPEN	Prob 7.10.4

Table 7.3: Our results for metapuzzles in The Witness: computational complexity with various sets of environmental features.

Nonclue constraints. In Section 7.9, we consider how additional features in The Witness beyond clues can affect (the complexity of) Witness puzzles. Specifically, these features include visual obstruction caused by the surrounding 3D environment (which blocks some edges), symmetry puzzles (where inputting one path causes a symmetric copy to be drawn as well), and intersection puzzles (where multiple puzzles must be solved simultaneously by the same solution path).

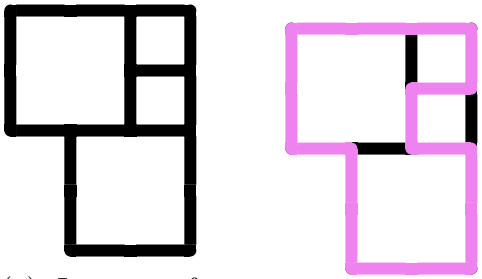
Metapuzzles. We also consider some of the *metapuzzles* formed by the 3D environment in The Witness, where the traversable geometry changes according to 2D single-panel puzzles. See Section 7.10 for details of these interaction models. Table 7.3 lists our metapuzzle results, which are all PSPACE-completeness proofs following the infrastructure of [10] (from FUN 2014).

Puzzle design. In Section 7.11, we consider designing 2D Witness puzzles using the variety of clues available. In particular, we introduce the *Witness puzzle design problem* where the goal is to find a puzzle whose solution is a specific path or set of paths. This problem naturally arises in metapuzzles, or if we were to design a Witness puzzle font (in the style of [45, 13]). Although many versions of this problem remain open, we present some basic universality results and limitations.

Open Problems. Finally, Section 7.12 collects together the main open problems from this chapter.

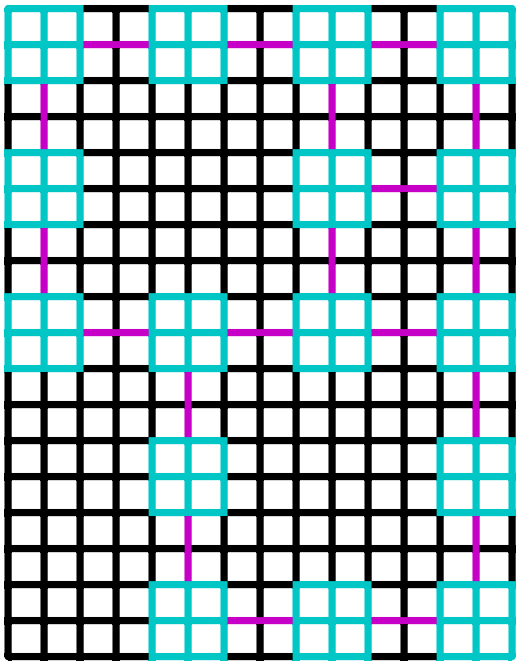
7.2 Hamiltonicity Reduction Framework

We introduce a framework for proving NP-hardness of Witness puzzles by reduction from Hamiltonian cycle in a grid graph G of maximum degree 3. Roughly speaking, we scale G by a constant scale factor s, and replace each vertex by a block called a chamber; refer to Figure 7-3. Precisely, for each vertex v of G at coordinates (x, y), we construct a $(2r+1)\times(2r+1)$ subgrid of vertices $\{sx-r, \ldots, sx+r\}\times\{sy-r, \ldots, sy+r\}$, and all induced edges between them, called a *chamber* C_v. This construction requires $2r < s$ for chambers not to overlap. For each edge $e = \{v, w\}$ of G, we construct a straight path in the grid from sv to sw, and define the *hallway* $H_{v,w}$ to be the subpath



(a) Instance of grid-graph Hamiltonicity

(b) A possible solution to (a)



(c) Corresponding chambers (turquoise) and hallways (pink)

Figure 7-3: An example of the Hamiltonicity framework with $r = 1$ and $s = 4$.

connecting the boundaries of v's and w's chambers, which consists of $s - 2r$ edges. Figure 7-3 illustrates this construction on a sample graph G.

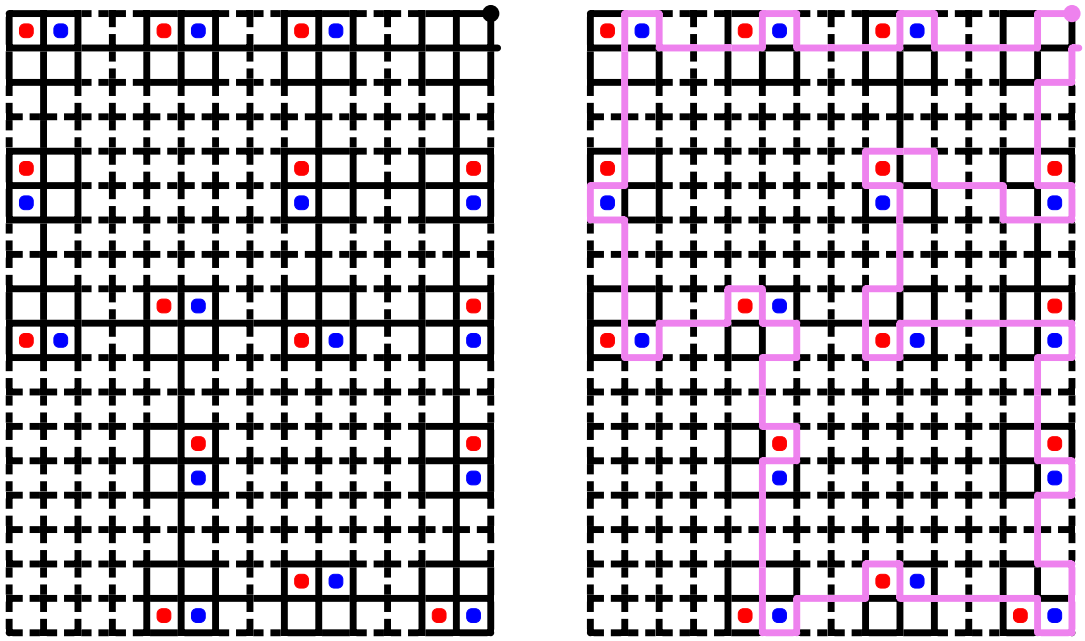
In each reduction, we define constraints to force the solution path to visit (some part of) each chamber at least once, to alternate between visiting chambers and traversing hallways that connect those chambers, and to traverse each hallway at most once. Because G has maximum degree 3, these constraints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

7.2.1 Simple Applications of the Hamiltonicity Framework

In this section, we briefly present some simple constructions based on this Hamiltonicity framework. All of these results are subsumed by stronger results presented formally in later sections, so we omit the details of these simpler NP-hardness proofs. Membership in NP for all puzzles except antibodies will be proved in Observation 7.8.2.

Corollary 7.2.1. *It is NP-complete to solve Witness puzzles containing broken edges and squares of two colors.*

Proof. We make hallways with broken edges and chambers with one square of each color. See Figure 7-4. □



(a) Unsolved (b) Solved

Figure 7-4: Example of the Hamiltonicity framework applied to Witness with squares and broken edges.

Corollary 7.2.2. *It is NP-complete to solve Witness puzzles containing broken edges and stars of arbitrarily many colors.*

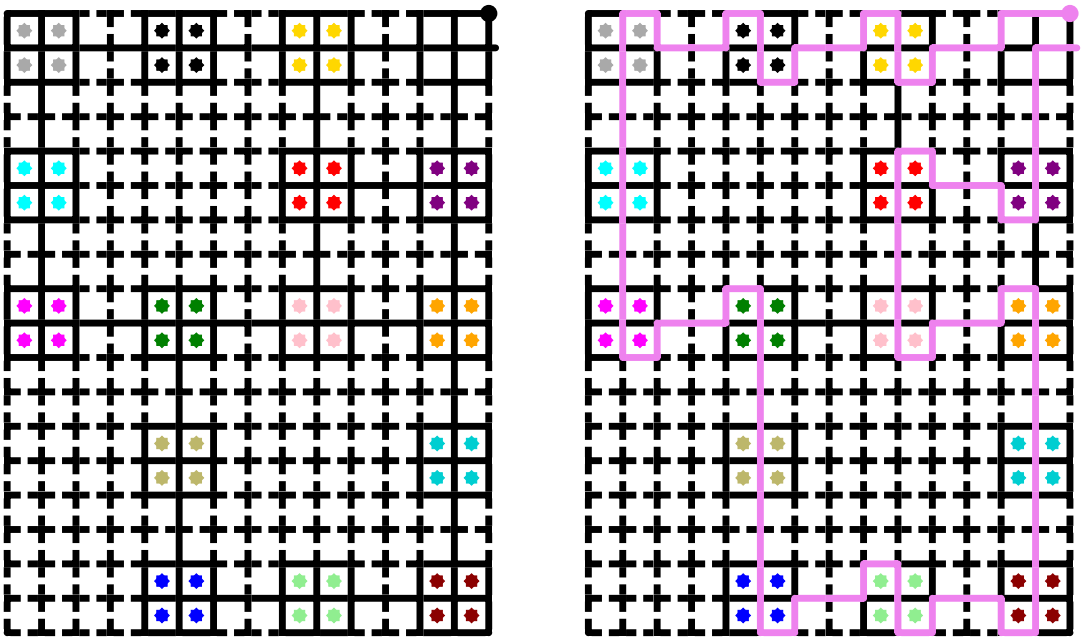
Proof. We make hallways with broken edges and chambers with four stars of one color (a different color for each chamber). See Figure 7-5. □

Corollary 7.2.3. *It is NP-complete to solve Witness puzzles containing broken edges and k-triangle clues, for any (single) $k \in \{1, 2, 3\}$.*

Proof. We make hallways with broken edges and chambers with a single k-triangle clue. See Figure 7-6. □

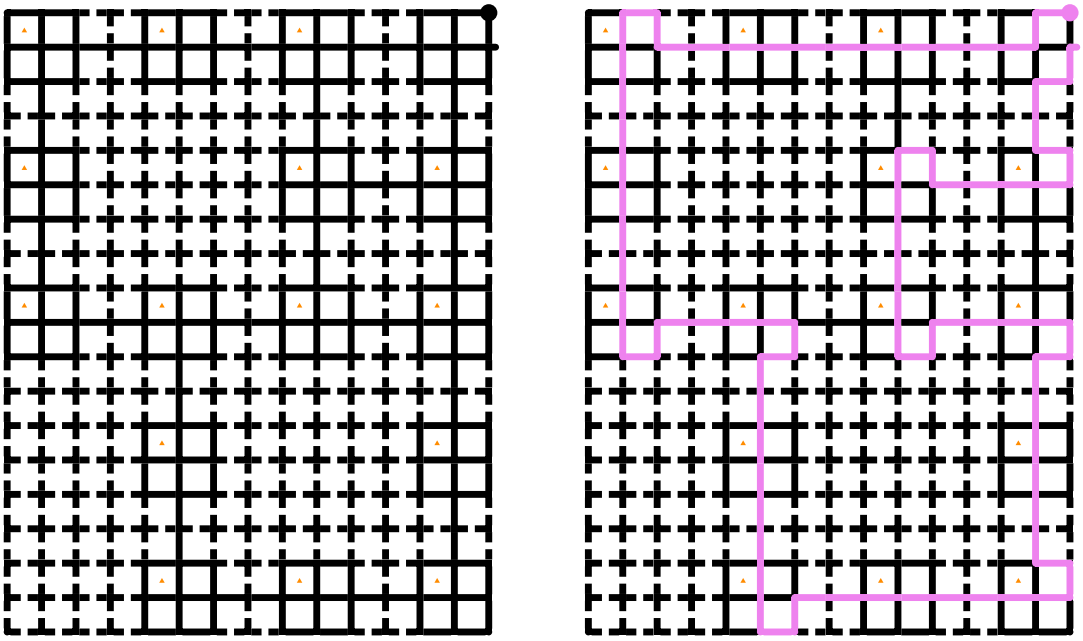
7.3 Hexagons and Broken Edges

Hexagons are placed on vertices or edges of the graph and require the path to pass through all of the hexagons. Broken edges are edges which cannot be included in the path. In this section, we show two positive results and two negative results. On the positive side, we show that puzzles with just broken edges are solvable in L (Section 7.3.1), and puzzles with hexagons just on the boundary of the puzzle (even when the boundary is not rectangle) and arbitrary broken edges are solvable in P (Section 7.3.4). On the negative side, we show that puzzles with just hexagons on vertices and broken edges are NP-complete (Section 7.3.2), and puzzles with just hexagons on edges (and no broken edges) are NP-complete (Section 7.3.3). We leave open the complexity of puzzles with just hexagons on vertices (and no broken edges):



(a) Unsolved (b) Solved

Figure 7-5: Example of the Hamiltonicity framework applied to Witness with stars and broken edges.



(a) Unsolved (b) Solved

Figure 7-6: Example of the Hamiltonicity framework applied to Witness with 1-triangles and broken edges.

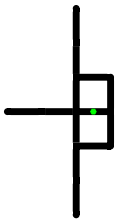


Figure 7-7: 2×3 vertex gadget.

Open Problem 7.3.1. *Is there a polynomial-time algorithm to solve Witness puzzles containing only hexagons on vertices?*

7.3.1 Just Broken Edges

Observation 7.3.2. *Witness puzzles containing only broken edges, multiple start circles, and multiple end caps are in L.*

Proof. We keep two pointers and a counter to track which pairs of starts and ends we have tried. For each start and end pair, we run an (s, t) path existence algorithm, which is in L. If any of these return YES, then the answer is YES. Thus, we have solved the problem with a quadratic number of calls to a log-space algorithm, a constant number of pointers, and a counter, all of which only require logarithmic space. □

7.3.2 Hexagons and Broken Edges

The following trivial result motivates Open Problem 7.3.1 (do we need broken edges?).

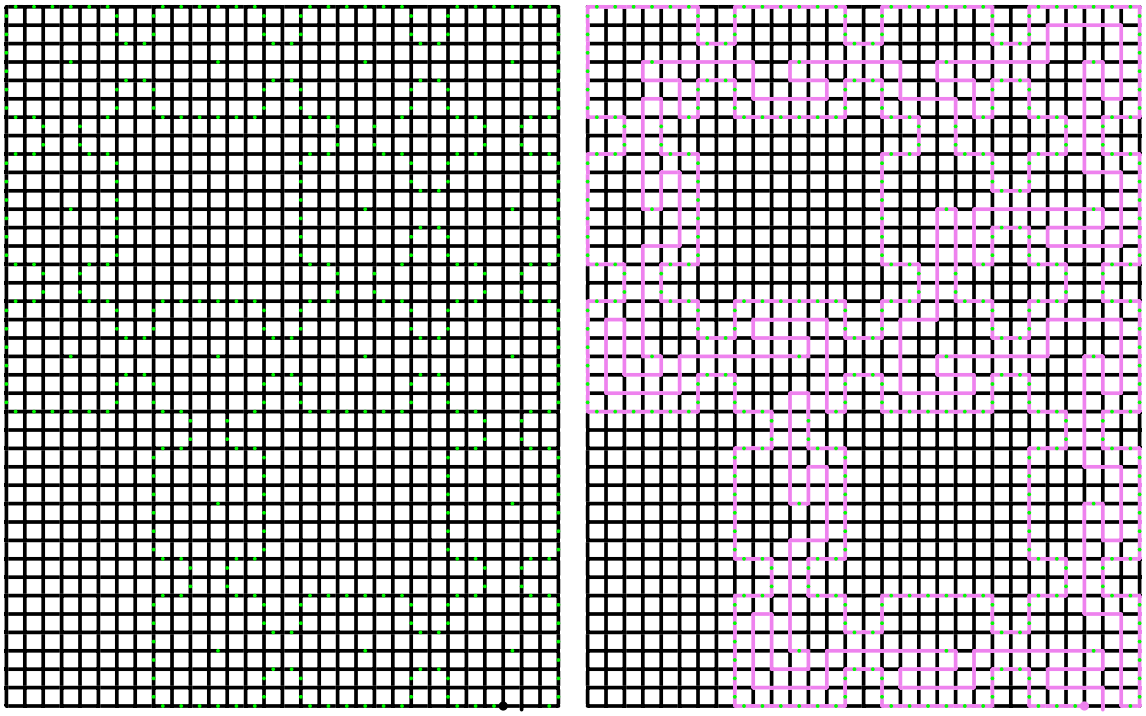
Observation 7.3.3. *It is NP-complete to solve Witness puzzles containing only broken edges and hexagons on vertices.*

Proof. Hamiltonian path in grid graphs [83] is a strict subproblem. □

For edge hexagons, we first present a simple application of the Hamiltonicity framework that uses broken edges; this result will be subsumed by Theorem 7.3.5 by a more involved application that avoids broken edges.

Theorem 7.3.4. *It is NP-complete to solve Witness puzzles containing only broken edges and hexagons on edges.*

Proof. Apply the Hamiltonicity framework with scale factor $s = 5$ and chamber radius $r = 1$. All edges within chambers and hallways are unbroken, and all other edges are broken, forcing hallways to be traversed at most once. Within each chamber, we place a hexagon on one of the edges incident to the center of the chamber. This hexagon forces the chamber to be visited, while having enough empty space to enable connections however desired. In fact, a smaller 2×3 chamber with a hexagon on the middle edge also suffices, as shown in Figure 7-7. □



(a) Instance corresponding to Figure 7-3a (b) Solution corresponding to Figure 7-3b

Figure 7-8: Example of the Hamiltonicity framework applied to Witness with edge hexagons.

7.3.3 Just Edge Hexagons

Theorem 7.3.5. *It is NP-complete to solve Witness puzzles containing only hexagons on edges (and no broken edges).*

Proof. Apply the Hamiltonicity framework with scale factor $s = 8$ and chamber radius $r = 2$; refer to Figure 7-8. As before, within each chamber, we place a hexagon on one of the edges incident to the center of the chamber. Consider the grid graph G formed by the chambers and hallways, and its complement grid graph $\bar{G}$ (induced by all grid points in $\mathbb{Z}^2 \setminus G$).

To constrain the solution path to remain mostly within G, we add hexagons as follows. For each connected component C of $\bar{G}$ (including the region exterior to G), we add a cycle of hexagons on the boundary edges of C. (These cycles outline the chambers and hallways, without intersecting them.) We break each cycle by removing one or two consecutive hexagons, leaving a path of hexagons called a *wall*. The removed hexagon(s) are adjacent to the *holey chamber* of the cycle: for a non-outer cycle, the holey chamber is the leftmost topmost chamber adjacent to the cycle, and for the outer cycle, the holey chamber is the rightmost bottommost chamber adjacent to the cycle. For each non-outer cycle, we remove one hexagon, from the horizontal edge immediately below the bottom-right cell of the holey chamber. For the outer cycle, we remove two hexagons, from the horizontal edges immediately below the two rightmost cells in the bottom row of the holey chamber; these two horizontal edges

share a vertex called the *gap vertex*. Thus, in all cases, the removed hexagon(s) of a cycle are below the bottom right of its holey chamber, so each chamber is the holey chamber of at most one cycle.

We place the start circle and end cap at the bottom of the diagram, with the start circle at the left endpoint of the outer wall, and the end cap below the gap vertex. (Thus, the rightmost bottommost chamber is the start/end chamber.)

Witness solution → Hamiltonian cycle. We claim that any solution to this Witness puzzle contains each wall as a contiguous subpath. Let $e_1, e_2, \ldots, e_k$ be the path of edges with hexagon clues forming a wall, and let $v_0, v_1, \ldots, v_k$ be the corresponding vertices on the path. When the solution path visits an edge e_i, where $1 \leq i < k$, the solution path must visit edge e_{i+1} immediately before or after; otherwise, e_{i+1} could not be visited at another time on the solution path (contradicting its hexagon constraint) because its endpoint v_i has already been visited. By induction, any solution path visits the wall edges consecutively, as either $e_1, e_2, \ldots, e_k$ or $e_k, e_{k-1}, \ldots, e_1$.

Next we claim that any solution path must enter and exit each non-outer wall from its corresponding holey chamber. The cycle corresponding to the wall is the boundary of a connected component C of $\bar{G}$. The wall contains all vertices of the boundary of C, so the only way for the solution path to enter or exit the interior of C is by entering or exiting the wall. But the start circle and end cap are not interior to C or on the wall, so the solution path must enter and exit the wall from the exterior of C. The only such neighbors of the wall endpoints are in the holey chamber.

By the previous two claims, any solution to this Witness puzzle cannot go strictly inside any connected component of $\bar{G}$, except the outer component, and traversing a wall starts and ends in the same chamber. Hence, a solution can traverse from chamber to chamber only via hallways, and it must visit every chamber to visit the hexagon in the middle. The solution effectively begins and ends at the rightmost bottommost chamber, the gap vertex being the only way in from the outside. Therefore, any solution can be converted into a Hamiltonian cycle in the original grid graph.

Hamiltonian cycle → Witness solution. To convert any Hamiltonian cycle into a Witness solution, first we route the Hamiltonian cycle within the chambers and hallways so that, in every chamber, the routed cycle visits the hexagon in the middle of the chamber as well as the bottom edge of the bottom-right cell of the chamber. The routing along each hallway is uniquely defined. To route within a chamber, we connect one visited hallway along a straight line to the central vertex of the chamber; then traverse the incident edge with the hexagon unless we just did; then walk clockwise or counterclockwise around the remainder of the chamber in order to visit the bottom edge of the bottom-right cell of the chamber before reaching the other visited hallway.

Next we modify this routed cycle into a Witness solution path by including the walls. For each wall, within the corresponding holey chamber, we replace the bottom edge e of the bottom-right cell of the chamber by the two vertical edges below e. For non-outer walls, the two endpoints of the wall attach to these two vertical edges, effectively replacing the edge e with the wall path. Thus, before we modify the holey

chamber of the outer wall, we still have a cycle. For the outer wall, the right endpoint of the wall attaches to the right vertical edge, while the left endpoint of the wall is the start circle; the left vertical edge attaches to the gap vertex, which leads to the end cap. Thus, we obtain a path from the start circle to the end cap. □

7.3.4 Boundary Hexagons and Broken Edges

In this section, we solve Witness puzzles with arbitrary broken edges and hexagons just on boundary of the puzzle. To make this result more interesting, we allow a generalized type of Witness puzzle (also present in the real game) where the board consists of an arbitrary simply connected set C of cells (instead of just a rectangle), and hexagons can be placed on any vertices and/or edges on the outline of C.

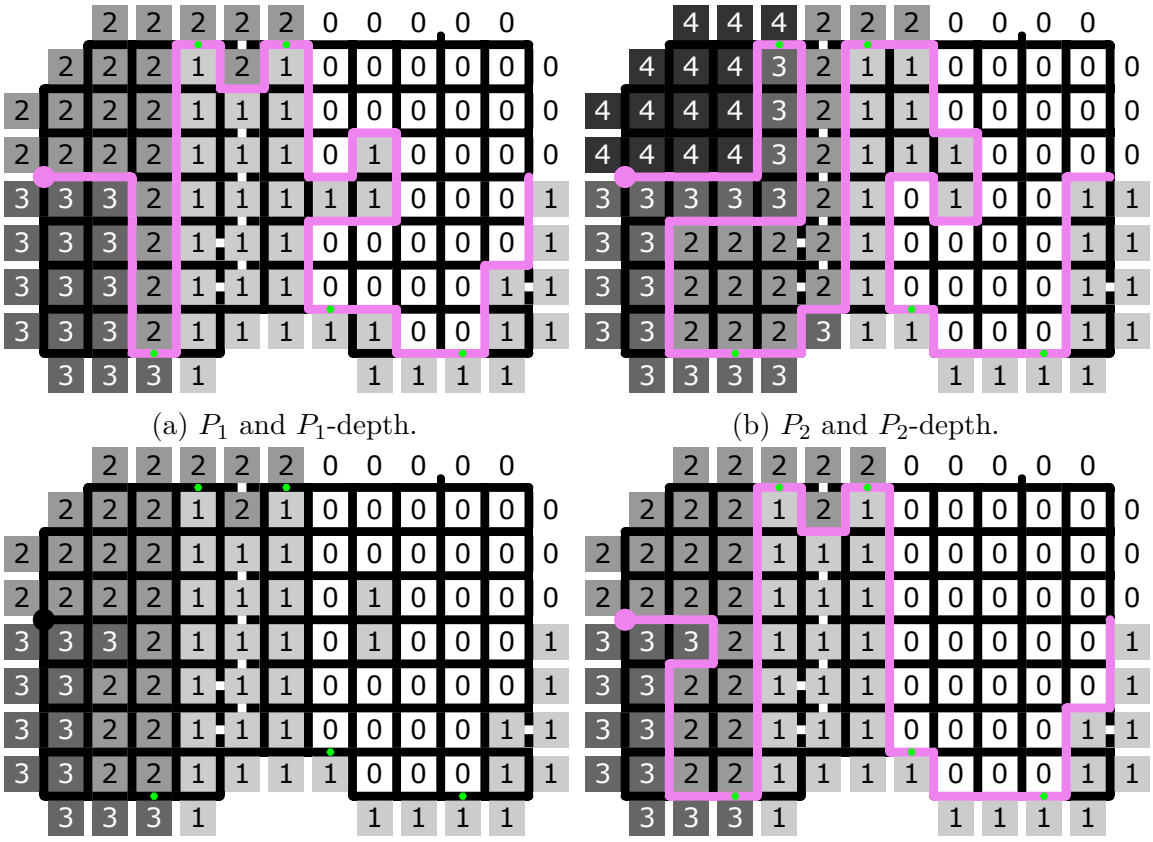
This result is essentially a polynomial-time algorithm for solving *subset Hamiltonian path* — find a simple path visiting a specified subset of vertices and/or edges — on planar graphs when the subset lies entirely on the outside face of the graph. This problem is a natural variation of *subset TSP* — find a minimum-length not-necessarily-simple cycle visiting a specified subset of vertices. A related result is that *Steiner tree* — find a minimum-length tree visiting a specified subset of vertices — can be solved in polynomial time on planar graphs when the subset lies entirely on the outside face of the graph [57]. This result is a key step in the first PTAS for Steiner tree in planar graphs [29]. It seems that simple paths are trickier to find than trees, so our algorithm is substantially more complicated. Hopefully, our result will also find other applications in planar graph algorithms.

Define a *forced division* (C, F) to consist of a simply connected set C of empty cells (possibly with broken edges between them) together with a set F of *forced* vertices and edges (i.e., vertices and/or edges with hexagon clues) on the outline of C. A (C, F)*-path* is a simple path using only vertices and edges incident to cells in C that traverses all vertices and edges in F.

To enable a dynamic program for finding (C, F)-paths when they exist, we prove a strong structural result about (C, F)-paths that leave the "most room" for future paths, which may be of independent interest.

For any two vertices u and v on the outline of C, any (C, F)-path P from u to v decomposes the cells of C into one or more connected components. For each vertex t on the outline of C, if P does not visit t, then there is a unique connected component $R_t(P)$ incident to t. We call $R_t(P)$ the t*-remainder* of P. The intent is for the path to continue on from v into the t-remainder, so we are interested in the case where v is incident to $R_t(P)$, which is equivalent to preventing the path from going "backward" along the outline of C between v and t.

Lemma 7.3.6. *Given a forced division (C, F) with possibly broken edges, for distinct vertices u, v, t appearing in clockwise order on the outline of C, if there is any (C, F)-path from u to v that does not visit the outline of C in the clockwise interval $(v, t]$, then there exists a unique* maximum t-remainder R^* *over all such paths. More precisely, the t-remainder of any such path is a subset of R^*, and there is such a path with t-remainder exactly R^*. The same result holds for any forbidden interval (v, x) of the*



(a) P_1 and P_1-depth.

(b) P_2 and P_2-depth.

(c) Depth (the minimum of P_1-depth and P_2-depth). Note the 1-cavity inside the depth-0 area.

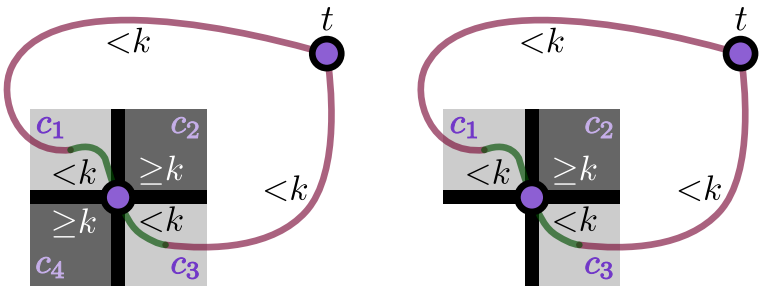
(d) P_3. After filling the cavity, the ledges form a path from u to v.

Figure 7-9: Example remainder computation, where F is represented by hexagons, u is the start circle, v is the other endpoint of the paths, and t is the end cap. Maximizing the remainder leaves the maximum freedom for completing the path. (Note that the two edges in each reflex corner have two exterior depths; they happen to be the same in this example, but they may differ in general.)

outline of C that contains $(v, t]$.

Proof. Let P_1 and P_2 be any two (C, F)-paths from u to v that do not visit the outline of C in the clockwise interval $(v, t]$ (or any larger forbidden interval). We will construct another such path P_3 (built entirely from edges of P_1 and P_2) whose t-remainder is a superset of the t-remainders of P_1 and P_2. Refer to Figure 7-9 for an example. By applying this construction repeatedly to all such paths, we obtain a single such path whose t-remainder is a superset of the t-remainder of all such paths.

Define the *P_i-depth* of a cell $c \in C$ to be the minimum possible number of crossings of P_i by a curve within C from a point interior to c to t. (This number is most easily seen to be well-defined by allowing the curve to cross cell boundaries only at edges, not vertices.) Define the *depth* of a cell $c \in C$ to be the minimum of the P_1-depth and P_2-depth of c.



(a) Alternating filled depths $< k, \geq k, < k, \geq k$.

(b) Alternating filled depths $< k, \geq k, < k$ on the boundary.

Figure 7-10: Two impossible situations in the proof of Lemma 7.3.6: alternating filled depths.

Define a *k-cavity* to be a connected set of cells of depth $\geq k$ (with at least one cell of depth exactly k) "surrounded" by cells of depth $< k$: every edge on the outline of the cavity must have an incident exterior cell that has depth $< k$. The outline of a cavity therefore shares no edges with the outline of C. (On the other hand, this definition allows the outline of a cavity to share a vertex with the outline of C, though we will argue later that this is impossible.) Furthermore, by this definition, every cavity is a *maximal* connected set of cells of depth $\geq k$.

Define the *filled depth* of cells $c \in C$ to be the result of the following process. Start with filled depth equal to depth. If there is a k-cavity with the current notion of (filled) depth, set the filled depth of all cells in the cavity to $k - 1$. Repeat until there are no more cavities.

Next we claim that we never have cells c_1, c_2, c_3, c_4 in clockwise order around a vertex x with filled depths of $< k, \geq k, < k, \geq k$ respectively; refer to Figure 7-10a.[5] First, neither c_1 nor c_3 belongs to a filled cavity, or else c_2 and c_4 would belong to the same cavity, so filling this cavity would place c_2 and c_4 at the same depth as either c_1 or c_3. Thus, c_1 and c_3 have depths equal to filled depths which are $< k$. By definition of depth, for $i \in \{1, 3\}$, we can draw a simple curve from c_i to t that remains in depth $< k$. Because filling only decreases depths, these simple curves also remain in filled depth $< k$. Connecting these two curves at x, we obtain a simple closed curve through x and t that traverses cells only of filled depth $< k$. By planarity, this closed curve must contain either c_2 or c_4 of filled depth $\geq k$, so there must in fact be a k-cavity that is not yet filled, a contradiction, proving the claim.

Similarly, we claim that we never have a vertex x on the outline of C with incident cells c_1, c_2, c_3 in clockwise order interior to C with filled depths of $< k, \geq k, < k$ respectively; refer to Figure 7-10b. Otherwise, as above, for $i \in \{1, 3\}$, c_i was not filled, and we can draw a simple curve from c_i to t that remains in depth $< k$; connect these curves into a simple closed curve through x and t that traverses cells only of

[5] For simplicity, we assume here that every vertex has degree at most 4, as in The Witness, though this argument can easily be generalized to arbitrary planar graphs: replace each c_i with an interval of cells around x, so that c_i is indeed adjacent to c_{i+1}, and generalize to ≥ 4 alternations.

filled depth $< k$; and by planarity this cycle must contain c_2 of filled depth k, forming an unfilled k-cavity and a contradiction.

As a consequence of the previous claim, no k-cavity can touch a vertex x of the outline of C. Furthermore, the previous claim holds for P_i-depths as well as filled depths: by the same argument, we get a simple closed curve (through x and t) that traverses cells only of P_i-depth $< k$, yet containing a cell of P_i-depth $\geq k$, contradicting that P_i is a simple path.

For an edge e on the outline of C, define the *interior depth* of e to be the depth of the incident cell of C; the *exterior P_i-depth* of e to be the minimum possible number of crossings of P_i by a curve that starts just outside e, immediately crosses e, and continues within C to t; and the *exterior depth* of e to be the minimum of the exterior P_1-depth and exterior P_2-depth of e. Note that the exterior depth of e is always at least the interior depth of e. (These notions do not need to distinguish between depth and filled depth, because we never fill a cell incident to an edge on the outline of C.)

We claim that the exterior P_1-depth and exterior P_2-depth of e have the same parity. By the definition of exterior P_i-depth, we can draw a simple curve starting at a point q just outside C, immediately crossing e, and continuing within C to t, crossing P_i that many times. We can close these curves without adding P_i crossings by adding a simple curve exterior to C from q to t. The resulting simple closed curves for both P_1 and P_2 enclose the same interval of the outline of C, so u is either inside or outside of both closed curves, and similarly for v. If u and v are on the same side of the closed curves, then P_i from u to v must cross the closed curve an even number of times, so e has even exterior P_i-depth; while if u and v are on different sides, e has odd exterior P_i-depth. In either case, the parities are the same for both i.

We claim that two consecutive edges e_1, e_2 of the outline of C have the opposite exterior depth parity if and only if their common endpoint is u or v. Consider the curve proceeding around the common endpoint x from just outside e_1, immediately crossing e_1, crossing all cells incident to x, and crossing e_2 to just outside e_2. This curve crosses P_i zero or two times unless x is one of P_i's endpoints, in which case it crosses exactly once. Thus, the exterior P_i-depths of e_1 and e_2 have the same parity unless x is u or v, in which case they have opposite parity. The exterior depths of e_1 and e_2 therefore satisfy the same parity relationship.

Define *ledges* as follows: an edge e on the outline of C is a ledge if it has different interior and exterior depths, and an edge e interior to C is a ledge if its two incident cells have different filled depths. Filled depth changes exactly at ledges, by ± 1, so in particular every ledge is an edge of P_1 or P_2 or both.

We claim that, at every vertex of C except u or v, the number of incident ledges is 0 or 2; and at u and v, the number of incident ledges is 1. By the $< k, \geq k, < k, \geq k$ and $< k, \geq k, < k$ claims, every vertex has at most two incident ledges. For a vertex interior to C, we must therefore have zero or two incident ledges: there cannot be just one change in a cycle of numbers. A vertex x on the outline of C might have zero, one, or two incident ledges. Let e_1 and e_2 be the two (consecutive) edges of the outline of C incident to x. As in the previous claim, consider the curve proceeding around x from just outside e_1, immediately crossing e_1, crossing all cells incident to x, and crossing e_2 to just outside e_2. As we traverse this curve, the filled/exterior depth

changes by ± 1 at ledge edges, and at no other edges (by definition of ledge). Thus, x has exactly one incident ledge if and only if the exterior depths of e_1 and e_2 have opposite parity, which by the previous claim must happen exactly when $x \in \{u, v\}$; otherwise, x has zero or two incident ledges as desired.

By the previous claim, the set of ledges forms a path P_3 between u and v plus zero or more disjoint simple cycles. We claim that, in fact, there can be no cycles of ledges. Suppose for contradiction that there were such a cycle X. There are two cases:

1. If X touches the outline of C at just one vertex (which cannot be t by its degree bound) or not at all, then X is surrounded by cells of a constant filled depth k (because filled depth changes exactly at ledges), and the filled depths of cells interior and incident to X must have filled depth either $< k$ or $> k$. In fact, no cells interior to X can have filled depth $< k$, as they would have a curve to t (which is exterior to X) that only visits cells with filled depth $< k$, contradicting the filled depth of the cells surrounding X. But if all cells interior to X have filled depth $> k$, then X is the outline of a $> k$-cavity, contradicting that we already filled all cavities.

2. If X touches the outline of C at two or more vertices then, because X is disjoint from P_3, there is a subpath Y of X whose endpoints y_1, y_2 lie on the outline of C separating the rest of X from P_3 and thus u and v. Refer to Figure 7-11. Label y_1 and y_2 to be closer to u and v respectively on the outline of C, i.e., so that their clockwise order is either y_1, u, v, y_2 or u, y_1, y_2, v. Because P_1 and P_2 and thus all ledges are not on the outline of C in the clockwise interval $(v, t]$, t is on the same side of Y as v. Therefore Y is a ledge transition from some filled depth k on the u, v, t side to filled depth $k+1$ on the X side (locally on either side of Y). Indeed, the entire X side of Y has filled/exterior depth $> k$; otherwise, there would be a curve to t (and thus crossing Y) of filled depths $\leq k$. Therefore, the other (nonempty) path of X, $X \setminus Y$, must be a ledge transition from filled depth $k+1$ to $k+2$, so there must be a filled/exterior depth of $k+2$ on the X side of Y.

   Now consider the transition from filled depth k to $k+1$ on the edge e of Y incident to y_1, and let c_k and c_{k+1} be the cells of C incident to e of filled depths k and $k+1$ respectively (which are exterior and interior respectively to X). In fact, c_k and c_{k+1} have depth k and $k+1$ respectively: cell c_k could not have been filled as it has a neighbor c_{k+1} of higher depth, and cell c_{k+1} could not have been filled because that would have made its filled depth match c_k's. By definition of depth, for some $i \in \{1, 2\}$, the P_i-depth of c_k is k while the P_i-depth of c_{k+1} is $k+1$. To achieve this transition, P_i must have a subpath Q_i between y_1 and some vertex z on the outline of C, with e as its first or last edge, where one side of Q_i locally has P_i-depth k while the other side of Q_i locally has P_i-depth $k+1$. Because the entire X side of Y has filled/exterior depth $> k$, Q_i cannot strictly enter the X side of Y, so Q_i must be nonstrictly on the t side of Y. Viewed from the other side, X must be nonstrictly on the $k+1$ side of Q_i, while t must be strictly on the k side of Q_i. By our labeling of y_1 and y_2, z lies on the interval

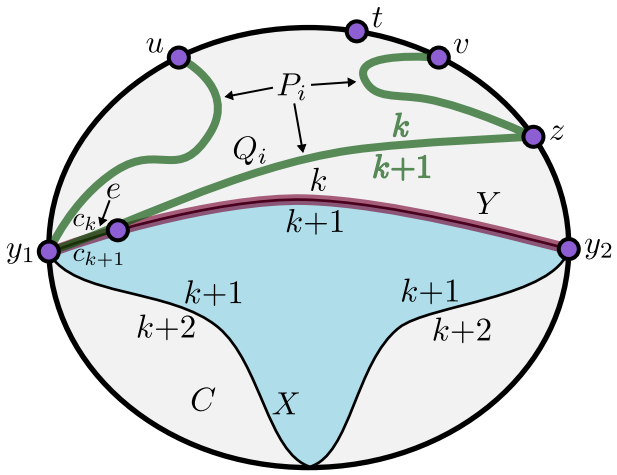


Figure 7-11: Case 2 of the proof of Lemma 7.3.6.

of the outline of C between t and y_2 not containing y_1 (or u), so u and v lie on the same (k) side of Q_i as t. Now beyond the endpoints y_1, z of Q_i, the simple path P_i must proceed on the k side of Q_i which contains u and v, so P_i never goes strictly on the $k+1$ side of Q_i, which is nonstrictly on the t side of Y. But this contradicts that there must be a transition from P_i-depth $k+1$ to $k+2$ on $X \setminus Y$, which is on the X side of Y.

