Games, Puzzles, and Computation

by

Robert Aubrey Hearn

B.A., Rice University (1987)
S.M., Massachusetts Institute of Technology (2001)

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

June 2006

Author .....
Department of Electrical Engineering and Computer Science
May 23, 2006

Certified by .....
Erik D. Demaine
Esther and Harold E. Edgerton Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Gerald J. Sussman
Matsushita Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Games, Puzzles, and Computation

by

Robert Aubrey Hearn

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Abstract

There is a fundamental connection between the notions of game and of computation. At its most basic level, this is implied by any game complexity result, but the connection is deeper than this. One example is the concept of alternating nondeterminism, which is intimately connected with two-player games.

In the first half of this thesis, I develop the idea of game as computation to a greater degree than has been done previously. I present a general family of games, called *Constraint Logic*, which is both mathematically simple and ideally suited for reductions to many actual board games. A deterministic version of Constraint Logic corresponds to a novel kind of logic circuit which is monotone and reversible. At the other end of the spectrum, I show that a multiplayer version of Constraint Logic is undecidable. That there are undecidable games using finite physical resources is philosophically important, and raises issues related to the Church-Turing thesis.

In the second half of this thesis, I apply the Constraint Logic formalism to many actual games and puzzles, providing new hardness proofs. These applications include sliding-block puzzles, sliding-coin puzzles, plank puzzles, hinged polygon dissections, Amazons, Konane, Cross Purposes, TipOver, and others. Some of these have been well-known open problems for some time. For other games, including Minesweeper, the Warehouseman's Problem, Sokoban, and Rush Hour, I either strengthen existing results, or provide new, simpler hardness proofs than the original proofs.

Thesis Supervisor: Erik D. Demaine
Title: Esther and Harold E. Edgerton Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Gerald J. Sussman
Title: Matsushita Professor of Electrical Engineering

Acknowledgments

This work would not have been possible without the contributions of very many people. Most directly, I started on this line of research at the suggestion of Erik Demaine, who mentioned that the complexity of sliding-block puzzles was open, and that Gary Flake and Eric Baum's paper on the complexity of Rush Hour might be a source of good ideas for attacking the problem. This intuition proved to be spot-on, but neither Erik nor I had any idea how many more results would follow in natural progression.

But before this work began, I was already well-suited to head down this path. I acquired an early interest in games and puzzles, particularly with a mathematical flavor, primarily through Martin Gardner's "Mathematical Games" column in Scientific American, and his many books. My early interest in the theory of computation is harder for me to pin down, but it was certainly dependent on having access to computers, which most children of my generation did not. I have my parents, particularly my father, to thank for this. I also have my parents to thank for exposure to the Martin Gardner books. During high school my good friend Warren Wood was a fellow traveler; we invented many an interesting (and often silly) mathematical game together.

Later, I grew to love the beautiful mathematics of Combinatorial Game Theory, developed by Elwyn Berlekamp, John Conway, and Richard Guy. I am also grateful for many enjoyable and useful discussions with Elwyn, John, and Richard, which arose later during this work's development.

My interest in the theory of computation was rekindled when I returned to school after several years in the software industry, and took Michael Sipser's Theory of Computation course. Before this, I viewed NP-completeness as deep, mysterious math. I understood the concept, but the thought that *I* might someday show a real problem to be NP-complete—or even harder—was not one I had seriously entertained. In Michael's course I gained a clearer appreciation for how such reductions were done. It was in this context that I had the good fortune to TA MIT's Introduction to Algorithms course for Charles Leiserson and Erik Demaine. I was put in charge of the class programming contest, and I chose puzzle design as the domain. This led to discussions of game and puzzle complexity with Erik, and eventually to all of the present work. I also learned the great value of teaching from Charles and Erik. There is nothing like having to teach MIT students algorithms to keep you on your toes and make the material come alive for you.

Meanwhile, however, I was working on my "primary" research, implementing Marvin Minsky's "Society of Mind". I thank Marvin for his encouragement, and Gerry Sussman for his patience as I discovered as so many before me have that solving AI is not the task of a few years in grad school. I also have Gerry to thank for ultimately encouraging me to assemble my game and puzzle work into a Ph.D. thesis, and I also thank Erik for respectfully refraining from a similar suggestion until I raised the possibility with him, based on his understanding of my reason for being at MIT.

After my initial results on puzzles, Albert Meyer and Shafi Goldwasser served as my RQE committee, and offered many useful perspectives, as well providing me with

an extra sense of the legitimacy of my work and my ability to communicate it.

I am grateful to my coauthors on the work presented here: Erik Demaine, Michael Hoffman, Greg Frederickson, Martin Demaine, Rudolf Fleischer, and Timo von Oertzen. I learned a lot through collaboration that it would be virtually impossible to learn otherwise. Marty Demaine has also always had something interesting going on to discuss, and has offered excellent life advice on many occasions.

As I began to get results, I met many other interesting people in the mathematical games community. I am especially fortunate to have met John Tromp and Michael Albert, who have become good friends, in addition to providing many valuable insights. Other "games people" I have enjoyed many discussions with and learned from include Richard Nowakowsi, Aviezri Fraenkel, J. P. Grossman, Aaron Seigel, Cyril Banderier, and Ed Pegg.

I want to especially thank Ivars Peterson for helping to popularize some of my work in Science News and Math Trek, and making me aware of the wider interest in this kind of work. Similarly, thanks are due to Michael Kleber for inviting me to write an article for Mathematical Intelligencer.

Of my many fellow grad students at MIT, I want to thank Jake Beal, Justin Werfel, Attila Kondacs, Radhika Nagpal, Paulina Varshavskaia, Rebecca Frankel, and Keith Winstein for helping to make the journey more pleasant. Two former students, Erik Rauch, who was my officemate, and Push Singh, who was a good friend and sometimes mentor in my Society of Mind work, both left this world before their time. Both, however, had a big positive effect on me, and many others.

My wife Liz deserves special thanks for being patient as I kept trying to solve impossible problems on the one hand, and screwing around with silly games on the other. Finally, Liz, I've made something of all that playing around with games!

Last but not least, I am very appreciative of my committee. Gerry Sussman was my advisor from the moment I arrived at MIT, and it has been my great privilege to share many hours of discussion with Gerry on virtually every subject that is interesting, and many evenings of fine astronomy as well. Erik Demaine has been incredible to know and to work with. He has an amazing ability to point people in just the right direction, so that they will find interesting things. Every time I had a new result, I would take it to Erik, and he would say,"yes, cool! Now, did you think about this?", and I would be off on another hunt. Marvin Minsky has been a major source of inspiration for me for more than 20 years. I hope to revive my Society of Mind work; I still feel that this book is the single best source of insights on how to think about intelligence. I am still somewhat astounded that I am able to just talk to Marvin like an ordinary person; he is an architect of a huge part of modern computer science and AI. Similarly, it has been an honor to have Patrick Winston on my committee. I have learned a lot about effective writing and communication, as well as AI, from Patrick. Marvin and Patrick deserve extra thanks for remaining on my committee when I switched topics from AI to game complexity. Finally, Michael Sipser, though a late addition to the committee, has been a valuable presence. I learned a great deal in his course—including a lot that I thought I already knew! And Michael also had valuable comments for me when he graciously agreed to join my committee, in spite of his load as head of the Mathematics department.

Contents

List of Figures

Chapter 1

Introduction

In this thesis I argue that there are deep connections between the idea of game and the idea of computation, and indeed that games should be thought of as a valid and important model of computation, just as Turing machines are.