Therefore the set of ledges forms exactly a path P_3 between u and v.

We claim that P_3 visits all forced vertices and edges in F. First, P_3 visits every forced edge because such an edge is in both P_1 and P_2 and thus is a ledge via differing internal and external depths, which are unaffected by cavity filling. Second, we claim that P_3 visits every vertex x on the outline of C that is visited by both P_1 and P_2, and thus every forced vertex. Specifically, we show that x is incident to two different depths (cell or exterior), which implies (because cavities cannot touch a vertex x on the outline of C) that x is incident to two different filled depths, and thus incident to a ledge. Suppose for contradiction that every depth incident to x is equal to the same k, and thus every incident P_i-depth is $\geq k$ and every cell or edge exterior has either P_1-depth or P_2-depth equal to k; refer to Figure 7-12. Because the P_i-depth changes by ± 1 across each edge of P_i, x must be incident to a P_i-depth of $> k$ as well. Let e_1 and e_2 be the two edges of the outline of C incident to x in counterclockwise order around the outline of C, so that the edges incident to x proceed clockwise from e_1 to e_2. Assume by symmetry that the exterior P_1-depth of e_1 is k. By the $< k, \geq k, < k$ claim applied to P_1-depth at x, the exterior P_1-depth of e_2 cannot be k; otherwise, there would be a P_1-depth of $> k$ in between e_1 and e_2 of exterior P_1-depths k. Thus, the P_1-depths clockwise around x must proceed $k, k+1, k+2$. The exterior depth of e_2 is k, while the exterior P_1-depth of e_2 is $k+2$, so the exterior P_2-depth of e_2 must be k. By a symmetric argument, the P_2-depths clockwise around x must proceed $k+2, k+1, k$. Because there are no incident depths $> k$, the transition of P_2-depth from $k+2$ to k must occur fully before (counterclockwise of) any of the transition of P_1-depth from k to $k+2$. Because of the transition from P_i-depth k to P_i-depth

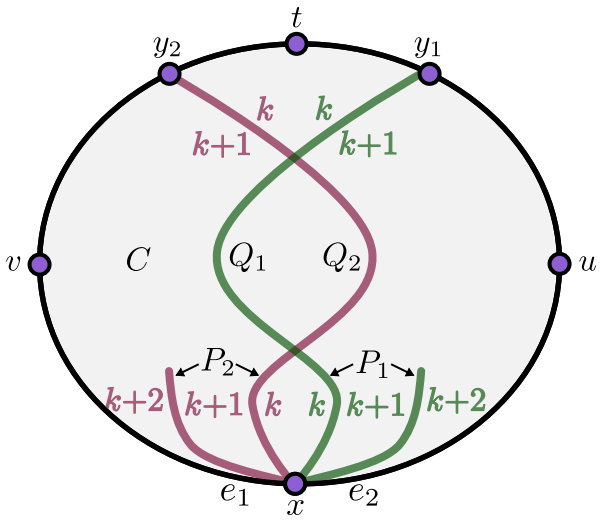


Figure 7-12: An impossible situation in the proof of Lemma 7.3.6: vertex x visited by both paths P_1 and P_2 incident to only a single depth k.

$k+1$, each P_i has a subpath Q_i between x and a point y_i on the outline of C, without visiting any vertices on the outline of C in between, where Q_i has P_i-depth k locally on one side and P_i-depth $k+1$ locally on the other side. Because of the transition from P_i-depth $k+1$ to P_i-depth $k+2$, locally at x, P_i proceeds strictly to the $k+1$ side of Q_i, and by planarity and simplicity of P_i, one endpoint of P_i (u or v) must be strictly on the $k+1$ side of P_i. Because u, v, t appear in clockwise order on the outline of C, u must be strictly on the $k+1$ side of Q_1 while v must be strictly on the $k+1$ side of Q_2. Because t is on the k side of both Q_1 and Q_2, we must have t, y_1, u, x, v, y_2, t appearing in clockwise order on the outline of C. Because P_2 does not visit the clockwise interval $(v, t]$, we must have $v = y_2$. But then P_2 consists of Q_2 preceded by a path strictly on the $k+1$ side of Q_2, so P_2 cannot reach a vertex on the outline of C on the $k+1$ side of Q_1, so in particular cannot reach u, a contradiction.

Now that P_3 is a path from u to v that visits all forced vertices and edges, we just need to check a few more properties. The remainder of P_3 contains all cells of filled depth 0 because such a cell can be connected to t via a curve that does not cross any ledges; therefore, the remainder of P_3 includes all cells of depth 0, and thus all cells of P_i-depth 0, which is the remainder of P_i. Every edge in path P_3 is a ledge and thus an edge of P_1 or P_2, so P_3 uses no broken edges and shares the property of not visiting the outline of C in the clockwise interval $(v, t]$ (or any larger forbidden interval). Therefore, P_3 is the path we were searching for. □

Now we give our dynamic programming algorithm for generalized Witness puzzles with boundary hexagons and broken edges.

Theorem 7.3.7. *Given a forced division (C, F) with possibly broken edges, for any two vertices s and t on the outline of C, we can decide in polynomial time whether there exists a (C, F)-path from s to t.*

Proof. We use dynamic programming to solve this problem. For every clockwise

interval $[a, b]$ of the outline of C containing s and not strictly containing t (i.e., $t \notin (a, b)$), we define two subproblems based on Lemma 7.3.6:

- find a maximum-remainder $(C, F \cap [a, b])$-path from s to b that does not visit the outline of C in the clockwise interval (b, a), or report that no such path exists; and
- find a maximum-remainder $(C, F \cap [a, b])$-path from s to a that does not visit the outline of C in the clockwise interval (b, a), or report that no such path exists.

To solve the original problem, we apply either subproblem on the clockwise interval $[t, t]$ (understood to mean the entire outline of C, starting and ending at t). In this case, $a = b = t$ and the clockwise interval $(b, a) = \emptyset$, and the remainder is undefined, so the subproblem statement is exactly to find a (C, F)-path from s to t.

To find a maximum-remainder $(C, F \cap [a, b])$-path P^* from s to b not visiting (b, a), if such a path exists, we guess (try all options for) the last vertex x before b of the outline of C visited by the path P^*. Such a vertex $x \in [a, b)$ exists because s is a candidate for x, except in the base case $s = b$, where we simply return the empty path from s to s. Vertex x splits P^* into a path P_1^* from s to x followed by a path P_2^* from x to b. The latter path P_2^* visits the outline of C only at its endpoints, except when P_2^* is a single edge $(b - 1, b)$ where $b - 1$ denotes the clockwise previous vertex before b on the outline of C. We divide into two overlapping cases based on the relation between $x \in [a, b)$ and s:

1. If x is in the clockwise interval $[s, b)$, then there must not be any forced vertices/edges in the clockwise interval (x, b), except possibly a single forced edge $(b - 1, b)$ when $x = b - 1$. Otherwise, P_2^* could not include any such forced vertices/edges (as it cannot visit the outline of C in that interval) and P_2^* effectively "hides" such forced vertices/edges from P_1^* (as s is outside the cycle formed by P_2^* and $[x, b]$). Thus, P_1^* can visit forced vertices/edges only in the interval $I = [a, x]$.

2. If x is in the clockwise interval $[a, s]$, then there must not be any forced vertices/edges in the clockwise interval (a, x). Otherwise, P_2^* cannot include any such forced vertices/edges (as it cannot visit the outline of C in that interval) and P_2^* effectively "hides" such forced vertices/edges from P_1^* (as s is outside the cycle formed by P_2^* and $[b, x]$). Furthermore, the edge $(b - 1, b)$ cannot be a forced edge unless $x = b - 1 = s$: it could not be visited by P_2^* (as it includes no edges of the outline of C) nor by P_1^* (or else P^* would be a cycle, not a simple path). Thus, P_1^* can visit forced edges only in the interval $I = [x, b - 1]$.

3. If $x = s$, then the constraints of both above cases must hold: there must not be any forced vertices/edges in the clockwise interval (a, b), except possibly a single forced vertex s and a single forced edge $(b - 1, b)$ when $x = b - 1 = s$. Furthermore, P_1^* must be the trivial path from s to s, so it can visit forced vertices/edges only in the trivial interval $I = [s, s]$.

If any of the stated constraints do not hold, then this choice of x fails.

Otherwise, we recursively find a maximum-remainder $(C, F \cap I)$ path P_1 from s to x that does not visit the outline of C outside the interval I defined above in each case. Note that I is strictly contained in $[a, b]$ in all cases, so the recursive calls cannot form a cycle. By a cut-and-paste argument, we can assume $P_1 = P_1^*$, i.e., that the path P_1 we find matches the prefix P_1^* of the desired path P^*. Otherwise, modify P^* by replacing P_1^* with P_1, resulting in an equally suitable maximum-remainder $(C, F \cap [a, b])$-path P^* that does not visit (b, a). First, the remainder of P^* is the remainder of P_2^*, so remains unchanged. Second, P_2^* lies in the remainder of P_1^*, which is contained within the (maximum) remainder of P_1, so P^* remains non-self-crossing.

Next we depth-first search using the left-hand rule to find a maximum-remainder path P_2 from x to b while avoiding all broken edges, all vertices on the outline of C, and all vertices of P_1. If the depth-first search fails to find such a path, then this choice of x fails. Otherwise, we claim that $P_2 = P_2^*$. At each step, the depth-first search makes the leftmost choice that can be completed into a path. Hence, if P_2^* deviates from P_2 at any step, P_2^* must be to the right of P_2 at that step. But that deviation is incident to a cell in P's remainder that is not in P^*'s remainder (because P_2 does not visit the outline of C until b, and the interval (b, a) has not been visited by P_1 or P_2), contradicting that P^* is maximum-remainder. Here we crucially use Lemma 7.3.6 that there is a unique maximum remainder according to the subset relation, not two possible remainders that are mutually incomparable.

Finally, we concatenate P_1 and P_2 to form a path P, which is a candidate for P^* that (as argued above) equals P^* assuming our guess for x was correct. By trying all choices for x, and returning the resulting path P that has the maximum remainder, we are sure to find P^* if it exists.

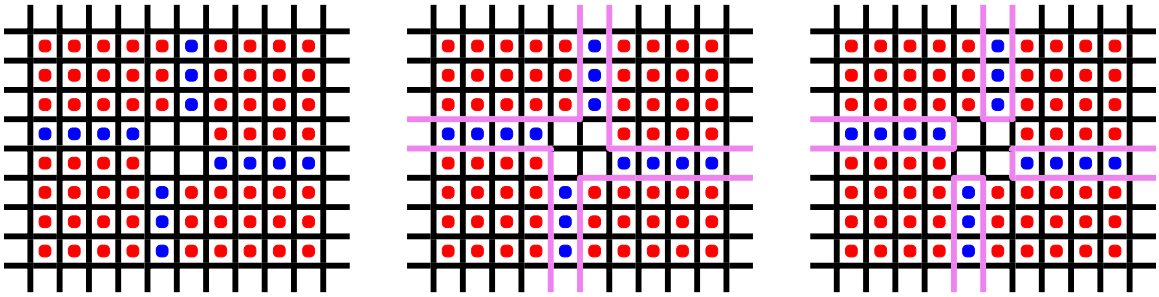
The other type of subproblem, to find a maximum-remainder $(C, F \cap [a, b])$-path P^* from s to a not visiting (b, a), is symmetric.

Overall, if C has n cells and $m = O(n)$ edges on the outline, then there are $O(m^2)$ subproblems, $O(m)$ choices per subproblem, and the depth-first search spends $O(n)$ time per subproblem and choice. Therefore, the total running time is $O(m^3 n) = O(n^4)$. □

7.4 Squares

Each square clue has a color and is placed on a cell of the puzzle. Each region formed by the solution path and puzzle boundary must have at most one color of squares. In this section, we prove NP-hardness of square clues of two colors. This result is tight given the following:

Observation 7.4.1. *Witness puzzles containing only squares of one color do not constrain the path, so are always solvable.*



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

Figure 7-13: Breakable degree-4 vertex gadget.

7.4.1 Tree-Residue Vertex Breaking

Our reduction is from *tree-residue vertex breaking* [51]. Define *breaking* a vertex of degree d to be the operation of replacing that vertex with d vertices, each of degree 1, with the neighbors of the vertex becoming neighbors of these replacement vertices in a one-to-one way. The input to the tree-residue vertex breaking problem is a planar multigraph in which each vertex is labeled as "breakable" or "unbreakable". The goal is to determine whether there exists a subset of the breakable vertices such that breaking those vertices (and no others) results in the graph becoming a tree (i.e., destroying all cycles without losing connectivity). This problem is NP-hard even if all vertices are degree-4 breakable vertices [51].

7.4.2 Squares of Two Colors

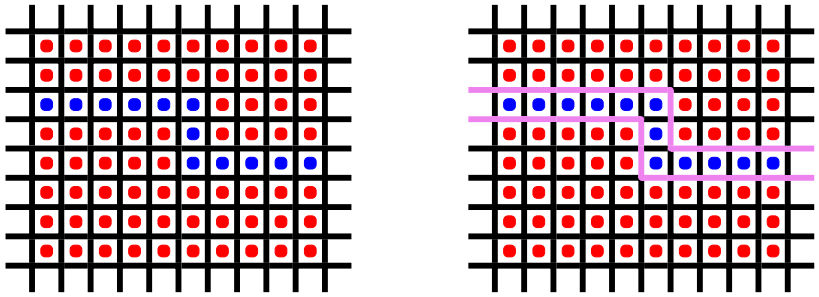
Theorem 7.4.2. *It is NP-complete to solve Witness puzzles containing only squares of two colors.*

Concurrent work [90] also proves this theorem. However, we prove this by showing that the stronger *Restricted Squares Problem* is also hard, which will be useful to reduce from in Section 7.5.

Problem 7.4.3 (Restricted Squares Problem)**.** *An instance of the* Restricted Squares Problem *is a Witness puzzle containing only squares of two colors (red and blue), where each cell in the leftmost and rightmost columns, and each cell in the topmost or bottommost rows, contains a square clue; and of these square clues, exactly one is blue, and that square clue is not in a corner cell; and the start vertex and end cap are the two boundary vertices incident to that blue square; see Figure 7-15a.*

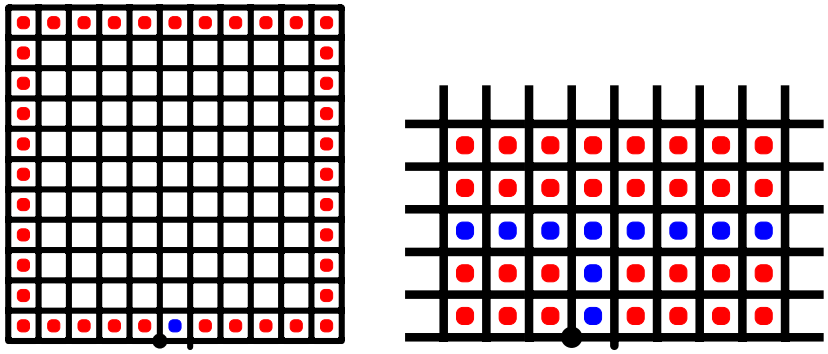
Theorem 7.4.4. *The Restricted Squares Problem is NP-complete.*

Proof. We reduce from tree-residue vertex breaking on a planar multigraph with breakable vertices of degree 4. The overall plan is to fill the board with red squares, and then embed the multigraph into the strict interior of the board (i.e., without



(a) Edge gadget (b) Local solution

Figure 7-14: Edge gadget.



(a) Boundary of the Restricted Squares Problem (b) Connecting the boundary to the center

Figure 7-15: Boundary of the Restricted Squares Problem.

using the extreme rows and columns). The gadgets representing the graph use a combination of blue squares and empty cells. Figures 7-13 and 7-14 illustrate the gadgets for a breakable vertex of degree 4 and an edge (including turns), respectively. The square constraints dictate that any solution path must traverse the edge shared by two cells with squares of opposite color. By this property, the only possible local solutions to the gadgets are the ones shown in the figures.

To connect these gadgets together, we first lay out the multigraph orthogonally on a linear-size grid with turns along the edges (e.g., using [124]), scale by a constant factor,[6] place the vertex gadgets at the vertices, and route the edges to connect the vertex gadgets (roughly following the drawing, but possibly adding extra turns). Finally, we choose any edge on the bounding box of the embedded multigraph and connect it to the boundary as shown in Figure 7-15b. This effectively introduces an unbreakable vertex of degree 3, which does not affect the solvability of the tree-residue vertex breaking instance. We place the start vertex and end cap at the two boundary vertices at the end of this boundary connection, as required by the Restricted Squares Problem.

[6]For example, a scale factor of 8 in each dimension suffices. The center of a vertex gadget is at most 2×2, and three more rows and columns on each side suffice to wiggle the incident edges to a desired row or column to match an adjacent gadget. More efficient scaling is likely possible.

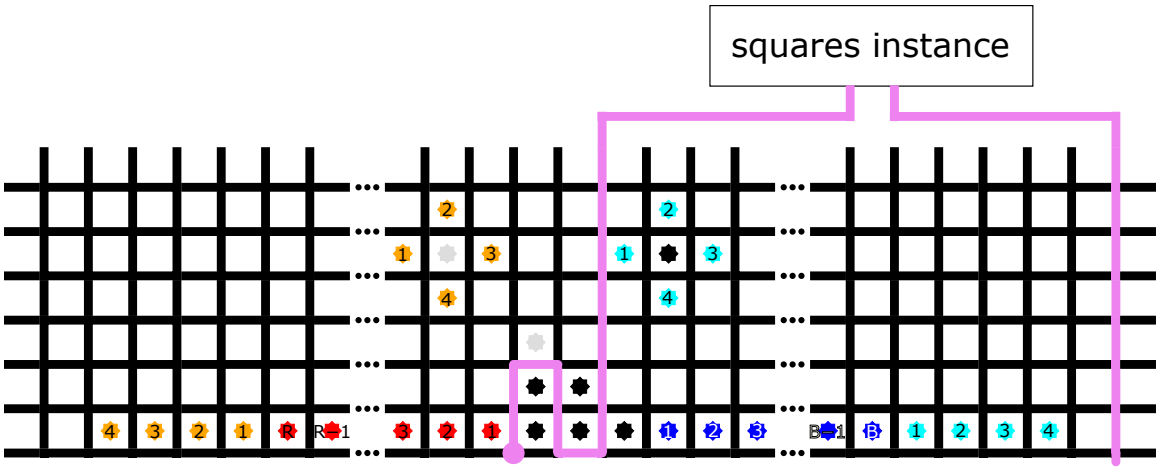


Figure 7-16: The boundary of the reduction. Each visual (color, number) pair represents a distinct color in the constructed instance. All stars depicted as blue correspond to blue squares in the source instance and must be in the inside region. Stars depicted as red correspond to red squares and must be in the outside region. The other stars enforce this.

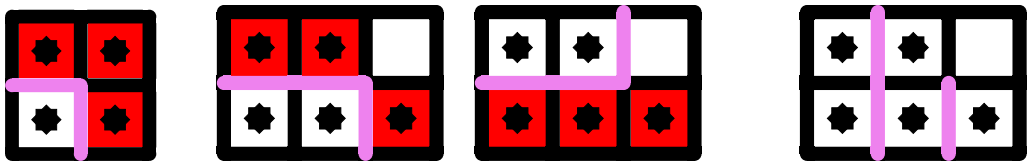
We claim that there is a bijection between solutions of the constructed Restricted Squares Problem and solutions to the given instance of tree-residue vertex breaking: a vertex gets broken if and only if we choose the locally disconnected solution (Figure 7-13c) in the corresponding vertex gadget. It remains to show that breaking a subset of vertices results in a tree if and only if the corresponding local solutions form a global solution path to the Witness puzzle. The local solutions mimic an Euler tour of the planar graph after breaking the subset of vertices, and this Euler tour is a single connected cycle if and only if the graph is connected and acyclic. The one difference is the subdivided edge with an extra unbreakable vertex of degree 3 connected to the boundary (Figure 7-15b), which transforms an Euler tour into a solution path starting at the start vertex and ending at the end cap. Provided that a vertex breaking solution exists, the corresponding solution path will satisfy all square constraints because it will have all blue squares on its interior and all red squares on its exterior. □

7.5 Stars

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

Theorem 7.5.1. *It is NP-complete to solve Witness puzzles containing only stars (of arbitrarily many colors).*

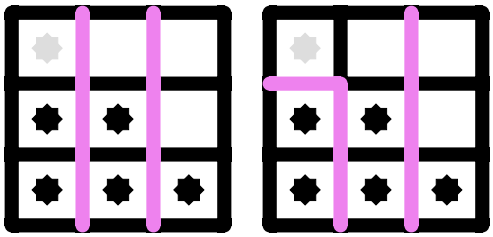
Proof. We reduce from the Restricted Squares Problem (Problem 7.4.3 above). For every square in the source instance I, we use exactly one pair of stars of a distinct



(a) After visiting the vertex in the center of four stars of the same color, the path cannot turn. (Rotations omitted.)

(b) The star in the right column prevents the path from splitting the four stars horizontally.

(c) Thus, the path must split the four stars vertically and separate the right star by using these forced edges, so the rightmost star cannot be in the same region as the middle stars.



(d) No valid assignment of stars to regions places the white star in the same region as the rightmost black star.

Figure 7-17: Analysis of the group of contiguous black stars.

color corresponding to that square, as well as ten auxiliary colors. Figure 7-16 shows the high-level structure of the reduction: if the given squares instance I has R red squares and B blue squares, then in the stars instance, we use $R+B+10$ colors: black, white, c_1, c_2, c_3, c_4 (all drawn as cyan), o_1, o_2, o_3, o_4 (all drawn as orange), $r_1, r_2, \ldots, r_R$ (all drawn as red), and $b_1, b_2, \ldots, b_B$ (all drawn as blue). A subrectangle S of the constructed puzzle is designated for recreating I. For each pair of stars corresponding to a square, we place one of the two stars on the boundary of the puzzle, and the other in S in the same position as the corresponding square in I. We will show that the solution path is forced to divide the overall puzzle into exactly three regions—an "inside", an "outside", and a 2-cell region of just black squares—such that all of the boundary stars corresponding to red squares are on the outside and all of the boundary stars corresponding to blue squares are on the inside. Then, inside of S, the solution path must ensure that all stars corresponding to red squares are in the outside region and all stars corresponding to blue squares are in the inside region, or else the star constraint will be violated. Then any solution path for the puzzle, restricted to S, must correspond exactly to a solution path in I.

Consider the group of five contiguous black stars. Figure 7-17 shows that any solution must place the two leftmost black stars in the same region, the two middle

black stars in another region, and the two rightmost black stars (including the star outside the group) in a third region. Now consider the extragroup black star, which is adjacent to a c_1 star, a c_2 star, a c_3 star, and a c_4 star. At least one of the four edges of the extragroup black star is absent, so the extragroup star is in the same region as at least one of its neighboring cyan stars, and thus also in the same region as the singular other star of that cyan color in the far right of the puzzle. By transitivity, the rightmost intragroup black star (named b in the following) is in the same region as at least one far-right cyan star.

Consider the edge e to the left of b. Any solution path passes through e to separate b from the star to its left. Then consider the maximal part P of the solution path that contains e but not any boundary edges. Orient P so that it starts with edge e; note that this orientation may be opposite the orientation of the whole solution path. If P's other endpoint lies on the part of the boundary between the black stars and the far-right cyan stars, then b and those stars are in different regions, a contradiction. Therefore, all the boundary cells between b and the far-right cyan stars are in the same region. In particular, the blue stars of colors $b_1, b_2, \ldots, b_B$ are all in the same region as b, that is, on the right of P.

Now consider the white star w adjacent to the group of black stars. As shown in Figure 7-17d, w is not in the same region as b, so w is to the left of P (though not necessarily in the region immediately to the left of P), and thus the other (leftmost) white star is also left of P. As with the cyan stars, at least one orange star among those adjacent to the leftmost white star lies on the left of P, and so does at least one far-left orange star. As with the cyan, black, and blue stars, this implies that the red stars of colors $r_1, r_2, \ldots, r_R$ are on the left of P.

Thus, we have shown that the red stars are on a different side of P from the blue stars. This is possible only if the original Restricted Squares Problem instance has a solution. Conversely, if the original squares instance is possible to solve, then we can augment that solution into a solution to the constructed stars Witness puzzle as shown in Figure 7-16. □

Open Problem 7.5.2. *Is it NP-hard to solve Witness puzzles containing only a constant number of colors of stars? Or just a single color of stars?*

7.6 Triangles

Triangles are placed in cells. The number of edges on the solution path that are incident to that cell must match the number of triangles. This constraint is similar to Slitherlink, which is known to be NP-complete [152]. Table 7.4 summarizes known and new results for Slitherlink puzzles with clues chosen from $\{0, 1, 2, 3\}$. The one difference is that The Witness does not allow 0-triangle clues. Unfortunately, the proof in [152] relies critically on being able to force zero edges around a cell using 0 clues. We can simulate 0 clues using broken edges, as in Table 7.1. To avoid broken edges, we develop new proofs that 1-triangle, 2-triangle, and 3-triangle clues are NP-hard.

Clue types	*Complexity*
0	P [152]
1	**NP-complete [Theorem 7.6.1]**
2	**NP-complete [Theorem 7.6.2]**
3	**NP-complete [Theorem 7.6.3]**
4	P [trivial]
0 and (1 or 2 or 3)	NP-complete [152]

Table 7.4: Summary of Slitherlink / Witness triangle constraints. New results are bold. Recall that "0" is not a valid clue in The Witness. (The proof in [152] is for 0 and 1 clues, but can be straightforwardly adapted to replace 1 clues with 2 or 3 clues.)

7.6.1 1-Triangle Clues

Proving hardness of Witness puzzles containing only 1-triangle clues is challenging because it is impossible to (locally) force turns on the interior of the puzzle. Any rectangular subpuzzle with any set of 1-triangle clues can be satisfied by a set of disjoint paths containing either every second row of horizontal edges in that rectangle or every second column of vertical edges in that rectangle, regardless of the configuration of 1-triangle clues. Therefore, any local arguments about gadgets in the interior of the puzzle must confront the possibility of local solutions which are comprised of just horizontal or vertical paths straight through. 2-triangle clues, discussed in Section 7.6.2, present a similar challenge. Nonetheless, we are able to prove NP-hardness:

Theorem 7.6.1. *It is NP-complete to solve Witness puzzles containing only 1-triangle clues.*

Proof sketch. We reduce from X3C, making use of the fact that the solution path must be a single closed path. Refer to Figure 7-18. Sets and elements are represented by sets of rows in a puzzle almost completely filled with 1-triangle clues. Boundary conditions alone force any solution to form disconnected cycles and one path consisting largely of horizontal lines, but we can build gadgets by deleting 1-triangle clues in specific locations. Set–element connection gadgets allow connecting the horizontal lines of a chosen set exactly when we connect the horizontal lines of a specific element, but only if the element is so covered exactly once, and the 3-set gadget allows connecting the horizontal lines of an unchosen set. The cycles can be connected to form a single solution path exactly when the X3C instance has a solution. □

Proof. We reduce from Exact Cover by 3-Sets, which we abbreviate X3C [65]: given a set X of n elements, and given a family C of m cardinality-3 subsets of X, decide whether there exists a subset $P \subseteq C$ such that P is a partition of X (i.e., P covers every element in X exactly once). Without loss of generality, let $X = \{0, 1, \ldots, n-1\}$. Figure 7-18 shows how to fit together the gadgets we will now describe.

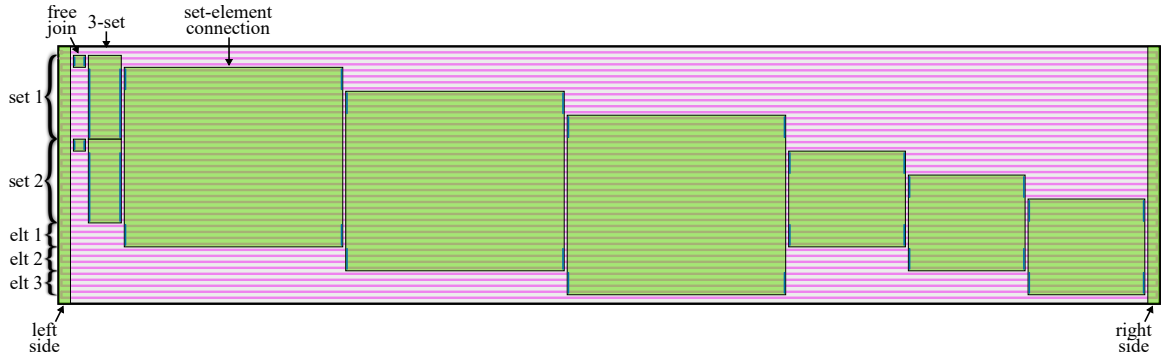


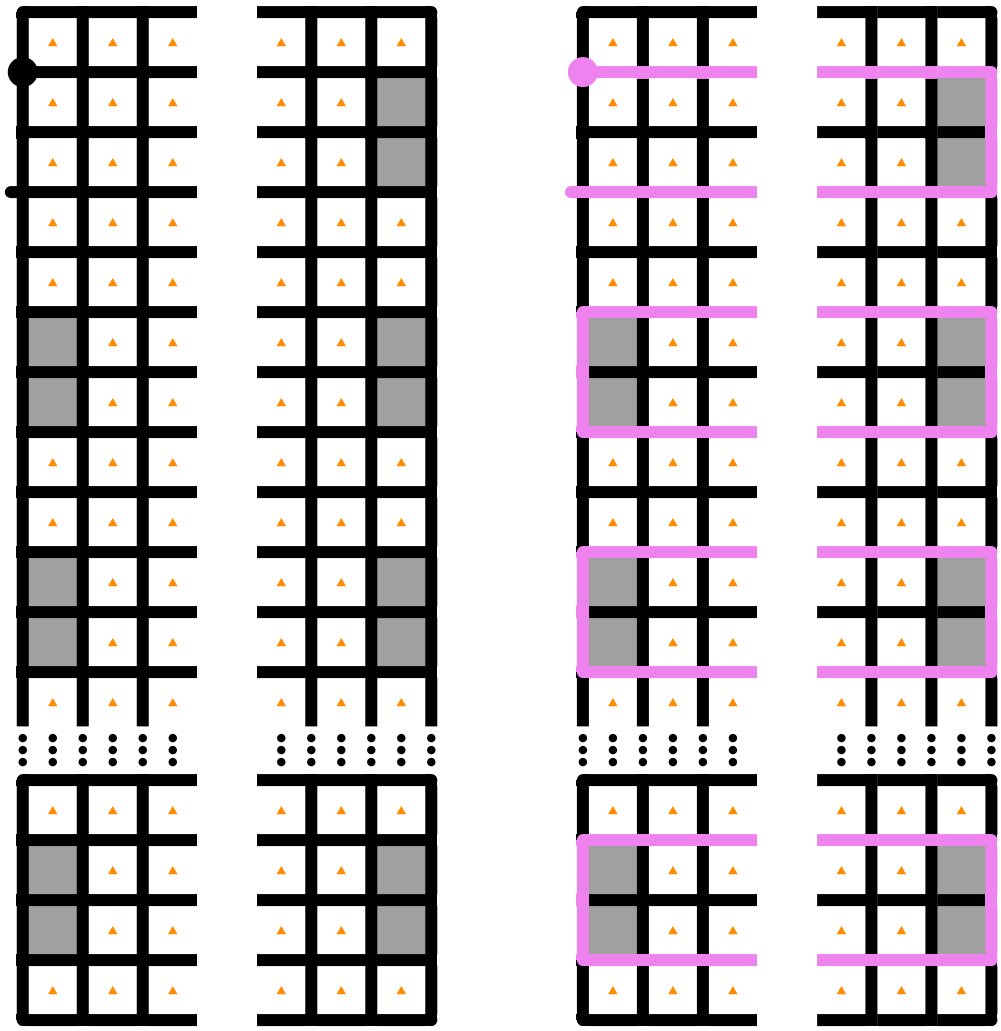
Figure 7-18: Full example schematic (drawn roughly to scale) of the reduction from X3C to 1-triangle clues of Theorem 7.6.1, with three elements and two 3-sets each consisting of all three elements. Pink lines indicate the "solution" of the initial puzzle filled with 1-triangles in all but the leftmost and rightmost columns. Each box represents one of the gadgets in the proof, with vertical lines to indicate the horizontal lines that could be connected by the gadget.

Left and right sides of the construction. We start from (and later modify) a Witness puzzle where all cells have a 1-triangle clue except for the leftmost and rightmost columns of cells, as shown in Figure 7-19a. The rightmost column follows the repeating pattern (1-triangle, empty, empty, 1-triangle)^k, while the leftmost column has the same pattern except for the top four rows which instead of two empty cells have the start circle and end cap, in the bottom-left of the first and third row of cells respectively. The number of rows of cells is $4k = 2 + 28m + 8n + 2$.

If we relax the "solution" to allow multiple paths and/or cycles, then we claim that this puzzle has the unique solution shown in Figure 7-19b. (Thus, with the usual constraint of a single path, there is no solution.) Even stronger, for any modified puzzle still having 1-triangles throughout the topmost two rows and the same leftmost three columns and rightmost three columns as Figure 7-19a, we claim that any solution must look like Figure 7-19b in those columns.

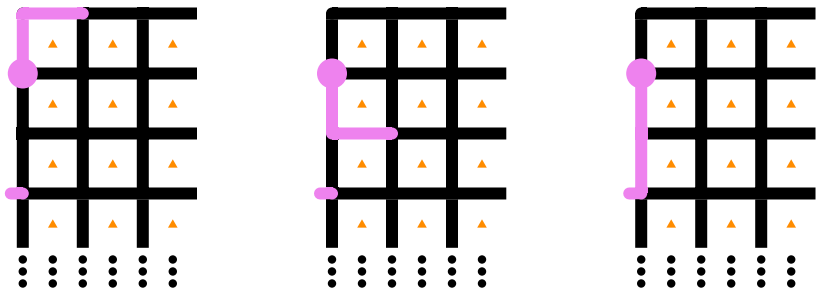
First we claim that the path goes right from the start vertex; refer to Figure 7-20. If the path goes up into the top-left corner, then it must then go right, violating the 1-triangle clue in the top-left cell. If the path goes down, the 1-triangle forces it to go down again, but then it has reached the end cap without satisfying most of the 1-triangles in the puzzle. Therefore, the path must initially go right.

Next, by repeated application of Rule 1 of Figure 7-21 in the topmost two rows, the path must continue right from the start circle until reaching the right side of the puzzle. By repeated application of Rule 3 of Figure 7-21, rows of cells alternate between having the path on their bottom and having the path on their top, thereby making $2k$ *horizontal lines* of path. If we perfectly pair up adjacent rows of cells (pairing every odd row, including the first, with the row below it), then there is exactly one horizontal line in between each pair of rows. These horizontal lines must join using the edges incident to blank cells in the leftmost and rightmost columns (except at the start and end cap).



(a) Unsolved. (b) Solved.

Figure 7-19: Leftmost and rightmost three columns of the puzzle. Empty cells are shaded gray for emphasis.

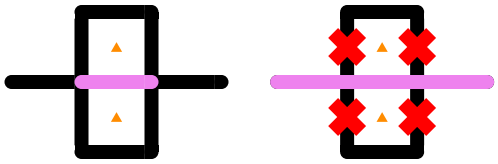


(a) Failed attempt to start by going up.

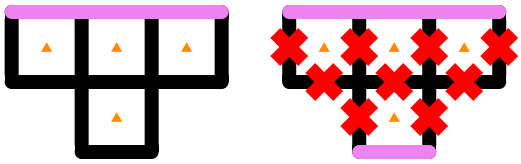
(b) Failed attempt to start by going down then right.

(c) Failed attempt to start by going down twice.

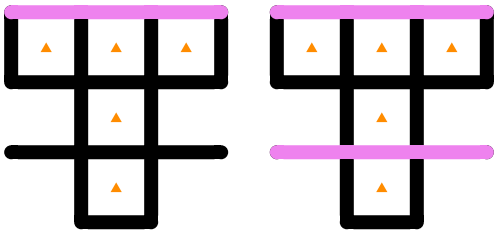
Figure 7-20: Nonsolutions to the leftmost columns.



(a) Rule 1: When an edge adjacent to two 1-triangle clues is present in the solution path, then it must extend at least one more step in both directions.



(b) Rule 2: When the three edges along the long side of four 1-triangle clues arranged in a T shape are present in the solution path, then the edge opposite them must also be present.

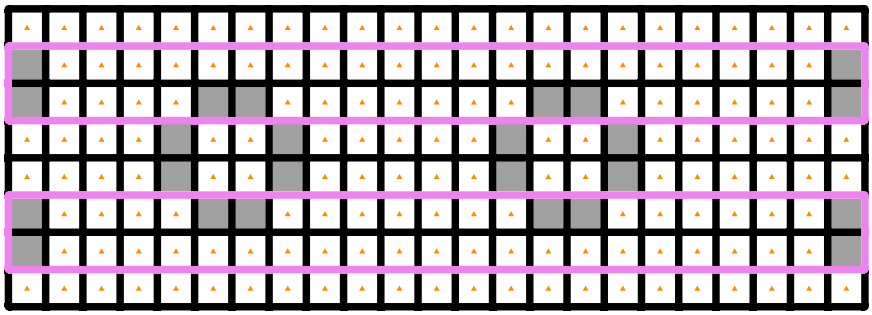


(c) Rule 3: An application of Rule 2 followed by an application of Rule 1.

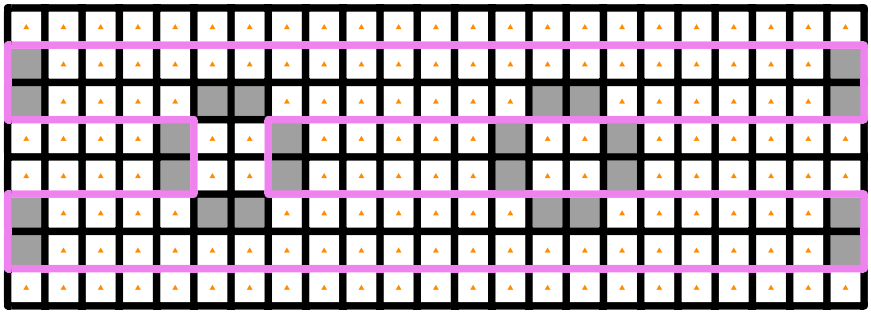
Figure 7-21: Local rules for 1-triangle clues. A red X means that the edge cannot be in the solution path due to constraints imposed by triangles.

Hence, the unique "solution" to this initial puzzle consists of k connected components, one start-to-end path in the top four rows and one cycle in every following consecutive four rows. In the remainder of the proof, we will place gadgets in the interior of the puzzle (removing some 1-triangle clues) to enable connecting these solution components together into one path exactly when the X3C instance has a solution. This modification will not touch the topmost two rows, the bottommost two rows, the leftmost three columns, or the rightmost three columns, thereby still forcing those columns to look like Figure 7-19b.

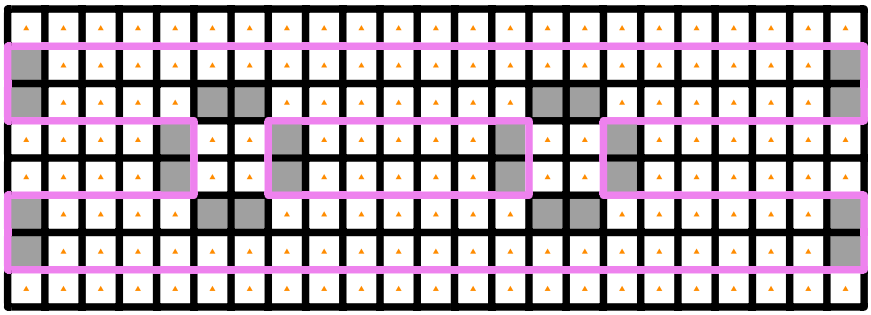
Free-join gadget. Figure 7-22 shows one gadget for joining components of the "solution" together, called the *free-join gadget*. By placing a 4×4 "circle" of empty cells, we allow the solution to optionally deviate from horizontal lines and thereby connect together two components by joining the bottom line of one component with the top line of the next component. (This gadget, and all future gadgets, can be argued to have exactly two local solutions using Rules 1–3 of Figure 7-21.) Later gadgets will perform similar joins between adjacent pairs of components, but with more constraints on when the join is allowed. Figure 7-22c shows that performing two such joins on the same two lines results in an isolated cycle in the middle of the puzzle, which in our construction can never be connected to other components. Therefore, to form one connected solution path, we will need to make exactly one join between each even-numbered horizontal line (in particular, excluding the first line) and the line immediately below it.



(a) Lines unjoined.

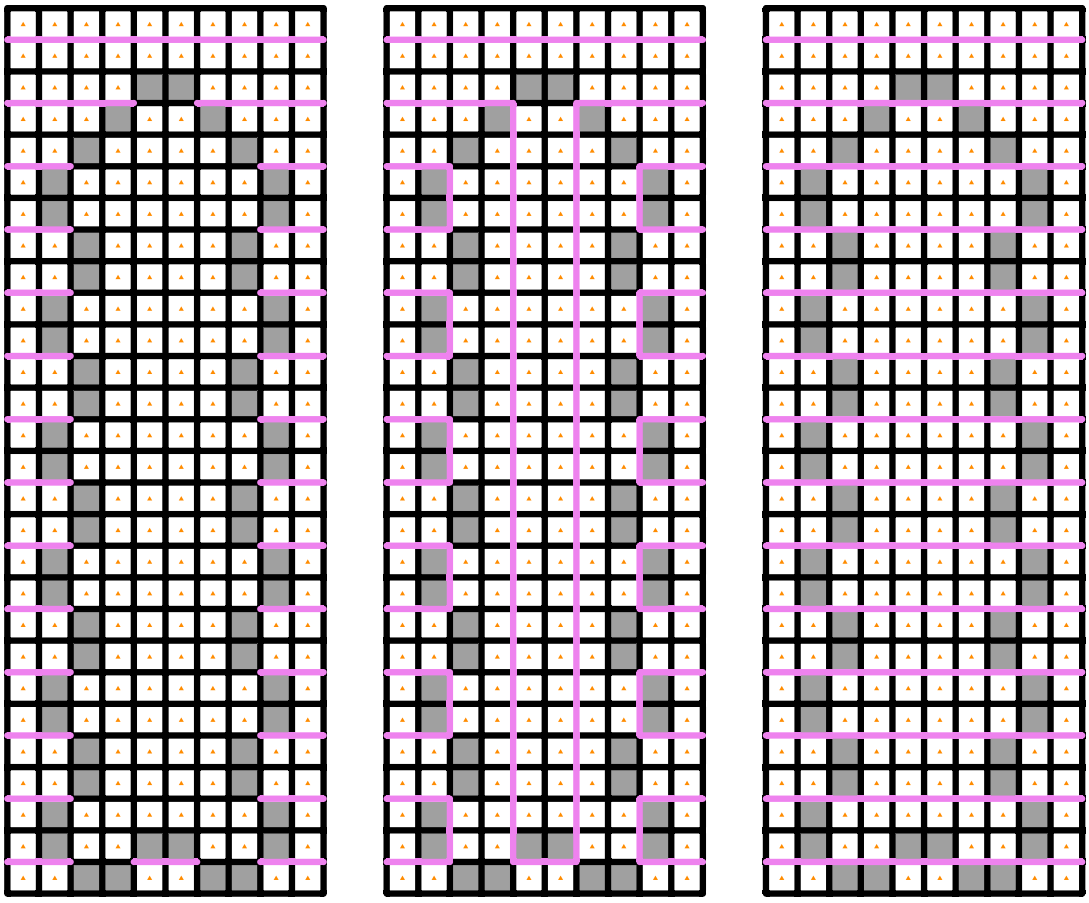


(b) Lines joined once.



(c) Lines joined twice (invalid).

Figure 7-22: Free-join gadget and how it interacts with the left and right boundaries. In fact, only one of the two 4×4 circles are needed for this gadget, but this form illustrates the negative effect of joining twice.



(a) Unsolved. (b) Solution when the set is unchosen. (c) Solution when the set is chosen.

Figure 7-23: The 3-set gadget consists of 28 rows of cells: 4 "support" on top to enable this gadget, followed by $8 \cdot 3$ rows, where each group of 8 rows connects to one element.

3-set gadgets. We do not place any gadgets in the topmost two rows of cells (and thus do not touch the topmost horizontal line), effectively shifting the parity of lines in all gadgets relative to the left and right sides of the puzzle. In the remaining rows, the top $28m$ rows of the puzzle represent the m 3-sets, where each group of 28 consecutive rows represents one 3-set. Intuitively, the top 4 rows of a 3-set are "support" rows (containing exactly 2 lines), while the following $24 = 8 \cdot 3$ rows leave 8 rows (containing 4 lines) for each of the 3 connections to elements. The primary *3-set gadget* is shown in Figure 7-23; we place it near the left side of every 3-set. This gadget will enforce that each set is either entirely unchosen (Figure 7-23b) or entire chosen (Figure 7-23c). We also place a free-join gadget to join together the two horizontal lines in the four support (top) rows of each 3-set.

Element gadgets. The following $8n$ rows, which reach down to all but the bottom-most two rows of the puzzle, represent the elements. Each element is represented simply by 8 consecutive rows (containing 4 horizontal lines).

Connecting sets to elements. For each occurrence of an element in a set, we connect the 8 rows of the element to the corresponding 8 rows of the set (among the $24 = 3 \cdot 8$ nonsupport rows) using the *set–element connection gadget* in Figure 7-24. This gadget has two solutions. When the set is not chosen, all lines proceed horizontally through the gadget. When the set is chosen, the solution is as shown in Figure 7-24b, which connects together adjacent pairs of horizontal lines among the 4 lines in the set and the 4 lines in the element, while leaving all other horizontal lines unaffected (topologically in terms of which ends they connect, even though they bend geometrically). Each element connects to potentially several sets in this way, but referring to Figure 7-22, the element's lines will be properly connected if and only if exactly one set containing the element is chosen while the rest are unchosen.

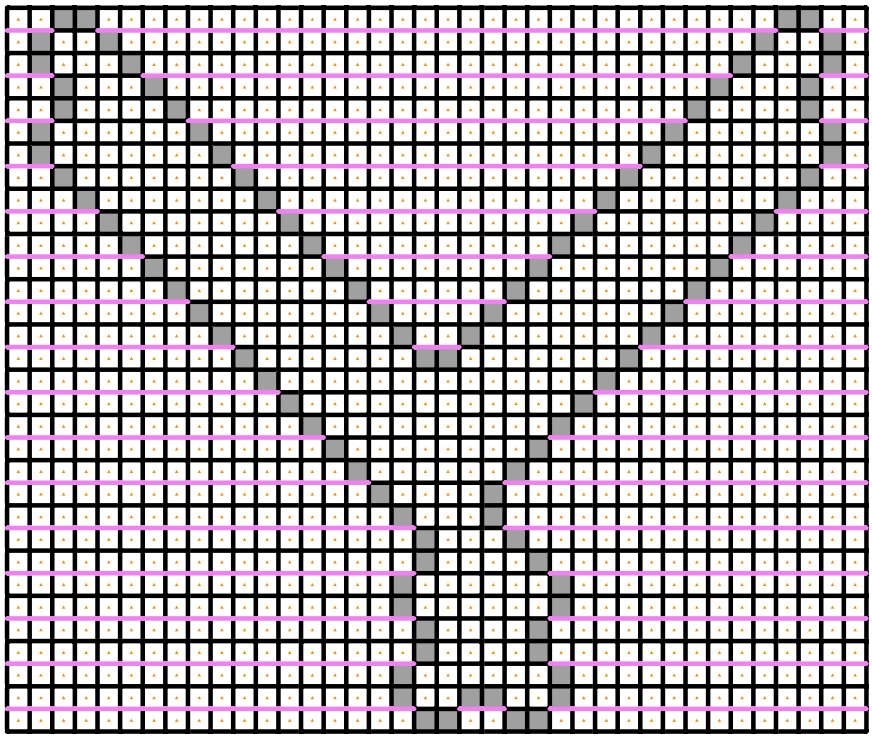
The set–element connection gadget can be extended to larger sizes than drawn in Figure 7-24 by adding an equal number of rows and columns, and extending the pattern of the four "diagonals" of empty cells in the top half of the gadget. Similarly, it can be reduced in size. This resizing enables us to connect together any set's 8 rows with any element's 8 rows. Because the width of the gadget depends upon its height, the nonoverlapping placement of these gadgets affects the total number of columns in the puzzle. Because we use only a polynomial number of polynomial-sized gadgets, the constructed puzzle remains polynomial size.

Equivalence to X3C. As argued above, the partial solution in Figure 7-19b joins together into a single solution path if and only if every even-numbered horizontal line (skipping the first line) has exactly one join with the line immediately below it (as in Figure 7-22). The top two lines of each 3-set can always join together (exactly once) via their free-join gadget (and in no other way). The remaining ten lines of each 3-set are joined exactly once if and only if the 3-set gadget and all three incident set–element connection gadgets either all use the "unchosen" solution (in which case the connections are from the 3-set gadget) or all use the "chosen" solution (in which case the connections are from the set–element connection gadgets). The four lines of each element are joined exactly once if and only if exactly one of the incident set–element connection gadgets uses the "chosen" solution. Therefore, solving the Witness puzzle is equivalent to solving the X3C instance. □

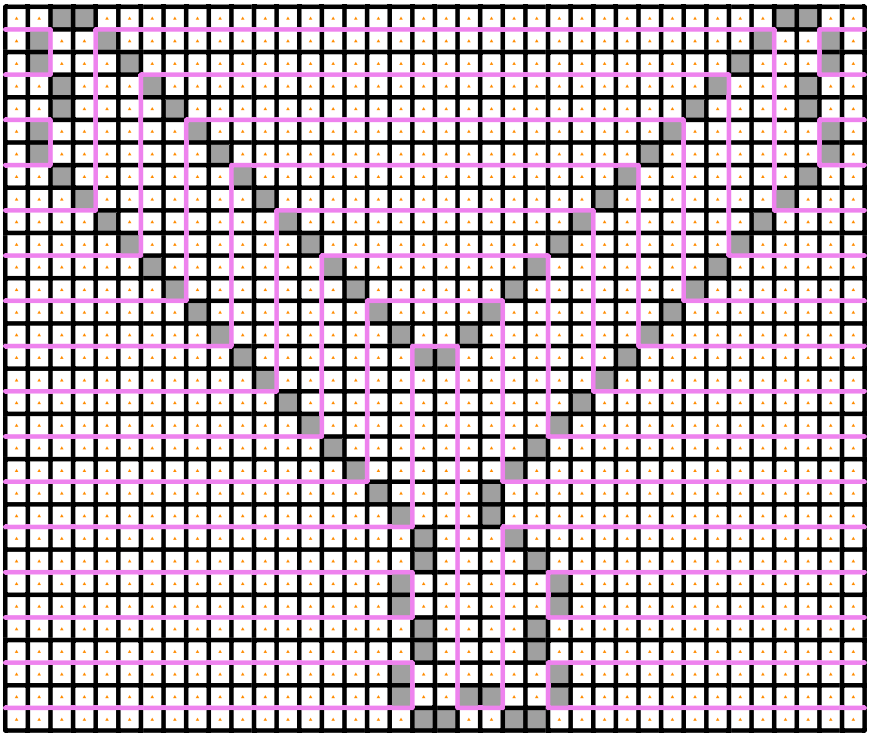
7.6.2 2-Triangle Clues

As with 1-triangle clues, any rectangle containing only 2-triangle clues can be locally satisfied by a set of disjoint path segments containing either all the vertical edges or all the horizontal edges in the rectangle. However, compared to 1-triangle-only puzzles, locally-satisfying paths can turn with more flexibility even in areas completely filled with 2-triangle clues. Accordingly, our proof for puzzles containing only 2-triangle clues is based around connecting concentric cycles instead of straight path segments.

Theorem 7.6.2. *It is NP-complete to solve Witness puzzles containing only 2-triangle clues.*

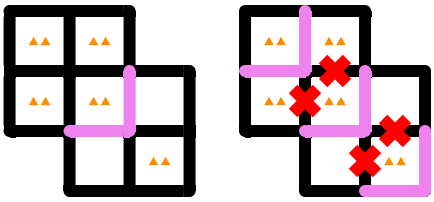


(a) Unsolved.

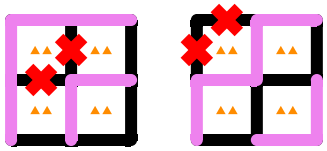


(b) Solution when the set is chosen.

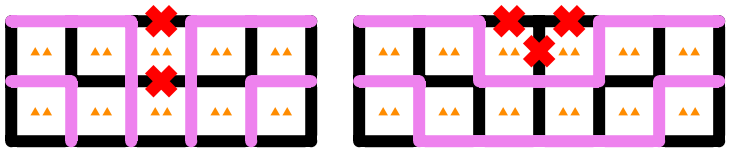
Figure 7-24: Set–element connection gadget. When this set is unchosen, the solution simply proceeds horizontally across the gadget.



(a) A turn between clues forces further turns in both directions along the ray.



(b) The corners of the puzzle always emit one or three rays.



(c) Rays can reflect off the puzzle boundary in two ways (here, the top boundary).

Figure 7-25: Local rules for 2-triangle clues regarding wave propagation.

Proof sketch. We reduce from X3C, making use of the fact that the solution path must be a single closed path. The puzzle is almost completely filled with 2-triangle clues. Local conditions alone force any solution to form disconnected concentric cycles, but we can build gadgets by deleting 2-triangle clues. Elements are represented by rows in the bottom quadrant of the puzzle, where sets are represented by gadgets intersecting these rows (see Figure 7-26). Used set gadgets connect the concentric cycles of their element rows, and cleanup gadgets allow connecting the cycles not corresponding to elements. The cycles can be connected to form a single solution path exactly when the X3C instance has a solution. □

Proof. As in the 1-triangle reduction of Theorem 7.6.1, we reduce from X3C. Consider an instance (X, C) with $|X| = n$ elements and $|C| = m$ cardinality-3 subsets of X, and assume without loss of generality that $X = \{0, 1, \ldots, n-1\}$.

Similar to the previous reduction, we start by considering a puzzle completely filled with 2-triangle clues together with a clue-satisfying set of disjoint paths, namely concentric squares in a square grid, and placing gadgets (arrangements of empty cells) such that these paths can be joined into a full solution path if and only if the X3C instance has a solution.

Wave propagation. A path can turn in puzzle areas filled with 2-triangle clues, but when it does so, it becomes a wave that continues to propagate, turning along a ray directed based on where the turns "point" (see Figure 7-25a). When a path turns on a 2-triangle clue cell, and the ray points to a diagonally adjacent 2-triangle clue cell, then the path must also turn in the same direction on that cell, extending the ray outwards. Similarly, the wave propagates inward (opposite the ray) when there are only 2-triangle clues nearby, as the path would otherwise produce an isolated 2×2 square. The solution path necessarily turns at the corner of the puzzle, so each corner

emits one ray (directed into the corner) or three rays (one out and two in) along the main diagonals of the puzzle (see the top-right of Figure 7-25b).

Wave propagation imposes structure on any possible solution path: in an area full of 2-triangle clues, the solution path divides the area into horizontally- and vertically-oriented subareas (containing parallel horizontal or vertical path edges respectively) separated by the rays of the waves. By the propagation rules, rays can only start and end at either an empty cell or the boundary of the puzzle, and cannot intersect other rays propagating in a different direction except at an empty cell or at the boundary of the puzzle.

Overall layout. Figure 7-26 shows the overall layout of the produced Witness puzzle. The rays emitted from the corners of the puzzle divides the puzzle into four quadrants. Because the puzzle is full of 2-triangle clues except in our gadgets, the path cannot turn except along those main diagonal rays, so the initial "solution" to the puzzle is a set of concentric square cycles. In the remainder of the proof, we add gadgets (remove 2-triangle clues) to enable connecting these cycles into a single path exactly when the X3C instance has a solution.

We associate some pairs of concentric squares with elements in the X3C instance and add X3C gadgets that allow connecting the square pairs associated with each of the elements in a 3-set. The filler diamond gadgets connect the cycles between the element-representing cycles. The bulk connector gadgets connect the cycles in the rest of the puzzle and resolve the collision of the main diagonal rays at the very center of the puzzle.

Each of these gadgets contains some empty cells, so there is a risk that rays could start in one gadget and end in another, or travel between distant empty cells in the same gadget in unintended ways, disrupting the pattern of concentric squares. To contain the rays, we place gadgets such that there are no empty cells outside the gadget along the diagonals of the gadget's empty cells (see Figure 7-27). We also take into account possible reflections off of the puzzle boundary (see Figure 7-25c) by adding buffer space below the gadget and protecting additional diagonals to its left and right. By placing gadgets in this way, any ray escaping a gadget will crash into one of the main diagonal rays (possibly after reflecting off of the puzzle boundary); as there are no empty cells along the main diagonals (except the very center cell), there is no way to avoid violating the 2-triangle clue at the ray intersection.

We isolate the different types of gadgets from one another by placing them in different quadrants (separated by a main diagonal). Besides being convenient for layout, using separate quadrants is necessary because the bulk connector gadget contains empty cells on most diagonals.

X3C gadget. For each $x \in X$, we associate an adjacent pair of concentric squares p_x with radii r_x and $r_x + 1$, separated from other pairs such that $r_{x+1} = 4 + r_x$. For each $\{i, j, k\} \in C$, we place an X3C gadget, which is built to permit one nontrivial solution where each of p_i, p_j, and p_k merge, representing using the set in the cover. The trivial solution, with all paths continuing horizontally through the gadget, represents not

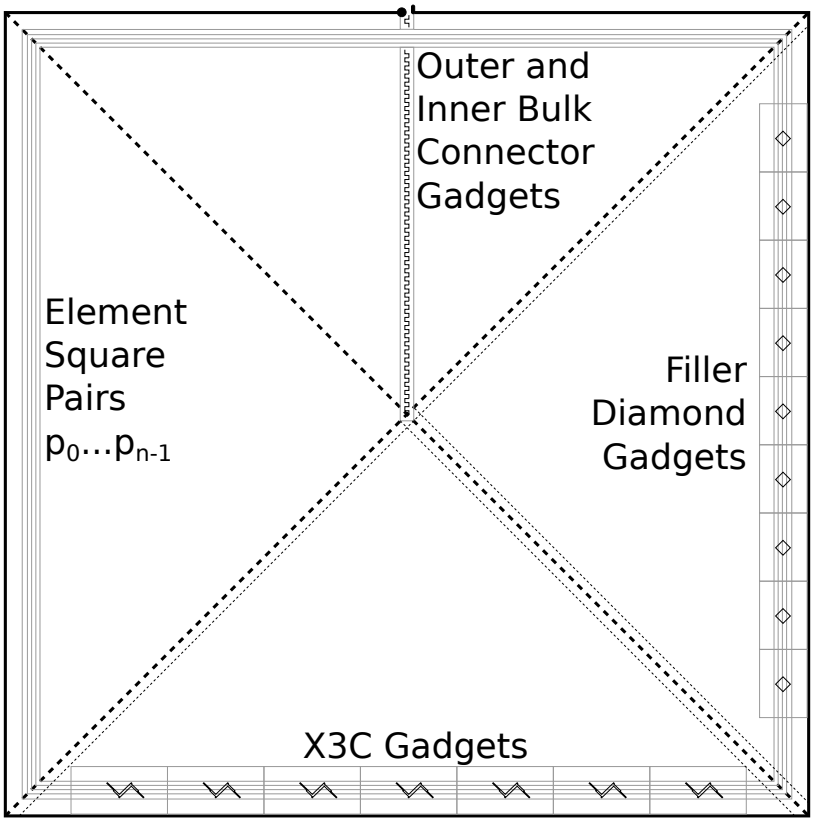


Figure 7-26: The layout of gadgets within the X3C reduction puzzle. Quadrants defined by corner rays separated by thick dashed lines, and ray escape risk areas by thin dashed lines.

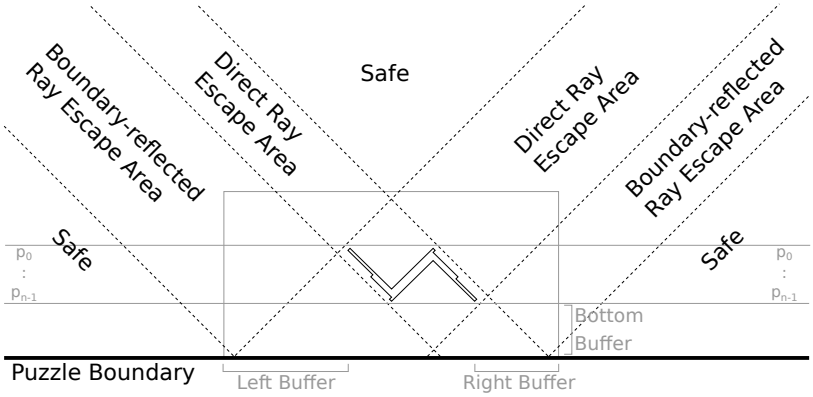


Figure 7-27: How to safely place a gadget (e.g., an X3C gadget) anywhere within p_0 to p_{n-1}. Possible ray escape areas are delimited by dashed lines, and the reserved space for the gadget by a gray box.

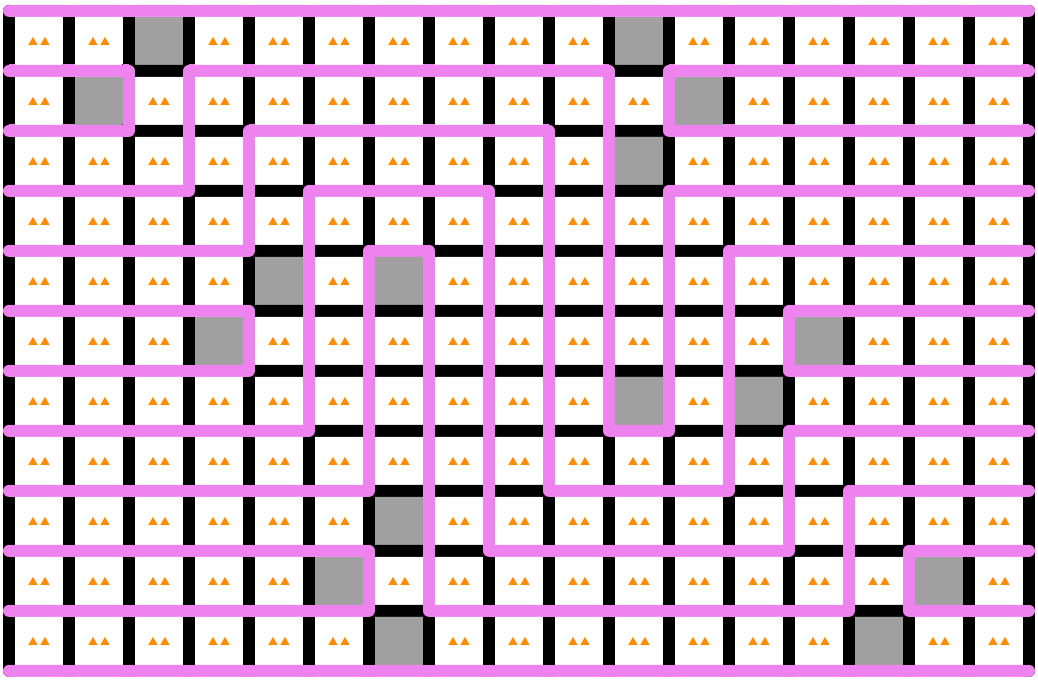


Figure 7-28: The X3C gadget for $\{x, x+1, x+2\}$, solved to join 3 pairs of adjacent paths p_x, p_{x+1}, and p_{x+2}. All other traversing paths' connections are unaffected. In the other solution, all paths continue horizontally across the gadget.

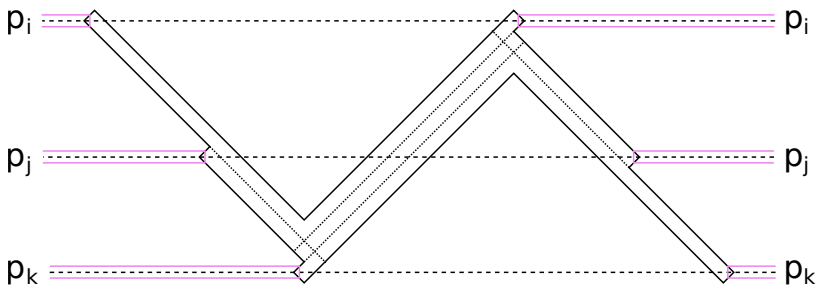
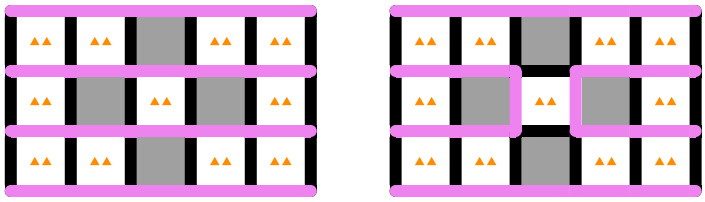


Figure 7-29: A blueprint for constructing an X3C gadget for any $\{i, j, k\}$. The polygon represents the border of the vertically oriented interior of the gadget: its points are the centers of the empty cells, and its edges are the rays, all on unique diagonals of the grid. The dotted lines mark the width of each interior section.



(a) Unused (not joining any cycles).

(b) Used to join the center pair of cycles into a single cycle.

Figure 7-30: The two local solutions of a diamond gadget.

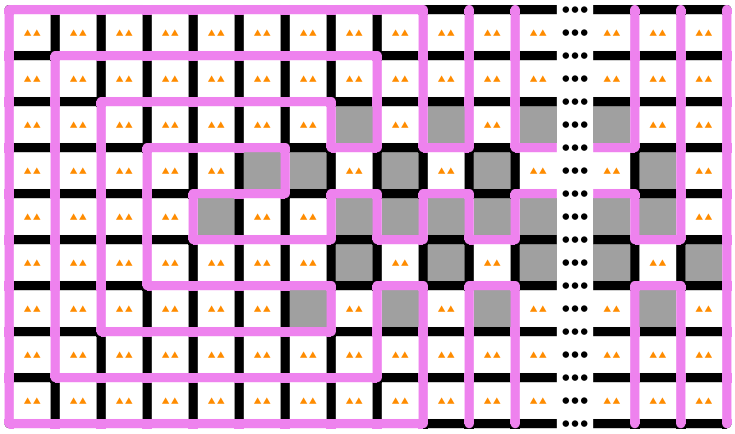
using the set in the cover. Figure 7-28 shows the smallest X3C gadget, and a general X3C gadget can be constructed by stretching it to cover any p_i, p_j, and p_k pairs, as detailed in Figure 7-29. Each empty cell in an X3C gadget shares a diagonal with exactly two other empty cells, each on a different diagonal. Because our overall layout leaves no way to terminate a ray that leaves the gadget, when the nontrivial solution is used, the rays are forced to trace out the unique polygon that merges the three pairs of paths and preserves connectivity of other paths.

Diamond gadget. The diamond gadget, shown in Figure 7-30, is the simplest gadget that can connect adjacent cycles in a controlled way. Again, due to our overall layout, only the trivial fully-horizontal/vertical local solution and the nontrivial local solution shown in the figure are possible in a global solution. We place diamond gadgets in the right quadrant to connect the three adjacent pairs between each p_x and p_{x+1}. As these gadgets are the only way to connect those pairs, the nontrivial local solution is always used.

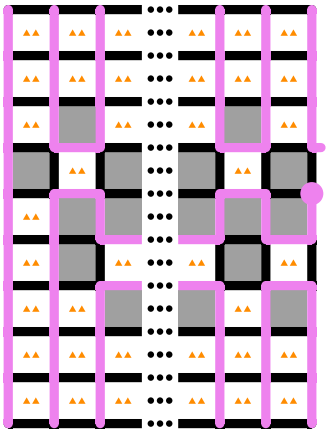
Bulk Connector Gadget. The bulk connector gadget is an extensible pattern connecting all concentric squares smaller or larger than a desired radius (see Figure 7-31). The X3C gadgets take care of connecting $p_0, \ldots, p_{n-1}$, and the diamond gadgets connect the pairs in between; we use bulk connector gadgets in the top quadrant to join all squares with radii from 1 to r_0 and from $r_{n-1}+1$ to the boundary. This places an empty cell at the center of the puzzle to terminate the main diagonal rays.

Analysis. If (X, C) is a YES instance of X3C, then the puzzle can be solved using the partition $P \subseteq C$ of X. Start with the concentric squares set of disjoint paths, which satisfies all two-triangle clues, then use the bulk connector gadget and all filler diamond gadgets to join all adjacent squares except $p_0, \ldots, p_{n-1}$. For each $\{i, j, k\} \in P$, use the corresponding X3C gadget to join p_i, p_j, and p_k. Because each $x \in X$ appears in exactly one $S \in P$, each p_x will be joined exactly once. Because using a gadget preserves clue satisfaction, and every concentric square is joined into one path, the path is a solution to the Witness puzzle.

Likewise, if the puzzle has a solution path, then (X, C) must be a YES instance of X3C. As argued, gadgets are sufficiently spaced apart and the main diagonals of



(a) The inner bulk connector gadget. The leftmost empty cell is the puzzle's center cell.



(b) The outer bulk connector gadget, incorporating the start circle and end cap.

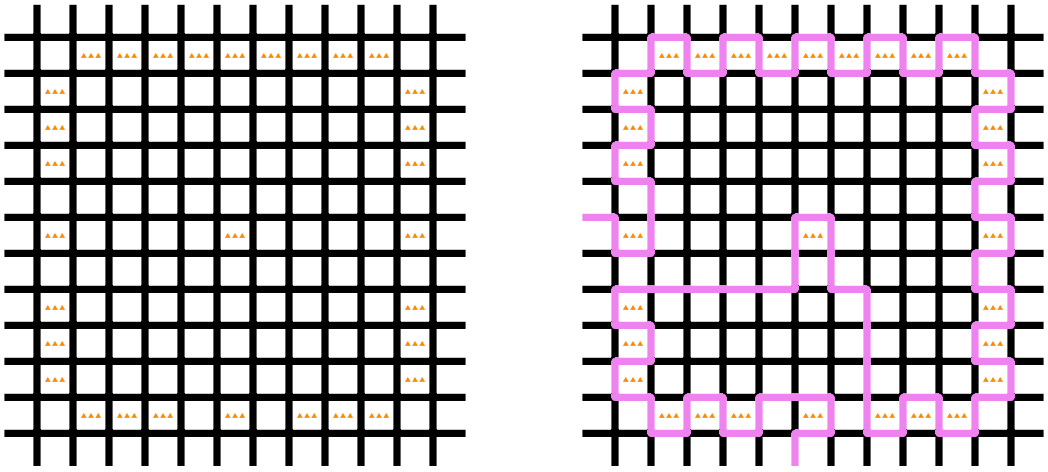
Figure 7-31: The bulk connector gadget used to connect all concentric squares smaller or larger than a desired radius. The bulk connector gadget can be instantiated in any orientation; in the orientation shown here, vertical pairs are connected. The concentric squares representing elements lie between the inner and outer bulk connector gadgets.

the puzzle isolate each quadrant, so any ray leaving a gadget would intersect with a main diagonal ray (possibly after reflecting off of the puzzle boundary), violating the 2-triangle clue at the intersection. Because X3C and diamond gadgets only have one nontrivial solution not emitting any rays, and using X3C gadgets is the only way to join adjacent concentric squares $p_0, \ldots, p_{n-1}$ without violating 2-triangle clues, the solution path defines a $P \subseteq C$ by its use of the X3C gadgets. P must cover every $x \in X$ otherwise the solution path would be disjoint, and it cannot double-cover any $x \in X$ otherwise the solution path would create a disconnected cycle out of the concentric squares in p_x, so P is a partition of X.

With respect to the complexity of the reduction, notice that the bounding box of the X3C gadget for $\{i, j, k\}$ is $\Theta(k-i) \times \Theta(k-i)$ cells, for $i < j < k \leq n-1$. The size of the buffer space surrounding each X3C gadget is $\Theta(n) \times \Theta(n)$ cells and the total lineup of X3C gadgets is $\Theta(mn)$ cells wide and $\Theta(n)$ tall. Similarly, each of the $3n$ filler diamond gadgets takes up $\Theta(n) \times \Theta(n)$ cells of buffer space, for a total of $\Theta(n^2)$ cells tall and $\Theta(n)$ cells wide. Thus, the side length of the puzzle is $\Theta(nm + n^2)$ cells to fit the simple-to-construct gadgets along the border without interference, and therefore the puzzle can be constructed in polynomial time. □

7.6.3 3-Triangle Clues

3-triangle clues admit a much simpler construction than those needed for 1-triangles and 2-triangles, as 3-triangles are constraining enough to build gadgets whose properties can be verified purely locally. The proof roughly follows the Hamiltonicity framework



(a) Unsolved.

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

Figure 7-32: A chamber with edges to the left, right, and below.

of Section 7.2, using adjacent 3-triangle clues to force the solution path to form impassable "walls".

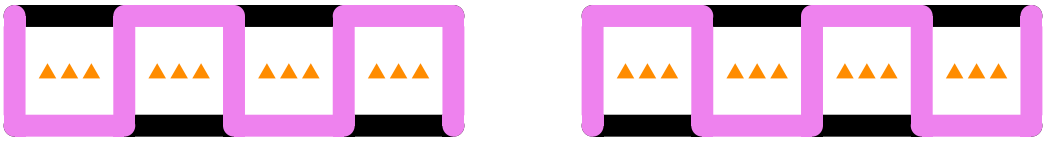
Theorem 7.6.3. *It is NP-complete to solve Witness puzzles containing only 3-triangle clues.*

Proof. We use a variation of the Hamiltonicity framework—reducing from Hamiltonicity in a maximum-degree-3 grid graph, G, by scaling by a constant factor 12, and representing each vertex by a square-region *chamber*—but we will represent edges slightly differently than hallways. Each chamber is a 11×11 grid of cells. Initially, we place 3-triangle clues in the center cell and in every boundary cell except the four corners. Because the scale factor is 12, there is exactly one row or column of empty cells between adjacent chambers. If there is an edge between vertices in G corresponding to two chambers, then we remove the 3-triangle clues from the two cells adjacent to the center cell of the walls of both chambers (i.e., the fourth and sixth cells of each wall), as depicted in Figure 7-34.

The rightmost bottommost chamber is the special start/end chamber. We remove the center 3-triangle clue on its bottom edge, and place the start circle and end cap just below, as can be seen in the full graph instance shown in Figure 7-35.

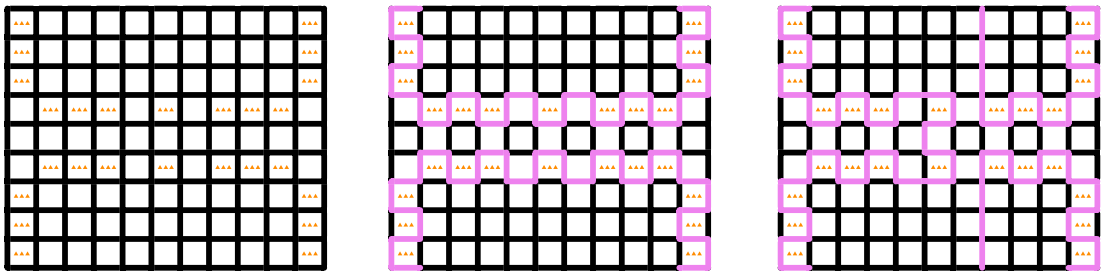
Now we prove that every solution to this Witness puzzle corresponds to a Hamiltonian path. The key observation is that k consecutive 3-triangle clues (in a row or column) force the solution path to have one of two local zig-zag patterns, shown in Figure 7-33.

Each chamber gadget is surrounded by such consecutive sequences of 3-triangle clues. Pairs of these sequences meet diagonally adjacently at each corner of the chamber, which in fact forces the solution path to use a particular zig-zag for these two sequences. As a result, the solution path's behavior is forced around all 3-triangle



(a) One possible local solution. (b) The other possible local solution.

Figure 7-33: The only local solutions to a string of contiguous 3-triangle clues.



(a) Boundary between two chambers (unsolved). (b) Solution when edge is unused. (c) Solution when edge is used.

Figure 7-34: The boundary between two chambers with an edge between them.

clues except the lone clues in the center of each side of a chamber corresponding to an edge in G.

Every chamber must be visited in order to satisfy its center 3-labeled cell. When a chamber is visited, all of its 3-labeled wall cells can be satisfied by following the walls around from the entry point to just before the intended exit point, then traversing through the center 3-labeled cell back to the other side of the entry point, and then around to and out the exit point, as shown in each chamber of Figure 7-35.

If there is an edge in G between vertices corresponding to two chambers a and b, then the solution path can travel from a to b (or vice versa) as shown in Figure 7-34c.

Examining Figure 7-32, a simple case analysis shows that the path satisfying the two 3-triangle cells comprising each corner of a chamber is completely forced, and this in turn forces the path around the rest of the 3-labeled cells comprising the walls. Similarly, case analysis on the middle of the walls in Figure 7-34 shows that the path cannot escape to another chamber, so the path can only travel between adjacent chambers. Finally, there isn't enough space to use an edge more than once, and therefore exactly two edges incident to each chamber must be traversed, because G has max degree 3 and if the solution path traverses only one edge incident to a chamber, it could not have left that chamber and therefore could not have made it to the end cap. □

7.7 Polyominoes

This section covers various types of *polyomino* and *antipolyomino* clues, giving both positive and negative results. Polyomino clues can generally be characterized by the

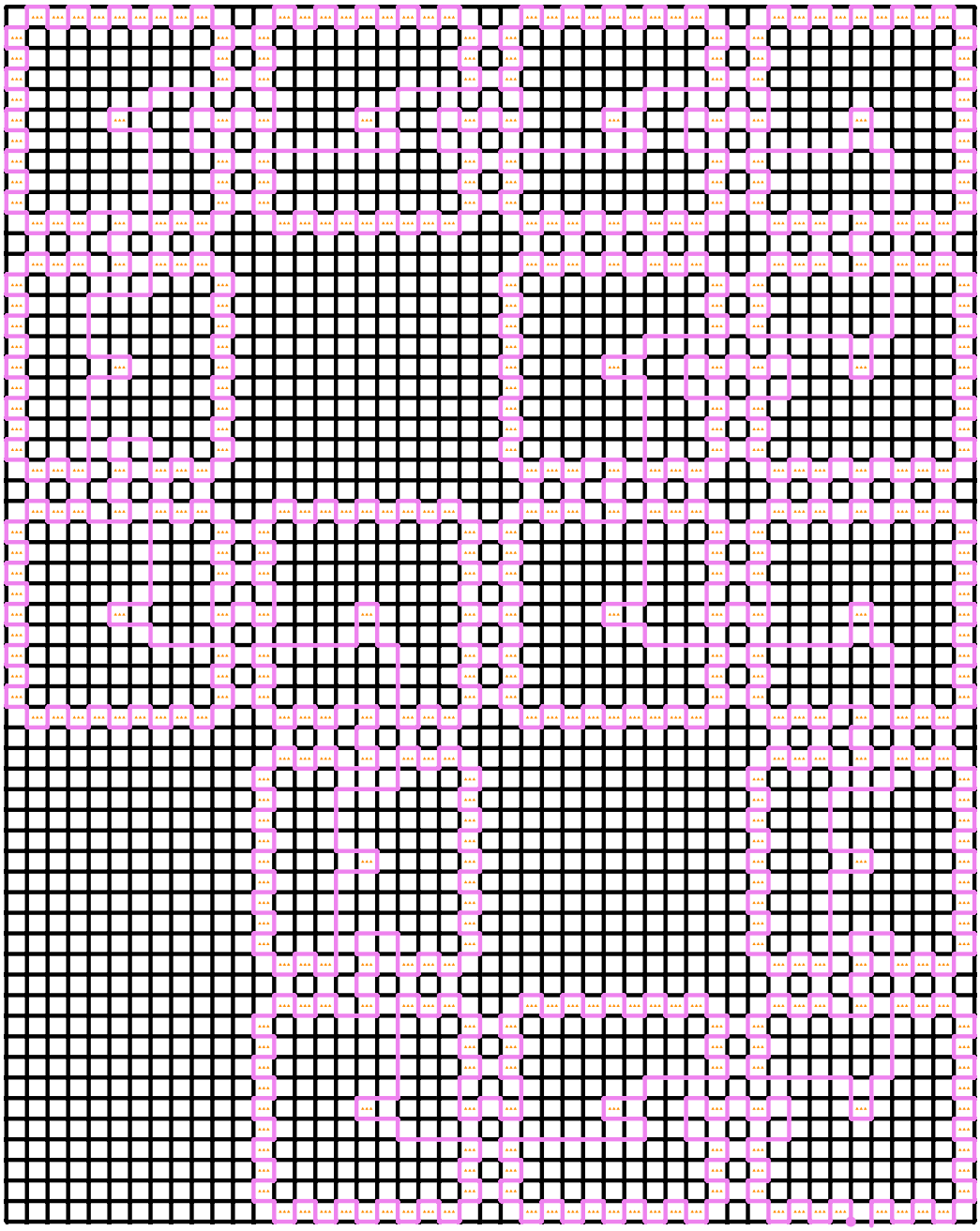


Figure 7-35: A complete 3-triangles Hamiltonicity framework instance with solution.

size and shape of the polyomino and whether or not they can be rotated (vs.). For each region, it must be possible to place all polyominoes and antipolyominoes depicted in that region's clues (each placed as exactly one copy, not necessarily within the region) so that, for some $i \in \{0, 1\}$, each cell inside the region is covered by exactly i more polyominoes than antipolyominoes and each cell outside the region is covered by the same number of polyominoes and antipolyominoes. In Section 7.7.1, we prove that monomino clues alone can be solved in polynomial time, even in the presence of broken edges, generalizing a result of [90]. In Sections 7.7.2, 7.7.3, and 7.7.4 we

give several negative results showing that some of the simplest (anti)polyomino clues suffice for NP-completeness.