It is trivially true that any problem shown to be NP-hard, PSPACE-hard, etc. has an inherent computational flavor, merely by virtue of the (often indirect) reduction from some kind of resource-bounded Turing machine. Given this, the fact that many games (such as Chess, Checkers, and Go) have been shown to be hard would not seem to argue for a special connection between games as such and computation. One might as well, it seems, argue that graphs are inherently computational, because there are many hard graph problems.

However, it is a curious fact that various kinds of games seem to be in especially direct correspondence with particular models of computation. For example, the notion of nondeterminism inherent in NP-completeness nicely matches the feature of puzzles that a player must choose a sequence of moves or piece placements to satisfy some global property. Even more striking is the correspondence between alternation, the natural extension to nondeterminism, and two-player games [8].

These connections have been pointed out before. Reif [64], and Peterson, Reif, and Azhar [60], suggest that games should be considered as a model of computation, and they explicitly list a taxonomy of games types and corresponding models of computation, culminating in team games of private information, which correspond to unbounded deterministic Turing machines.

My primary contribution, developed in Part I, is to present an explicit, uniform model of abstract game, called Constraint Logic, which has as natural specializations zero-player games (deterministic simulations), one-player games (puzzles), two-player games, et cetera. In each case I show that my game is complete for the appropriate complexity class, or, in the case of team games of private information, undecidable. In fact, this last game is arguably the first game, in a particular, reasonable sense,[1]

[1]Specifically, the game must have a finite number of positions, and players must alternate turns. No previously known undecidable problem, to my knowledge, has both these properties. Peterson and Reif's game *Team-Peek* [60] is described to have those properties, but there is an implicit assumption in the construction that players do not take turns in sequence. (Also there is a mistake in the definition; as defined, the game is trivially decidable.) See Chapter 7.

to be shown undecidable. The fact that there are such undecidable games using finite resources highlights the difference between games and Turing machines, and is the linchpin in the argument for viewing games as a model of computation.

In addition to filling a natural slot in the game complexity hierarchy, the deterministic version of Constraint Logic also turns out to be a new, interesting model of computation in its own right. It is reversible and monotone, and may potentially have practical application for building real computers with very low power consumption.

Constraint Logic is also naturally suited for reductions to actual board games, and an additional contribution of this thesis, in Part II, is a large collection of new game hardness proofs based on the various flavors of Constraint Logic. Constraint Logic is a game played on a directed graph; a move is to reverse the direction of an edge. One especially useful property of these graphs is that in all cases but one, the completeness results hold even for planar graphs. This greatly simplifies many game reductions, in some cases making trivial what was previously intricately complex.

Another curious property of games (or perhaps of their players), which becomes evident in Part II, is that a given game will tend to be as hard as possible given its general characteristics. For example, if a game is a one-player puzzle with a bounded length, odds are it is NP-complete. If it is a two-player game with an unbounded length, it will generally be EXPTIME-complete. And so on. This is perhaps a reflection on human psychology as much as anything; a game would not be interesting if it were not hard.

Hardness results are called "negative" results, as opposed to "positive" results showing how to efficiently solve problems; one feeling in the mathematical games community is that once a game has been shown hard, there is no point studying it further. But from the computational perspective of this thesis, and taking the preceding paragraph into consideration, showing a game to be hard *validates* it as an object worthy of interest and further study.

Part I

Games in General

Part I of this thesis develops the Constraint Logic model of computation in its various flavors. These comprise instances in a two-dimensional space of game categories, shown in Figure 1-1. The first dimension ranges across zero-player games (deterministic simulations), one-player games (puzzles), two-player games, and team games with private information. The second dimension is whether the game has (polynomially) bounded length. In all cases, the games use bounded space; the basic idea is that a game involves pieces moving around, being placed, or being captured, on a board or other space of fixed size.

	Zero player (simulation)	One player (puzzle)	Two player	Team, imperfect information
Unbounded	PSPACE	PSPACE	EXPTIME	Undecidable
Bounded	P	NP	PSPACE	NEXPTIME

Figure 1-1: Table of Constraint Logic game categories and complexities. Each game type is complete for the indicated class. (After [61].)

Chapter 2 considers various notions of games, and describes the kinds of games, *generalized combinatorial games*, that will be studied here. Chapter 3 defines the general Constraint Logic model of computation. Chapters 4–7 develop notions of game ranging from deterministic simulations to team games of private information, and provide corresponding complexity results for appropriate versions of Constraint Logic.[2]

Chapter 8 summarizes the results in Part I, and explores some of their implications.

[2]One result, NEXPTIME-completeness for Bounded Team Private Constraint Logic, is merely conjectured.

Chapter 2

What is a Game?

The goal of this thesis is to investigate the computational characteristics of games and puzzles such as Chess, Checkers, Go, Hex, Bridge, sliding-block puzzles, Conway's Game of Life, ... but what exactly is a game?

Game Theory. There are innumerable kinds of activities and situations that might be described as games. A large portion of these may be studied within the context of classical game theory. However, the games I wish to consider are both more specialized and more general than what is traditionally addressed by game theory. More specialized, because I shall only be concerned with determining the winner of a game, and not with other issues of interest in game theory such as maximizing payoff, studying cooperative strategies, etc. More general, because game theory is concerned only with the interactions of two or more players, whereas I will consider games with only one player (puzzles) and even with no players at all (simulations). Other differences are that game theory generally formulates games in either "strategic" form (by exhaustively listing the strategies for each player) or "extensive" form (by analyzing the explicit game tree). But for the games I consider both forms would be exponentially large, or even infinite.

Combinatorial Games. Instead, in order to explore the connection between games and computation, I will study games from a computational complexity standpoint. Naturally, then, these games will have a combinatorial aspect. Indeed, we might simply call them "combinatorial games", except for the fact that there is already an established field called Combinatorial Game Theory [4, 9], and many of the kinds of games I will consider fall outside the domain of this field. A combinatorial game is defined to be a two-player, perfect-information game with no chance elements [25]. By so restricting the notion of game, Combinatorial Game Theory is able to draw out beautiful and unexpected relationships between games and numbers—indeed, in that framework a number is simply a special kind of game. But other "games" with a combinatorial flavor, such as sliding-block puzzles, or Conway's Game of Life, or Bridge, violate these restrictions, yet are interesting to study from a computational perspective.

Notion of a Game. At the risk of being insufficiently specific, I will avoid formally defining a notion of game. The reason is that it is difficult, if not impossible, to be sure there is not some further reasonable generalization of the concept of game that could not be studied by further generalization of the techniques presented in this thesis.

For example, in a seminal paper on game complexity [81], Stockmeyer and Chandra define a notion of "reasonable game", and demonstrate formula games that are "universal" for such games. These formula games provably require exponential time to determine the winner.

However, shortly afterward Reif [64] generalized the notion of game to include imperfect information,[1] relabeled Stockmeyer and Chandra's "reasonable games" as "reasonable games of perfect information", and demonstrated formula games that are "universal for all reasonable games", where the notion of imperfect information has been added. Reif's formula games are complete in doubly-exponential time—exponentially harder to decide than even the provably intractable games that previously were all that was "reasonable".

And then again, shortly afterwards, Peterson and Reif [61] noted that "the generalization to more than two players is to have two teams, A and B", and showed that such games can in general be undecidable.[2] This is, in my view, a very deep result, and one I will explore in detail in Chapters 7 and 8.

But the point is that there always seems to be another generalization of what constitutes a "reasonable" game. The limit may seem to have been reached with Peterson and Reif's undecidable games, but perhaps there are other ways to generalize games to get undecidability, or perhaps there is a kind of game with a higher degree of undecidability.

Requirements for Games. Notwithstanding the above, I must give some general criteria for what constitutes a game for this thesis to have any meaning. Otherwise, one could potentially argue that anything could be viewed as a game. Informally, the characteristics listed below correspond to what I will call a *generalized combinatorial game*.

The most important criterion is that there be a finite number of *positions*. This is what fundamentally distinguishes games from conventional models of computation, such as Turing machines. A Turing machine has an infinite tape; in principle, such a machine cannot exist in any finite physical space. The computers that sit on our desktops correspond more closely to space-bounded Turing machines. However, games with finitely many positions *can* actually exist in our world.

This requirement already rules out one undecidable problem that has a puzzle-like flavor: the Post Correspondence Problem (PCP) [62]. An instance of PCP consists

[1] Reif's paper is called "Universal Games of Incomplete Information", but technically in game theory *incomplete information* refers to games where the payoffs or possible strategies are not known to all players. In contrast, *imperfect information* refers to games where players do not know all actions taken by the other players. Reif's games are actually games of imperfect information.

[2] However, there appear to be several technical problems with this result, which I will explain and correct in Chapter 7.

of a set of dominos, each with two character strings, one on each side. The question is, is there a sequence of dominos from that set, allowing repetitions, such that the concatenated sequence of strings on each side is the same? This is a well-known undecidable problem, which is indeed framed as a kind of puzzle. But allowing a given domino to be reused any number of times in a sequence makes it unphysical, and violates the spirit of what I will consider to be a game. There is no finite bound on the number of possible sequences of dominos.[3]

Next, clearly there should be some number of *players*. The players take *turns* in some fashion, mapping one position into another (by making *moves*. In the interest of including simulations such as the game of Life, and because such games fit naturally into the framework I will develop, zero is an acceptable number of players. (A zero-player game can be thought of as a one-player game where all the moves are forced.) Generally, adding players adds more nondeterminism to the corresponding model of computation.

For each game, the standard decision question will be along the lines of, "from this position, does player X have a forced win?". The meaning of "forced win" is often not stated explicitly in game problem statements (see e.g. [34]). However, the meaning may be taken to be, does player X have a *strategy*—a rule dictating play in all circumstances—such that for all possible ways the other players play, player X wins? What specifically constitutes a strategy will depend on the kind of game. (Also, we assume that when one player wins, the game ends, so no other player may then win as well.)

Finally, we would like it to be possible, and easy, to determine the legal moves from a given position.

Some actual games skirt the boundaries informally laid out above. For example, Go as played in China and in the USA has a *superko* rule, which stipulates that no former board position may be recreated. This rule ensures that games will not loop, and will eventually finish. Thus, it has practical utility. But it does mean that it is not possible to determine the legal moves from the current position. Of course, the entire history of the game could be taken to be the current position, but this violates our intuition of what normally constitutes a position. Also, complexity results would then have to be taken relative to an exponentially larger space of possible histories, versus possible board configurations, and thus would not be particularly interesting. Or, perhaps, we could consider an equivalent game, where moves that recreate former positions are legal, but losing. Does this solve the problem, and is Go with superko thus a valid game in the above sense? It is not so clear. It is worth noting that the complexity of Go with superko is an interesting open problem [69]; there is reason to believe it could be harder than Go with the traditional Japanese ko rule.

Formal Problem Statements. Each category of game considered *will* be formally defined. But the notion of generalized combinatorial game sketched here is intended to be more heuristic, and potentially suggestive of kinds of games not treated explicitly.

[3]Of course, a bounded version of PCP, for example with some given bound on the sequence length, is a perfectly good puzzle.

Other Kinds of Games. There are other kinds of games one could consider than the generalized combinatorial games addressed in this thesis. In addition to unbounded games, such as PCP, or Life on an infinite grid, there are games with random elements. Papadimitriou [58] has shown that in some cases a source of random moves is equivalent to another player. There are also continuous games. The techniques presented in this thesis have been used to show that many questions about the manipulations of hinged dissections of polygons are PSPACE-complete [42], regardless of the fact that those are continuous problems. In the domain of classical game theory, there are also complexity results: it was recently shown [14] that computing Nash equilibria is PPAD-complete. (All finite games have Nash equilibria for mixed strategies [57]; PPAD-completeness refers not to a decision problem, but to computing a function efficiently.)

I choose to focus on generalized combinatorial games, because it is there that the computational character of games is expressed most plainly, and that the key differences from traditional models of computation are most apparent.

Chapter 3

The Constraint Logic Formalism

The general model of games I will develop is based on the idea of a *constraint graph*; the rules defining legal moves on such graphs are called *Constraint Logic*. In later chapters the graphs and the rules will be specialized to produce zero-player, one-player, two-player, etc. games. A game played on a constraint graph is a computation of a sort, and simultaneously serves as a useful problem to reduce to other games to show their hardness.

In the game complexity literature, the standard problem used to show games hard is some kind of game played with a Boolean formula. The Satisfiability problem (SAT), for example, can be interpreted as a puzzle: the player must existentially make a series of variable selections, so that the formula is true. The corresponding model of computation is determinism, and the natural complexity class is NP. Adding alternating existential and universal quantifiers creates the Quantified Boolean Formula problem (QBF), which has a natural interpretation as a two-player game [80, 79]. The corresponding model of computation is alternation, and the natural complexity class is PSPACE. Allowing the players to continue to switch the variable states indefinitely creates a formula game of unbounded length, raising the complexity to EXPTIME. And so on. Most game hardness results (e.g., Instant Insanity [67], Hex [65, 22], Generalized Geography [72], Chess [26], Checkers [70], Go [53, 68]) are direct reductions from such formula games or their simple variants, or else even more explicit reductions directly from the appropriate type of Turing machine (e.g., Sokoban [13]).

One problem with such reductions is that the geometric constraints typically found in board games do not naturally correspond to any properties of the formula games. By contrast, the constraint graph games I will present are all, with one exception, games played on planar graphs, so that there is a natural correspondence with typical board game topology. Furthermore, the required constraints often correspond very directly to existing constraints in many actual games. As a result, the various flavors of Constraint Logic are often much more amenable to reductions to actual games than are the underlying formula games. As evidence of this, I present a large number of new games reductions in Part II. The prototypical example is sliding-block puzzles, where the physical constraints of the blocks—that two blocks cannot occupy the same space at the same time—are used to implement the appropriate kind of Constraint

Logic.

Constraint Logic also seems to have an advantage in conceptual economy over formula games. Formula games require the concepts of variables and formulas, but Constraint Logic games require the single concept of a constraint graph. In the various reductions I will give from formula games of various types to equivalent constraint graph games, the variables and the formulas are represented uniformly as graph elements. This conceptual economy translates to simpler game reductions; often fewer gadgets must be built to show a given game hard using Constraint Logic.

Appendix A reviews Boolean formulas and the Satisfiability and Quantified Boolean Formulas problems; other formula games are defined in the text as they are needed.

3.1 Constraint Graphs

The material in this section is joint work with Erik Demaine [46, 47].

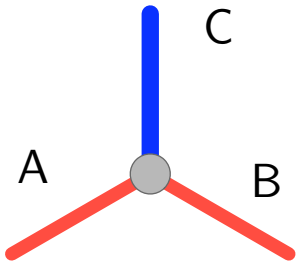
A *constraint graph* is a directed graph, with edge weights $\in \{1, 2\}$. An edge is then called *red* or *blue*, respectively. The *inflow* at each vertex is the sum of the weights on inward-directed edges. Each vertex has a nonnegative *minimum inflow*. A legal configuration of a constraint graph has an inflow of at least the minimum inflow at each vertex; these are the *constraints*. A legal move on a constraint graph is the reversal of a single edge, resulting in a legal configuration. Generally, in any game, the goal will be to reverse a given edge, by executing a sequence of moves. In multiplayer games, each edge is controlled by an individual player, and each player has his own goal edge. In deterministic games, a unique sequence of reversals is forced. For the bounded games, each edge may only reverse once.