7.7.1 Monominoes and Broken Edges

In this section, we prove the following theorem:

Theorem 7.7.1. *Witness puzzles containing only monominoes and broken edges are in P.*

Concurrent work [90] gives a polynomial-time algorithm for Witness puzzles where *every* cell has a square clue of one of two colors. Such puzzles are equivalent to puzzles with only monomino clues, by replacing one color of square with monominoes and the other color with empty cells (or vice versa for the reverse reduction). The only constraint on the two puzzle types is that there can be no region with a mix of square colors or, equivalently, monomino clues and empty cells. Accordingly, our proof of Theorem 7.7.1 is similar. In our case, we reduce to the result from Section 7.3.4 that path finding is easy with broken edges and forced edges on the boundary of the puzzle, which we considered in Section 7.3.4 in the context of boundary edge hexagons.

Proof. When a puzzle contains only monomino clues and broken edges, every region outlined by the solution path that contains a monomino clue must have a monomino clue in every cell in the region to satisfy the polyomino area constraint. Any solution path must therefore include all of the edges shared between an empty cell and a cell containing a monomino clue. If any of these forced edges is broken, or the forced edges form a cycle, then we can immediately reject the puzzle as a NO instance. In particular, we reject if any connected component of monomino clues either does not touch the boundary or has a hole of empty cells. The solution path is forced to trace the outline of each clue-containing component except where it meets the boundary. Thus, the problem reduces to connecting the endpoints of these forced outline paths into a path from the start circle to the end cap.

Some connected components of monomino clues divide the puzzle into disconnected groups of cells. We call these groups *divisions*, and the connected components of clues that give rise to them *dividers*. A division may contain more than one region, but a region is never split across divisions. Call a division *empty* if it does not contain a monomino clue, the start circle, or the end cap, and call it *nonempty* otherwise. Because of our rejection rule above, every division is simply connected.

When a solution path visits the forced outline edges of a divider (from one of the incident divisions), the path must visit all of the forced edges before entering one of the incident divisions, because it can never exit that division without revisiting a vertex. In particular, if any divider neighbors three or more nonempty divisions, then we reject the puzzle as a NO instance: only two divisions can be visited by a solution path (before and after the forced edges, respectively), and each nonempty division must be visited to visit either some forced edge, the start circle, and/or the end cap.

Because each divider neighbors at most two nonempty divisions, we can define a graph where nodes correspond to the nonempty divisions and edges correspond to

dividers. Each edge in this graph can only be traversed once by the solution path because doing so consumes all the forced edges surrounding that divider. Therefore if this graph is not a path, or if the endpoints of this path are not the divisions containing the start vertex and end cap, respectively, then we reject the puzzle as a NO instance, because no path from the start vertex to the end cap can visit all of the nonempty divisions while using each adjacency graph edge only once.

For each division D, let F_D be forced edges between an empty cell and a cell containing a monomino clue within the division, that is, the subset of the edge outline separating D from its neighboring dividers and the nonboundary edge outline of any nondivider components of monomino clues in the division. Because there is a forced edge where each divider's outline touches the puzzle boundary, the path can enter and exit this division at each of two possible vertices (at opposite ends of the dividers). The start circle is the sole entry vertex for the first division and the end cap is the sole exit vertex for the last division.

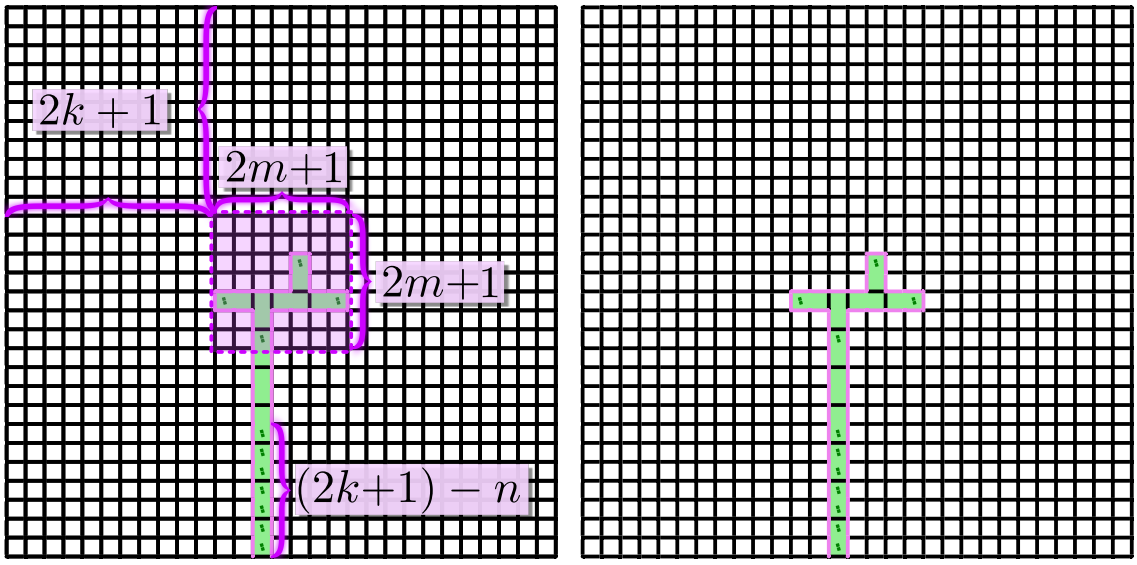
We build a connectivity graph G whose nodes represent the entry and exit vertices of all nonempty divisions (including the start circle and end cap). For each division D, let C_D be the subset of empty cells in D. Set C_D is connected because any set of cells (containing monominoes) separating C_D into two components would be a divider. By our first rejection rule, C_D is also simply connected. Every edge in F_D is incident to a single empty cell, so F_D is a subset of the edge outline of C_D. Hence, (C_D, F_D) is a forced division as defined in Section 7.3.4. For each pair (s, t) of entry and exit vertices of D, respectively, we apply Theorem 7.3.7 to decide whether there is a path from s and t entirely within C and traversing all forced edges in F_D. Whenever there is such a path, we add an edge in G between the nodes corresponding to s and t.

The forced edges of each divider, plus some puzzle boundary edges interior to the divider, form paths connecting each possible exit vertex to an entry vertex in the next division, and we add the corresponding edges to G. Then we can find a puzzle solution by searching for a path in G from the start circle to the end cap, and replacing each edge from that path in G with the Witness puzzle path it represents. If there is no such path in G, then the Witness puzzle has no solution. □

7.7.2 Rotatable Dominoes

Theorem 7.7.2. *It is NP-complete to solve Witness puzzles containing only rotatable dominoes.*

Proof. We reduce from the rectilinear Steiner tree problem [64]: given n points with integer coordinates (x'_i, y'_i) in the plane, $i \in \{1, 2, \ldots, n\}$, and given an integer k, decide whether there exists a rectilinear tree connecting the n points having total length at most k. Hanan's Lemma [71, Theorem 4] states that this is equivalent to the existence of a rectilinear tree of length at most k on the integer grid, or even on the grid formed by a horizontal and a vertical line through each of the n points. By globally translating the points by $(-\min_i x'_i, -\min_i y'_i)$, we can find an equivalent instance of n points (x_i, y_i), $i \in \{1, 2, \ldots, n\}$, with all coordinates nonnegative and where $\min_i y_i = 0$. Let $m = \max_i\{x_i, y_i\}$. Because this problem is strongly NP-hard



(a) Unsolved. (b) Solved.

Figure 7-36: Example of the rotatable dominoes NP-completeness proof, where $k = 5$, $n = 4$, and $m = 3$.

[64], [65, Problem ND13, p. 209], we can assume $m = n^{O(1)}$.

We construct a $(2m+4k+3)\times(2m+4k+3)$ Witness puzzle with a $(2m+1)\times(2m+1)$ center C_m; refer to Figure 7-36. Within C_m, we include exactly n dominoes: for every point (x_i, y_i) in the rectilinear Steiner tree input, we include a domino in cell $(2x_i, 2y_i)$, where these coordinates are relative to the lower-left corner of C_m. Consider the leftmost point with a y coordinate of zero, whose corresponding domino clue is on the bottom row of C_m; call this cell the *root*. Define the *trunk* to be the $2k + 1$ cells in the same column as and below the root. We include exactly $(2k + 1) - n$ dominoes in the trunk, starting at the bottom of the board. We place the start circle and end cap on the boundary vertices adjacent to the trunk (in either order).

Steiner tree → Witness solution. First suppose that the rectilinear Steiner tree instance has a solution S with total length at most k. By Hanan's Lemma, we can assume that the vertices and horizontal/vertical segments of S lie on the integer grid, with all y coordinates nonnegative. Scale the tree S by a factor of 2, and consider the set P of integer grid points that lie on the vertices or segments of this scaled tree. If two points in P have distance 1 on the grid, then (by the scaling) they must have come from a common segment of the tree S. Thus, P induces a grid graph that is a tree. This grid tree is the dual of a set $C \subseteq C_m$ of cells. Because S is a Steiner tree on the n points, C contains all the cells with domino clues in C_m. In particular, one vertex of S corresponds to the root clue, and we define this vertex to be the root of the tree S.

Consider a segment (u, v) of the tree S of length ℓ, oriented so that u is closer than v to the root. Vertices u and v map to cells c_u and c_v in C in the same row or column, with $2\ell - 1$ cells from C between them. We can tile these $2\ell - 1$ cells together with c_v

with exactly ℓ dominoes. By our orientation of segments, dominoes from different segments do not overlap, so we obtain a tiling of C except the root using exactly L dominoes.

Next we construct a set C' of cells, by adding to C the $2k+1$ cells in the trunk (which includes all remaining domino clues). The trunk has an odd number of cells in a single column, so together with the root it can be tiled by exactly $k+1$ dominoes. In total, we have tiled C' with $L+k+1$ dominoes.

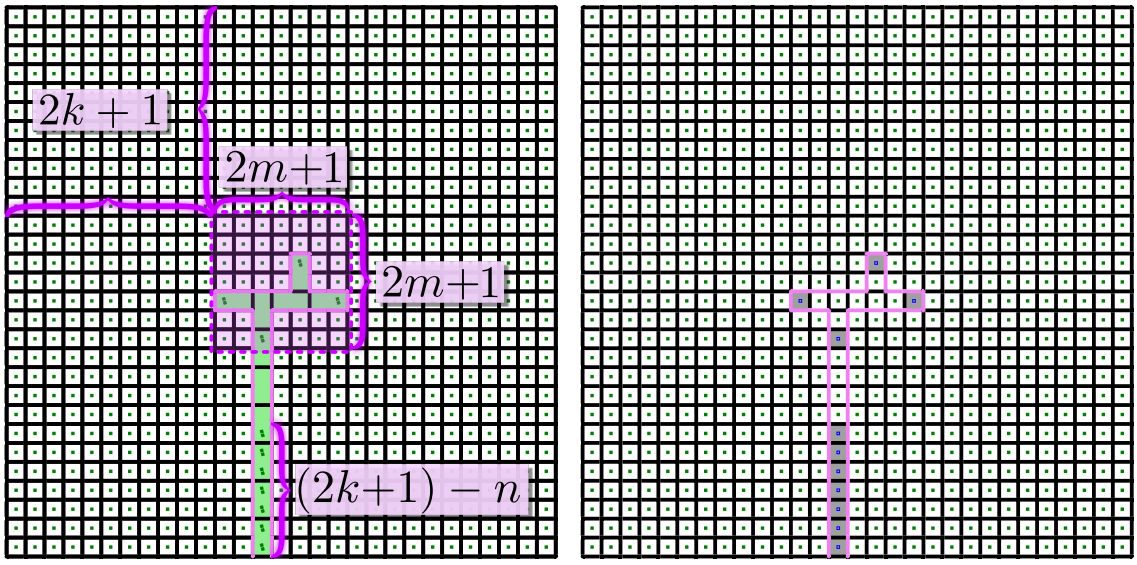
Finally, we construct a set C'' by adding to C' the cells of $k-L$ disjoint dominoes immediately right of the trunk (e.g., starting from the bottom). Set C'' can be tiled by exactly $2k+1$ dominoes, is connected and hole-free, and contains all domino clues. The border of C'' is therefore a cycle on the integer grid; removing the bottommost edge of the trunk leaves a path which solves the Witness puzzle.

Witness solution $\rightarrow$ Steiner tree. Now suppose that the Witness puzzle has a solution.

Consider the cells containing domino clues in C_m. Every such cell is in a region that borders the boundary of the board and can be tiled by dominoes equal in number to the number of domino clues inside that region. Each such region has area strictly greater than $2k+1$ because the region must reach from the boundary of the board into C_m. But the total number of domino clues in the whole board is $2k+1$, so the total area in all such regions is $2(2k+1)$. Therefore, there is exactly one region A that contains all the domino clues in C_m.

We now show that there is a connected subset B_m of C_m, of size at most $2k+1$, containing all the domino clues of C_m. We start with $B_{m+(2k+1)} = A$ and inductively define sets B_i for all $m \leq i \leq m+(2k+1)$ such that B_i (1) is a subset of the $(2i+1)\times(2i+1)$ center C_i of the board, (2) is connected, (3) contains all the domino clues in C_m, (4) has size at most $i-m+(2k+1)$, and (5) is connected to one of the four boundary sides of C_i.

Given B_{i+1}, define B_i to include all the cells of B_{i+1} that are in C_i plus, for each cell of B_{i+1} on the boundary of C_{i+1}, the nearest cell on the boundary of C_i. (1) B_i is, by definition, a subset of C_i. (2) To prove B_i connected, consider the $(2i+3)\times(2i+3)$ grid graph representing the dual of C_{i+1}, with cells of B_{i+1} *marked*. Then transforming from B_{i+1} to B_i is equivalent to contracting dual edges with exactly one endpoint on the boundary of C_{i+1} and the eight dual edges with one endpoint at a corner of C_{i+1} and then marking each contracted vertex if and only if at least one of the endpoints of the contracted edge was marked. This transformation preserves connectivity of the marked vertices, so B_i is connected. (3) B_{i+1} contained all the domino clues in C_m, so because those are all in C_i, B_i still contains them. (4) Because B_{i+1} was connected to a boundary of the board and C_m, it contains at least one adjacent pair of squares on the boundaries of C_{i+1} and C_i, so at least one dual edge with both endpoints marked is contracted, making B_i smaller than B_{i+1} by at least one square. Hence B_i has size at most $|B_{i+1}|-1 \leq (i+1-m+(2k+1))-1 = i-m+(2k+1)$. (5) By the existence of the same edge, B_{i+1} and hence also B_i has a cell on the boundary of C_i, completing the induction.



(a) Unsolved. (b) Solved.

Figure 7-37: Example of the monominoes + antimonominoes NP-completeness proof, converted from Figure 7-36.

We claim that any spanning tree T of the dual graph of B_m is a factor-2 scaling of a Steiner tree of length at most k. Any such T has $|B_m| \leq 2k + 1$ vertices and hence at most $2k$ edges. Each edge has length 1 in the grid, so T is a tree of total rectilinear length at most $2k$. In addition, we know that T touches the points with coordinates $(2x_i, 2y_i)$ for $i \in \{1, 2, \ldots, n\}$. Scaling T by a factor of $\frac{1}{2}$ yields a tree of total length at most k which touches the points with coordinates (x_i, y_i) for $i \in \{1, 2, \ldots, n\}$, solving the given rectilinear Steiner tree instance. □

7.7.3 Monominoes + Antimonominoes

Theorem 7.7.3. *It is NP-complete to solve Witness puzzles containing only monominoes and antimonominoes.*

Proof. Like Theorem 7.7.2, we reduce from the rectilinear Steiner tree problem; refer to Figure 7-37. Given an instance I of this problem, let W be the Witness puzzle that is constructed from I by the reduction in Theorem 7.7.2; W contains only domino clues. Modify W as follows: in each cell with a domino clue, instead place an antimonomino clue, and in each cell that was previously empty, add a monomino clue. This modified puzzle, W', which contains only monomino and antimonomino clues, is the output of our reduction.

Consider any region in a solution of W' which contains an antimonomino. The total area of the antipolyomino clues in this region is less than the total area of the region. Thus, the total area of the clues must be exactly zero in order to satisfy the polyomino constraint. In other words, the area of the region must be exactly double the number of antimonominoes in the region.

Therefore every region must either contain only monominoes, or its area must equal twice the number of antimonomino clues (i.e., the region must contain an equal number of monominoes and antimonominoes). In fact, as long as a path satisfies this area condition, it is a solution.

This condition is strictly weaker than that imposed by W, so any solution to W is also a solution to W'. So if the Steiner tree instance is solvable, so is W'.

Conversely, suppose W' is solvable. In the **Witness solution → Steiner tree** direction of the proof of Theorem 7.7.2, we relied only on area conditions: that the area of every region with a domino equals twice the number of domino clues in the region. This exactly corresponds to the area conditions imposed by W', so an identical proof shows that any solution to W' can be converted into a Steiner tree that solves I. □

7.7.4 Nonrotatable Dominoes

Theorem 7.7.4. *It is NP-complete to solve Witness puzzles containing only nonrotatable vertical dominoes.*

Proof. We reduce from planar rectilinear monotone 3SAT [87], which is a special case of 3SAT where the formula can be represented by the following planar structure. Each variable has an associated *variable segment* on the x axis, and the variable segments are disjoint. Each clause has an associated *clause segment* which is horizontal but not on the x axis. Each clause segment is connected by vertical segments to the variable segments corresponding to the literals in the clause. Clause segments above the x axis represent *positive* clauses whose literals are all positive (not negated), and clause segments below the x axis represent *negative* clauses whose literals are all negated.

We will convert such an instance into a Witness puzzle with nonrotatable vertical domino clues. Our construction will have the following properties:

Property 7.7.1. *No two domino clues are in vertically adjacent cells.*

Property 7.7.2. *There is exactly one cell ℓ in the leftmost column that contains a domino clue; and there are no other domino clue cells in the rightmost column, top two rows, or bottom two rows.*

Property 7.7.3. *The start circle is the bottom-left corner of ℓ, and the end cap the top-left corner of ℓ.*

Domino tiling problem. Before describing the construction, we show how solving a Witness puzzle satisfying the properties above can be rephrased as solving a special kind of domino tiling problem: choosing one domino tile to cover each domino-clue cell (and the cell either above or below it) such that the boundary of the union of the domino tiles forms a simple cycle.

Any solution path to the constructed Witness puzzle must divide the board into regions such that each region containing domino clues can be exactly tiled by a number of domino tiles equal to the number of contained domino clues. In such a tiling, every domino-clue cell must be covered by exactly one domino tile. By Property 7.7.1, no

domino tile can cover multiple cells with domino clues. Therefore, there is a one-to-one correspondence between domino clues and domino tiles in any solution, where each domino tile covers the cell with the corresponding domino clue. Equivalently, a valid such tiling can be specified by choosing whether the domino tile covering a domino-clue cell also covers the cell above or below.

By Property 7.7.2, exactly one domino tile can be adjacent to the board boundary. Every tiled region in the solution must touch the board boundary, and must therefore contain the unique domino tile adjacent to the board boundary. Thus, there is exactly one region containing all the domino clues, which must be tiled by all corresponding domino tiles. The solution path must (as a subpath) outline the part of the boundary of this region that lies interior to the board.

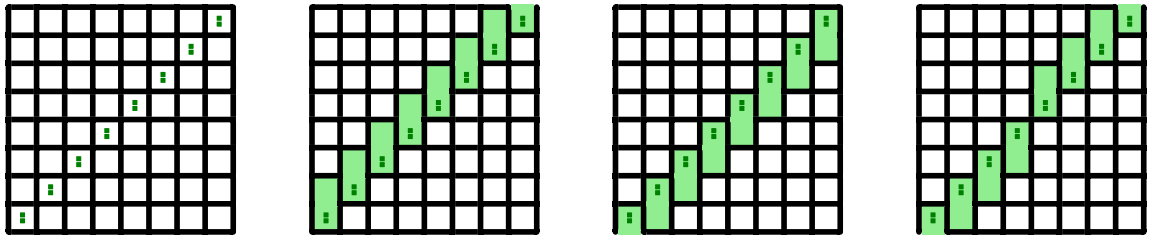
We claim that solving the Witness puzzle is equivalent to choosing one domino tile to cover each domino-clue cell (and the cell either above or below it) such that the boundary of the union of the domino tiles forms a simple cycle. On the one hand, any simple boundary cycle can be converted into a solution path by removing the left edge of ℓ, by Property 7.7.3. On the other hand, for any solution path, we can take the subpath outlining the tiled region and add two edges (the left edges of the domino tile covering ℓ) to form a simple cycle that is the boundary of the tiled region.

Variable gadget. For a variable appearing in k clauses, its *variable gadget* consists of $4k$ domino clues in a positive-slope diagonal line, with each clue immediately above and to the right of the previous one. Divide these clues into k *blocks* of four contiguous clues each. Each block will be used to connect to one of the clause gadgets corresponding to a clause in which the variable appears. Suppose the variable appears in k^+ positive clauses $C_1^+, C_2^+, \ldots, C_{k^+}^+$ and in k^- negative clauses $C_1^-, C_2^-, \ldots, C_{k^-}^-$ (so $k^+ + k^- = k$). We assign the leftmost k^- blocks for connections to the negative clauses, followed by the remaining k^+ blocks for connections to the positive clauses.

Dominoes in the variable gadget that have the path extending one cell above them ("up" dominoes) represent an assignment of TRUE to the corresponding variable and dominoes with the path extending one cell below them ("down" dominoes) represent an assignment of FALSE. We implement the connections to the clause gadgets such that all other domino clues are at least three cells away from the diagonal. Under this property, the path may switch mid-variable-gadget from up to down dominoes, but not from down to up (the path would intersect itself). Because connections to the negative clauses precede connections to the positive clauses, switching from up to down is never beneficial. See Figure 7-38.

Clause gadget. The clause gadget is a horizontal line of domino clues with horizontal extent exactly matching the extent of the blocks allocated from variable gadgets for connections to this clause. See Figure 7-39.

Layout and connections. Given an instance I of planar rectilinear monotone 3SAT, we construct a Witness puzzle W whose only clues are 1×2 nonrotatable dominoes; refer to Figure 7-41.



(a) Unset. (b) True. (c) False. (d) False to true (invalid).

Figure 7-38: A 2-block-wide variable gadget and its possible domino tilings. Switching from false to true mid-gadget is impossible.

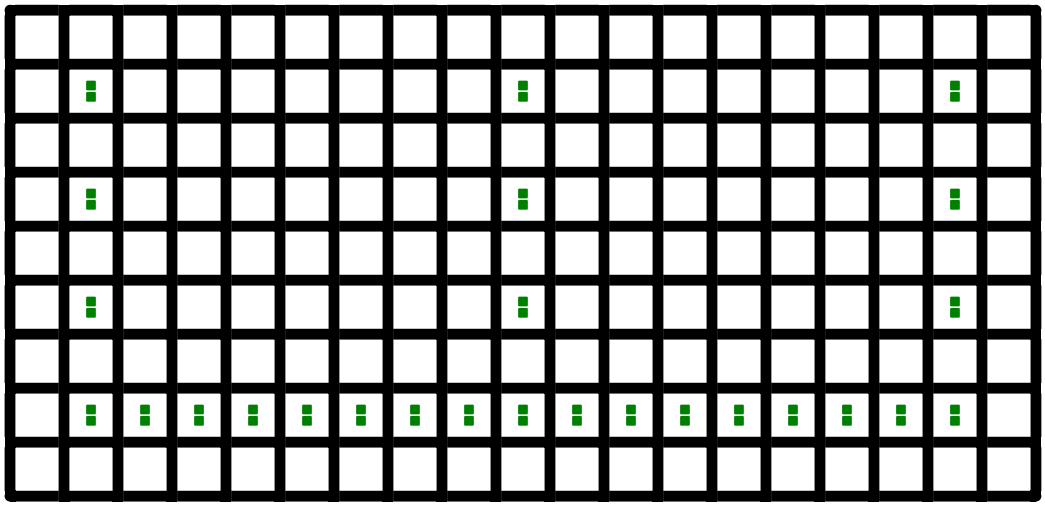
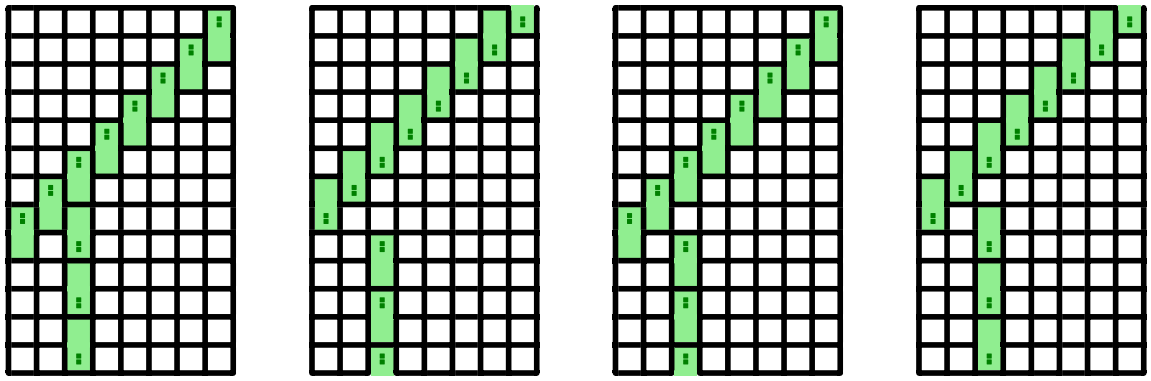


Figure 7-39: A negative clause gadget and its three literal connections.



(a) Satisfied. (b) Unsatisfied. (c) Invalid. (d) Invalid.

Figure 7-40: A negative literal connection to a variable gadget. The other two possibilities are invalid because the solution path would touch itself.

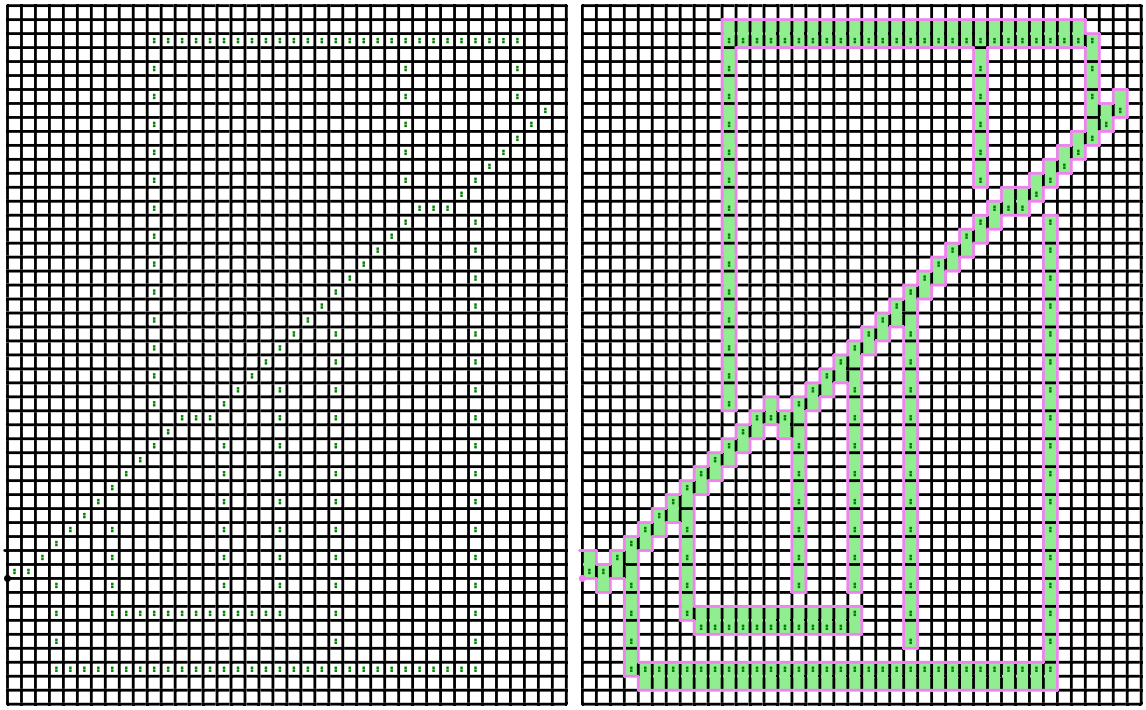
We place the variable gadgets in the same order as the corresponding variable intervals appear on the x-axis in I. The bottom-left clue of each variable gadget is two cells to the right of the top-right clue of the previous variable gadget, with a domino clue in the cell between allowing the path to switch freely from up to down or down to up at the transition point between gadgets.

We place the clause gadgets with their left and right ends aligned with the left and right ends of the outermost blocks in the variable gadgets they connect to and sorted vertically with respect to the variable gadgets and other clause gadgets in the same way as the corresponding clause line segments in I. We scale the vertical distance between clause line segments such that every pair of clause gadgets is separated by a vertical distance of at least five cells. In particular, positive (negative) clause gadgets are above (below) the variable gadgets.

Each connection between a variable and a clause is allocated to one block of four clues in the corresponding variable gadget. Of the two center columns of that block, we choose the column in which the variable's clue's vertical parity is opposite to the vertical parity of the line of clues that constitute the clause gadget. We place domino clues in every square of the clause line's vertical parity in that column between the clause and variable gadgets, except the one such square adjacent to a clue in the variable. See Figure 7-40.

We place one domino clue in the square S immediately to the left of the bottom-left clue in the leftmost variable gadget. The left boundary of the board is immediately left of S, while the other three boundaries are three empty cells away from the most extreme clue in that direction. We put the start circle and end cap at the bottom-left and top-left vertices of S.

Witness solution $\rightarrow$ Satisfying assignment. Suppose the Witness puzzle produced by our reduction has a solution: a path from the start circle to the end cap such that each induced region that contains domino clues is exactly tiled by that many (vertical) dominoes. As a result, every square containing a domino clue is covered



(a) The puzzle. (b) The solution.

Figure 7-41: A Witness puzzle produced from $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg y)$ and its solution (x and y are FALSE, z is TRUE). Shaded cells show the domino tiling on the interior of the path.

by a domino. Because no two domino clues are vertically adjacent, we can associate each clue with the domino that covers it. By this correspondence, those are the only dominoes in the tiling.

By construction, the domino covering S is the only domino touching the boundary of the board (all other domino clues are too far from the boundary), and so only one region containing domino clues touches the boundary. Because solution paths are simple paths, every region in a Witness puzzle touches the boundary, so all domino clues are in a single region.

We can read off a truth assignment from the position of the dominoes covering the clues in the variable gadget: TRUE if the domino is up and FALSE if the domino is down. As explained above, the path may switch from up to down in the middle of a variable gadget, but such switches do not help in satisfying clauses, so we take the first domino in each variable gadget to determine the assignment of that variable.

It remains to show that this assignment satisfies the formula. Because all domino clues are in a single region, the clause gadgets are all in the region. For the path to reach and encircle a positive (negative) clause gadget, it must follow a connection up (down) from a variable gadget domino that is up (down) and then return along the other side of that connection; any other path to or from a clause gadget results in a region too large to cover with the domino clues. That variable gadget, connection and clause gadget correspond to the variable that satisfies the literal in that clause. Thus, the recovered truth assignment satisfies the formula.

Satisfying assignment → Witness solution. Given a satisfying assignment, we can construct a domino tiling of a tree of cells rooted at S, the boundary of which is a solution path. Refer to Figure 7-41. We begin by covering each domino clue in the variable gadgets with a domino positioned according to the value of that variable (up for TRUE, down for FALSE). We cover the connections corresponding to satisfied positive literals with up dominoes and satisfied negative literals with down dominoes, such that the connection connects the variable and clause gadgets. Then for all but one of the satisfying literals, we move the two dominoes furthest from the clause gadget to the other orientation (further away from the variable gadget), disconnecting the connection to avoid forming a closed loop that would make drawing the solution path impossible. We cover the unsatisfied positive literals with down dominoes and satisfied negative literals with up dominoes, leaving them connected to the clause gadget but not the variable gadget. Then we cover the remaining clause gadget domino clues with up dominoes for positive clauses and down dominoes for negative clauses. See Figure 7-42.

This domino tiling is a tree: the variable gadgets are connected in a line because they have adjacent domino clues, exactly one connection connects each clause gadget to exactly one variable gadget, and the remaining connections connect to either their variable gadget or their clause gadget (not both). S is covered by the tiling because it contains a domino clue. Then the boundary of this domino tiling (deleting S's left edge) is a solution path in the Witness puzzle. □

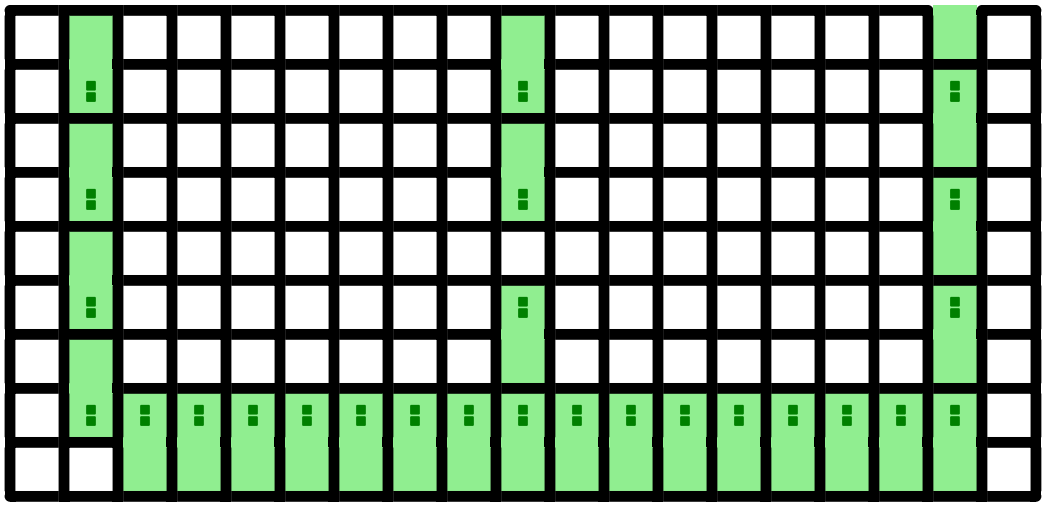


Figure 7-42: A negative clause gadget with the left two literals satisfied and the right literal unsatisfied. The left and middle literal connections connect to their variable gadgets, while the right literal does not. The middle literal could be used to satisfy this clause (instead of the left literal) by moving the bottom two dominoes in that column up and the bottom two dominoes of the left literal's connection column down.

7.8 Antibodies

An antibody () eliminates itself and one other clue in its region. For the antibody to be satisfied, this region must *not* be satisfied without eliminating a clue; that is, the antibody must be necessary. An antibody may be colored, but its color does not restrict which clues it can eliminate.[7] Very few Witness puzzles contain multiple antibodies, making the formal rules for the interactions between antibodies not fully determined by the in-game puzzles. We believe the following interpretation is a natural one: each antibody increments a count of clues that must necessarily be unsatisfied for their containing region to be satisfied. If there are k antibodies in a region, then there must be k clues which can be eliminated such that those k clues were unsatisfied and all other clues were satisfied; furthermore, there must not have been a set of fewer than k unsatisfied clues such that all other clues are satisfied[8]. Antibodies cannot eliminate other antibodies. The choice of clue to eliminate need not be unique; for instance, a region with three white stars and one antibody is satisfied, even though the stars are not distinguished. Formally:

Definition 7.8.1 (Simultaneous Antibodies)**.** *A region with k antibody clues is satisfied if and only if there exists a set S of k non-antibody clues such that eliminating all clues in S and all k antibodies leaves the region satisfied, and there does not exist*

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

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

a set S' of non-antibody clues with $|S'| < k$ such that eliminating all clues in S' and only $|S'|$ of the antibodies leaves the region satisfied.

7.8.1 Positive Results

This section gives algorithms and arguments showing various types of Witness puzzles can be solved in NP or Σ_2. We begin with a simple case: puzzles with no antibodies are in NP. Then we show that puzzles with antibodies but without polyominoes or antipolyominoes is also in NP. Next we restrict to a single antibody and add polyomino clues back in, which remains in NP. Finally, we show that puzzles with all clue types are in Σ_2. These last three results provide tight matches for our lower bounds in Section 7.8.2.

Observation 7.8.2. *Witness puzzles containing all clue types except antibodies are in NP.*

Proof. For all clues except polyominoes and antipolyominoes, a witness to such puzzles is the solution path. (Because the solution path must be simple, it has linear size.) For each region defined by the solution path that has a polyomino or antipolyomino clue, we also include the (anti)polyomino packing as part of the witness. This packing can be encoded as (x, y) coordinates for the topmost leftmost pixel of each polyomino and each antipolyomino. These coordinates can be assumed to be polynomial in the input size: given any solution packing, delete any rows or columns not intersecting the board nor the bounding box of any polyomino. Because some polyomino in the packing covers a cell of the region in the board, the other polyominoes can be no farther away than the number of polyominoes times the maximum size of the board or a polyomino's bounding box. □

Theorem 7.8.3. *Witness puzzles containing all clue types except polyominoes and antipolyominoes are in NP.*

Proof. We give a polynomial-time algorithm to verify a claimed solution path. Each region induced by the path can be verified separately because no clues can interact with clues in other regions. For regions not containing antibodies, we simply check that all clues in the region are satisfied, which can be done in polynomial time.

An *elimination set* is a set of eliminated non-antibody clues. An elimination set is *proper* if it has size equal to the number of antibodies in the region and *improper* if it is smaller. A region is satisfied under an elimination set if the non-eliminated non-antibody clues are satisfied; we say the region is *properly satisfied* if the elimination set is proper and *improperly satisfied* otherwise. Then the region is satisfied if it is properly satisfied under at least one elimination set (all of the antibodies can be used) and not improperly satisfied under any elimination set (all of the antibodies are necessary).

We proceed by *marking* non-antibody clues that must be in any elimination set that satisfies the region (properly or improperly). When we have a choice of clues to mark, we mark as few clues as possible, or if it cannot be known how many clues

to mark, we evaluate the subproblem resulting from each choice. After processing all clue types, we check whether the elimination set of the marked clues properly or improperly satisfies the region (or neither).

To begin, we mark all edge and vertex hexagons in the interior of the region (i.e., not visited by the solution path) and all triangle clues not adjacent to the corresponding number of edges in the solution path. Hexagons and triangles do not interact with each other or other clue types, so we have no choices to make here.

The remaining allowed clue types are squares, stars and antibodies, which interact by their color. Within a region, clues of these types with the same color are interchangeable because their location is irrelevant to their satisfaction, so we record only how many clues of each type and color have been marked. We assume for the moment that all antibodies are used and thus eliminate themselves. Of stars and squares, a star can be satisfied by pairing with a square of the same color, but the presence or absence of stars has no effect on squares, so we consider squares first. Squares are only satisfied if the region is monochromatic in squares, so we must mark all clues of all but one color of square. We evaluate the subproblems resulting from each choice of surviving square color.

Only clues of the same color are relevant to satisfying stars, so we process each star color independently. For each color of star, we have a choice between marking all stars of that color (zero stars survive), all but one star of that color and all but one square of that color (only possible if the star matches the surviving square color), or all but two stars of that color and all squares of that color (if any). We choose the number of surviving stars that results in marking the fewest clues; if there is a tie, we choose the smaller number of surviving stars.[9]

At this point, all unmarked clues are satisfied. We now check whether the region is satisfied under the elimination set consisting of the marked clues.

- If the region contains more marked clues than antibodies, then the region is not satisfied under this elimination set. We continue with the next choice of square color.

- If this elimination set is improper, then some of the antibodies are not used, contrary to our assumption. Unused antibodies do not eliminate themselves, so they may cause a star to be unsatisfied. Let u be the number of unused antibodies and v be the number of antibody clues not sharing a color with a surviving star clue. If $u > v$, at least one of the unused antibodies causes a star to be unsatisfied, so the region is not satisfied under this elimination set, and we continue with the next choice of square color. Otherwise ($u \leq v$), the region is improperly satisfied under the marked clues, so we reject the solution path. Note that our tie-breaking preference for zero surviving stars maximizes v without decreasing u, ensuring that we detect improper satisfaction.

- If this elimination set is proper, then we need to check one special case to discharge our assumption that all antibodies are used: if the region contains an

[9] We must choose zero surviving stars if possible; between one and two surviving stars there is no difference.

antibody of the same color as a marked star and all other non-antibody clues of that color (if any) are marked, then the region is also satisfied if that star survives and that antibody is unused. In that case, the region is improperly satisfied after that star is removed from the elimination set, so we reject the solution path. Our tie-breaking preference for zero surviving stars ensures that we detect this. That is the only way an unused antibody can satisfy another clue, so otherwise our assumption that all antibodies are used holds, and we record that the region can be properly satisfied. We still have to evaluate any remaining choices of square color to verify they do not allow the region to be improperly satisfied.

If, after checking all choices of surviving square color, we did not reject the solution path, and the region was properly satisfied under at least one elimination set, then this region is satisfied. The path is a solution exactly when all regions are satisfied.

It remains to show this algorithm runs in polynomial time. There are only polynomially many regions to check. For each region, there are only polynomially many squares in the region, and thus at most polynomially many colors of squares. For each color of square, we do work proportional to the number of colors of stars, but similarly there are only polynomially many stars in the region and so only polynomially many colors of stars. The work done at the end of each iteration requires only counting the number of clues of a particular type or color that are marked or unmarked. Then an NP algorithm to solve Witness puzzles containing all clue types except polyominoes and antipolyominoes follows directly from this polynomial-time verification algorithm. □

Theorem 7.8.4. *Witness puzzles containing all clue types except antipolyominoes and for which at least one solution eliminates at most one polyomino in each region are in NP.*

Proof. We give a polynomial-time algorithm to verify a certificate. Similar to Observation 7.8.2, the certificate includes a solution path and a polyomino packing for each region defined by the solution path that has an uneliminated polyomino. In addition, the certificate includes which clue gets eliminated by each antibody, satisfying the promise that at most one polyomino from each region gets eliminated. (The NP-hardness of polyomino packing [44] necessitates this promise; otherwise, verifying a packing resulting from using two antibodies to eliminate two polyominoes would require checking that there is no packing resulting from using just one of the antibodies to eliminate one (larger) polyomino.)

We first verify the packing witnesses. The certificate is invalid if it contains a packing witness for a region with no polyomino clues or if the specified arrangement of the polyominoes does not actually pack the region. Otherwise, for each region containing polyomino clues:

- If the certificate does not provide a packing witness for this region, this region must contain exactly one polyomino clue and one antibody, and the polyomino clue must not have the shape of the entire region (because then the polyomino is satisfied and can't be eliminated); otherwise, the certificate is invalid. We record the antibody as eliminating the polyomino clue.

- If the certificate contains a packing witness omitting the position of more than one polyomino clue (requiring two polyomino clues to be eliminated), the certificate is invalid. (The solution path in the certificate may be a valid solution to the Witness puzzle, but this algorithm cannot verify it. By the promise that there exists a solution eliminating at most one polyomino per region, there is some other certificate attesting that this Witness puzzle is a YES instance that this algorithm can verify.)

- If the certificate contains a packing witness omitting the position of exactly one polyomino clue, this region must contain an antibody; otherwise, this certificate is invalid. We record the antibody as eliminating the omitted polyomino clue. (We know the antibody necessarily eliminates the omitted polyomino clue because otherwise the total area of the polyominoes would be greater than the size of the region.)

- If the certificate contains a packing witness specifying the position of every polyomino clue, the polyominoes enforce no further constraint.

Now consider the Witness puzzle obtained by taking the initial instance and removing all polyomino clues and the antibodies we recorded as eliminating polyomino clues. The resulting puzzle contains neither polyominoes nor antipolyominoes, so we can apply the algorithm given in the proof of Theorem 7.8.3 to verify that the remaining clues in each region are satisfied under the solution path. The certificate is valid exactly when the solution path is valid for the resulting puzzle. Then an NP algorithm to solve Witness puzzles containing all clue types except antipolyominoes and for which at least one solution eliminates at most one polyomino in each region follows immediately from this polynomial-time verification algorithm. □

Theorem 7.8.5. *Witness puzzles containing any set of clue types (including polyominoes, antipolyominoes, and antibodies) are in* Σ_2.

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

Our verification algorithm begins checking the certificate of validity by verifying the packing witnesses and checking that the antibody mapping specifies distinct eliminated clues in the same region as each antibody. Then we remove all antibody clues, polyomino and antipolyomino clues, and eliminated clues from the Witness

puzzle and (like the proof of Theorem 7.8.3) verify in polynomial time that the remaining clues in each region are satisfied under the solution path.

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

7.8.2 Negative Results

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

Problem 7.8.6 (Adversarial-Boundary Edge-Matching)**.** *A* signed color *is a sign (+ or −) together with an element of a set C of colors. Two signed colors* match *if they have the same element of C and the opposite sign. A* tile *is a unit square with a signed color on each of its edges.*

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

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

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

This problem is similar to a known Σ_2-complete problem called *finite tiling extension* [52, 121]. The main difference is that, in our adversarial-boundary edge-matching game, each given tile must be used exactly once, whereas in finite tiling extension, each given tile (type) can be used any number of times (including zero).

Lemma 7.8.7. *Adversarial-boundary edge-matching is Σ_2-hard, with or without tile rotation, even when the first player has a losing strategy.*

See Section 7.13 for the proof. Both our reduction and the related past results [52] are inspired by Berger's famous undecidability result for tiling the infinite plane [27]; the main issue in our case is collecting garbage to ensure every tile gets used exactly once.

Theorem 7.8.8. *It is Σ_2-complete to solve Witness puzzles containing two antibodies and polyominoes.*

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

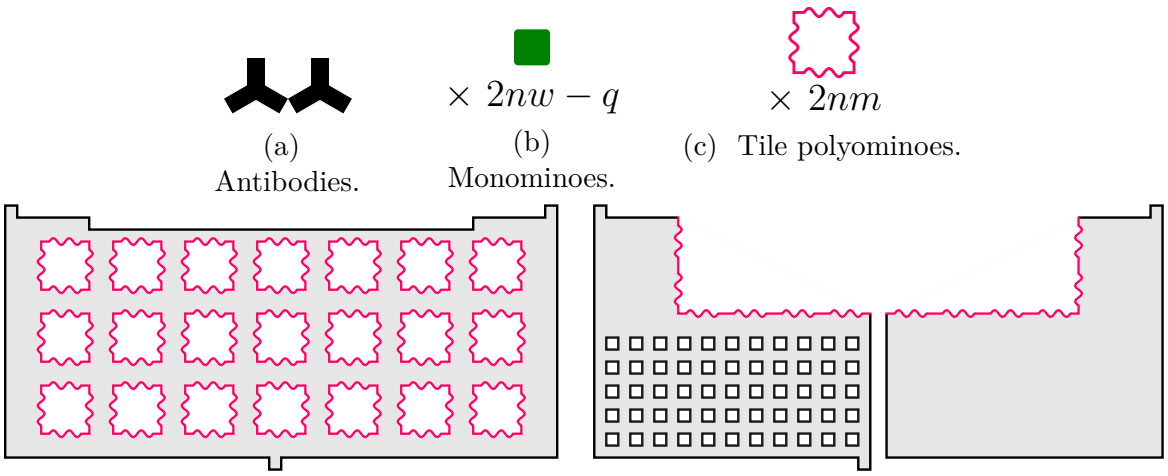
Encoding signed colors. We encode signed colors on the edges of polyominoes in binary as unit-square tabs (for positive colors) or pockets (for negative colors) [44, Figure 7]. If the input adversarial-boundary edge-matching instance has c colors, we need $\lceil \log_2(c+1) \rceil$ bits to encode the color[10]. To prevent pockets at the corners of a tile from overlapping, we do not use the 2×2 squares at each corner to encode colors, so tiles are built out of squares with side length $w = \lceil \log_2(c+1) \rceil + 4$[11].

Clue sets. We consider the clues in the Witness puzzle to be grouped into two clue sets, A and B, which we place far apart on the board. We will argue that any solution path must partition the puzzle into two regions, such that each set is fully contained in one of the regions. The clue sets are shown in Figures 7-43 and 7-44.

Clue set A contains:

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

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



(a) Antibodies.

(b) Monominoes.

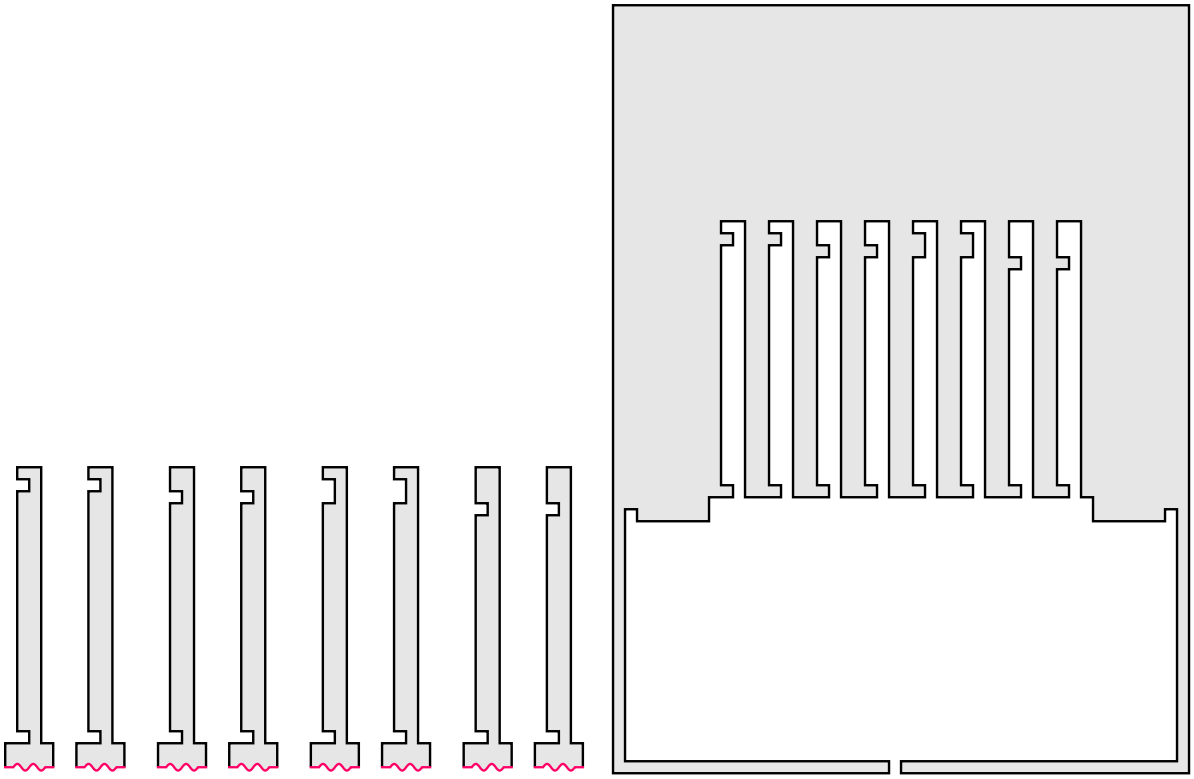
(c) Tile polyominoes.

(d) The medium polyomino.

(e) The left board-frame polyomino.

(f) The right board-frame polyomino.

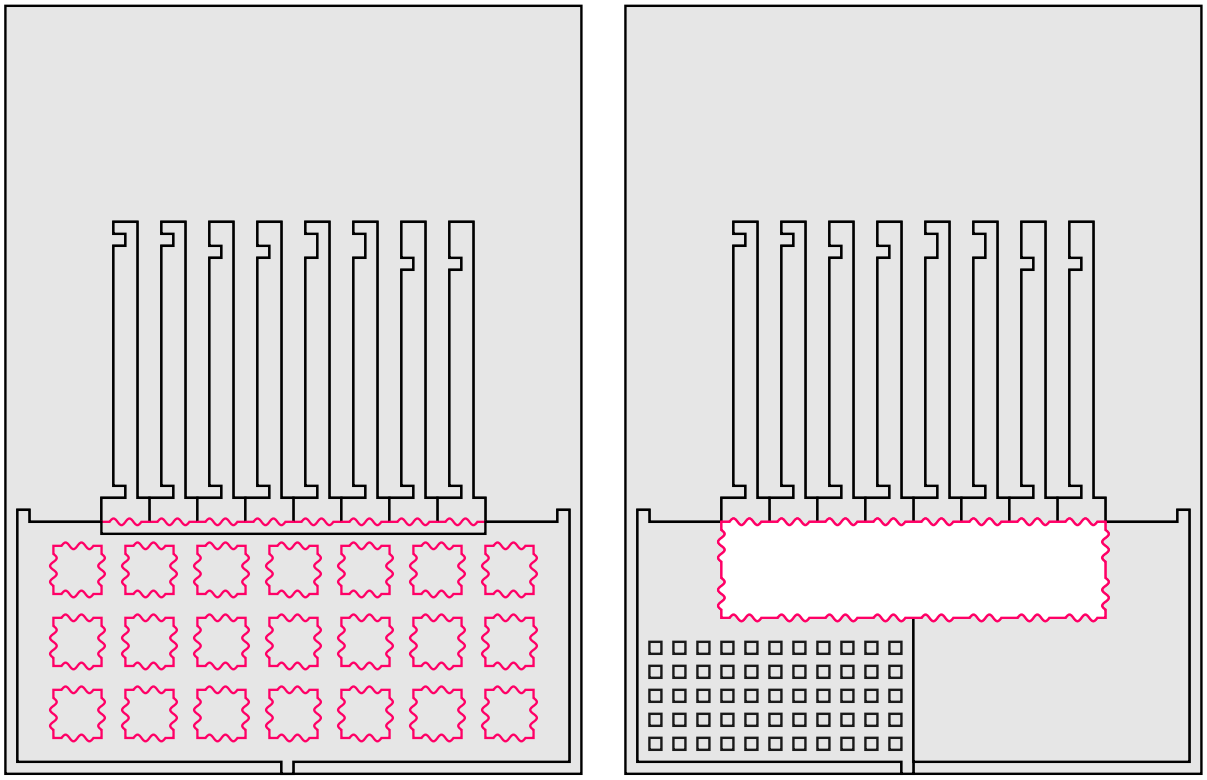
Figure 7-43: The contents of clue set A in the proof of Theorem 7.8.8 (not to scale). Pink wavy edges bear tabs and pockets encoding signed colors.



(a) $2n$ stamps.

(b) The large polyomino.

Figure 7-44: The contents of clue set B in the proof of Theorem 7.8.8 (not to scale). Pink wavy edges bear tabs and pockets encoding signed colors.



(a) The trivial packing of the puzzle after eliminating both board-frame polyominoes. The medium polyomino slots inside the large polyomino and the tiles fill the medium polyomino's holes. The stamps fill in their matching handle slots in the large polyomino and the monominoes fill in the pockets and any extra space in the stamp accommodation zone. This packing is always possible.

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

Figure 7-45: The two possible polyomino packings.

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

  Finally, for each monomino in this clue set, cut a pixel out of the left board-frame polyomino, starting from the second-bottommost pixel in the second column, continuing across every other column, then continuing with the fourth-bottommost pixel in the second column, and so on. The left board-frame polyomino has width $nw + 3n$, we cut pixels out of every other column, and we do not cut holes in its left or right columns, so we cut pixels out of $\frac{nw+3n-2}{2}$ columns. Below the mw-tall cutout and allowing two rows to ensure cut pixels do not join with pockets encoding signed colors along the edges of the cutout, we can cut pixels out of $\frac{3w-1}{2}$ rows (or $\frac{3w}{2}$, depending on parity). This allows up to $(\frac{nw+3n-2}{2})(\frac{3w-1}{2}) = \frac{n(w-4)^2+2w(nw-3)+13n+2}{4} + 4nw - 8n$ pixels to be cut out, but

there are at most $4nw - 8n$ monominoes, so we can always cut enough pixels without interfering with any other cuts.

Clue set B contains:

- A *stamp* polyomino for each of the $2n$ edge segments of the top edge of the boundary-colored board. Each stamp is composed of a $w \times 2$ rectangle modified to encode the signed color on the corresponding edge segment (called the *die*), a pixel centered above that rectangle, and a $2 \times h$ rectangular *handle* whose bottom-right pixel is immediately above that pixel, where $h = \max(m(w+3)+7, n)$. Stamps corresponding to 1-indexed edge segments $2i$ and $2i+1$ have pockets encoding i in binary cut into the left edge of their handle, starting from the second-to-top row of the handle.

- A "large" sized polyomino built from a $2n(w+3)+1 \times t$ rectangular polyomino, where t is the total area of all other polyominoes so far defined. Modify this polyomino by cutting out the middle pixel of the bottom row, the $2n(w+3)-1 \times m(w+3)+3$ horizontally-centered rectangle immediately above that removed pixel, and the pixels above the upper-left and upper-right removed pixels. (That is, cut out space for the medium polyomino, including the horns and tail but not including the stamp accommodation zone.) Then cut out the image of each stamp in the order of their corresponding edge segments in the adversarial-boundary edge-matching instance, aligning the leftmost-bottom pixel of the first stamp's die two pixels to the right of the upper-left removed pixel and aligning successive dies immediately adjacent to one another.

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

Placement of A clues. We place a monomino from clue set A in the cell having the start circle and end cap as vertices, then place an antibody above that monomino, surrounded by a monomino in each of its other three neighbors. We then place the other antibody, surrounded by monominoes in its neighboring cells, three cells above the first antibody. (See Figure 7-46.) It is always possible to surround the antibodies in this way because there are at least $8n$ monominoes. We place the remaining clues from clue set A inside the $2n(w+3)-1 \times m(w+3)+3$ rectangle one row above the bottom of the puzzle; this is always possible because $|A| \leq 4nw - 8n + 2nm + 5$.

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

Argument. In any solution to the resulting puzzle, the large polyomino is not eliminated. If it were, it must be in the same region as an antibody. Because each antibody is surrounded by monomino clues, the number of polyomino clues in this region is strictly greater than the number of antibodies, so the region must be packed

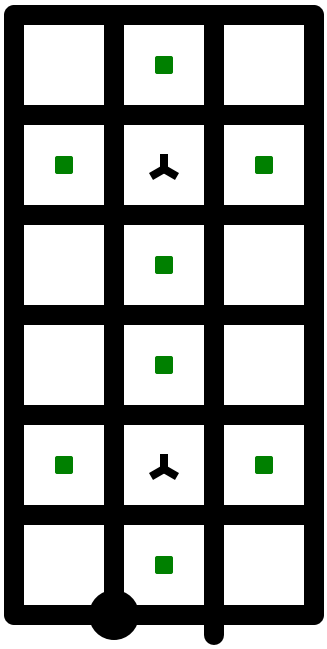


Figure 7-46: Because both antibodies are surrounded by monominoes, any region containing an antibody also contains at least one monomino.

by the non-eliminated polyomino clues. The nearest (upper) antibody is $t-4$ columns and $nw+3n$ rows away from the large polyomino clue, so this region has area at least t. Recall that t is the total area of all polyomino clues except the large polyomino. If the large polyomino is eliminated, there is no way to pack this region, even if all other polyomino clues are used.

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

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

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

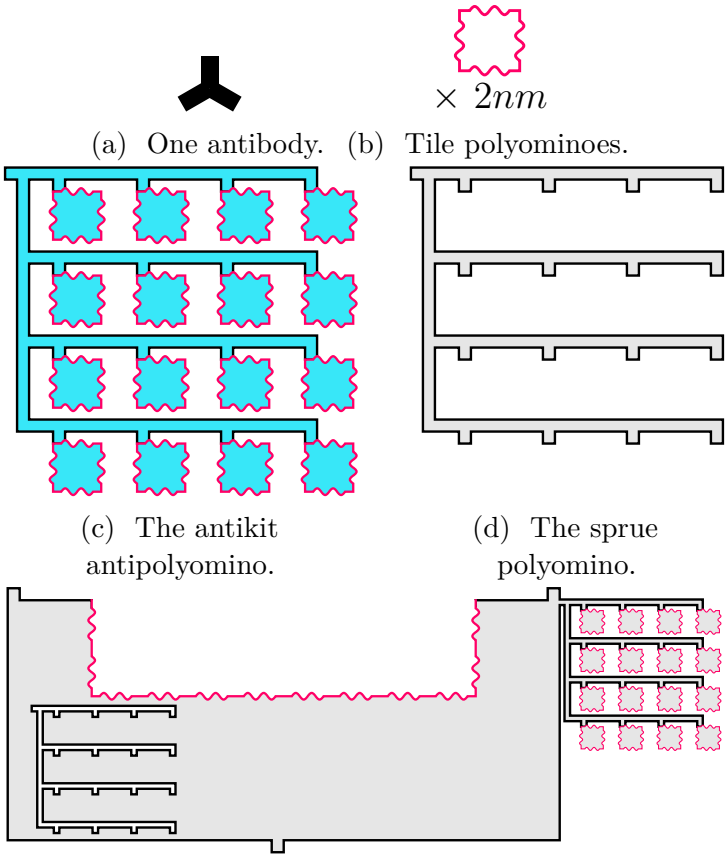
All clues in set A are in the other region. The monomino clue in the cell having both the start circle and end cap as vertices cannot be in the same region as the large polyomino (else the path could not divide the puzzle into two regions). Because each antibody is surrounded by monomino clues, the number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

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

When using both antibodies to eliminate the board-frame polyominoes, the remaining polyominoes always pack their region (see Figure 7-45a). The medium polyomino's placement is fixed by the horns and tail; the stamp accommodation zone ensures this placement is legal regardless of the pattern of tabs on the dies of the stamps. The tile polyominoes fit directly into the cutouts in the medium polyomino and there are exactly enough monominoes to fill in the uncovered area in the stamp accommodation zone and the pockets of the dies.

The path is a solution only if both antibodies are necessary. When using one antibody to eliminate the medium polyomino, the board-frame polyominoes' position is forced by the horns and tail. The monominoes are the only way to fill the single-pixel holes in the left board-frame polyomino and there are exactly enough monominoes to do so. Then the dies of the stamps and the edges of the rectangular cutout in the board-frame polyominoes models the boundary-colored board of the input adversarial-boundary edge-matching instance (see Figure 7-45b). The tile polyominoes cannot pack this area, necessitating the second antibody and making the path a solution, exactly when the set of top-boundary swaps is a winning strategy in the adversarial-boundary edge-matching instance. □

Theorem 7.8.9. *It is Σ_2-complete to solve Witness puzzles containing one antibody,*



(a) One antibody. (b) Tile polyominoes.

(c) The antikit antipolyomino.

(d) The sprue polyomino.

(e) The medium polyomino. The attached kit polyomino and sprue cutout are to scale with each other, but not with the body of the medium polyomino.

Figure 7-47: The contents of clue set A in the proof of Theorem 7.8.9 (not to scale). Pink wavy edges bear tabs and pockets encoding signed colors.

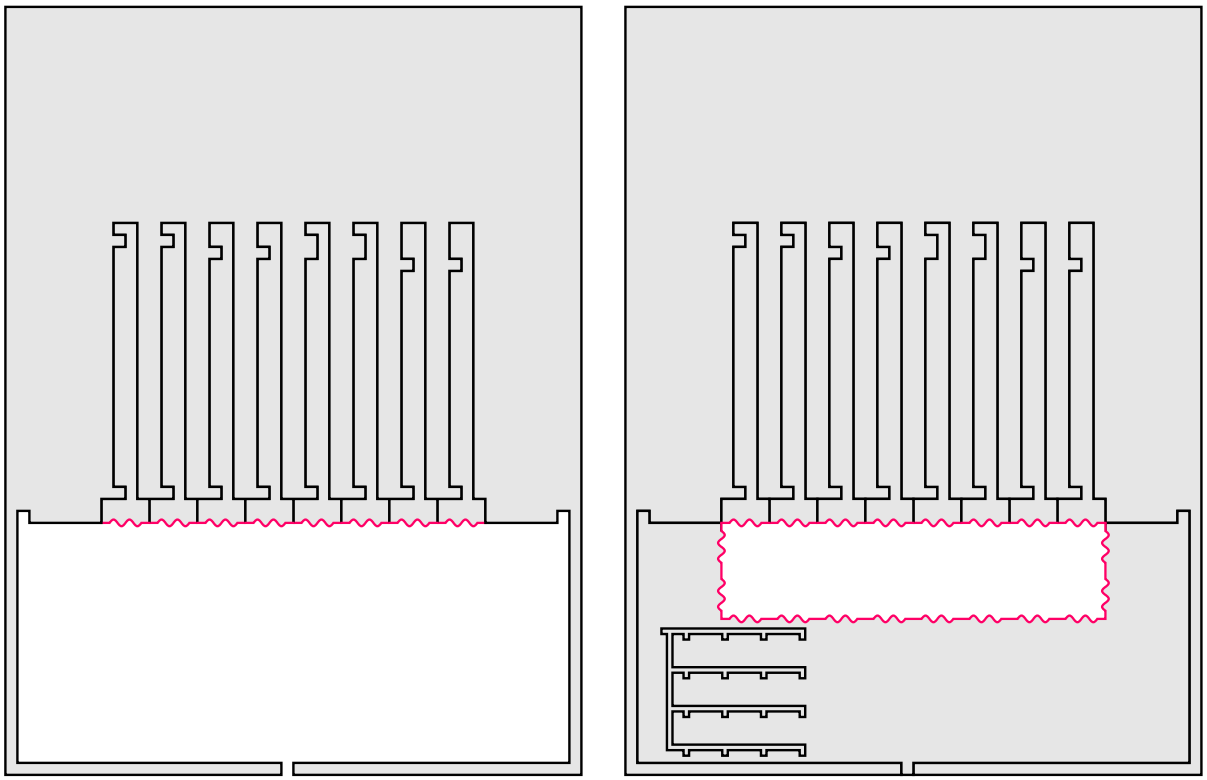
polyominoes and antipolyominoes.

Proof. As in the previous proof, we reduce from adversarial-boundary edge-matching (though we do not need the first player to have a losing strategy), and the reduction is similar. The primary difference is that the medium polyomino is also the (singular) board-frame polyomino.

Clue sets. As before, we have two clue sets. Clue set A is shown in Figure 7-47. Clue set B is nearly the same as it is in the previous proof (see Figure 7-44), only with the large polyomino being slightly wider (not visibly different at the scale of the figure).

Clue set A contains:

- One antibody.
- A $w \times w$ square polyomino for each of the $2nm$ tiles in the adversarial-boundary edge-matching instance. The edges of each polyomino are modified with tabs



(a) The trivial packing of the puzzle after eliminating the medium polyomino. The sprue and tiles annihilate with the antikit antipolyomino. The bottom region is unconstrained because it does not contain any surviving polyomino clues. After the stamps are placed in their handle slots, the path simply traces the exterior of the surviving polyominoes. This packing is always possible.

(b) When the antibody is not used, the antikit antipolyomino annihilates part of the medium polyomino, allowing it to fit inside the large polyomino. The sprue polyomino fills the cutout in the medium polyomino. The stamps fill in their matching handle slots in the large polyomino, leaving only the boundary-colored board for the simulated adversarial-boundary edge-matching instance.

Figure 7-48: The two possible polyomino packings.

and pockets encoding the signed colors on the corresponding edges of the corresponding tile. Call the upper-left corner of the $w \times w$ square the *key pixel* of that polyomino (even if tabs caused other pixels to be further up or to the left).

- An *antikit* antipolyomino, which we define in terms of a kit polyomino (which is not itself a clue). Start with m copies of a $2(n-1)(w+5)+3 \times 1$ rectangle, vertically aligned and spaced at $w+5$ intervals. Connect these rectangles with $1 \times w+4$ rectangles in the left column. Align the key pixels of the tile polyominoes to a $2n \times m$ grid spaced at $w+5$ intervals, then place this grid such that the key pixel of the upper-left polyomino is three pixels below and to the right of the upper-left pixel of the polyomino being built. (A tile polyomino having tabs on each edge is $w+2$ wide and tall, but there is $w+4$ width and height in the grid, so this placement is always possible.) Connect each tile polyomino to the rectangle above it by adding two pixels immediately above the key pixel. Finally, add one pixel immediately to the left of the upper-left pixel. The antikit antipolyomino is just the antipolyomino with an antipixel for each pixel in the kit polyomino. (See Figure 7-47c.)

- A *sprue* polyomino shaped like the sprue[12], the non-tile-polyomino area of the kit polyomino.

- A medium polyomino built from a $2n(w+5)+5 \times m(2w+5)+2$ rectangle polyomino. Add a pixel directly below the center pixel of the rectangle (the tail). Cut out a $2nw \times mw$ rectangle whose upper-left pixel is the third pixel in the top row of the rectangle. The left, right and bottom edges of this cutout are modified with tabs and pockets encoding the signed colors on the corresponding sides of the boundary-colored board in the adversarial-boundary edge-matching instance. Then add the kit polyomino, placing the leftmost pixel of the kit to the right of the upper-rightmost pixel of the rectangle. Then cut out the shape of the sprue polyomino, aligning the leftmost pixel in the second column and the bottommost pixels in the second-to-bottom row of the rectangle. This is always possible because the sprue is bounded by a $2(n-1)(w+5)+4 \times (m-1)(w+5)+2$ rectangle, so the sprue cutout fits within the medium polyomino even after the board cutout.

Clue set B contains:

- A *stamp* polyomino for each of the $2n$ edge segments of the top edge of the boundary-colored board. The stamps are exactly as specified in the previous proof.

- A large polyomino built from a $2n(w+5)+7 \times t$ rectangular polyomino, where t is the total area of all other polyominoes so far defined. Modify this polyomino by

[12] In molding, the sprue is the waste material that cools in the channels through which the material is poured or injected into the mold. Specifically, in plastic model kits, the sprue is the frame from which the model pieces are snapped out.

cutting out the middle pixel of the bottom row and the $2n(w+5)+5 \times m(2w+5)+2$ horizontally-centered rectangle immediately above the removed pixel. Then cut out the image of each stamp in the order of their corresponding edge segments in the adversarial-boundary edge-matching instance, aligning the leftmost-bottom pixel of the first stamp's die one pixel above and two pixels to the right of the upper-left pixel of the rectangular cutout and aligning successive dies immediately adjacent to one another. (This is the same procedure as in the previous proof, but with slightly different dimensions.)

Puzzle. The Witness puzzle is a $2n(w+5)+7 \times t$ rectangle. The start circle and end cap are at the middle two vertices of the bottom row of vertices.

Placement of A clues. We place the antibody in the cell having the start circle and end cap as vertices, then place the medium polyomino above the antibody. We place the remaining $2nm+2$ clues from clue set A contiguously, directly to the left and right of the medium polyomino, half on each side.

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

Argument. The medium polyomino clue is wider than the puzzle, so any solution must eliminate it. There is only one antibody, so the large polyomino is not eliminated in any solution.

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

Suppose the solution path places the entire puzzle into a single region; that is, suppose the solution path proceeds (in either direction) from the start vertex to the end cap without leaving the boundary. After eliminating the medium polyomino, the remaining polyominoes and antipolyomino have less net area than the puzzle, so any path resulting in a single region is not a solution to the puzzle. Thus, there are exactly two regions.

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

All clues in set A are in the other region. The antibody clue is in the cell having both the start circle and end cap is vertices, so it cannot be in the same region as the large polyomino (else the path could not divide the puzzle into two regions). The

medium polyomino is eliminated, so it must be in the same region as the antibody. The remaining A clues (the sprue polyomino, tile polyominoes and antikit antipolyomino) have net area 0. If a polyomino from this group is in the first region, the other region's clues require negative area to be satisfied; if the antikit is in the first region, the other region's clues do not have enough area to pack the area of the puzzle not already covered by the large polyomino or the stamps. Thus, the remaining clues are in the other region (and, because there are no other non-eliminated polyomino clues in the region, are satisfied).

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

By the above arguments, any solution path of this form satisfies all non-antibody clues when the antibody is used to eliminate the medium polyomino (see Figure 7-48a), but the antibody is only satisfied if it is necessary. If the antibody is not used to eliminate the medium polyomino, the antikit antipolyomino must annihilate the portion of the medium polyomino shaped like a kit polyomino. The rest of the medium polyomino is exactly the right size to fit inside the large polyomino and the sprue polyomino fills the sprue cutout in the medium polyomino. Then the dies of the stamps and the edges of the rectangular cutout in the medium polyomino models the boundary-colored board of the input adversarial-boundary edge-matching instance (see Figure 7-48b). The tile polyominoes cannot pack this area, necessitating the antibody and making the path a solution, exactly when the set of top-boundary swaps is a winning strategy in the adversarial-boundary edge-matching instance. □

By Theorem 7.8.4, Theorem 7.8.8 and Theorem 7.8.9 are tight.

7.9 Nonclue Constraints

In this section, we analyze Witness puzzle constraints that are not represented by clues on vertices, edges, or cells. This is not an exhaustive list of elements included in puzzles in The Witness. In most cases, these elements do not appear to change the computational complexity of the problem. However, containment in L for broken edges is no longer obvious in many cases. In addition, it is unclear how to modify some of our reductions to obey the symmetry constraint, though we believe the problem still remains hard for all prior cases.

7.9.1 Visual Obstruction

Some panels are placed in the 3D world in such a way that obstacles obstruct the player's vision of some of the panel. Because the player can only draw paths along edges they can see, the obstructions constrain the panel's solution set, similar to broken edges. Indeed, we show how to reduce visual obstructions to broken-edge puzzles:

Theorem 7.9.1. *Witness puzzles constrained only by visual obstruction are in P.*

Proof. We can determine the combinatorially different viewing positions (where different sets of vertices and/or edges are obstructed) in $O(n^3)$ time by building the arrangement of the planes extending the obstacle faces [56]. For each viewing position reachable in the environment, we can reduce the visual obstruction puzzle to an equivalent puzzle using only broken edges: starting from a puzzle with the same size, start circles, and end caps, break each obstructed edge and break all edges incident to each obstructed vertex.[13] This set of broken edges exactly captures the obstruction's restrictions on path drawing. Puzzles containing only broken edges are in L (Observation 7.3.2), and there are only polynomially many viewing positions, so we can solve from all positions in polynomial time. The visual obstruction puzzle is a YES instance exactly when at least one of the broken-edge puzzles is a YES instance. □

Some puzzles constrained only by visual obstruction from The Witness can obviously only be viewed from a single position in the game world, so building the plane arrangement is unnecessary. Such puzzles are in L if there exists an L algorithm to decide whether the projection onto the puzzle panel of some face of the obstacle overlaps a particular edge or vertex.

More generally, for any puzzle type allowing broken edges from P or larger complexity classes, adding visual obstructions results in puzzles in the same complexity class, as we can simply try all of the polynomially many viewpoints by the proof of Theorem 7.9.1.

7.9.2 Symmetry

In symmetry puzzles, there are an even number of start circles (usually exactly two), and drawing a path from one of them also causes one reflectionally symmetric path (reflected across one axis, or reflected across both axes which is equivalent to 180° rotation) to be drawn from another start circle. The paths must not intersect and both paths must reach an end cap. The paths induce regions that must satisfy all clues just as in no-symmetry puzzles. In some symmetry puzzles, the paths are colored yellow and blue, as are some vertex or edge hexagons; each path must visit the hexagons of its color (and any hexagons of other colors must be visited by either path). Path color is not relevant for other clue types.

We now discuss the implications of adding symmetry to each of the clue sets we have considered so far in this chapter.

Containment. Our proofs for containment in NP or in Σ_2 are based on algorithms for verifying certificates. Our verifiers can be straightforwardly modified to check that the paths in the certificates are indeed symmetric, do not intersect, and visit all

[13] Some puzzles in The Witness have end caps halfway along edges instead of at vertices. If only one half-edge or adjacent vertex is obstructed for such an end cap, replace the edge with an end cap at the other vertex instead of breaking edges.

hexagons of their color (if any). From then on it is irrelevant that there are two paths instead of one, so the containments still hold when the symmetry constraint is added.

Observation 7.3.2 establishes that puzzles with only broken edges, even with multiple start circles and end caps, are in L. For each start circle, we can determine the symmetric start circle in L, and the connectivity algorithm can follow the progress of both paths (if necessary, by running another instance of the algorithm in lockstep) to ensure paths are extended only if both edges are not broken. However, it is not obvious how to ensure that the paths do not intersect each other (as there is not enough space to store them). Thus, we have only that Witness puzzles with only broken edges and symmetry are in P.

Theorem 7.3.7 and Theorem 7.7.1 establish polynomial-time algorithms for edge hexagons on the boundary of a puzzle and monominoes, respectively. While we suspect that these algorithms can be extended to solve symmetric puzzles of these types, we leave these as open problems.

Hardness. To prove that adding symmetry does not make puzzles easier, we need to modify the constructions from our proofs to tolerate the addition of a symmetric copy. For most of our constructions, this is straightforward:

- **Vertex hexagons and broken edges** (Observation 7.3.3): Under the symmetry constraint, Hamiltonian path is no longer a strict special case. We can add a symmetric copy of the puzzle if we separate it from the original by breaking all the edges in a row or column between the copies. This wall of broken edges ensures the two paths cannot interact, making Hamiltonian path a special case again.

- **Edge hexagons** (Theorem 7.3.5): We can add a symmetric copy in any position with a row and/or column of empty cells between them to ensure that the paths do not touch. The rightmost bottommost chamber, which contains the start circle and end cap, is on the bottom boundary of the board, so the symmetric copy's start circle and end cap are also on the boundary of the board (possibly the top boundary). Then each path is confined to its copy by the existing argument.

- **Squares** (Theorem 7.4.2): If we consistently use the same colors inside and outside the path, we can add a symmetric copy of the puzzle in any position. The boundary of red squares ensures that each path is confined to its copy of the puzzle.