It is natural to view a game played on a constraint graph as a computation. Depending on the nature of the game, it can be a deterministic computation, or a nondeterministic computation, or an alternating computation, etc. The constraint graph then *accepts* the computation just when the game can be won.

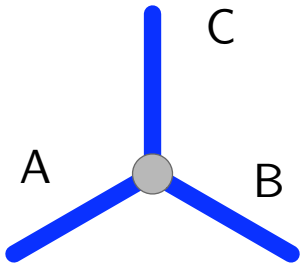
AND/OR Constraint Graphs. Certain vertex configurations in constraint graphs are of particular interest. A vertex with minimum inflow constraint 2 and incident edge weights of 1, 1, and 2 behaves as a logical AND, in the following sense: the weight-2 (blue) edge may be directed outward if and only if both weight-1 (red) edges are directed inward. Otherwise, the minimum inflow constraint of 2 would not be met. I will call such a vertex an *AND vertex*.

Similarly, a vertex with incident edge weights of 2, 2, and 2 behaves as a logical OR: a given edge may be directed outward if and only if at least one of the other two edges is directed inward. I will call such a vertex an *OR vertex*. AND and OR vertices are shown in Figure 3-1. Blue edges are drawn thicker than red ones as a mnemonic for their increased weight.

It turns out that for all the game categories, it will suffice to consider constraint graphs containing only AND and OR vertices. Such graphs are called *AND/OR* constraint graphs.



(a) AND vertex. Edge C may be directed outward if and only if edges A and B are both directed inward.



(b) OR vertex. Edge C may be directed outward if and only if either edge A or edge B is directed inward.

Figure 3-1: AND and OR vertices. Red (light gray) edges have weight 1, blue (dark gray) edges have weight 2, and vertices have a minimum in-flow constraint of 2.

For some of the game categories, there can be many sub-types of AND and OR vertex, because each edge may have a distinguishing initial orientation (in the case of bounded games), and a distinct controlling player (when there is more than one player). In some of these cases I will present alternate sets of basis vertices, which are not strictly AND and OR vertices, but which can lead to simpler game reductions.

Directionality; Fanout. As implied above, although it is natural to think of AND and OR vertices as having inputs and outputs, there is nothing enforcing this interpretation. A sequence of edge reversals could first direct both red edges into an AND vertex, and then direct its blue edge outward; in this case, I will sometimes say that its *inputs* have *activated*, enabling its *output* to activate. But the reverse sequence could equally well occur. In this case we could view the AND vertex as a splitter, or *FANOUT* gate: directing the blue edge inward allows both red edges to be directed outward, effectively splitting a signal.

In the case of OR vertices, again, we can speak of an active input enabling an output to activate. However, here the choice of input and output is entirely arbitrary, because OR vertices are symmetric.

Circuit Interpretation. With these AND, OR, and FANOUT vertex interpretations, it is natural to view an AND/OR graph as a kind of digital logic network, or circuit. (See Figure 4-6 for examples of such graphs.) One can imagine signals flowing through the graph, as outputs activate when their input conditions are satisfied. This is the picture that motivates my description of Constraint Logic as a model of computation, rather than simply as a set of decision problems. Indeed, it is natural to expect that a finite assemblage of such logic gadgets could be used to build a sort of computer.

However, several differences between AND/OR constraint graphs and ordinary digital logic circuits are noteworthy. First, digital logic circuits are deterministic. With the exception of zero-player Constraint Logic, a Constraint Logic computation exhibits some degree of nondeterminism. Second, with the above AND and OR vertex

interpretations, there is nothing to prohibit "wiring" a vertex's "output" (e.g. the blue edge of an AND vertex) to another "output", or an "input" to an "input". In digital logic circuitry, such connections would be illegal, and meaningless, whereas they are essential in Constraint Logic. Finally, although we have AND- and OR-like devices, there is nothing like an inverter (or NOT gate) in Constraint Logic; inverters are essential in ordinary digital logic.

This last point deserves some elaboration. The logic that is manifested in constraint graphs is a *monotone* logic. By analogy with ordinary numerical functions, a Boolean formula is called monotone if it contains only literals, ANDs, and ORs, with no negations. The reason is that when a variable changes from false to true, the value of the formula can never change from true to false. Likewise, Constraint Logic is monotone, because inflow is a monotone function of incident edge orientations. Reversing an edge incident at a given vertex from in to out can never enable reversal of another edge at that vertex from in to out. That is what would be required by a NOT vertex. One of the more surprising results about Constraint Logic is that monotone logic is sufficient to produce computation, even in the deterministic case.

Flake and Baum [24] require the use of inverters in a similar computational context. They define gadgets ("both" and "either") that are essentially the same as our AND and OR vertices, but rather than use them as primitive logical elements, they use their gadgets to construct a kind of dual-rail logic. With this dual-rail logic, they can represent inverters, at a higher level of abstraction. We do not need inverters for our reductions, so we may omit this step.

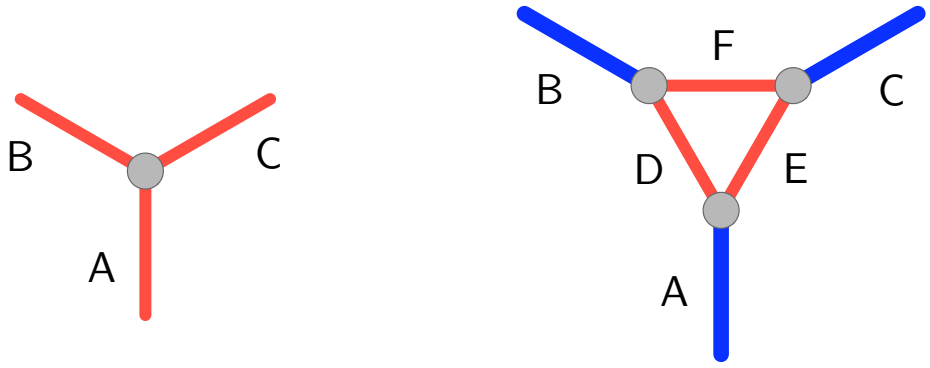
3.2 Constraint Graph Conversion Techniques

Often it will be convenient to work with constraint graphs that are not strictly AND/OR graphs, but that can be easily converted to equivalent AND/OR graphs. The three such "shorthands" that will occur most frequently are the use of *CHOICE* (red-red-red) vertices, degree-2 vertices, and loose edges.

CHOICE Vertices. A *CHOICE* vertex, shown in Figure 3-2(a), is a vertex with three incident red edges and an inflow constraint of 2. The constraint is thus that at least two edges must be directed inward. If we view A as in input edge, then when the input is inactivated, i.e., A points down, then the outputs B and C are also inactivated, and must also point down. If A is then directed up, either B or C, but not both, may also be directed up. In the context of a game, a player would have a choice of which path to activate.

The AND/OR subgraph shown in Figure 3-3(b) has the same constraints on its A, B, and C edges as the CHOICE vertex does. Suppose A points down. Then D and E must also point down, which forces B and C to point down. If A points up, D and E may as well (using vertex A-D-E as a FANOUT). F may then be directed either left or right, to enable either B or C, but not both, to point up.