- **Rotatable dominoes** (Theorem 7.7.2): We can add a symmetric copy of the puzzle in any position. The distance between the copies is twice the existing distance from the Steiner tree area to the boundary, so the copies cannot interact. Then hardness preservation for monominoes and antimonominoes (Theorem 7.7.3) follows in the same way as the no-symmetry construction.

- **Nonrotatable dominoes** (Theorem 7.7.4): We can add a symmetric copy of the puzzle without affecting the proof if we add padding rows and/or columns to ensure that there are at least five empty cells between the closest domino clues. This preserves Properties 7.7.1, 7.7.2, and 7.7.3 (rotated and/or reflected in the symmetric copy), so solving the puzzle still requires solving the domino tiling problem. Then because there are at least five empty cells between clues, the symmetric paths cannot cross from their tree of domino clues to the other.

- **1-triangle clues** (Theorem 7.6.1): We can add a symmetric copy above the original construction. The original copy's path is forced to proceed from the start circle entirely across the puzzle, forming a barrier preventing the two paths from interfering.

- **3-triangle clues** (Theorem 7.6.3): We can add a symmetric copy of the puzzle in any position because each path is confined to the chambers of its copy.

But for others of our constructions, adapting to a symmetry puzzle is not so simple, and we leave solutions as open problems:

- **Stars** (Theorem 7.5.1): While Figure 7-16 is not to scale, it does accurately depict that there is plentiful blank space outside the simulated squares instance; note that the number of cells occupied by the red and blue stars is equal inside and outside the simulated instance, but the outside stars are arranged in a line instead of a rectangle. Thus, there is space for the symmetric path to disrupt the argument about what colors of stars must be in which region. It may be possible to fix this by packing the outside stars more densely, so that the simulated squares instance occupies the full width or height of the puzzle, blocking the symmetric path from interfering.

- **2-triangle clues** (Theorem 7.6.2): Our construction relies on forcing paths throughout the entire puzzle, particularly wave propagation from the four corners of the puzzle. To add a symmetric copy, we need some way to force simulated corners exactly halfway across the puzzle.

- **Antibodies** (Theorems 7.8.8 and 7.8.9): Obviously, the symmetric copy includes a copy of the antibody clue(s), so adding symmetry weakens the theorems to require four and two antibodies for hardness (respectively). Substantively, both proofs rely on the large polyomino not being eliminated and having a unique placement. With a symmetric copy, this is no longer true: one large polyomino could be eliminated and the other placed in the middle of the board instead of up against the boundary, or the two large polyominoes could exchange the intended placements. Then arguments that polyominoes must be in the same region because the placement of the large polyomino overlaps the cells containing their clues no longer hold.

7.9.3 Intersection

In intersection puzzles, multiple puzzles must be solved simultaneously by the same solution path. That is, the set of solutions to an intersection puzzle is the intersection of the sets of solutions to its component puzzles. The Witness contains one intersection puzzle in the mountain consisting of six panels. Drawing a path on any of these panels causes the path to be drawn on all of them, and the game only accepts a path if it satisfies all the panels. (Initially only the first panel is energized, with each additional panel activating when all the previous panels are solved, but this is not relevant for the complexity analysis.)

Intersection puzzles are typically in the same complexity class as the hardest of their component puzzles. Solving an intersection puzzle containing only broken edges is the same as solving a single broken-edges puzzle (which is in L, Observation 7.3.2) where an edge is broken if it is broken in any component puzzle; the path existence algorithm simply checks that the edge is present in every component puzzle. If the hardest component puzzle is in NP or Σ_2, membership for the intersection puzzle follows by combining the certificate verification algorithms (as in Section 7.9.2). An interesting challenge is monomino puzzles (Theorem 7.7.1); it is unclear whether intersection puzzles of monominoes can still be solved in polynomial time.

7.9.4 Recursion

In recursive puzzles, one or more panels are embedded in the cells of a larger panel. The inner panels contain non-antibody clues and at least one antibody. The inner panels are solved independently and any surviving clue(s) from each inner panel "bubble up" to the cell in the outer puzzle containing that panel. If a surviving clue is a rotatable polyomino clue, it bubbles up as a nonrotatable clue oriented as it was used in the inner panel.[14] Once all of the inner panels have been solved, the outer panel is solved just like a normal puzzle with the surviving clues in those cells (along with any other clues in the outer puzzle). As antibodies are only satisfied if they are necessary, an inner panel always provides the same number of clues to its outer panel regardless of how it is solved.

Because clues are only promoted from inner to outer levels, recursive puzzles are contained in the larger of the largest class among the complexity classes of their inner puzzles and the largest class possible for the outer puzzle (considering all possible sets of surviving clues). For example, consider an outer puzzle containing two antibodies and inner puzzles each containing one antibody, one polyomino clue, and one nonpolyomino clue. Then the inner puzzles are in NP, and when the inner puzzles all provide the nonpolyomino clue, the resulting outer puzzle contains only antibodies and nonpolyominoes, and so is also in NP. But because some possible resulting outer puzzles may contain polyominoes, we obtain the weaker bound that the recursive

[14] See https://gaming.stackexchange.com/q/253987. As this puzzle (on the bottom floor of the mountain) is the sole recursive puzzle in The Witness, another consistent interpretation is that the solution paths of the inner puzzles are what matters, not the surviving clues, but that definition does not generalize to nonpolyomino clue types.

puzzle is in Σ_2.

7.10 Metapuzzles

Figure 7-49: A screenshot from The Witness, featuring a panel whose solution controls a sliding bridge.

In this section, we analyze several of the *metapuzzles* that appear in The Witness. Metapuzzles are puzzles which have one or more puzzle panels as a subcomponent of the puzzle, and in which solving the puzzle panel affects the surrounding world in a way that depends on the choice of solution that was used to solve the panel. Figure 7-49 shows one example from The Witness.

Unlike the single-panel pathfinding puzzles, metapuzzles are naturally a kind of reconfiguration puzzle with no obvious bound on the number of moves (solving and re-solving panels). Indeed, we show PSPACE-completeness for three different metapuzzles: sliding bridges, elevators and ramps, and power cables and doors. Refer to Table 7.3.

Our PSPACE-hardness results use the *door framework* of [10]. Specifically, [10, Section 2.2] shows that the following gadgets suffice to prove a one-robot motion planning puzzle PSPACE-hard:

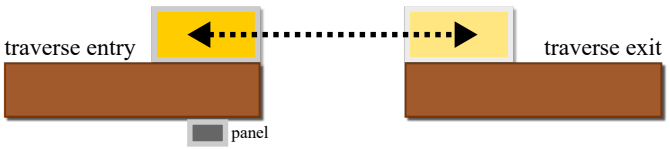
1. **One-way**: allows the robot/player to traverse from a point A to a point B but not from B to A.

2. **Door**: has three paths, *open*, *traverse*, *close* and two states *open* and *close*. The *open* and *close* paths transition the gadget to the *open* and *close* states respectively. The *traverse* path can be traversed if and only if the door is in the *open* state and is blocked otherwise.

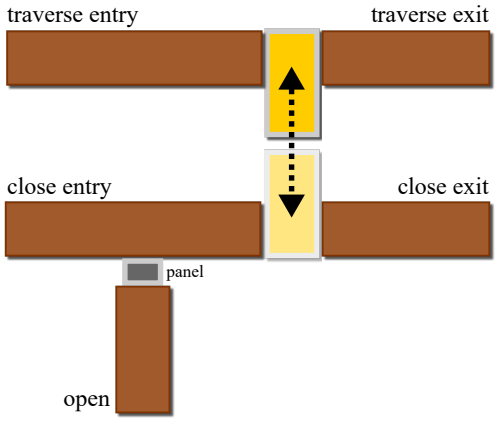
3. **Crossover**: has two non-interacting paths which can be independently traversed and which geometrically cross (in projection). This gadget is trivial to build in a 3D game like The Witness with ramps and tunnels. (The Witness is inherently nonplanar.)

7.10.1 Sliding Bridges

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



(a) One-way gadget



(b) Door gadget

Figure 7-50: Sliding-bridge gadgets for Theorem 7.10.1, drawn from a birds-eye view.

Theorem 7.10.1. *It is PSPACE-complete to solve Witness metapuzzles containing sliding bridges.*

Proof. We apply the door framework described above using the gadgets in Figure 7-50.

The one-way gadget consists of a sliding bridge that can be adjacent to one of two docks; see Figure 7-50a. The entry dock has a puzzle panel which controls the bridge. When the player is at the entry dock, they can change the bridge's position by solving the panel. To cross to the exit dock, the player stands on the bridge and solves the panel to set the bridge's position to the exit dock. The panel is not viewable from the exit dock, so once the bridge reaches the exit dock, the player cannot return to the entry dock.

The door gadget consists of five docks, named *traverse-entry*, *traverse-exit*, *close-entry*, *close-exit* and *open*, and a single bridge with two states, spanning the gap between either the two traverse docks or the two close docks; see Figure 7-50b. There is also a one-way gadget on either side of the close-entry and close-exit so the player can only enter from the close-entry side and exit from the close-exit side. In either bridge state, the player can cross between the docks adjacent to the bridge, but not the other pair of docks. We place the controlling puzzle panel visible from only the *open* and *close-entry* docks, allowing the player to set the state of the bridge from those two docks only. The panel is far enough away from the end of the *close-entry* dock to prevent the player from running onto the bridge after sending the bridge to the gap between traverse docks. The player opens the door by using the panel from the *open* dock to move the bridge to the traverse docks, and can then traverse that path; to cross between the close docks, the player is forced to use the panel from the *close-entry* dock to move the bridge to the close docks, closing the door.

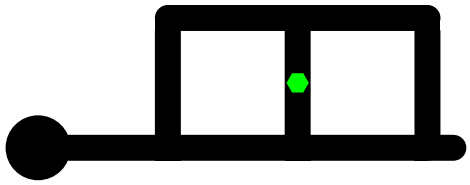


Figure 7-51: Elevator control puzzle panel for the door gadget for Theorem 7.10.2.

Using these gadgets, we can construct any instance of TQBF using the framework from [10] and so The Witness metapuzzles with just sliding bridges are PSPACE-hard. Sliding bridge puzzles have only a polynomial amount of state (given by the position of each bridge and the position of the player) so these puzzles are also in PSPACE. Thus, sliding bridge puzzles are PSPACE-complete. □

7.10.2 Elevators and Ramps

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

Theorem 7.10.2. *It is PSPACE-complete to solve Witness metapuzzles containing elevator reconfiguration, even when each panel controls at most one elevator.*

Proof. Again we apply the door framework, building our one-way and door gadgets by modifying the gadgets from the sliding-bridges proof.

For the one-way gadget, we replace the bridge with an elevator controlled by a panel on the lower level. When the elevator is on the lower level, the player can access the puzzle panel to raise the elevator and reach the upper level. From the upper level, the player cannot see the panel at all and cannot move the elevator to reach the lower level. This gadget's requirement that the player travel from the lower level to the upper level is not a constraint, as we can freely add elevators controlled by a panel visible from both levels.

For the door gadget, we use the puzzle panel in Figure 7-51 to allow the player to flip between the two states. (We show a panel using an edge hexagon, but equivalent (possibly larger) panels can be constructed using squares, stars, triangles, or monominoes instead.) Now the moving bridge in the previous reduction (Figure 7-50b) is replaced with two up/down platforms which are constrained by the puzzle

panel such that exactly one of them can be up at any given time, and the player can only traverse between an entry and exit dock if the corresponding block is down. □

7.10.3 Power Cables and Doors

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

Theorem 7.10.3. *It is PSPACE-complete to solve Witness metapuzzles containing power cables and doors.*

Proof. Again we wish to apply the door framework. One difficulty is that power cables in The Witness are initially off until their powering puzzle has been solved, so initially all doors are closed. We use an additional construction to allow the player to open the doors that are initially open in the one-way-and-doors instance we are reducing from.

This gadget is akin to an airlock: there are two doors in sequence (an entry and an exit) which are both initially closed. There is a puzzle panel just outside the entry door with two solutions connected by power cables to the entry and exit doors. When the player approaches the one-way gadget, the player can open the entry door (closing the exit door if it is open) using the panel and walk through the door. To leave via the exit door, the player must solve the panel with the other solution while standing inside the airlock looking through the open entry door, thereby opening the exit door but closing the entry door. Once the entry door is closed, the panel is not visible from the airlock, preventing the player from traversing from the exit to the entry door.

The door gadget is almost identical to that of Theorem 7.10.1 (shown in Figure 7-50b), except that instead of a moving bridge, the puzzle panel is connected by cables to two doors located in the positions where the bridge could have been. As before, in order to traverse the *close* path, the player must solve the panel puzzle such that the bottom door is open, thereby closing the top door. The *open* path can be used to open the top door, and finally, the *traverse* path is traversable if and only if the top door is open.

The one-way-and-doors instance we are reducing from may have some of the door gadgets initially open. Because The Witness is three-dimensional, we can build a skybridge above the rest of the puzzle allowing the player to open any door gadgets that should be initially open. For each such door gadget, a path branches off the skybridge to an overlook point with line of sight to the puzzle panel controlling that gadget. The player can use this overlook to open the door from the skybridge. Using walls along the sides of the skybridge (including its branches), we block line of sight to the panels controlling door gadgets that should be initially closed. The skybridge ends in a one-way gadget leading to the starting point of the one-way-and-doors instance, ensuring the player cannot return to the skybridge. Note that setting a door gadget to the *open* state allows the player to traverse strictly more paths than the *closed* state, so the player cannot benefit from leaving any skybridge-visible doors closed. □

7.10.4 Light Bridges

Light-bridge metapuzzles consist of a large void between two platforms, with a controller panel at one or both ends. Drawing a path on the controller panel causes a "hard light" bridge to follow a corresponding path between the two platforms. As with all panels, solution paths remain displayed on the panel after the player finishes drawing them, so the player can traverse the light bridge to the other platform. If there is a controller panel on that side of the void, the player can draw another path to create another light bridge. Each controller panel can only control the bridge originating on its side, but both paths appear on the panel, so the first bridge's path constrains the second, and any clues in the panels must be satisfied in each configuration. The objective of the metapuzzle is to position the bridge(s) so that the player can view (and solve) panels located around the void, or to use the bridge(s) to reach an exit platform on a third side of the void.

The Witness has two light-bridge metapuzzles inside the mountain. The first metapuzzle has a single bridge. There is a pillar in the center of the void that blocks the player from traversing straight through it, but permits the player to enter from the bridge and exit at a 90-degree angle, slightly divorcing the player's path from the path drawn on the panel. Besides panels on the platform across the void, there are additional panels mounted on the walls of the void area, so a secondary goal is to have the bridge pass in front of these panels (as if the panel has "virtual" edge hexagons). Neither of these features is interesting for hardness.

The second light bridge metapuzzle has two bridges. The objective of this metapuzzle is to exit through a door on the side of the void. This door is only open when both bridges are in place, and even if it were always open, it is not possible to draw a solution path on the first controller panel directly to the door (because the panel's clues cannot be satisfied). Instead the player must cross to the other platform, draw the second bridge, return to the first platform and adjust the first bridge, and so on. The optimal solution requires crossing and returning twice before the first bridge can be drawn to the exit.

The hardness of light-bridge reconfiguration metapuzzles remains open. We speculate that they are PSPACE-complete, as many reconfiguration problems are.

Open Problem 7.10.4. *Is it PSPACE-complete to solve Witness light-bridge reconfiguration metapuzzles?*

7.11 Puzzle Design Problem

The previous sections analyzed the hardness of solving Witness puzzles. In this section, we consider the *Witness puzzle design problem*: creating a Witness puzzle having a specified set of solution paths. While most of the panels in The Witness only need to be interesting to solve, the controller panels for metapuzzles depend critically on having a specific set of solutions corresponding to the positions of the environmental objects they control. That is, the panel controlling a sliding bridge metapuzzle that brings the player from one platform to another must have solutions representing the

bridge being at each end. The controller also must *not* have a solution representing the bridge in the middle, lest the player get stuck on one platform by moving the bridge to the middle while not on it.

Open Problem 7.11.1. *What is the complexity of the Witness puzzle design problem?*

7.11.1 Path Universality

More specifically, it is natural to ask whether there can exist an algorithm using only a subset of the available clue types and nonclue constraints. Ideally, we would hope for a small clue set having *path universality*, the ability to express all possible sets of solution paths. Unfortunately, an information-theoretic argument rules out path universality for reasonable clue sets.

Observation 7.11.2. *A rectangular board with n cells admits $2^{\Theta(n)}$ simple paths and $2^{2^{\Theta(n)}}$ sets of simple paths, even if we fix the endpoints to opposite corners of the puzzle.*

Proof. Fixing the endpoints reduces the number of paths by $\Theta(n^2)$, so we focus on the fixed-endpoint case. For the upper bound, a simple path makes $O(n)$ steps with at most three choices (turn left, turn right, or go straight) at each step, beyond the first step with four choices (north, west, south, east), so there are $O(3^n)$ paths. For the lower bound, we consider the case that the fixed endpoints are diagonally opposite, namely top-left and bottom-right, and describe a family of $2^{\Omega(n)}$ such paths. Suppose the puzzle is $x \times y$, and assume by symmetry that $x \geq y$. We start with 2^{x-1} possible paths within the top row of cells, from the top-left corner to the bottom-right corner of the row, by choosing whether each cell except the rightmost is left or right of the path (while the rightmost is forced to be on the left, to ensure the correct endpoint). Then we continue the path down one unit, and apply the same construction to the third row of cells from the top-right corner to the bottom-left corner. We repeat this process, zig-zagging back and forth, until we must stop to leave room to get to the bottom-right corner of the puzzle. In total, we obtain

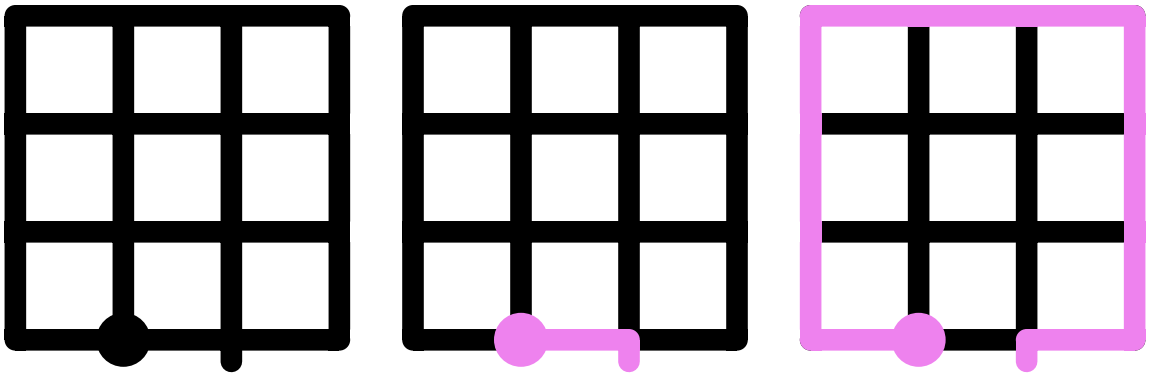
$$(2^{x-1})^{2\lfloor (y-1)/4 \rfloor + 1} \geq 2^{(x-1)((y-1)/2+1)} = 2^{(x-1)(y+1/2)} = 2^{\Omega(n)}$$

paths. Combining the upper and lower bounds gives $2^{\Theta(n)}$ paths, and so the powerset of the set of paths has size $2^{2^{\Theta(n)}}$. □

For clues of constant size, there are $c = O(1)$ choices of clue for each vertex, edge, and cell (including not placing a clue at that position), and $O(n)$ clue positions, so there are only $c^{O(n)}$ puzzles. Even if we allow polyominoes and antipolyominoes of size $k(n)$, there are at most $c^{O(n \cdot k(n))}$ puzzles. By Observation 7.11.2, most sets of solution paths do not have a corresponding puzzle.

7.11.2 k-Path Universality

Instead, we must settle for *k-path universality*, the ability to express all sets of up to k solution paths, for some small k. Here we focus on $k = 1$.



(a) Empty puzzle (b) Short path (c) Long path

Figure 7-52: Path universality requires distinguishing the short and long boundary paths.

Observation 7.11.3. *Broken edges are* 1*-path-universal.*

Proof. Break all edges not in the desired solution path. □

Observation 7.11.4. *Edge hexagons are* 1*-path-universal.*

Proof. Place edge hexagons on exactly those edges in the desired solution path. When the path visits an edge, it visits one of the vertices incident to the next edge, so if the path ever deviates from the edges with hexagons, at least one of the hexagons cannot be visited because one of its incident vertices has already been visited. □

Many clue sets are not even 1-path-universal, often because they cannot distinguish between different paths along the boundary. Consider the empty puzzle in Figure 7-52a. The following observations are based on trying to add clues that distinguish the *short path* in Figure 7-52b from the *long path* in Figure 7-52c.

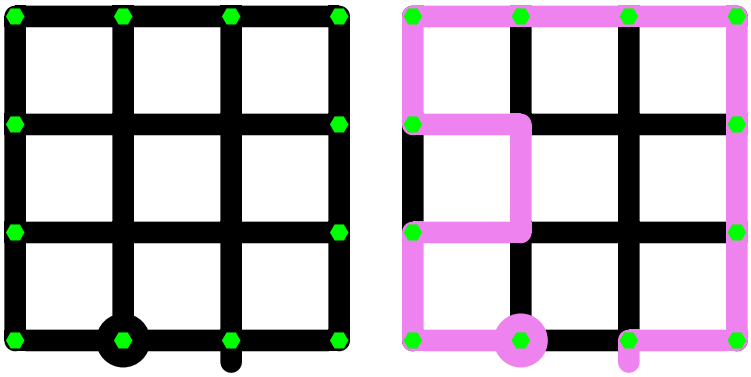
Observation 7.11.5. *Squares, stars, polyominoes, and antipolyominoes, in any combination, are not* 1*-path-universal.*

Proof. These clue types cannot distinguish the short and long paths because their satisfaction depends only on the other clues contained in or the shape of their region, not the path that induced the region. □

Observation 7.11.6. *Vertex hexagons are not* 1*-path-universal.*

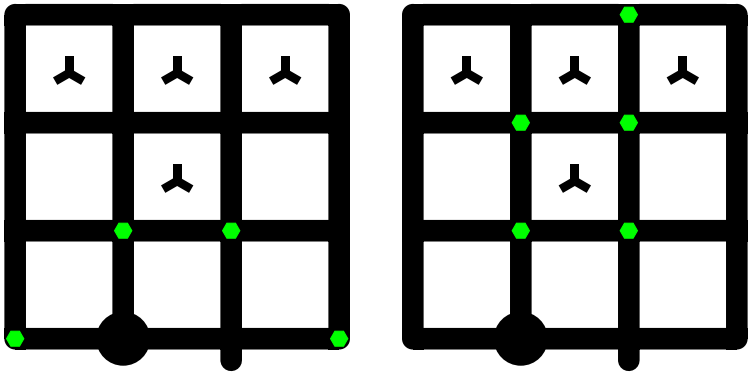
Proof. Placing vertex hexagons at each boundary vertex (Figure 7-53a) rules out the short path, but does not force the long path because the path can enter the interior and return to the boundary at the next hexagon (Figure 7-53b). Vertex hexagons are entirely redundant for enforcing the short path because the vertices bearing the start circle and end cap are already required to appear in the path. □

Vertex hexagons combined with antibodies can force the short and long paths as shown in Figure 7-54. This example shows that adding antibodies can increase the set



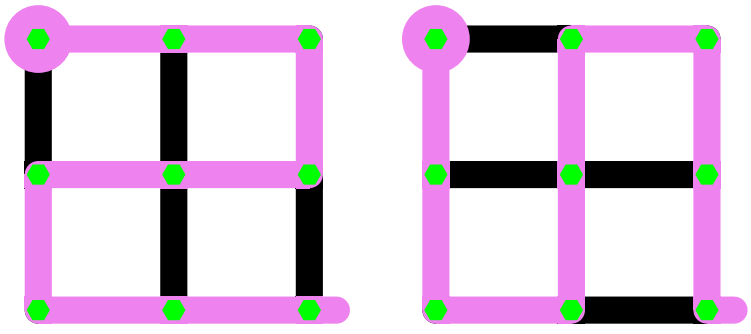
(a) Puzzle (b) Unintended solution

Figure 7-53: Vertex hexagons can rule out the short path, but not force the long path.



(a) Short path (b) Long path

Figure 7-54: Vertex hexagons with antibodies can force the short and long paths.



(a) Horizontal zigzag (b) Vertical zigzag

Figure 7-55: Vertex hexagons cannot distinguish these solutions, and antibodies cannot help.

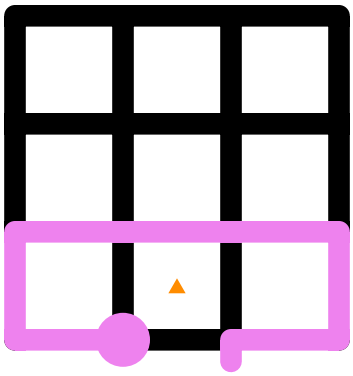


Figure 7-56: 1-triangle clues can rule out the long path, but not force the short path.

of expressible paths. However this combination is still not 1-path-universal as vertex hexagons cannot distinguish the paths in Figure 7-55 and there are no unsatisfied hexagons to be eliminated by antibodies.

Observation 7.11.7. *1-triangle clues are not 1-path-universal.*

Proof. Adding a 1-triangle clue in the cell adjacent to both the start circle and end cap rules out the long path, but doesn't force the short path, as shown by the unintended solution in Figure 7-56. □

7.11.3 Region Universality

As paths along the boundary are the source of problems, we can settle for specifying the same interior edges as a given path, but possibly with different boundary edges. This is equivalent to the ability to specify all *region decompositions* (sets of regions inducible by some path and whose union is the entire puzzle), so we call it *region universality*. There are $O(n^2)$ paths consistent with a given region decomposition (see Lemma 7.11.8 below), so Observation 7.11.2 also roughly counts the number of (sets of) region decompositions, so the same information-theoretic argument makes general region universality impossible. Therefore, we consider *k-region universality* where up to k region decompositions must be realized, and focus on $k = 1$.

Lemma 7.11.8. *In polynomial time, we can decide whether a given set $D = \{D_1, D_2, \ldots, D_m\}$ of disjoint sets of cells is a region decomposition, and if so, produce a path that induces that region decomposition.*

Proof. If the union of the sets of cells is not the set containing all cells in the puzzle, D is not a region decomposition. If any D_i does not contain a cell adjacent to the boundary, D is not a region decomposition, because the path along D_i's outline is a cycle. If any vertex in the puzzle is incident to cells in more than two regions, D is not a region decomposition, because more than two edges incident to that vertex are forced to be in any solution.

Otherwise, we will search for a path P inducing D. Initially, P consists of all edges between adjacent pairs of cells not in the same region (all paths inducing D must contain these edges). Because every D_i contains at least one cell on the boundary, each path segment in P begins and ends at a boundary vertex. To avoid inducing additional regions, the edges we add to complete P must lie entirely on the boundary.

For each pair (s, t) of endpoints of (possibly different) segments, consider the other endpoint s' of the segment starting at s. The path can go clockwise or counterclockwise along the boundary, connecting s' to the next endpoint in that direction. After this choice, the boundary segments must alternate between not being in the path and being in the path, so that every endpoint except s and t has degree 2. If there is such a choice of s, t and boundary segment parity, then the resulting path induces D; otherwise, D is not a region decomposition. There are only $2i$ choices for s and t and two choices for parity, so this algorithm runs in polynomial time. □

Any 1-path-universal set of clues is also 1-region-universal. We revisit the examples above that are not 1-path-universal:

Observation 7.11.9. *Vertex hexagons are not* 1*-region-universal.*

Proof. The puzzle in Figure 7-55 contains the maximal set of vertex hexagons and admits at least two region decompositions (induced by the solutions shown in the figure). Removing vertex hexagons from a puzzle can only add additional solutions and region decompositions, not remove them, so vertex hexagons cannot express only one of the region decompositions shown in the figure. □

Observation 7.11.10. *Squares are not* 1*-region-universal.*

Proof. Squares cannot force the entire puzzle to be a single region. If the puzzle contains only squares of a single color, the squares do not constrain the path or the region decomposition. If the puzzle contains multiple colors of squares, all solutions induce at least two regions. □

Observation 7.11.11. *Stars are not* 1*-region-universal.*

Proof. Stars cannot express some region decompositions in which a region consists of a single cell. Consider a desired decomposition in which the bottom-left cell forms a singleton region. Any cell containing a star cannot be in a singleton region in any solution, so the bottom-left cell must be empty. But stars are not affected by empty cells in their region, so if there is a solution path placing the bottom-left cell alone in a region, the path remains a solution after being locally modified to place the bottom-left cell in the adjacent region instead. □

Open Problem 7.11.12. *Are polyominoes* 1*-region-universal?*

It is tempting to think that monominoes are 1-region-universal by 2-coloring the regions and placing monominoes in every cell in one color class. While this forces any solution path to visit all the specified region boundaries, it leaves the path free to further subdivide those regions.

Nonclue constraints. Previously uninteresting nonclue constraints become more interesting for universality. For example, intersection puzzles inherently take the intersection of the solution sets of their component puzzles, so adding intersection to a set of clue types may allow specifying more solution sets than before, possibly making them universal.

Theorem 7.11.13. *Intersection puzzles having*

- *one component puzzle containing only squares of* 2 *colors, or*
- *one component puzzle containing only monominoes;*

and

- *two component puzzles containing only stars of* q *colors, or*
- $q - 1$ *component puzzles containing only stars of* 1 *color, or*
- *one component puzzle containing nonmonomino polyominoes, or*
- $q - 1$ *component puzzles containing only monominoes and antimonominoes, or*
- $q - 1$ *component puzzles containing only antibodies and antimonominoes,*

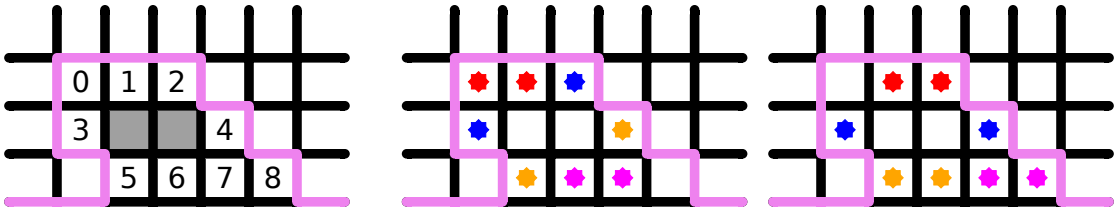
are 1*-region universal, where* r *is the number of regions in the input region decomposition and* q *is half of the maximum number of exterior cells in any region in the input region decomposition, rounded down.*

Proof sketch. The clue types in the first set of component puzzles constrain groups of cells to be in different regions (*disjointness*). Conversely, the clue types in the second set of component puzzles constrain groups of cells to be in the same region (*cohesion*). Intersection puzzles allow us to overcome the one-clue-per-cell limit to fully specify the region decomposition using these groupwise constraints. For colored clue types, there is a trade-off between the number of components and the number of colors. □

Proof. We give an algorithm taking as input a region decomposition $D = \{R_1, R_2, \ldots, R_r\}$ and producing an intersection puzzle for which all solutions induce D (and there is at least one such solution). Each component puzzle is shaped like the union of the R_i. By Lemma 7.11.8, we can produce a path P inducing D. In every component puzzle, we place the start circle and end cap at the endpoints of P.

Disjointness. The component puzzle enforcing disjointness constrain the solution path to induce D or a refinement of D (splitting some R_i into multiple regions).

- **Squares of** 2 **colors:** We 2-color D and create one component puzzle in which each cell contains a square of that cell's color. Regions must be monochromatic in squares, so this puzzle enforces disjointness.
- **Monominoes:** We 2-color D and create one component puzzle having monomino clues in all the cells in one color class, leaving the cells in the other class empty. By the polyomino area constraint, no solution can place a cell containing a monomino clue in the same region as an empty cell, so this puzzle enforces disjointness.



(a) Numbering the non-interior cells. The interior cells are shaded.

(b) The two component puzzles implementing cohesion.

Figure 7-57: An example of implementing cohesion using stars of q colors.

Cohesion. The component puzzle(s) enforcing cohesion constrain the solution to place cells in the same R_i in the same region. Note that our constructions for cohesion can assume disjointness, because it is enforced by the first component puzzle.

The constructions for some sets depend on a constant q, defined as follows. First, define the *interior cells* of R_i to be the cells in R_i having all four neighbors also in R_i, and call the remaining cells of R_i *non-interior*. Then q is half of the maximum number of non-interior cells in any R_i, rounded down.

- **Stars of q colors:** Create two component puzzles. Arbitrarily number the non-interior cells of each R_i starting at 0. In each R_i, place stars of color c in the even-parity cell pairs $(2c, 2c+1)$ in the first component puzzle and in the odd-parity cell pairs $(2c+1, 2c+2)$ in the second component puzzle. See Figure 7-57 for an example.

  There may be up to $2r$ stars of each color in each component puzzle, but any path satisfying disjointness cannot pair any star in R_i with a star in R_j for $i \neq j$, so each star color is effectively unique within R_i. Then cells numbered $2c$ and $2c+1$ must be in the same region to satisfy the stars in the first component puzzle, and cells numbered $2c+1$ and $2c+2$ must be in the same region to satisfy the stars in the second component puzzle. Then by transitivity, all non-interior cells of each R_i must be in the same region in any solution. Any path under which the non-interior cells are in the same region also places the interior cells in that region because the edges separating the interior cells from the non-interior cells constitute a cycle. Thus, all cells in each R_i must be in the same region.

- **Stars of 1 color:** Create $q-1$ component puzzles. As in the previous construction, number the non-interior cells of each R_i starting at 0. In puzzle p, in each R_i containing a $(p, p+1)$ cell pair, place a star in those cells.

  In each puzzle, each R_i contains only two stars, so those cells must be in the same region in any solution, and so by transitivity all non-interior cells in each R_i are in the same region. Then the interior cells are also in the region by the previous argument, so all cells in each R_i must be in the same region.

- **Nonmonomino polyominoes:** Create one component puzzle. For each R_i containing more than one cell, place a polyomino clue shaped exactly like R_i

in the leftmost bottommost cell in R_i. A path satisfying disjointness can only satisfy these polyomino clues by refraining from further dividing the R_i. (We do not place monomino clues in single-cell R_i because they cannot be divided further in any case.)

- **Monominoes and antimonominoes:** Use the construction for stars of 1 color, except when placing pairs of stars, place a monomino and an antimonomino instead. In each of the $q-1$ component puzzles, each R_i contains only a single monomino and antimonomino, so they must be in the same region in any solution. Then follow the same argument.

- **Antibodies and antimonominoes:** Use the construction for stars of 1 color, except when placing pairs of stars, place an antibody and an antimonomino instead. In each of the $q-1$ component puzzles, each R_i contains only a single antibody and antimonomino, so they must be in the same region in any solution. Then follow the same argument.

The first set of component puzzles constrains any solution to place cells in different R_i in different regions and the second set constrains any solution to place cells in the same R_i in the same region. Thus, any solution path must induce D, and P is such a path. □

7.11.4 Further Variants

All of the analysis above looked for exact matches to a path or region decomposition. We can also consider applying a scale factor, where each edge in the specified path is represented by s consecutive edges, or each cell in the specified region decomposition is represented by an $s \times s$ square, in the produced puzzle.

Returning to the motivation of specifying $k > 1$ paths or region decompositions, metapuzzles differ in the size of the sets they require. Light-bridge metapuzzles, for example, require only a small constant number of paths, because the player will be spending time traversing the bridge and interacting with any panels or other objects in the void. But the moving ramp and log claw in the sawmill can be moved to a linear number of positions, so we might hope for $O(n)$-path or -region universality.

Open Problem 7.11.14. *Classify sets of clue types and/or nonclue constraints by their degree of path and region universality, or more generally, characterize the sets of paths and regions they can express.*

7.12 Open Problems

We restate here the open problems defined in the previous sections, then briefly suggest additional possible lines of inquiry.

Open Problem 7.3.1. *Is there a polynomial-time algorithm to solve Witness puzzles containing only hexagons on vertices?*

Open Problem 7.5.2. *Is it NP-hard to solve Witness puzzles containing only a constant number of colors of stars? Or just a single color of stars?*

Open Problem 7.10.4. *Is it PSPACE-complete to solve Witness light-bridge reconfiguration metapuzzles?*

Open Problem 7.11.1. *What is the complexity of the Witness puzzle design problem?*

Open Problem 7.11.12. *Are polyominoes* 1*-region-universal?*

Open Problem 7.11.14. *Classify sets of clue types and/or nonclue constraints by their degree of path and region universality, or more generally, characterize the sets of paths and regions they can express.*

Parsimony. Which hardness reductions can be made (or are already) parsimonious, and thus prove #P-hardness and ASP-hardness? For example, Theorem 7.6.1 is a parsimonious reduction from X3C, and X3C is #P- and ASP-complete [80]. On the other hand, Theorem 7.4.2 is a parsimonious reduction from tree-residue vertex breaking, but is tree-residue vertex breaking #P/ASP-hard?

Fixed-parameter tractability with respect to board height. We conjecture that many Witness puzzles can be solved in polynomial time (XP) when there are a constant number of rows in the puzzle, using dynamic programming over the possible vertical cross-sections and how they are topologically connected on either side. In particular, we believe this to be possible for broken edges, hexagons, triangles, squares, stars, and constant-sized polyominoes and antipolyominoes. This leaves three clue types: antibodies and unbounded polyominoes and antipolyominoes. Antibodies seem difficult to handle because of the need to verify that they are necessary. Unbounded polyomino clues are NP-hard in $2 \times n$ puzzles by a simple reduction from 3-Partition (similar to [54, Theorem 3]), so are not in XP unless P = NP.

Beyond just membership in XP as above, are any of these problems fixed-parameter tractable with respect to the number of rows? For example, we conjecture that this is possible for broken edges, hexagons, triangles, and squares and stars of $O(1)$ colors.

7.13 Adversarial-Boundary Edge-Matching Problem is Σ_2-complete

Proof of Lemma 7.8.7. We reduce from QSAT_2, which is the Σ_2-complete problem of deciding a Boolean statement of the form $\exists x_1 : \exists x_2 : \cdots : \exists x_n : \forall y_1 : \forall y_2 : \cdots : \forall y_n : f(x_1, x_2, \ldots, x_n; y_1, y_2, \ldots, y_n)$ where f is a Boolean formula using AND ($\wedge$), OR ($\vee$), and/or NOT ($\neg$). We convert this formula into a circuit, lay out the circuit on a square grid, and implement each circuit element as a set of tiles. The first player's boundary-edge swaps encode a setting of true or false for the first player's variables. Then, as part of solving the edge-matching problem, the second player must exhibit a setting of their variables that makes the formula false; otherwise the first player wins.

Conversion to circuit. We convert the formula into a Boolean circuit with AND, OR, and NOT gates for the Boolean operations, plus $2n$ INPUT gates to represent the x_is and y_is, binary FAN-OUT gadgets to make copies of these values, and one OUTPUT gate to represent the formula output (which must be TRUE).

Circuit layout. We construct a planar layout of this circuit on an integer grid. Each cell in the grid views itself as being "connected" to zero, one, two, or three of its edge neighbors:

- Each AND and OR gate maps to a three-neighbor cell (two neighbors for the inputs, one neighbor for the output).
- Each FAN-OUT gate maps to a three-neighbor cell (one neighbor for the input, two neighbors for the outputs).
- Each NOT gate maps to a two-neighbor cell (one neighbor for the input, one neighbor for the negated output).
- Each INPUT gate maps to a one-neighbor cell (one neighbor for the output).
- Each OUTPUT gate maps to a one-neighbor cell (one neighbor for the input).
- Each wire between gates maps to a (possibly empty) path of two-neighbor WIRE cells (one neighbor for the input, one neighbor for the copied output).
- All other cells are zero-neighbor NULL cells.

We construct the layout as follows. Let g be the number of (AND, OR, FAN-OUT, NOT, INPUT, and OUTPUT) gates. Because the graph has maximum degree 3, we can find an orthogonal layout in a $(\frac{g}{2}+1) \times \frac{g}{2}$ grid of cells [114].

We want each x_i input to appear in the top row of the grid, in an even column, with its output on the south side, with all other cells in the top row (including all odd columns) being NULL cells. To guarantee this, we add a top row, move each x_i INPUT gadget to an even column in this top row, and connect the old location of that gadget to the new location via an orthogonally routed edge by adding $O(1)$ new empty rows and columns.

Then we replace each orthogonally routed edge by a path of WIRE cells. Additionally, we scale the coordinates by a constant factor, and replace each crossover between pairs of edges with a crossover gadget built out of AND and NOT gates, implementing a NAND crossover which can be built from the standard 3-XOR crossover [68].

Constructing the board. The adversarial-boundary edge-matching board is divided into two regions, an upper *circuit* region and a lower *garbage* region. In between, we have a single row of tiles called the *wasteline* that is forced to separate the regions (explained later). The circuit region is tileable only if the circuit is satisfied. The garbage region is always tileable; it provides space for tiles that weren't used in the circuit region to be placed.

Circuit region. In the circuit region, each cell in the circuit layout is mapped to a set of tiles, one of which will be used in that position in the solution. The other tile choices for that cell will be garbage collected in the garbage region. The first player's boundary-edge swaps set the x_i inputs to the circuit. The second player will then try to solve the resulting problem by choosing a value for the y_i inputs, then evaluating the circuit; all other tile choices are forced by the inputs.

In the circuit region of the board, all tile edges encode a circuit signal from the set $\{\text{true}, \text{false}, \text{null}\}$. To preserve the circuit layout in all legal placements of the tiles, the top and bottom edges of each tile in the circuit region also encode their position. The tile at position (x, y) (where the origin is the upper-left tile) encodes (x, y) on its top edge and $(x, y + 1)$ on its bottom edge. In this way, we can associate each circuit cell with the set of tiles that may appear in the corresponding slot in the tiling problem.

The set of tiles produced for each cell depends on its type. Unless explicitly mentioned otherwise, the position encoding on the top and bottom edges is as above.

- AND, OR: four tiles encoding the truth-table operation on the edges connecting to other cells. For example, an AND cell with inputs on its top and left edges and output on its right edge would produce four tiles with (F, F, F), (F, T, F), (T, F, F) and (T, T, T) as the circuit signals on its top, left and right edges respectively, with null on its bottom edge.

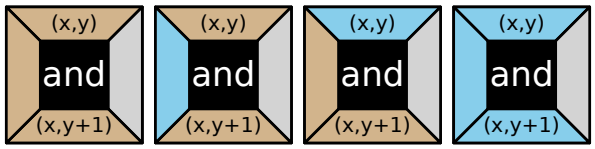
- NOT, WIRE, FANOUT: two tiles encoding their truth table on the edges connecting to other cells and null on the other edges.

- OUTPUT: one tile with the false circuit signal on its connected edge and null on its other edges. Because there is no tile for this cell with true on its connected edge, the tiling can only be completed when the circuit evaluates to false.

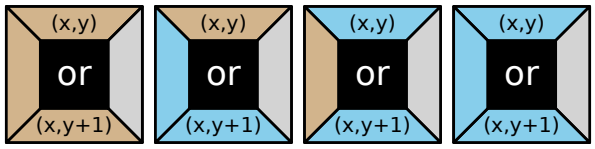
- INPUT representing x_i: two tiles that propagate the circuit signal selected by the first player's top-boundary swap. By the layout process above, these cells are always in the top row and an even column. These tiles encode their column pair number instead of their position on their top edge, along with a circuit signal (so either $(\frac{x}{2}, T)$ or $(\frac{x}{2}, F)$). The bottom edge uses the usual position encoding and propagates the circuit signal (so $((x, 1), T)$ or $((x, 1), F)$, respectively). Their left and right edges always encode the null circuit signal.

- INPUT representing y_i: two tiles, one with true and the other with false on the connecting edge, and null on the other edges.

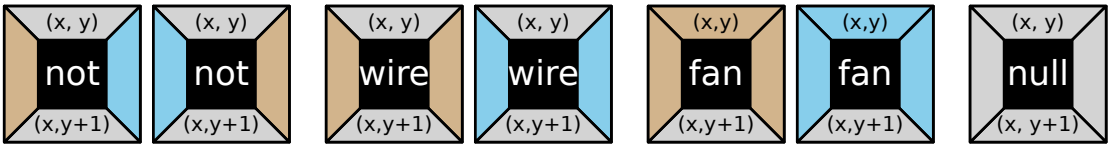
- NULL: one tile with the null circuit signal on all edges. For cells in the top row, the top color encodes $\lfloor \frac{x}{2} \rfloor$ instead of the usual $(x, 0)$ position, so that the first player's swapping does not affect the tile's placeability. Cells in the top row immediately to the right of an INPUT cell representing x_i instead have two tiles in their set, encoding true and false circuit signals on their top edge (and null on their other edges), to terminate whichever signal the first player did not assign to x_i.



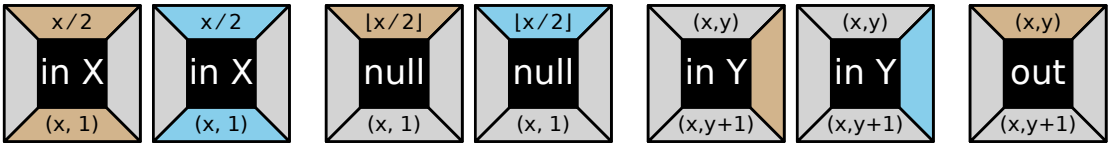
(a) AND with inputs from up and left and output down.



(b) OR with inputs from up and left and output down.

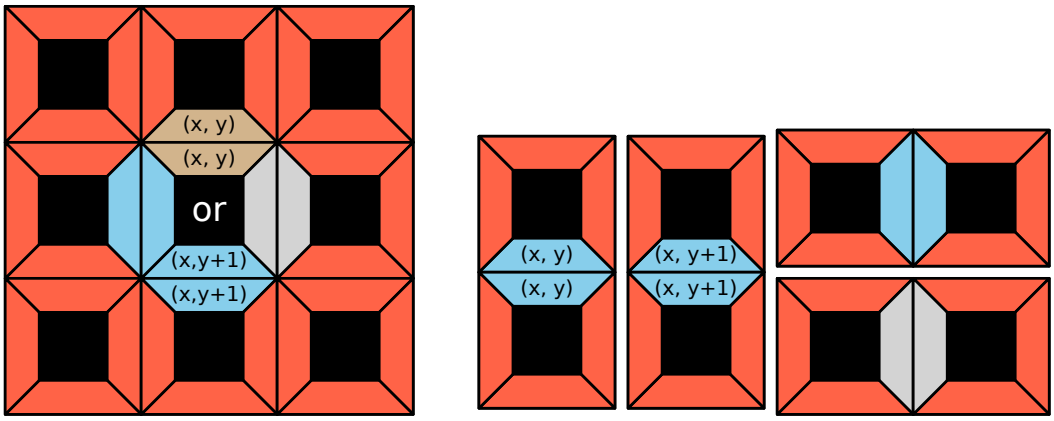


(c) Horizontal NOT. (d) Horizontal WIRE. (e) FANOUT. (f) NULL.



(g) INPUT representing x_i. (h) Special NULLs to the right of INPUTs representing x_i. (i) INPUT representing y_i, output on the right. (j) OUTPUT, input from up.

Figure 7-58: Examples of tile sets produced for each type of cell in the circuit region. x and y represent the position of the cell and tan, blue and gray represent the false, true and null circuit signals respectively. Except for the INPUTs representing x_i and their special NULL neighbors, each tile type can be instantiated with connecting edges in any direction.



(a) One of the three 3×3 garbage blocks encapsulating the unused tiles.

(b) The garbage dominoes collecting the garbage collection tiles that would have been used for the both-inputs-true OR tile had it not been placed in the circuit region.

Figure 7-59: Collecting garbage for the OR cell with tile set shown in Figure 7-58b, assuming the rightmost (both inputs true) tile was placed in the circuit region.

Wasteline. The wasteline is a row of tiles that transitions from the circuit region to the garbage region. The top of each tile encodes the null circuit signal and the position of the cell in the same way as in the circuit region. The left and right edges in each tile are unique colors encoding the column of the tile, forcing the wasteline to occur in order as a contiguous row in the tiling. The bottom of each tile is the garbage color g.

Garbage region. The garbage region provides space for the unused tiles from each set to be placed. For each tile t in the circuit region (even if it's the only tile in its set), we emit four garbage tiles, each having one of t's edge colors on one side and g on the other three sides, and four tiles having g on all four edges, allowing that tile to be encapsulated into a 3×3 block having g on its exterior, as in Figure 7-59a.

Consider the boundary edges of the circuit region (that is, the entire top board boundary, parts of the left and right board boundary, and the top edges of the wasteline). In a valid tiling, these edges will always match a circuit tile edge, so for each of those edges we remove a corresponding garbage collection tile (with that boundary color and three g edges) that we just produced.

We then need to provide a way to garbage collect the garbage collection tiles for the used tiles. Each nonboundary edge of each used tile corresponds to an edge of a neighboring used tile, so we can pair the corresponding garbage collection tiles to create a vertical or horizontal domino having g on its exterior, as in Figure 7-59b.

Finally, we need to provide exactly enough tiles having g on all four sides to fill the garbage region. If the circuit region is w tiles wide and h_c tiles tall (assuming $w \geq 3$) and we produced t total circuit tiles, we have $t - wh_c$ 3×3 blocks, $2wh_c - 2w$ vertical dominoes and $2wh_c - 2h_c$ horizontal dominoes to pack. Using a simple strategy of packing the 3×3 blocks horizontally in as many rows as necessary, followed by the

vertical dominoes and horizontal dominoes, each type starting on a new row, the garbage region has height $h_g = 3\lceil\frac{3(t-wh_c)}{w}\rceil + 4(h_c - 1) + \lceil\frac{4h_c(w-1))}{w}\rceil$. We provide $wh_g - 9(t - wh_c) - 2(4wh_c - 2w - 2h_c)$ tiles with g on all four sides to pack the remaining area in the garbage region.

Boundary-colored board. The board's width is determined by the width of the circuit region, which is in turn determined by the circuit layout. The height is the circuit region height, plus 1 for the wasteline, plus the garbage region height (i.e., $h_c + 1 + h_g$). The top edge of the board is colored $(\lfloor\frac{x}{2}\rfloor, T)$ and $(\lfloor\frac{x}{2}\rfloor, F)$ above the INPUT cells corresponding to x_i variables and the NULL cells immediately to their right, and $(\lfloor\frac{x}{2}\rfloor, N)$ elsewhere. The left and right edges of the board above the wasteline have the null circuit signal color. The left and right edges at the wasteline have the unique colors on the left (right) edge of the leftmost (rightmost) wasteline tile. Below the wasteline, the left, right and bottom edges are all colored g.

Argument. The first player swaps pairs of top-boundary edges, then the second player attempts to tile the resulting board.

Garbage tiles cannot be placed in the circuit region. Because each garbage tile has g on three or four of its edges, there is no placement of garbage tiles with non-g colors on its entire exterior. Thus, if any garbage tiles are placed in the circuit region, the entire region must be filled with garbage tiles, but this is impossible because the corners of the boundary of the circuit region have two non-g edges.

The wasteline's placement is fixed by the unique colors on the left and right boundary. The wasteline tiles encode their position on their top edge, so this also fixes all the tiles above in place; each tile in the circuit region must be from the set created for the corresponding circuit cell. Then the tiles corresponding to the x_i INPUT cells must correctly conduct the circuit signal (true or false) from the boundary. (Swaps of the boundary above the NULL cells in the top row change nothing because those edges have the same color.)

If the second player knows an assignment to the y_i variables that makes the circuit evaluate to false, they can select the tiles conducting that circuit signal from the sets corresponding to their INPUT cells. After that, the tiling in the circuit region is forced; from the sets of tiles corresponding to gate and wire cells, only one tile can be placed (based on the inputs from its connecting cells), and its output connection forces further cells. If the circuit indeed evaluates to false, then the tiling can be completed at the OUTPUT cell, whose only tile requires its input to be false. In that case, the remaining tiles can be garbage collected (by the construction of the garbage region given above), and the second player wins the game.

Otherwise, if the circuit evaluates to true under all y_i variable settings, there is no way to place a tile that connects to the OUTPUT cell's tile, and the first player wins the game.

Losing strategy guarantee. We can easily adapt this proof to guarantee that the first player always has a *losing* strategy in the produced adversarial-boundary

edge-matching instance (regardless of whether they have a winning strategy). Simply amend the QSAT_2 instance by adding an additional existential variable x_{n+1} which is conjoined with the formula $(f(\dots) \wedge x_{n+1})$. Performing the boundary-edge swap corresponding to setting x_{n+1} to FALSE is a losing strategy.

Tile rotation. The reduction above remains valid when tile rotation is allowed. The wasteline's position and orientation are fixed by the unique boundary colors, then the position encoding on the top boundary and top edge of the wasteline prevents the circuit tiles from being rotated. Allowing the garbage collection tiles to rotate is harmless because they still cannot be placed in the circuit region.

Signed colors. The reduction above produces an unsigned edge-matching instance, but every unsigned edge-matching puzzle can be converted into a signed edge-matching puzzle with a constant-factor increase in size and the number of colors [44, Theorem 4]. □

Chapter 8

A Proposal for Parsing Screenshots of Grid-based Games[1]

8.1 Motivation

Video games are a rich source of interesting search problems like those considered in this thesis, but extracting the problem instances from games is a source of accidental challenge. Transcribing instances by hand is tedious and error-prone. Extracting instances by reverse-engineering the game's data files requires a different skillset from the study of the search problem, and is not possible if the instances are procedurally generated. The most obvious approach is to parse screenshots from the game; unsophisticated parsers based on testing the colors of specific pixels are inflexible and work only for simple games, while sophisticated computer vision processing again requires a different skillset.

As an enabling technology for the study of these search problems, we propose to build a generic framework for parsing grid-based video game images using a high-level description of the relationships between primitive elements in the screenshots. The framework will use existing techniques from image processing and computer vision to locate the primitive elements in the image and fit them to the high-level description. The parser can be used directly in a game-playing program ("bot"), or the parsed game state can be rendered into human- or machine-readable text or back into an image format such as SVG (useful for writing papers about the game or the resulting search algorithms).

To get an idea for what such a framework would need, we built several parsers tailor-made for specific games as part of writing programs that play those games.

LYNE. LYNE is a game played on dense grid graphs of triangles, squares, diamonds, and octagons (see Figure 8-1a). There are two terminals of each shape except octagons, denoted by a light color at the center of that shape. The objective is to connect the terminals of each shape via a path containing only nodes of that shape and octagons.

[1] Screenshots in this chapter are used under Fair Use for the educational purpose of describing an approach to parsing them. All trademarks are the property of their respective owners.

Each octagon must be visited by paths a number of times equal to the number of small squares inset in the octagon.

We have written a complete bot[2] that can automatically solve all levels in the game. The bot parses a puzzle by building connected component regions of RGB pixel values, filtering out shapes by their colors, and recovering the grid by aligning the centers of the connected components. The bot finds terminals and octagon pips by looking for regions of the terminal or pip colors inside the bounding boxes of the shapes. This parser does not attempt to recognize the shapes except by their color.

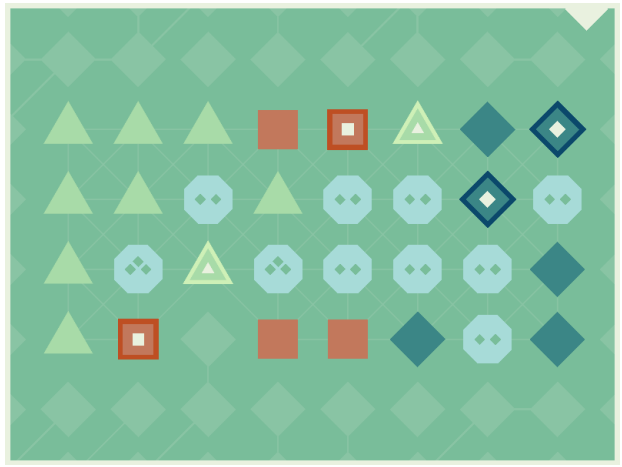
Sokobond. Sokobond is a Sokoban-like game in which the player moves a particular atom through the faces of a grid graph (see Figure 8-1b). When two atoms with free electrons become adjacent, they form a bond and move together from then on. Fixed objects located at grid vertices break bonds, form double or triple bonds, or rotate a molecule. The level is solved when no atom has free electrons, usually reached by forming a single molecule from all atoms.

We have written a bot[3] that solves some Sokobond puzzles. The bot's parser recognizes atoms by color, then finds free electrons by looking for connected components of white pixels inside the atom's black outline and finds bonds by looking between each pair of adjacent atoms. Empty squares, the puzzle boundaries, and grid vertex objects are similarly detected by color. Free electrons rotate around their parent atom, sometimes occluding bonds, so the bot takes multiple screenshots and checks that they agree before calling the solver.

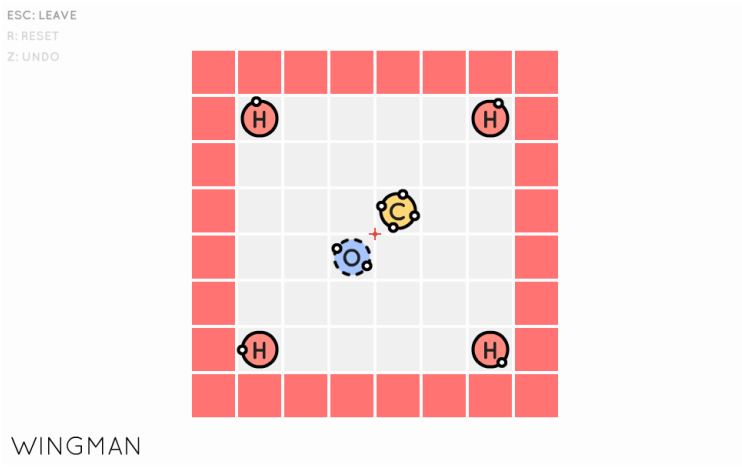
Hexcells. Hexcells is a series of Minesweeper-like games of logical deduction played on a hexagonal grid (see Figure 8-1c). Each cell in the puzzle is present or absent. Clues along axis-aligned rays through the grid indicate the number of present cells on that ray, and clues in absent cells (revealed only after marking the cell as absent) indicate the number of present cells in their neighborhood. The objective is to mark all the present cells. Each Hexcells game includes hand-crafted levels of increasing difficulty, and the last game in the series, Hexcells Infinite, includes a procedural puzzle generator.

We began to implement a bot that plays Hexcells, but stopped because we could not reliably parse puzzles. Our attempt at a Hexcells puzzle parser recognizes hexes by their color and recovers the grid structure by aligning the centers of the hexes (similar to our parser for LYNE puzzles). The parser collects clues along lines of hexes by considering the grid to extend for one further hex in each direction. The parser attempts to recognize clues (numbers possibly surrounded by braces or dashes) by nearest-neighbor comparison with a human-classified training set, but this recognition is unreliable due to antialiasing causing the same clue to render differently in different grid positions.

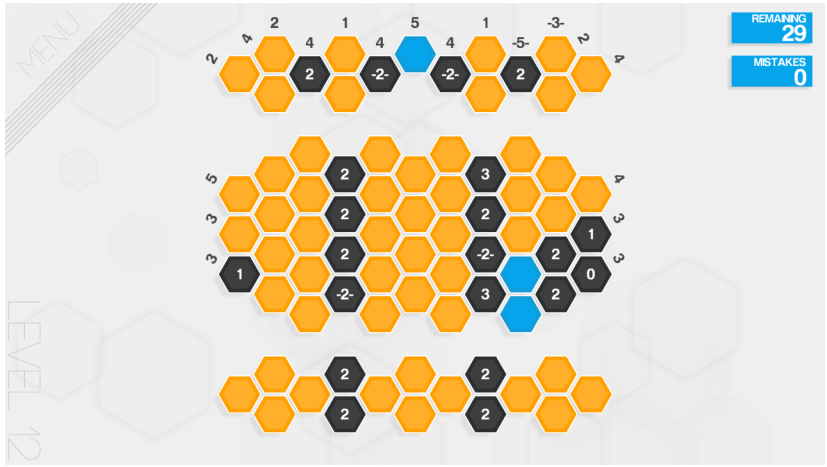
[2] https://github.com/jbosboom/lynebot
[3] https://github.com/jbosboom/sokobondbot



(a) LYNE



(b) Sokobond



(c) Hexcells

(d) Sigmar's Garden

Figure 8-1: Screenshots from games we have written bots for.

Sigmar's Garden. Sigmar's Garden is a marble-matching solitaire minigame (see Figure 8-1d) included in the alchemy-themed programming game Opus Magnum. Marbles of the four classical elements match with themselves or with salt, salt matches with itself, vitae matches with mors, and metals match with quicksilver in order from lead to gold. Only marbles with three contiguous adjacent empty spaces (including spaces off the edge of the board) can participate in a match, with other marbles being faded to indicate their inactivity. Puzzles are procedurally generated and every instance is winnable.

We have written a bot[4] that solves Sigmar's Garden puzzles. The bot's parser has hardcoded pixel coordinates of the centers of each hex on the game board, found by running a circle detector on a screenshot of an empty board. The bot classifies marbles by extracting a circular region from each hex, building four histograms, and comparing those histograms to average histograms from each marble class in human-classified training data (a nearest-centroid classifier). The first histogram is of RGB values, primarily serving to distinguish active marbles, as inactive marbles are faded and desaturated. The second histogram is of circle sizes found by a Hough circle detector; this histogram is intended to distinguish salt and quicksilver marbles, which are the same color but whose symbols include circles of different sizes. The remaining two histograms are of the radial distribution (bins on angle and distance from center) of keypoints found by the GFTT and MSER feature detectors; GFTT tends to find keypoints on the marble's symbol, while MSER is more sensitive to the non-symbol area of the marble. The bot uses the sum of OpenCV's "alternate" chi-squared distance between corresponding histograms as its distance score. This parser is reasonably but not perfectly accurate; the bot successfully obtained the achievement for solving 100 puzzles, but only with a human supervising and manually starting a new game when the bot misparsed a puzzle.

The Witness. The Witness is a first-person adventure and exploration game, but much of it revolves around two-dimensional line-drawing puzzles presented on panels in the three-dimensional environment. Solutions to a Witness puzzle are lines from a circle to a semicircle such that the constraints imposed by clues on grid vertices, edges, and faces are all satisfied. For a full description of Witness puzzles, see Section 7.1.

We have written a bot that solves most of the panel puzzles in the Challenge, an end-game area containing a timed series of randomized puzzles. Human input is still required to navigate from panel to panel in the environment. The bot's parser is specialized for the Challenge and does not generalize to other puzzles. For each puzzle in the Challenge, the panel's pixel coordinates, start and end vertices, and path and background colors are hardcoded in the parser, except for the puzzles containing triangle clues, where these values are dynamically determined based on the puzzle's vertices and edges being of a specific color. Clues are recognized by their color; while the colors are randomized, they are drawn from disjoint lists so there is no ambiguity, especially because the parser has hardcoded knowledge of which clues can appear in which puzzles. Four puzzles in the middle of the Challenge are displayed in random

[4] https://github.com/jbosboom/sigmars-gardener

locations; the parser determines which location it is at (and so which set of hardcoded coordinates to use) based on summing the total color distances of a hardcoded set of pixels (a nearest-neighbor classifier). The fourth puzzle is not parsed because it is laid on a table and the parser cannot correct for perspective distortion, and the cylindrical puzzles at the end of the Challenge are not parsed because the parser cannot stitch multiple screenshots together.

8.2 Parsing with Grammars

All of the games we have written parsers for, many other puzzle games, and some games of other genres (e.g, Advance Wars, a turn-based tactics game) are visually based on square or hexagonal grids. The Witness is even recursively grid-based, as polyomino and antipolyomino clues are grids within faces. The grid structure of these games makes them amenable to high-level description organized around the vertices, edges, and faces of the grid. In this section, we present a design for a framework for parsing screenshots of grid-based video games using a high-level, grid-structured description of the relationships between primitive elements in the screenshots.

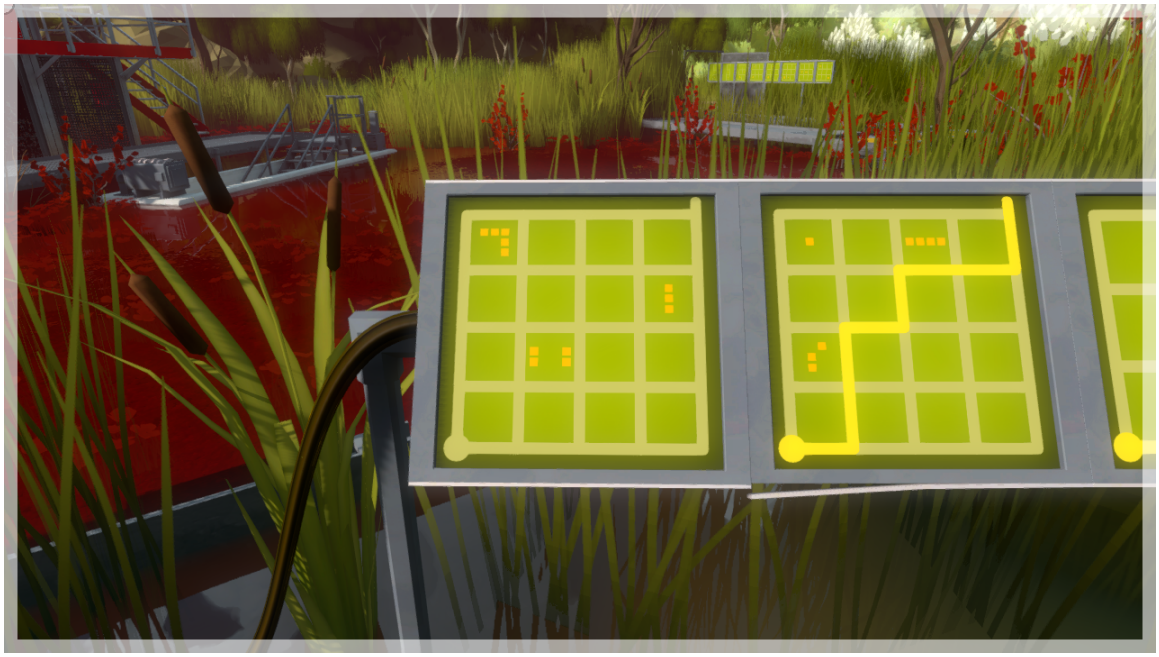


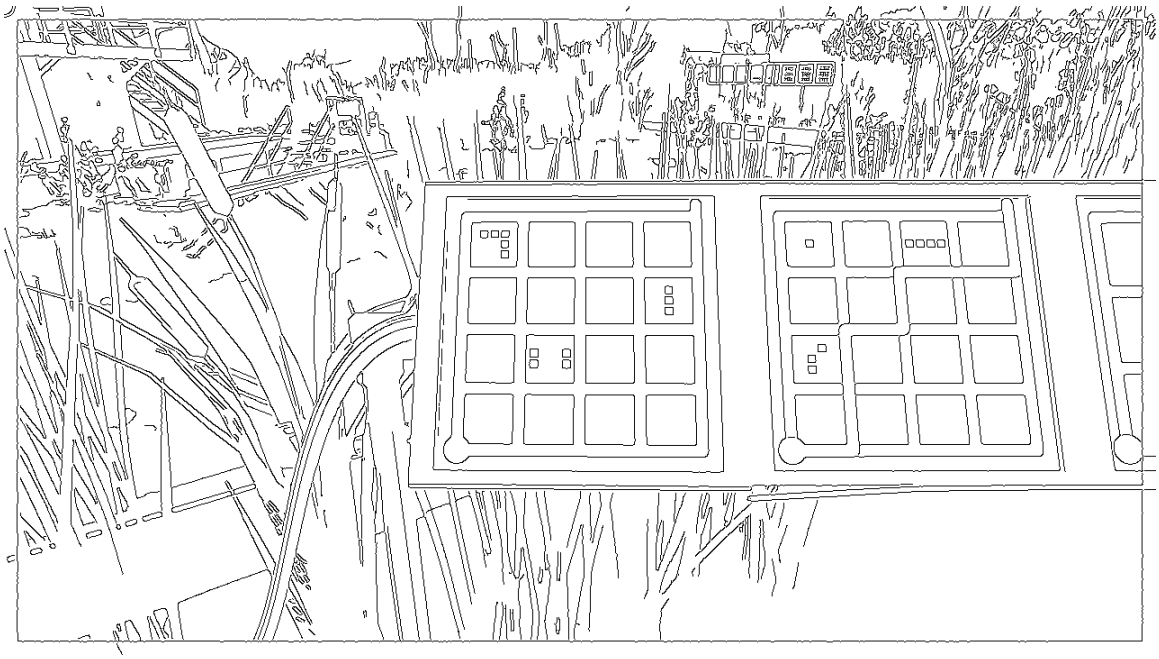
Figure 8-2: A screenshot from The Witness.

Consider the screenshot in Figure 8-2, which we would like to parse against the (partial) high-level description of a Witness puzzle in Listing 8.1. Together with the high-level description, the feature information shown in Figure 8-3 is enough information to parse the puzzle in the center of the image (and to reject the other puzzles):

1. First, we can localize the puzzles because they are low-energy areas: they have a low Canny edge density, low Hough circle density, and low Shi-Tomasi corner

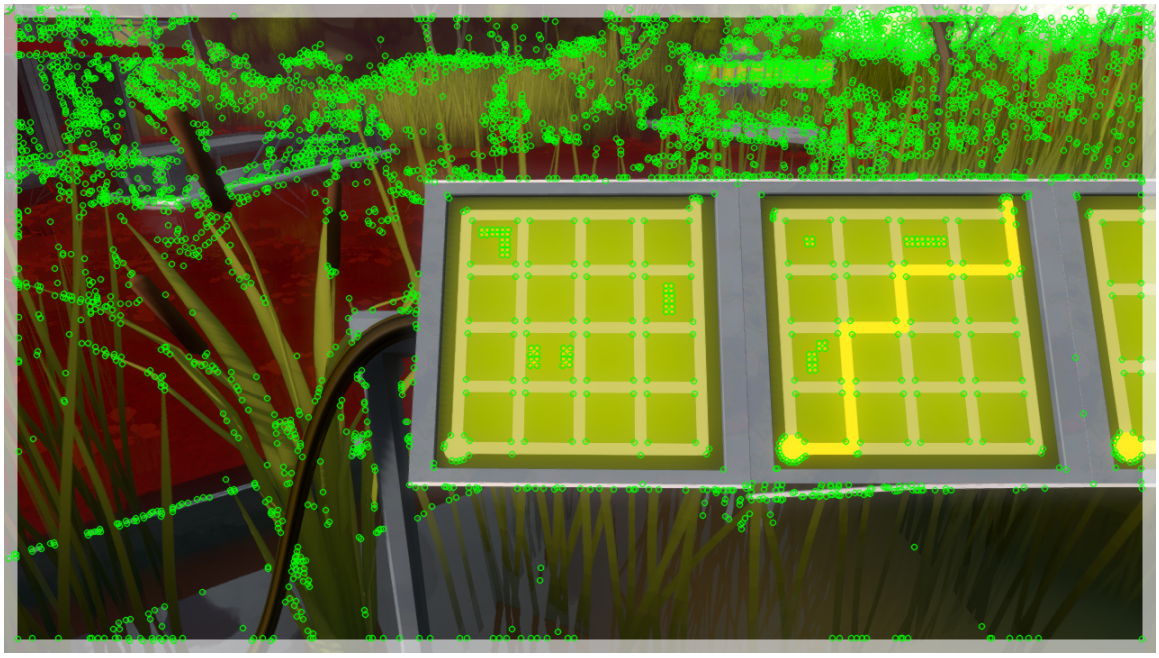
```
puzzle:
  shape: square grid with vertices, edges, faces
  location: usually centered
  grid:
    face rows: <= 20
    face cols: <= 20
    vertex:
      shape: square
      color:
        - consistent-within parent puzzle
        - same-as edge
        - different-than face
      children:
        start-circle:
          shape: circle
          color: same-as parent vertex
          diameter: >= 1.3 * parent vertex.width
          count: >= 1 within parent puzzle
        end-cap:
          shape: semicircle
          color: same-as parent vertex
          count: >= 1 within parent puzzle
    edge:
      shape: rectangle
      color:
        - consistent-within parent puzzle
        - same-as vertex
        - different-than face
    face:
      shape: square
      color:
        - consistent-within parent puzzle
        - different-than vertex
        - different-than edge
      width: >= 2 * min(edge.width within parent puzzle)
      height: >= 2 * min(edge.height within parent puzzle)
      children:
        polyomino:
          shape: square grid with vertices
          grid:
            sparse: vertices
            vertex rows: <= parent puzzle.face rows
            vertex cols: <= parent puzzle.face cols
            vertex:
              shape: square
              color:
                - different-than parent face
                - consistent-within parent face
                - usually consistent-within parent puzzle
                - usually near rgb(255,199,12)
```

Listing 8.1: Partial grammar for Witness puzzles. See Listing 8.3 for the full grammar.



(a) Canny edge detection.

(b) Hough circle detection.



(c) Shi-Tomasi corner detection.

(d) MSER feature detection.

Figure 8-3: The result of processing the screenshot in Figure 8-2.

density. (The lower-right corner of the image (below the puzzle panels) is also low-energy.)

2. The Shi-Tomasi corner detector finds the corners of each face of the grid, the MSER detector finds a region center in the center of each face, and the Canny edge detector finds the boundaries of each face and the outer boundary of each grid. From the puzzle definition, we know a puzzle is a square grid with vertices, edges, and faces. By fitting these points to a grid, we can recognize there are two full panels in the image, and that there is not a panel in the lower-right corner. We can also correct for perspective distortion and re-run the feature detectors, though doing so is not necessary in this example.

3. The Hough circle detector finds the start circle and end semicircle of the center panel, and the start circle of the right panel. This confirms there are two separate panels. (We can reject the stray circle inside the right panel's face because, according to the high-level description, circles cannot appear in faces; only squares in a sparse square grid can.)

4. The vertices and edges in the right panel are not all of the same color, apparent both by direct color comparison and by stray edges found by the edge detector. Additionally the end cap was not found in the right panel and the center panel is more centered, as the high-level description states is usually the case. Thus, between the two panels, we can conclude the center one is a better fit to the high-level description, and so probably the puzzle we seek to parse.

5. Each square in the polyomino clues in the faces of the center panel are detected as MSER regions and their corners are Shi-Tomasi corners. We can again fit a grid to the points within each face to recover the shape of the polyomino clues.

6. Using the constraint on the size of the grid in a polyomino clue and that previous steps determined the panel to be a 4×4 grid (of faces), the disconnected polyomino clue in the lower-right quadrant of the puzzle likely has two empty vertices separating its components. (If the panel was only three faces wide, then only one empty vertex could be present.)

This results in a parse tree mirroring the hierarchical structure of the puzzle description. Omitting empty vertices, edges, and faces for brevity, we get the puzzle definition in Listing 8.2. We could emit a more concise puzzle definition by writing code that traverses the parse tree. If we are using the parser as part of a bot, we would additionally retain the pixel coordinates of each puzzle element, so that the bot can draw a solution path by simulating a click on the start circle, drawing the path along edges and vertices, and finishing at the end semicircle.

The screenshots in Figure 8-3 were produced by OpenCV sample code with minimal modifications, and OpenCV already includes code for grid fitting and camera calibration (removing perspective distortion). OpenCV includes additional feature detectors not shown here, and should these not prove sufficient, there are many

```
puzzle:
  grid:
    face rows: 4
    face cols: 4
    edge rows: 5
    edge cols: 5
    vertex:
      row: 4
      col: 0
      start-circle: # no children
    vertex:
      row: 0
      col: 4
      end-cap: # no children
    face:
      row: 0
      col: 0
      polyomino:
        grid:
          vertex rows: 3
          vertex cols: 3
          vertex:
            row: 0
            col: 0
          vertex:
            row: 0
            col: 1
          vertex:
            row: 0
            col: 2
          vertex:
            row: 1
            col: 2
          vertex:
            row: 2
            col: 2
    face:
      row: 1
      col: 3
      polyomino:
        grid:
          vertex rows: 2
          vertex cols: 1
          vertex:
            row: 0
            col: 0
          vertex:
            row: 1
            col: 0
    face:
      row: 2
      col: 1
      polyomino:
        grid:
          vertex rows: 2
          vertex cols: 4
          vertex:
            row: 0
            col: 0
          vertex:
            row: 0
            col: 3
          vertex:
            row: 1
            col: 0
          vertex:
            row: 1
            col: 3
```

Listing 8.2: The result of parsing Figure 8-2.

algorithms described in the image processing literature not yet implemented in OpenCV. It should not be necessary to do novel image processing.

Integration system. Instead, the interesting part of our proposed framework is the system that integrates the low-level information to produce a puzzle fitting the high-level description. Some low-level feature information is naturally expressed in the high-level description, particularly shapes, colors and sizes. The features produced by feature detection algorithms such as SURF or ORB are less intuitive to humans. The programmer can annotate the high-level description with this information, but it may be more convenient to have the framework learn weights and parameters from a small set of screenshots and corresponding puzzles.

Because the goal of the framework is to relieve humans from manually transcribing puzzles, the training data available to learn from will necessarily be very limited, small enough that it might be more accurately called "fitting" to the data. For example, The Witness has approximately 300 panels[5], so transcribing 30 of them is already 10% of the entire task. We are relying on the high-level description to provide structure, reducing the number of parameters and weights that must be learned; for simple games we expect not to need any training data. We still expect machine learning to be useful for or classifying individual puzzle elements (for an example from our bots, learning to read the clues in Hexcells). Part of this research is to investigate the tradeoffs between learning from annotated images and annotating the high-level description.

If the parser is being used as part of a bot, the bot can ask a human to label puzzles where the parser had to ignore some features to fit the remaining features to the high-level description. If the bot can detect when the game rejects a puzzle solution, those images can also be presented to the human (in this case, the solver may also have a bug). In this way, each human annotation is known to be helpful, in that it corrects a mistake made by the parser. (Truly minimizing the amount of training data necessary requires a non-incremental approach.)

We can also learn from successful parses by increasing the weight of features that supported the successful parse and decreasing the weight of features that disagreed (reinforcement learning). Reinforcement is particularly applicable to puzzle games, as they usually have a difficulty curve for the benefit of human players; that is, puzzle elements are gradually introduced and puzzle complexity gradually increases. Easy puzzles are the most likely to parse unambiguously using only information from the high-level description (such as shape and color), so the parser has an opportunity to learn which features are useful before confronting more complex puzzles later in the game.

In our exploratory attempts at building the proposed framework, unifying bottom-up and top-down parsing was a challenge. Most of the parsers in our bots parsed the image bottom-up, first finding elements appearing at the leaves of the high-level description, then recovering the grid structure. In contrast, the approach to parsing

[5]The Witness actually has 523 panels, but many of them depend on the three-dimensional environment or are otherwise not grid-based.

The Witness presented earlier in this section works top-down, first localizing the puzzle in the image, then finding the grid structure, and finally recognizing elements inside the grid. An ideal parser would instead use Bayesian reasoning to parse at multiple levels at the same time, gaining confidence in its parse when the arrangement of features matches the high-level description. When used by bots that can detect when the game rejects a puzzle solution, such a parser could provide first the most likely parse, then if an error is detected, the next most likely parse.

Code generation. Aside from its use in parsing, the high-level description can also be used to generate high-level puzzle representations in various programming languages using existing systems for data representation synthesis [72, 95]. High-performance searches use sophisticated low-level representations to make common operations fast, often trading higher asymptotic complexity for reduced constant factors, so these representations will not be suitable within the search itself, but they allow the searches to take a convenient representation as input instead of a generic machine-readable text format. The synthesized representation could also be the basis of tools for visualizing a puzzle solution or for implementing puzzle transformations (for example, reductions between different problems).