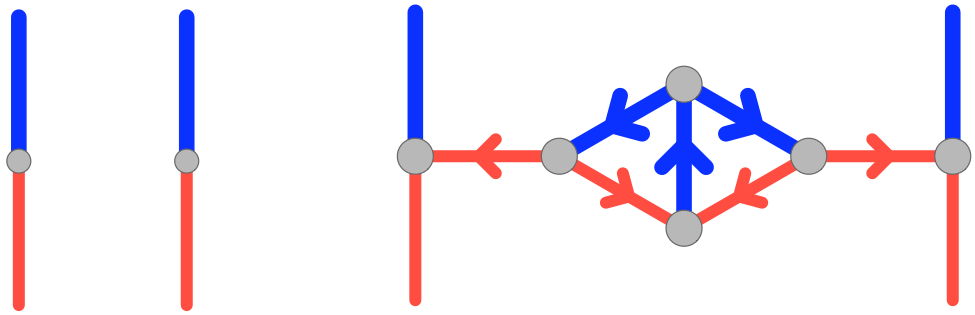
The replacement subgraph still may not be substituted directly for a CHOICE vertex, however, because its terminal edges are blue, instead of red. This brings us



(a) CHOICE vertex

(b) Equivalent AND/OR subgraph

Figure 3-2: CHOICE vertex conversion.



(a) Pair of red-blue vertices

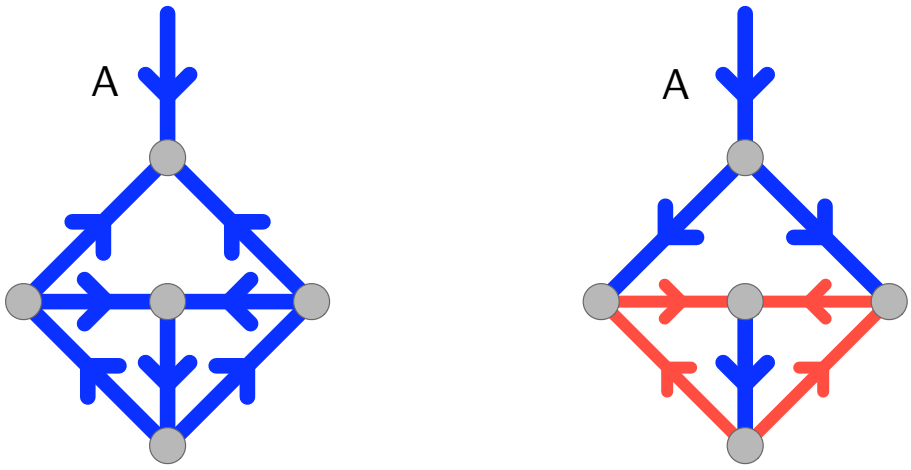
(b) Equivalent AND/OR subgraph

Figure 3-3: Red-blue vertex conversion. Red-blue vertices, which have an inflow constraint of 1 instead of 2, are drawn smaller than other vertices.

to the next conversion technique.

Degree-2 Vertices. Viewing AND/OR graphs as circuits, we might want to connect the output of an OR, say, to an input of an AND. We can't do this directly by joining the loose ends of the two edges, because one edge is blue and the other is red. However, we can get the desired effect by joining the edges at a red-blue vertex with an inflow constraint of 1. This allows each incident edge to point outward just when the other points inward—either edge is sufficient to satisfy the inflow constraint.

We would like to find a translation from such red-blue vertices to AND/OR subgraphs. However, there is a problem: in AND/OR graphs, red edges always come in pairs. The solution is to provide a conversion from *two* red-blue vertices to an equivalent AND/OR subgraph. This will always suffice, because a red edge incident at a red-blue vertex must be one end of a chain of red edges ending at another red-blue



(a) Free edge terminator (b) Constrained edge terminator

Figure 3-4: How to terminate loose edges.

vertex. The conversion is shown in Figure 3-3. Clearly, the orientations shown for the edges in the middle satisfy all the constraints except for the left and right vertices; for these, an inflow of 1 is supplied, and either the red or the blue edge is necessary and sufficient to satisfy the constraints.

Note that the degree-2 vertices are drawn smaller than the AND/OR vertices, as an aid to remembering that their inflow constraint is 1 instead of 2.

It will occasionally be useful to use blue-blue and red-red vertices, as well as red-blue. Again, these vertices have an inflow constraint of 1, which forces one edge to be directed in. A blue-blue vertex is easily implemented as an OR vertex with one loose edge which is constrained to always point away from the vertex (see below). Red-red edges will only occur in zero-player games. However, in that case special timing considerations also arise, so I will defer discussion of red-red vertices for now.

Loose Edges. Often only one end of an edge matters; the other need not be constrained. To embed such an edge in an AND/OR graph, the subgraph shown in Figure 3-4(a) suffices. If we assume that edge A is connected to some other vertex at the top, then the remainder of the figure serves to embed A in an AND/OR graph while not constraining it.

Similarly, sometimes an edge needs to have a permanently constrained orientation. The subgraph shown in Figure 3-4(b) forces A to point down; there is no legal orientation of the other edges that would allow it to point up.

Chapter 4

Zero-Player Games (Simulations)

We begin our study of specific abstract games with deterministic, or zero-player, games. We may think of such games as simulations: each move is determined from the preceding configuration. Examples that are often thought of as games include cellular automata, such as Conway's Game of Life [33].

More generally, the class of zero-player games corresponds naturally to ordinary computers, or deterministic space-bounded Turing machines—the kinds of computation tools we have available in the real world, at least until quantum computers develop further.

Constraint Logic. The Constraint Logic formalism does not restrict the set of moves available on a constraint graph to a unique next move from any given position. To consider a deterministic version, we must further constrain the legal moves. Rather than propose a rule which selects a unique next edge to reverse from each position, we apply determinism independently at each vertex, so that multiple edge reversals may occur on each deterministic "turn".

The basic idea is that each vertex should allow "signals" to "flow" through it if possible. So if both red edges reverse inward at an AND vertex, then on the next move the blue edge will reverse outward. For the bounded version, this idea is all we need. For the unbounded version, the rule is modified to allow inputs that can't flow through to "bounce" back. (They cannot do so in the bounded version, because each edge can only reverse once.) This allows the construction of arbitrary space-bounded computers—unbounded Deterministic Constraint Logic is PSPACE-complete. Furthermore, it turns out to represent a new style of reversible, monotone computation that could potentially be physically built, and could have significant advantages over conventional digital logic.

4.1 Bounded Games

A bounded zero-player game is essentially a simulation that can only run for a linear time. Admittedly it seems a stretch to call such simulations "games", but they do fit naturally into the overall framework sketched in Figure 1-1, and all the other slots

in that table are more arguably game-like. Bounded Deterministic Constraint Logic is included in this thesis merely for completeness. Conceivably there could be some solitaire games where the player has no actual choice, and the game is of bounded length, but such games would not seem to be very interesting.

This style of computation is captured by the notion of *Boolean circuits*, and more specifically, *monotone Boolean circuits*. A monotone Boolean circuit is a directed acyclic graph where the nodes are *gates* (*AND* or *OR*) or *inputs*, and the connections and edge orientations are as expected for the gate types. The gates are allowed to have multiple outputs (= outward directed edges); that is, there is fanout. One gate is the *output gate*. Each gate computes the appropriate Boolean function of its input. The *value* of the circuit, for a given assignment of Boolean input values, is the value computed by the output gate. An ordinary (non-monotone) Boolean circuit is also allowed NOT gates; these turn out not to add any computational power.

Essentially, then, a monotone Boolean circuit is just a representation of a monotone Boolean formula, that potentially allows some space savings by reusing subexpressions via fanout. The problem of determining the value of a monotone Boolean circuit, called *MONOTONE-CIRCUIT-VALUE*, is P-complete [36].

Bounded Deterministic Constraint Logic corresponds directly to *MONOTONE-CIRCUIT-VALUE*. To formally define the problem, first I define a constraint graph successor operation:

$$\begin{aligned} R_{i+1} &= \{e \mid e \text{ is a legal move in } G_i \text{ and } e \notin R_0 \cup \ldots \cup R_i\}, \\ G_{i+1} &= G_i \text{ with edges in } R_{i+1} \text{ reversed.} \end{aligned}$$

Recall that a legal move is the reversal of a single edge such that all constraints remain satisfied. Thus, this process effectively propagates signals through a graph until they can no longer propagate. It might appear that this definition could cause a legal constraint graph to have an illegal successor, since moves that are individually legal might not be simultaneously legal, but this will turn out not to be a problem.

BOUNDED DETERMINISTIC CONSTRAINT LOGIC (BOUNDED DCL)
INSTANCE: AND/OR constraint graph G_0; edge set R_0; edge e in G_0.
QUESTION: Is there an i such that e is reversed in G_i?

4.1.1 P-completeness

Theorem 1 *Bounded DCL is P-complete.*

Proof: Given a Boolean Circuit C, we construct a corresponding Bounded DCL problem, such that the edge in the DCL problem is reversed just when the circuit value is true. This process is straightforward: for every gate in C we create a corresponding vertex, either an AND or an OR. When a gate has more than one output, we use AND vertices in the FANOUT configuration. The difference here between AND and FANOUT is merely in the initial edge orientation. Where necessary, we use the red-blue conversion technique shown in Section 3.2. For the input nodes, we use terminators

as in Figures 3-4(a) and 3-4(b). The target edge e will be the output edge of the vertex corresponding to the circuit's output gate.

We must still address the issue of potential illegal graph successors. However, in the initial configuration the only edges that are free to reverse are those in the edge terminators and in the red-blue conversion subgraphs; all other vertices are effectively waiting for input. We add the edges in the red-blue conversion graphs to the initial edge set R_0, and we similarly add all edges in the edge terminators, except for the initial free edges that correspond to the Boolean circuit inputs. Then, no edges can ever reverse until the inputs have propagated through to them, and in each case the signals flow through appropriately. The only way to have an illegal graph successor would be to start in a configuration with all edges directed into an AND vertex, or with two edges directed into an OR, but these situations do not arise in the reduction.

Then, the Bounded DCL dynamics exactly mirror the operation of the Boolean circuit, and e will eventually reverse if and only if the circuit value is true. This shows that Bounded DCL is P-hard. Clearly it is also in P: we may compute G_{i+1} from G_i in linear time (keeping track of which edges have already reversed), and after a linear time no more edges can ever reverse. □

4.1.2 Planar Graphs

For all of the other kinds of games, it turns out that restricting the constraint graphs to planar configurations does not change their computational power. However, planar Bounded DCL seems to be weaker than unrestricted Bounded DCL. The reason is that, while *MONOTONE-CIRCUIT-VALUE* is P-complete, the planar monotone circuit value problem has been shown to lie in NC^3 [87], and it is believed that $NC^3 \subsetneq P$. Planarity is a useful property to have in Constraint Logic, because it greatly simplifies reductions to other games. In this case, however, there are no obvious games one would be interested in showing P-complete anyway, or if there are they have escaped my notice.

4.2 Unbounded Games

Unbounded zero-player games are simulations that have no *a priori* bound on how long they may run. Cellular automata, such as Conway's game of Life, are a good example. Since there is no longer a linear bound on the number of moves, it's not generally possible to determine the outcome in polynomial time. Indeed, Life has been shown to be "computation universal" [4, 85, 66] on an infinite grid; that is, it can simulate an arbitrary Turing machine. That means that there are decision questions about a Life game (for example "will this cell ever be born") that are undecidable. On a finite grid, the corresponding property is PSPACE-completeness. This result is not mentioned explicitly in the cited works, but it does follow directly, at least from [66].

Deterministic Constraint Logic (DCL) is the form of Constraint Logic that corresponds to these kinds of simulation. The definition is slightly more complicated than

for Bounded Deterministic Constraint Logic. Removing the restriction that each edge may reverse at most once means that signals would no longer naturally flow through vertices with the existing rule—when an edge reverses into a vertex, the rule would have it reverse out again on the next step, as well as whatever other edges it enabled to reverse. This would lead to illegal configurations.

Therefore, we add the restriction that an edge which just reversed may not reverse again on the next step, unless on that step there are no other reversals away from vertex that edge points to. Formally, we define a vertex v as *firing* relative to an edge set R if its incident edges which are in R satisfy its minimum inflow, and $F(G, R)$ as the set of vertices in G that are firing relative to R. Then, if we begin with graph G_0 and edge set R_0,

$$\begin{aligned} R_{i+1} &= \{e = (u, v) \mid e \notin R_i \text{ and } v \in F(G_i, R_i), \text{ or } e \in R_i \text{ and } v \notin F(G_i, R_i)\}, \\ G_{i+1} &= G_i \text{ with edges in } R_{i+1} \text{ reversed.} \end{aligned}$$

(Note that the R's are undirected edge sets here; if $(u, v) \in R$ then $(v, u) \in R$.)

The effect of this rule is that signals will flow through constraint graphs as desired, but when a signal reaches a vertex that it can't "activate", or "flow through", it will instead "bounce". (See Figure B-1 in Appendix B for an example.) For AND/OR graphs, this can only happen when a single red edge reverses into an AND vertex, and the other red edge is directed away. In DCL figures, edges that have just reversed are highlighted; they are a relevant part of the state.

This seems to be the most natural form of Constraint Logic that is unbounded and deterministic. It has the additional nice property that it is reversible. That is, if we start computing with G_{i-1} and R_i, instead of G_0 and R_0, we eventually get back to G_0. I omit the proof, but it is easy to verify by considering the small number of possible cases at a vertex. Each vertex may be considered independently here, since whether an edge reverses is a property only of the vertex it is directed towards.

The proof I present that DCL is PSPACE-complete, which is a reduction from Quantified Boolean Formulas, is not intended as a suggestion for how to actually perform computations with such circuits, were they to be physically realized. This is because that reduction entails an exponential slowdown from the computation performed on the corresponding space-bounded Turing machine. Instead, I later show how a practical reversible computer could be built using either dual-rail logic or Fredkin gates made with DCL components. Those constructions require the addition of a few (non-reversible, entropy-generating) elements that do not strictly fit within the DCL model, however, so they are not sufficient for showing PSPACE-completeness.

DETERMINISTIC CONSTRAINT LOGIC (DCL)
INSTANCE: AND/OR constraint graph G_0; edge set R_0; edge e in G_0.
QUESTION: Is there an i such that e is reversed in G_i?