8.3 Related Work

Video game definition languages. There are a number of languages that define video games for the purposes of procedural level generation [85, 99], general game playing agent research [55, 125], evolutionary or procedural game design [40, 97, 35, 34, 129], or as domain-specific programming languages for game development [96]. These languages necessarily focus on defining the semantics of the game; for example, VGDL includes a collision matrix defining what should happen when various types of sprites intersect. In contrast, our high-level description describes only what a puzzle looks like, not what it means; the example in Listing 8.1 defines `start-circle` and `end-cap`, but those are just identifiers with no meaning to the parser. Semantic information might be useful for resolving ambiguous or incorrect parses, but our parser relies on feedback from the solver when a parsed puzzle is logically inconsistent or when the game rejects a solution.

See also [135] for a survey of search-based procedural content generation for video games, language-based and otherwise.

Sketch understanding. The field of sketch understanding aims to make pen/stylus input a viable mode of user interaction with computers by understanding the structures being sketched, for example, chemical structures or circuit diagrams. This proposed work is closest to SketchREAD [11], which uses the LADDER [70] "hierarchical shape description language" to assign semantic meanings to shapes and relations, similar to our proposed high-level description of a puzzle. Besides the obvious differences between vector (pen) and raster (screenshot) inputs, pen input also has a temporal component (the sequence of strokes) that screenshots lack. SketchREAD exploits

this with a bottom-up Bayesian network inference approach, generating and pruning hypotheses incrementally with pen strokes. In contrast, we get our input all at once in the screenshot.

The ChemInk [112] parser for sketches of chemical structure incorporates domain knowledge about chemical valence to recover from incorrect parses. Similarly, our high-level description allows our parsers to determine that particular puzzle elements cannot appear in certain locations, and so some features must be disregarded.

Visual languages. There are many formal and semi-formal models of visual languages for human-computer interaction, visual programming, or computer vision; for a survey, see [98]. Our high-level descriptions can be viewed as a grammar in the sense that they have hierarchical structure. However, our descriptions also describe the primitive elements rather than treating them as abstract symbols, and our parsers also function as lexers in that they recognize individual elements. On the other hand, we do not attempt to give a complexity analysis of our parser framework nor to precisely define the class of languages it can parse.

Fusion of simple/generic features. Uijlings et al. [140] propose image regions representing three-dimensional objects in real-world images by evaluating multiple simple feature detectors in multiple color spaces in small regions and joining them bottom-up. A more expensive object recognizer is then applied to these regions such that regions proposed by multiple proposal algorithms are prioritized. We also use fusion of simple, game-independent features.

Direct play. Mnih et al. [105] describe a deep-learning-based system that plays Atari 2600 games directly taking only the raw pixels as input. While an impressive accomplishment, our objective is instead to extract the puzzles for separate study. Deep learning methods seem of limited value here due to a lack of training data. There are only 523 panels in The Witness, and we only expect to parse about half of those; if we were to apply deep learning using hand-labeled examples (either as training data or to evaluate the performance of reinforcement learning), we would likely need to transcribe all the puzzles we hope to parse, rendering our effort moot.

8.4 Full Grammar for The Witness

To give a deeper sense of the style of grammar to be used by the proposed framework, Listing 8.3 is a possible full grammar for parsing puzzles from The Witness.

```
1 puzzle:
2   shape: square grid with vertices, edges, faces
3   location: usually centered
4   grid:
5     face rows: <= 20
6     face cols: <= 20
7     vertex:
8       shape: square
```

```
      color:
        - consistent-within parent puzzle
        - same-as edge
        - different-than face
      children:
        start-circle:
          shape: circle
          color: same-as parent vertex
          diameter: >= 1.3 * parent vertex.width
          count: >= 1 within parent puzzle
        end-cap:
          shape: semicircle
          color: same-as parent vertex
          count: >= 1 within parent puzzle
        vertex-hexagon:
          shape: hexagon
          color: usually near rgb(90,90,90)
        vertex-symmetry-hexagon:
          shape: hexagon
          color:
            - usually near rgb(225, 174, 4) # orange
            - usually near rgb(137, 211, 136) # blue
          remember: color
          requires:
            - >= 2 start-circle within parent puzzle
            - >= 2 end-cap within parent puzzle
    edge:
      shape: rectangle
      color:
        - consistent-within parent puzzle
        - same-as vertex
        - different-than face
      children:
        edge-hexagon:
          shape: hexagon
          color: usually near rgb(90,90,90)
        edge-symmetry-hexagon:
          shape: hexagon
          color:
            - usually near rgb(225, 174, 4) # orange
            - usually near rgb(137, 211, 136) # blue
          remember: color
          requires:
            - >= 2 start-circle within parent puzzle
            - >= 2 end-cap within parent puzzle
        broken-edge:
          shape: rectangle
          color: different-than parent edge
    face:
      shape: square
      color:
        - consistent-within parent puzzle
        - different-than vertex
        - different-than edge
```

```
      width: >= 2 * min(edge.width within parent puzzle)
      height: >= 2 * min(edge.height within parent puzzle)
      children:
        square:
          shape: rounded square
          color: different-than parent face
          remember: color
        star:
          corners: 8
          color: different-than parent face
          remember: color
        antibody:
          corners: 9
          color: usually rgb(255, 255, 255)
          remember: color # relevant for star clues
        polyomino:
          shape: square grid with vertices
          grid:
            sparse: vertices
            vertex rows: <= parent puzzle.face rows
            vertex cols: <= parent puzzle.face cols
            vertex:
              shape: square
              color:
                - different-than parent face
                - consistent-within parent face
                - usually consistent-within parent puzzle
                - usually near rgb(255,199,12)
        rotatable-polyomino:
          shape: square grid with vertices rotated 15 degrees
          grid:
            sparse: vertices
            vertex rows: <= parent puzzle.face rows
            vertex cols: <= parent puzzle.face cols
            vertex:
              shape: square
              color:
                - different-than parent face
                - consistent-within parent face
                - usually consistent-within parent puzzle
                - usually near rgb(255,199,12)
        antipolyomino:
          shape: square grid with vertices
          grid:
            sparse: vertices
            vertex rows: <= parent puzzle.face rows
            vertex cols: <= parent puzzle.face cols
            vertex:
              shape: square
              stroke-color: near rgb(69, 95, 188)
              fill-color: parent face.color
        rotatable-antipolyomino:
          shape: square grid with vertices rotated 15 degrees
          grid:
```

```
sparse: vertices
vertex rows: <= parent puzzle.face rows
vertex cols: <= parent puzzle.face cols
vertex:
  shape: square
  stroke-color: near rgb(69, 95, 188)
  fill-color: parent face.color
```

Listing 8.3: Grammar for Witness puzzles.

Bibliography

[1] Zachary Abel, Jeffrey Bosboom, Michael Coulombe, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, Mikhail Rudoy, and Clemens Thielen. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. *CoRR*, abs/1804.10193, 2018. URL: http://arxiv.org/abs/1804.10193, arXiv:1804.10193.

[2] 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*, to appear.

[3] Zachary Abel, Jeffrey Bosboom, Erik D. Demaine, Linus Hamilton, Adam Hesterberg, Justin Kopinsky, Jayson Lynch, and Mikhail Rudoy. Who witnesses The Witness? Finding witnesses in The Witness is hard and sometimes impossible. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, pages 3:1–3:21, La Maddalena, Italy, June 2018.

[4] Aaron B. Adcock, Erik D. Demaine, Martin L. Demaine, Michael P. O'Brien, Felix Reidl, Fernando Sánchez Villaamil, and Blair D. Sullivan. Zig-zag Numberlink is NP-complete. *Journal of Information Processing*, 23(3):239–245, 2015. doi:10.2197/ipsjjip.23.239.

[5] Aviv Adler, Michael Biro, Erik Demaine, Mikhail Rudoy, and Christiane Schmidt. Computational complexity of numberless Shakashaka. In *Proceedings of the 27th Canadian Conference on Computational Geometry (CCCG 2015)*, Kingston, Canada, August 2015.

[6] Aviv Adler, Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu, and Jayson Lynch. Tatamibari is NP-complete. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, pages 1:1–1:24, La Maddalena, Italy, September 28–30 2020.

[7] Aviv Adler, Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Quanquan C. Liu, and Jayson Lynch. Tatamibari is NP-complete. *CoRR*, abs/2003.08331, 2020. URL: https://arxiv.org/abs/2003.08331, arXiv:2003.08331.

[8] Addison Allen, Daniel Packer, Sophia White, and Aaron Williams. Pencils and Sto-Stone are NP-complete [paper in review]. https://www.researchgate.net/project/Computational-Complexity-of-Video-Games-and-Puzzles/update/5aa7c402b53d2f0bba57bfb8, 13 March 2018.

[9] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Rijksuniversiteit Limburg te Maastricht, 1994.

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

[11] Christine Alvarado and Randall Davis. SketchREAD: a multi-domain sketch recognition engine. In Steven Feiner and James A. Landay, editors, *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology, Santa Fe, NM, USA, October 24-27, 2004*, pages 23–32. ACM, 2004. URL: http://doi.acm.org/10.1145/1029632.1029637, doi:10.1145/1029632.1029637.

[12] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic CSPs application to configuration. *Artif. Intell.*, 135(1-2):199–234, 2002. doi:10.1016/S0004-3702(01)00162-X.

[13] Walker Anderson, Erik D. Demaine, and Martin L. Demaine. Spiral Galaxies font. In Jennifer Beineke and Jason Rosenhouse, editors, *The Mathematics of Various Entertaining Subjects (MOVES 2017)*, volume 3, pages 24–30. Princeton University Press, 2019.

[14] Daniel Andersson. HIROIMONO is NP-complete. In *Proceedings of the 4th International Conference on FUN with Algorithms*, volume 4475 of *Lecture Notes in Computer Science*, pages 30–39, 2007. URL: http://www.brics.dk/RS/07/1/BRICS-RS-07-1.pdf.

[15] Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109(19):1145–1146, 2009.

[16] Joshua Ani, Jeffrey Bosboom, Erik D. Demaine, Yevhenii Diomidov, Dylan Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, pages 3:1–3:23, La Maddalena, Italy, September 28–30 2020.

[17] Joshua Ani, Jeffrey Bosboom, Erik D. Demaine, Yevhenii Diomidov, Dylan H. Hendrickson, and Jayson Lynch. Walking through doors is hard, even without staircases: Proving PSPACE-hardness via planar assemblies of door gadgets. *CoRR*, abs/2006.01256, 2020. URL: https://arxiv.org/abs/2006.01256, arXiv:2006.01256.

[18] Joshua Ani, Erik D. Demaine, Dylan H. Hendrickson, and Jayson Lynch. Trains, games, and complexity: 0/1/2-player motion planning through input/output gadgets. *CoRR*, abs/2005.03192, 2020. URL: https://arxiv.org/abs/2005.03192, arXiv:2005.03192.

[19] Gary Antonick. Roderick Kimball's path puzzles. *The New York Times: Numberplay*, 28 July 2014. https://wordplay.blogs.nytimes.com/2014/07/28/path-2/.

[20] Kenneth Appel and Wolfgang Haken. Every planar map is four colorable. Part I: discharging. *Illinois Journal of Mathematics*, 21(3):429–490, 1977.

[21] Kenneth Appel, Wolfgang Haken, and John Koch. Every planar map is four colorable. Part II: reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977.

[22] Jose Balanza-Martinez, Austin Luchsinger, David Caballero, Rene Reyes, Angel A. Cantu, Robert T. Schweller, Luis Angel Garcia, and Tim Wylie. Full tilt: Universal constructors for general shapes with uniform external forces. In Timothy M. Chan, editor, *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*, pages 2689–2708. SIAM, 2019. doi:10.1137/1.9781611975482.167.

[23] Kees Joost Batenburg and Walter A. Kosters. Solving nonograms by combining relaxations. *Pattern Recognit.*, 42(8):1672–1683, 2009. doi:10.1016/j.patcog.2008.12.003.

[24] BB+. The "grandfather" and "out-counts" methods of building tablebases. http://www.open-chess.org/viewtopic.php?f=5&t=779.

[25] Mihir Bellare, Oded Goldreich, and Madhu Sudan. Free bits, PCPs, and nonapproximability – towards tight results. *SIAM J. Comput.*, 27(3):804–915, 1998. doi:10.1137/S0097539796302531.

[26] Daniel Berend, Dolev Pomeranz, Ronen Rabani, and Ben Raziel. Nonograms: Combinatorial questions and algorithms. *Discret. Appl. Math.*, 169:30–42, 2014. doi:10.1016/j.dam.2014.01.004.

[27] Robert Berger. The undecidability of the domino problem. *Memoirs of the American Mathematical Society*, 66, 1966.

[28] Guillaume Bonfante and Joseph Le Roux. Intersection optimization is NP-complete. *Finite State Methods and Natural Language Processing*, page 74, 2007.

[29] Glencora Borradaile, Claire Kenyon-Mathieu, and Philip Klein. A polynomial-time approximation scheme for Steiner tree in planar graphs. In *Proceedings of*

the 18th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1285–1294, New Orleans, Louisiana, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283521.

[30] Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Adam Hesterberg, Roderick Kimball, and Justin Kopinsky. Path puzzles: Discrete tomography with a path constraint is hard. *CoRR*, abs/1803.01176, 2018. URL: http://arxiv.org/abs/1803.01176, arXiv:1803.01176.

[31] Jeffrey Bosboom, Erik D. Demaine, Martin L. Demaine, Adam Hesterberg, Roderick Kimball, and Justin Kopinsky. Path puzzles: Discrete tomography with a path constraint is hard. *Graphs Comb.*, 36(2):251–267, 2020. doi:10.1007/s00373-019-02092-5.

[32] Jeffrey Bosboom, Erik D. Demaine, and Mikhail Rudoy. Computational complexity of generalized Push Fight. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms, FUN 2018, June 13-15, 2018, La Maddalena, Italy*, volume 100 of *LIPIcs*, pages 11:1–11:21. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.FUN.2018.11.

[33] Jeffrey Bosboom, Erik D. Demaine, and Mikhail Rudoy. Computational complexity of generalized Push Fight. *CoRR*, abs/1803.03708, 2018. URL: http://arxiv.org/abs/1803.03708, arXiv:1803.03708.

[34] Cameron Browne. *Evolutionary Game Design*. Springer Briefs in Computer Science. Springer, 2011. doi:10.1007/978-1-4471-2179-4.

[35] Cameron Browne and Frédéric Maire. Evolutionary game design. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(1):1–16, 2010. doi:10.1109/TCIAIG.2010.2041928.

[36] David Caballero, Angel A. Cantu, Timothy Gomez, Austin Luchsinger, Robert Schweller, and Tim Wylie. Relocating units in robot swarms with uniform control signals is PSPACE-complete. In *Proceedings of the 32nd Canadian Conference on Computational Geometry (CCCG 2020)*, Saskatoon, Canada, August 2020.

[37] José M. Castaño and Rodrigo Castaño. A finite state intersection approach to propositional satisfiability. *Theor. Comput. Sci.*, 450:92–108, 2012. doi:10.1016/j.tcs.2012.04.030.

[38] Howard Chu. Life after BerkeleyDB: OpenLDAP's memory-mapped database. In *LinuxCon North America*, 2012. URL: http://www.lmdb.tech/media/20120829-LinuxCon-MDB-txt.pdf.

[39] Paolo Ciancarini and Gian Piero Favini. Playing the perfect Kriegspiel endgame. *Theor. Comput. Sci.*, 411(40-42):3563–3577, 2010. doi:10.1016/j.tcs.2010.05.019.

[40] Michael Cook and Simon Colton. Multi-faceted evolution of simple arcade games. In Sung-Bae Cho, Simon M. Lucas, and Philip Hingston, editors, *2011 IEEE Conference on Computational Intelligence and Games, CIG 2011, Seoul, South Korea, August 31 - September 3, 2011*, pages 289–296. IEEE, 2011. doi:10.1109/CIG.2011.6032019.

[41] Mark de Berg and Amirali Khosravi. Optimal binary space partitions in the plane. In My T. Thai and Sartaj Sahni, editors, *Proceedings of the 16th Annual International Conference on Computing and Combinatorics*, volume 6196 of *Lecture Notes in Computer Science*, pages 216–225, July 2010.

[42] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Budapest, Hungary, 2008. doi:10.1007/978-3-540-78800-3_24.

[43] Alberto Del Lungo and Maurice Navat. Reconstruction of connected sets from two projections. In Herman and Kuba [75], chapter 7.

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

[45] Erik D. Demaine and Martin L. Demaine. Fun with fonts: Algorithmic typography. *Theoretical Computer Science*, 586:111–119, June 2015.

[46] Erik D. Demaine, Martin L. Demaine, and David Eppstein. Phutball endgames are NP-hard. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 351–360. Cambridge University Press, 2002.

[47] Erik D. Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe, editors, *9th International Conference on Fun with Algorithms, FUN 2018, June 13-15, 2018, La Maddalena, Italy*, volume 100 of *LIPIcs*, pages 18:1–18:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.FUN.2018.18.

[48] Erik D. Demaine, Dylan H. Hendrickson, and Jayson Lynch. A general theory of motion planning complexity: Characterizing which gadgets make games hard. *CoRR*, abs/1812.03592, 2018. URL: http://arxiv.org/abs/1812.03592, arXiv:1812.03592.

[49] Erik D. Demaine, Dylan H. Hendrickson, and Jayson Lynch. Toward a general complexity theory of motion planning: Characterizing which gadgets make games hard. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*,

volume 151 of *LIPIcs*, pages 62:1–62:42. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ITCS.2020.62.

[50] Erik D. Demaine, Yoshio Okamoto, Ryuhei Uehara, and Yushi Uno. Computational complexity and an integer programming model of Shakashaka. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E97-A(6):1213–1219, 2014. URL: http://hdl.handle.net/10119/12147.

[51] Erik D. Demaine and Mikhail Rudoy. Tree-residue vertex-breaking: a new tool for proving hardness. In *Proceedings of the 16th Scandinavian Symposium and Workshops on Algorithm Theory*, pages 32:1–32:14, Malmö, Sweden, June 2018. arXiv:1706.07900.

[52] Bruno Durand and Anne-Cécile Fabret. On the complexity of deadlock detection in families of planar nets. *Theoretical Computer Science*, 215(1):225–237, 1999. URL: http://www.sciencedirect.com/science/article/pii/S0304397597001850, doi:https://doi.org/10.1016/S0304-3975(97)00185-0.

[53] M. E. Dyer and A. M. Frieze. Planar 3DM is NP-complete. *Journal of Algorithms*, 7(2):174–184, 1986. URL: http://www.sciencedirect.com/science/article/pii/0196677486900027, doi:https://doi.org/10.1016/0196-6774(86)90002-7.

[54] Martin Ebbesen, Paul Fischer, and Carsten Witt. Edge-matching problems with rotations. In Olaf Owe, Martin Steffen, and Jan Arne Telle, editors, *Proceedings of the 18th International Symposium on Fundamentals of Computation Theory*, volume 6914 of *Lecture Notes in Computer Science*, pages 114–125, Oslo, Norway, August 2011. doi:10.1007/978-3-642-22953-4_10.

[55] Marc Ebner, John Levine, Simon M. Lucas, Tom Schaul, Tommy Thompson, and Julian Togelius. Towards a video game description language. In Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius, editors, *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*, pages 85–100. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/DFU.Vol6.12191.85.

[56] Herbert Edelsbrunner, Joseph O'Rourke, and Raimund Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15(2):341–363, 1986. doi:10.1137/0215024.

[57] Ranel E. Erickson, Clyde L. Monma, and Arthur F. Veinott. Send-and-split method for minimum-concave-cost network flows. *Mathematics of Operations Research*, 12(4):634–664, 1987. URL: http://www.jstor.org/stable/3689922.

[58] Haw-ren Fang, Tsan-sheng Hsu, and Shun-chin Hsu. Construction of chinese chess endgame databases by retrograde analysis. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games, Second International Conference, CG 2000, Hamamatsu, Japan, October 26-28, 2000, Revised Papers*, volume

2063 of *Lecture Notes in Computer Science*, pages 96–114. Springer, 2000. doi:10.1007/3-540-45579-5_7.

[59] A. S. Fraenkel, M. R. Garey, D. S. Johnson, T. Schaefer, and Y. Yesha. The complexity of Checkers on an $N \times N$ board - preliminary report. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 55–64, Ann Arbor, Michigan, October 1978. URL: http://dx.doi.org/10.1109/SFCS.1978.36.

[60] Aviezri S Fraenkel and David Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n. *Journal of Combinatorial Theory, Series A*, 31(2):199–214, 1981. URL: http://www.sciencedirect.com/science/article/pii/0097316581900169, doi:http://dx.doi.org/10.1016/0097-3165(81)90016-9.

[61] Erich Friedman. Corral puzzles are NP-complete. http://www.stetson.edu/~efriedma/papers/corral/corral.html, August 2002.

[62] Erich Friedman. Pearl puzzles are NP-complete. http://www.stetson.edu/~efriedma/papers/pearl/pearl.html, August 2002.

[63] Erich Friedman. Spiral Galaxies puzzles are NP-complete. https://www2.stetson.edu/~efriedma/papers/spiral/spiral.html, March 2002.

[64] M. R. Garey and D. S. Johnson. The rectilinear Steiner tree problem is *NP*-complete. *SIAM Journal on Applied Mathematics*, 32(4):826–834, 1977. doi:DOI:10.1137/0132071.

[65] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[66] Ralph Gasser. Solving nine men's morris. In Richard Nowakowski, editor, *Games of No Chance*, pages 101–113. Cambridge University Press, Cambridge, 1996.

[67] Keith Golden and Wanlin Pang. Constraint reasoning over strings. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2003. doi:10.1007/978-3-540-45193-8_26.

[68] Leslie M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, July 1977. URL: http://doi.acm.org/10.1145/1008354.1008356, doi:10.1145/1008354.1008356.

[69] Linus Hamilton. Braid is undecidable. arXiv:1412.0784, 2014.

[70] Tracy Hammond and Randall Davis. LADDER: A language to describe drawing, display, and editing in sketch recognition. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 461–467. Morgan Kaufmann, 2003. URL: http://ijcai.org/Proceedings/03/Papers/069.pdf.

[71] M. Hanan. On Steiner's problem with rectilinear distance. *SIAM Journal on Applied Mathematics*, 14(2):255–265, 1966.

[72] Peter Hawkins, Alex Aiken, Kathleen Fisher, Martin C. Rinard, and Mooly Sagiv. Data representation synthesis. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 38–49. ACM, 2011. URL: http://doi.acm.org/10.1145/1993498.1993504, doi:10.1145/1993498.1993504.

[73] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A K Peters, July 2009.

[74] Gavin Henry. Howard Chu on Lightning Memory-Mapped Database. *IEEE Software*, 36(6):83–87, 2019. doi:10.1109/MS.2019.2936273.

[75] Gabor T. Herman and Attila Kuba, editors. *Discrete Tomography: Foundations, algorithms, and applications*. Birkhäuser, 2012.

[76] Michael Hoffmann. Motion planning amidst movable square blocks: Push-* is NP-hard. In *Proceedings of the 12th Canadian Conference on Computational Geometry*, pages 205–210, 2000.

[77] Jerry Holkins. Exposition. https://www.penny-arcade.com/news/post/2015/12/14/exposition, December 2015.

[78] Markus Holzer, Andreas Klein, and Martin Kutrib. On the NP-completeness of the NURIKABE pencil puzzle and variants thereof. In *Proceedings of the 3rd International Conference on FUN with Algorithms*, pages 77–89, Isola d'Elba, Italy, May 2004.

[79] Markus Holzer and Oliver Ruepp. The troubles of interior design—a complexity analysis of the game Heyawake. In *Proceedings of the 4th International Conference on FUN with Algorithms*, volume 4475 of *Lecture Notes in Computer Science*, pages 198–212, 2007.

[80] Harry B. Hunt, III, Madhav V. Marathe, Venkatesh Radhakrishnan, and Richard E. Stearns. The complexity of planar counting problems. *SIAM Journal on Computing*, 27(4):1142–1167, 1998. doi:10.1137/S0097539793304601.

[81] Geoffrey Irving. Pentago is a first player win: Strongly solving a game using parallel in-core retrograde analysis. *CoRR*, abs/1404.0743, 2014. URL: http://arxiv.org/abs/1404.0743, arXiv:1404.0743.

[82] Ayaka Ishibashi, Yuichi Sato, and Shigeki Iwata. NP-completeness of two pencil puzzles: Yajilin and Country Road. *Utilitas Mathematica*, 88:237–246, 2012.

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

[84] Chuzo Iwamoto. Yosenabe is NP-complete. *Journal of Information Processing*, 22(1):40–43, 2014. URL: http://dx.doi.org/10.2197/ipsjjip.22.40.

[85] Ahmed Khalifa and Magda Fayek. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*, 2015.

[86] Roderick Kimball. *Path Puzzles*. Roderick Kimball, 3rd edition, 2019.

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

[88] Timo Knuutila. Re-describing an algorithm by Hopcroft. *Theor. Comput. Sci.*, 250(1-2):333–363, 2001. doi:10.1016/S0304-3975(99)00150-4.

[89] Jonas Kölker. Kurodoko is NP-complete. *Journal of Information Processing*, 20(3):694–706, 2012. URL: http://dx.doi.org/10.2197/ipsjjip.20.694.

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

[91] Kouichi Kotsuma and Yasuhiko Takenaga. NP-completeness and enumeration of Number Link puzzle. *IEICE Technical Report*, 109(465):1–7, March 2010. URL: http://ci.nii.ac.jp/naid/110008000705/en/.

[92] Dexter Kozen. Lower bounds for natural proof systems. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 254–266. IEEE Computer Society, 1977. doi:10.1109/SFCS.1977.16.

[93] Ben Kuchera. Push Fight is the best board game you've never heard of. https://web.archive.org/web/20131211190946/http://penny-arcade.com/report/article/push-fight-is-the-best-board-game-youve-never-heard-of, May 2012.

[94] Mikael Z. Lagerkvist and Gilles Pesant. Modeling irregular shape placement problems with regular constraints. In *First Workshop on Bin Packing and Placement Constraints BPPC'08*, 2008. URL: http://www.gecode.org/paper.html?id=LagerkvistPesant:BPPC:2008.

[95] Calvin Loncaric, Emina Torlak, and Michael D. Ernst. Fast synthesis of fast collections. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 355–368. ACM, 2016. URL: http://doi.acm.org/10.1145/2908080.2908122, doi:10.1145/2908080.2908122.

[96] Giuseppe Maggiore, Alvise Spanò, Renzo Orsini, Michele Bugliesi, Mohamed Abbadi, and Enrico Steffinlongo. A formal specification for Casanova, a language for computer games. In Simone Diniz Junqueira Barbosa, José Creissac Campos, Rick Kazman, Philippe A. Palanque, Michael D. Harrison, and Steve Reeves, editors, *ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, Copenhagen, Denmark - June 25 - 28, 2012*, pages 287–292. ACM, 2012. URL: http://doi.acm.org/10.1145/2305484.2305533, doi:10.1145/2305484.2305533.

[97] Tobias Mahlmann, Julian Togelius, and Georgios N. Yannakakis. Towards procedural strategy game generation: Evolving complementary unit types. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcázar, Juan Julián Merelo Guervós, Ferrante Neri, Mike Preuss, Hendrik Richter, Julian Togelius, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation - EvoApplications 2011: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Torino, Italy, April 27-29, 2011, Proceedings, Part I*, volume 6624 of *Lecture Notes in Computer Science*, pages 93–102. Springer, 2011. doi:10.1007/978-3-642-20525-5_10.

[98] Kim Marriott, Bernd Meyer, and Kent B. Wittenburg. *A Survey of Visual Language Specification and Recognition*, pages 5–85. Springer New York, New York, NY, 1998. doi:10.1007/978-1-4612-1676-6_2.

[99] Andrew Martin, Andrew Lim, Simon Colton, and Cameron Browne. Evolving 3D buildings for the prototype video game Subversion. In Cecilia Di Chio, Stefano Cagnoni, Carlos Cotta, Marc Ebner, Anikó Ekárt, Anna Esparcia-Alcázar, Chi Keong Goh, Juan Julián Merelo Guervós, Ferrante Neri, Mike Preuss, Julian Togelius, and Georgios N. Yannakakis, editors, *Applications of Evolutionary Computation, EvoApplicatons 2010: EvoCOMPLEX, EvoGAMES, EvoIASP, EvoINTELLIGENCE, EvoNUM, and EvoSTOC, Istanbul, Turkey, April 7-9, 2010, Proceedings, Part I*, volume 6024 of *Lecture Notes in Computer Science*, pages 111–120. Springer, 2010. doi:10.1007/978-3-642-12239-2_12.

[100] Damiano Mazza. A functorial approach to reductions among decision problems. http://cl-informatik.uibk.ac.at/users/zini/events/dice18/abstracts/M.pdf.

[101] Brandon McPhail. The complexity of puzzles. Undergraduate thesis, Reed College, Portland, Oregon, 2003. URL: http://www.cs.umass.edu/~mcphailb/papers/2003thesis.pdf.

[102] Brandon McPhail. Light Up is NP-complete. http://www.mountainvistasoft.com/docs/lightup-is-np-complete.pdf, 2005.

[103] Brandon McPhail. Metapuzzles: Reducing SAT to your favorite puzzle. CS Theory talk, December 2007.

[104] Global constraints – the MiniZinc handbook. https://www.minizinc.org/doc-2.4.3/en/lib-globals.html#extensional-constraints-table-regular-etc.

[105] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[106] Eugene V. Nalimov, G. McC. Haworth, and Ernst A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000. doi:10.3233/ICG-2000-23304.

[107] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958. URL: http://www.jstor.org/stable/2033204.

[108] Nikoli Co., Ltd. タタミバリ（仮題） (Tatamibari (temporary title)). *Puzzle Communication Nikoli*, 107, 10 June 2004. See https://www.nikoli.co.jp/ja/publication/various/nikoli/back_number/nikoli107/ for a table of contents.

[109] Nikoli Co., Ltd. パズル通信ニコリ ＞ オモパリスト (Puzzle Communication Nikoli > Omopa List). https://www.nikoli.co.jp/ja/publication/various/nikoli/omopalist/, 2020.

[110] Nikoli Co., Ltd. Nikoli puzzles. http://www.nikoli.co.jp/en/puzzles/, 2020.

[111] Nicolas Oury, Michael Pedersen, and Rasmus L. Petersen. Canonical labelling of site graphs. In Ion Petre, editor, *Proceedings Fourth International Workshop on Computational Models for Cell Processes, CompMod 2013, Turku, Finland, 11th June 2013.*, volume 116 of *EPTCS*, pages 13–28, 2013. doi:10.4204/EPTCS.116.3.

[112] Tom Y. Ouyang and Randall Davis. ChemInk: a natural real-time recognition system for chemical drawings. In Pearl Pu, Michael J. Pazzani, Elisabeth André, and Doug Riecken, editors, *Proceedings of the 16th International Conference on Intelligent User Interfaces, IUI 2011, Palo Alto, CA, USA, February 13-16, 2011*,

pages 267–276. ACM, 2011. URL: http://doi.acm.org/10.1145/1943403.1943444, doi:10.1145/1943403.1943444.

[113] Nikolaos Papahristou and Ioannis Refanidis. Constructing pin endgame databases for the backgammon variant Plakoto. In Aske Plaat, H. Jaap van den Herik, and Walter A. Kosters, editors, *Advances in Computer Games - 14th International Conference, ACG 2015, Leiden, The Netherlands, July 1-3, 2015, Revised Selected Papers*, volume 9525 of *Lecture Notes in Computer Science*, pages 177–184. Springer, 2015. doi:10.1007/978-3-319-27992-3_16.

[114] Achilleas Papakostas and Ioannis G. Tollis. Improved algorithms and bounds for orthogonal drawings. In Roberto Tamassia and Ioannis G. Tollis, editors, *Proceedings of the DIMACS International Workshop on Graph Drawing*, pages 40–51, Princeton, New Jersey, October 1994. doi:10.1007/3-540-58950-3_355.

[115] Brett Picotte. Push Fight game. http://pushfightgame.com/, 2016. Accessed: 2017-06-22.

[116] Giovanni Pighizzini and Jeffrey O. Shallit. Unary language operations, state complexity and Jacobsthal's function. *Int. J. Found. Comput. Sci.*, 13(1):145–159, 2002. doi:10.1142/S012905410200100X.

[117] Claude-Guy Quimper and Toby Walsh. Decompositions of grammar constraints. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1567–1570. AAAI Press, 2008. URL: http://www.aaai.org/Library/AAAI/2008/aaai08-264.php.

[118] Neil Robertson, Daniel P. Sanders, Paul D. Seymour, and Robin Thomas. The four-colour theorem. *J. Comb. Theory, Ser. B*, 70(1):2–44, 1997. doi:10.1006/jctb.1997.1750.

[119] J. M. Robson. N by N Checkers is Exptime complete. *SIAM Journal on Computing*, 13(2):252–267, 1984. doi:10.1137/0213018.

[120] John W. Romein and Henri E. Bal. Solving Awari with parallel retrograde analysis. *IEEE Computer*, 36(10):26–33, 2003. doi:10.1109/MC.2003.1236468.

[121] Marcus Schaefer and Christopher Umans. Completeness in the polynomial-time hierarchy: A compendium. *ACM SIGACT News*, 33(3):32–49, 2002. Updated version available at http://ovid.cs.depaul.edu/documents/phcom.pdf.

[122] Jonathan Schaeffer, Yngvi Björnsson, Neil Burch, Robert Lake, Paul Lu, and Steve Sutphen. Building the checkers 10-piece endgame databases. In H. Jaap van den Herik, Hiroyuki Iida, and Ernst A. Heinz, editors, *Advances in Computer Games, Many Games, Many Challenges, 10th International Conference, ACG 2003, Graz, Austria, November 24-27, 2003, Revised Papers*, volume 263 of *IFIP*, pages 193–210. Kluwer, 2003.

[123] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *science*, 317(5844):1518–1522, 2007.

[124] Markus Schäffter. Drawing graphs on rectangular grids. *Discrete Applied Mathematics*, 63(1):75–89, 1995. URL: http://www.sciencedirect.com/science/article/pii/0166218X9400020E, doi:http://dx.doi.org/10.1016/0166-218X(94)00020-E.

[125] Tom Schaul. An extensible description language for video games. *IEEE Trans. Comput. Intellig. and AI in Games*, 6(4):325–331, 2014. doi:10.1109/TCIAIG.2014.2352795.

[126] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. Modeling and programming with Gecode. http://www.gecode.org/doc/6.0.1/MPG.pdf.

[127] Takahiro Seta. The complexities of puzzles, Cross Sum, and their Another Solution Problems (ASP). Senior thesis, University of Tokyo, 2002. URL: http://www-imai.is.s.u-tokyo.ac.jp/~seta/paper/senior_thesis/seniorthesis.pdf.

[128] Malte Skarupke. I wrote a faster sorting algorithm. https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/, 2016.

[129] Adam M. Smith and Michael Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In Georgios N. Yannakakis and Julian Togelius, editors, *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, Copenhagen, Denmark, 18-21 August, 2010*, pages 273–280. IEEE, 2010. doi:10.1109/ITW.2010.5593343.

[130] L. J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time (preliminary report). In *Proceedings of the 5th Annual ACM Symposium on Theory of Computing*, pages 1–9, 1973. URL: https://dl.acm.org/citation.cfm?id=804029.

[131] Thomas Ströhlein. *Untersuchungen über kombinatorische Spiele*. PhD thesis, Technische Hochschule München, 1970.

[132] Nathan R. Sturtevant. An argument for large-scale breadth-first search for game design and content generation via a case study of Fling! In *AI in the Game Design Process (AIIDE workshop)*, 2013.

[133] Ole Tange. *GNU Parallel 2018*. Ole Tange, March 2018. doi:10.5281/zenodo.1146014.

[134] Ken Thompson. Retrograde analysis of certain endgames. *ICGA Journal*, 9(3):131–139, 1986. doi:10.3233/ICG-1986-9302.

[135] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(3):172–186, 2011. doi:10.1109/TCIAIG.2011.2148116.

[136] Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discret. Appl. Math.*, 8(1):85–89, 1984. doi:10.1016/0166-218X(84)90081-7.

[137] Luca Trevisan, Gregory B. Sorkin, Madhu Sudan, and David P. Williamson. Gadgets, approximation, and linear programming. *SIAM J. Comput.*, 29(6):2074–2097, 2000. doi:10.1137/S0097539797328847.

[138] W. T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, s3-13(1):743–767, 1963. doi:10.1112/plms/s3-13.1.743.

[139] Nobuhisa Ueda and Tadaaki Nagao. NP-completeness results for NONOGRAM via parsimonious reductions. Technical Report TR96-0008, Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan, May 1996. URL: https://pdfs.semanticscholar.org/1bb2/3460c7f0462d95832bb876ec2ee0e5bc46cf.pdf.

[140] Jasper R. R. Uijlings, Koen E. A. van de Sande, Theo Gevers, and Arnold W. M. Smeulders. Selective search for object recognition. *International Journal of Computer Vision*, 104(2):154–171, 2013. doi:10.1007/s11263-013-0620-5.

[141] Leslie G Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.

[142] H. Jaap van den Herik and I. S. Herschberg. A data base on data bases. *ICGA Journal*, 9(1):29–34, 1986. doi:10.3233/ICG-1986-9104.

[143] Nageshwara Rao Vempaty. Solving constraint satisfaction problems using finite state automata. In William R. Swartout, editor, *Proceedings of the 10th National Conference on Artificial Intelligence, San Jose, CA, USA, July 12-16, 1992.*, pages 453–458. AAAI Press / The MIT Press, 1992. URL: http://www.aaai.org/Library/AAAI/1992/aaai92-070.php.

[144] Giovanni Viglietta. Lemmings is pspace-complete. In Alfredo Ferro, Fabrizio Luccio, and Peter Widmayer, editors, *Fun with Algorithms - 7th International Conference, FUN 2014, Lipari Island, Sicily, Italy, July 1-3, 2014. Proceedings*, volume 8496 of *Lecture Notes in Computer Science*, pages 340–351. Springer, 2014. doi:10.1007/978-3-319-07890-8_29.

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

[146] Ryan Williams. Applying practice to theory. *SIGACT News*, 39(4):37–52, 2008. URL: http://doi.acm.org/10.1145/1466390.1466401, doi:10.1145/1466390.1466401.

[147] Mårten Wiman. Improved inapproximability of max-cut through min-cut, 2018.

[148] Jan Wolter. Comparison of solvers on the n-Dom puzzles. https://www.webpbn.com/survey/dom.html.

[149] Jan Wolter. Survey of Paint-by-Number puzzle solvers. https://www.webpbn.com/survey/, 2011.

[150] I-Chen Wu, Der-Johng Sun, Lung-Ping Chen, Kan-Yueh Chen, Ching-Hua Kuo, Hao-Hua Kang, and Hung-Hsuan Lin. An efficient approach to solving nonograms. *IEEE Trans. Comput. Intell. AI Games*, 5(3):251–264, 2013. doi:10.1109/TCIAIG.2013.2251884.

[151] Ren Wu and Donald F Beal. A memory efficient retrograde algorithm and its application to chinese chess endgames. *ICCA Journal*, 42:213–227, 2002.

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

[153] Takayuki Yato. Complexity and completeness of finding another solution and its application to puzzles. Master's thesis, University of Tokyo, Tokyo, Japan, January 2003. URL: http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[154] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications, and Computer Sciences*, E86-A(5):1052–1060, 2003. Also IPSJ SIG Notes 2002-AL-87-2, 2002. URL: http://ci.nii.ac.jp/naid/110003221178/en/.

[155] Chiung-Hsueh Yu, Hui-Lung Lee, and Ling-Hwei Chen. An efficient algorithm for solving nonograms. *Appl. Intell.*, 35(1):18–31, 2011. doi:10.1007/s10489-009-0200-0.