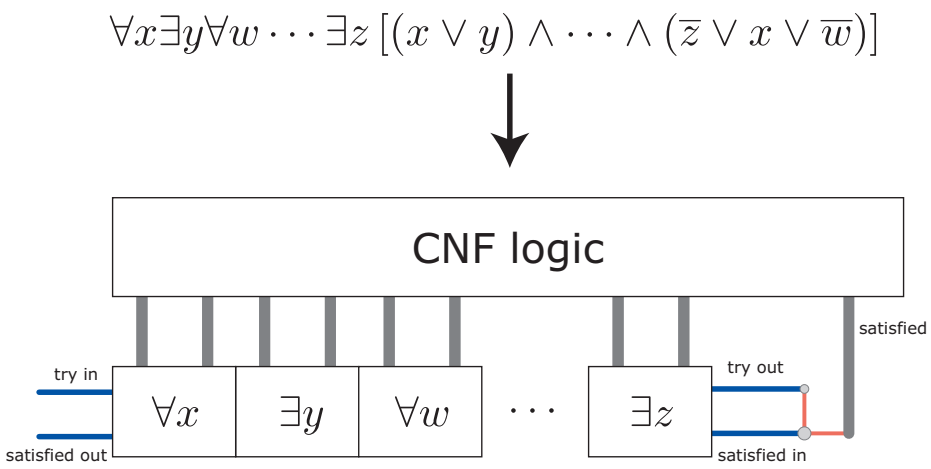


Figure 4-1: Schematic of reduction from QBF.

4.2.1 PSPACE-completeness

I show that DCL is PSPACE-complete via a reduction from Quantified Boolean Formulas (QBF; see Section A.5.3). Given an instance of QBF (a quantified Boolean formula F in CNF), we construct a corresponding constraint graph G such that iteration of the above deterministic rule will eventually reverse a target edge e if and only if F is true. The reduction is shown schematically in Figure 4-1.

This reduction is rather elaborate, even though many details are relegated to Appendix B. While this chapter is logically the first to deal with specific forms of Constraint Logic, the corresponding reduction in Chapter 5 is similar but more straightforward, and I recommend that the reader skip this reduction until after reading that one.

Reduction. One way to determine the truth of a quantified Boolean formula is as follows: Consider the initial quantifier in the formula. Assign its variable first to false and then to true, and for each assignment, recursively ask whether the remaining formula is true under that assignment. For an existential quantifier, return true if either assignment succeeds; for a universal quantifier, return true only if both assignments succeed. For the base case, all variables are assigned, and we only need to test whether the CNF formula is true under the current assignment.

The constructed constraint graph G operates in a similar fashion. Initially, the try in edge reverses into the left quantifier gadget, activating it. When a quantifier gadget is activated, it tries both possible truth values for the corresponding variable. For each, it send the appropriate truth value into the CNF logic circuitry. The CNF circuitry then sends a signal back to the quantifier gadget, having stored the value. The quantifier then activates the next quantifier's try in edge.

Eventually the last quantifier will set an assignment. Then, if the formula is satisfied by this total assignment, the last quantifier's satisfied in edge will activate. When a quantifier receives a satisfied in signal, if it is an existential quantifier, then it simply passes it back to the previous quantifier: the assignment has succeeded.

For a universal quantifier, when the first assignment succeeds, an internal memory bit is set. When the second assignment succeeds, if the memory bit is set, then the quantifier activates the previous quantifier's satisfied in input.

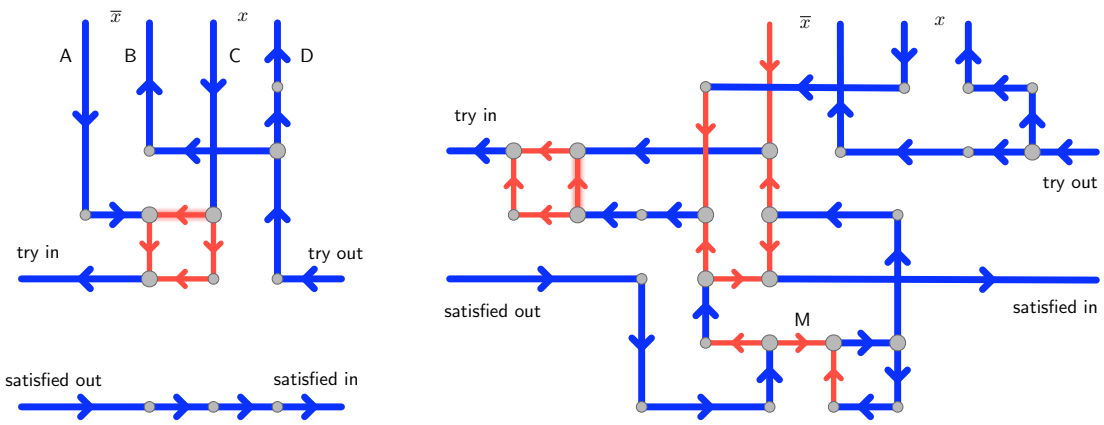
The leftmost satisfied out edge will eventually reverse if and only if the formula is true.

Timing. Since multiple signals can be bouncing around simultaneously in a DCL graph, and signals must arrive "in phase" to activate an AND vertex, timing issues are critical when designing DCL gadgets. To simplify analysis, all gadget inputs and outputs will be assumed to occur at times 0 mod 4. By this I mean that after an input has arrived, the first propagating edge reversal inside the gadget will be at time 1 mod 4, and the last edge reversal inside the gadget before the signal propagates out will be at time 0 mod 4.

Graph Shorthand. It will be most convenient to present the needed gadgets as subgraphs that are not strict AND/OR graphs. In particular, I will use red-blue vertices, blue-blue vertices (both already discussed in Section 3.2), and red-red vertices. As mentioned in Section 3.2, such degree-2 vertices are assigned an inflow constraint of 1, instead of the 2 used for ordinary degree-3 vertices, and the vertices are drawn smaller to reflect this. Essentially, such vertices are simply wires allowing a signal to flow from one edge to the other; we need these delay wires to create the correct signal phases. The following lemma shows that using these additional vertex types entails no loss of generality.

Lemma 2 *Every constraint graph G composed of AND, OR, red-blue, blue-blue, and red-red vertices has an equivalent (with respect to the DCL deterministic rule) AND/OR graph G'.*

Proof: We convert G to G' as follows. First, replace every edge with a sequence of four new edges: each blue edge is replaced by a chain of four blue edges; each red edge is replaced with a chain of four edges colored red, blue, blue, and red. However, wherever there is a red-red vertex in G, use a blue edge at those endpoints of the new chains, rather than red. That is, red-red becomes a chain red, blue, blue, blue, blue, blue, blue, red. The new graph has blue-blue and red-blue vertices, but no red-red vertices. For the blue-blue vertices, use the technique of Section 3.2: attach a constrained loose blue edge. This edge will permanently point away from the vertex, and thus the behavior will be identical to that at an actual blue-blue vertex. For the red-blue vertices, again use the subgraph given in Section 3.2. Here, we must be careful with timing. The two red edges which provide the extra inputs to the red-blue vertices (see Figure 3-3) will "bounce" each turn, as long as the blue edge is directed inward. However, we can easily arrange for the phase of the bounce to be such that whenever a signal arrives on the incoming red edge, the extra edge will also point into the vertex. This is because such reversals must always occur on a global odd time step. □



(a) Existential quantifier gadget

(b) Universal quantifier gadget

Figure 4-2: DCL quantifier gadgets.

Quantifier Gadgets. The existential and universal quantifier gadgets are shown in Figure 4-2. Their properties will be stated here; the simplest proof that they operate as described is simply a series of snapshots of states, each of which clearly follows from the previous by application of the deterministic rule. These snapshots are given in Appendix B. (All circuits were designed and tested with a DCL circuit simulator.) Note that edges that might appear to be extraneous serve to synchronize output phases, as described above. Each quantifier's try out edge is connected via a blue-blue vertex to the next quantifier's try in edge (except for the last quantifier, described later); similarly for satisfied in and satisfied out. The x and $\overline{x}$ edges likewise connect to the CNF logic circuitry, described later.

An existential quantifier assigns its variable to be first false, and then true. If either assignment succeeds, the quantifier activates its satisfied out edge. The simplest switching circuit that performs this task also assigns its variable to false again one more time, but this does not alter the result.

The gadget is activated when its try in edge reverses inward (at some time 0 mod 4). The internal red edges cause the signal to propagate to edge A three steps later. The signal then proceeds into CNF logic circuitry, described later, and returns via edge B. It then propagates to try out three steps later. Now, it is possible that satisfied in will later reverse inward; if so, satisfied out then reverses three steps later. Then, later, satisfied out may reverse inward, and satisfied in will then reverse outward three steps later. Here the sense of input and output is being reversed—the performed operation is being "undone".

Regardless of whether satisfied in was activated, later try out will reverse back inwards, continuing to unwind the computation. Then B will reverse again three steps later, and eventually A will reverse inward. Then, the switching circuit composed of the red edges will send the signal to C three steps later, effectively setting the variable to be true. Then, the same sequence as before happens, except that edges C and D are used instead of A and B. Finally, the switching circuit tries A and B again, and then at last try in is directed back outwards. The switching operates based on stored

internal state in the red edges; see Appendix B for details.

The universal quantifier gadget is similar, but more complicated. It uses the same switching circuit to assign its variable first to false and then to true, and then to false once more. However, if the false assignment succeeds – that is, if satisfied in is directed inward when the variable is set to false – then instead of propagating the signal back to the previous quantifier, the success is remembered by reversing some internal edges so that edge M is set to reversing every step. Then, if satisfied in is later directed in while M is in this state, and the variable is set to true, these conditions cause satisfied out to direct outward. Finally, in this case setting the variable to false again is useful; this causes M and the other internal edges to be restored to their original state, erasing the memory of the success when setting the variable false. Then, again, try in is directed back outward; the gadget has cleaned up and deactivated, waiting for the next assignment of the leftward variables.

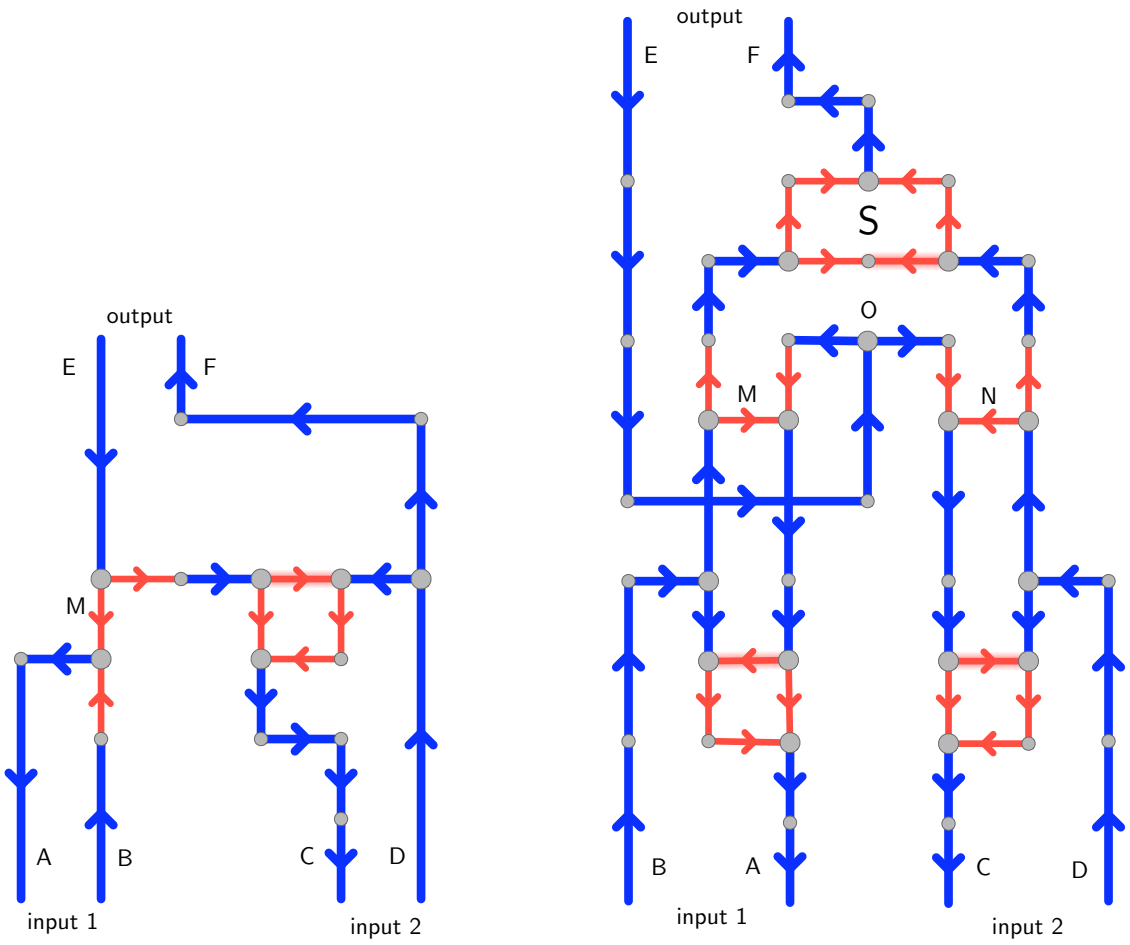
CNF Logic. We already have AND and OR vertices, so it might seem that we could simply feed the variable outputs from the quantifiers into a network of these that corresponds to the Boolean formula, and its output would activate only when the formula was satisfied by that assignment. However, the variable signals would all have to arrive simultaneously for that approach to work. Furthermore, ANDs that had only one active input would bounce that signal back, potentially confusing the timing in the gadget that sent the signal.

Rather than try to solve all such timing issues globally, I follow a strategy of keeping a single path of activation, that threads its way through the entire graph. Each gadget need merely follow the phase timing constraints described above.

I build abstract logic gates, AND′, OR′, and FANOUT′, that operate differently from their single-vertex counterparts. These gates receive inputs, remember them, and acknowledge receipt by sending a return signal. If appropriate, they also send an output signal, in a similar paired fashion, prior to sending the return signal. Later, the input signals are turned off in the reverse order they were turned on, by sending a signal back along the original return signal line; the gadgets then complete the deactivation by sending the signal back along the original input activation line.

This description will be made clearer by seeing some examples. As with the quantifier gadgets, correct operation of the CNF gadgets is demonstrated in the snapshots in Appendix B. These gadgets are connected together to correspond to the structure of the CNF formula: variable outputs feed OR′ inputs; OR′ outputs feed other OR′ inputs or AND′ inputs; AND′ outputs feed other AND′ inputs, except for the final formula output, which is combined with the final quantifier output as described later.

The AND′ gadget is shown in Figure 4-3(a). I assume that, if both inputs will arrive and the gate will activate, input 1 will arrive first; later I justify this assumption. Suppose a signal arrives at input 1, on edge A. Then, as in the universal quantifier gadget, edge M will be set bouncing to remember that this input has been asserted. If input 2 later activates, along edge C, then the same switching circuit as used in the quantifiers will send a signal so that it will arrive in phase with M, and activate the output on edge E. Later, when acknowledgment of this signal is received on edge F,



(a) AND′ gadget

(b) OR′ gadget

Figure 4-3: DCL AND′ and OR′ gadgets.

the return signal is propagated back to the second input via D.

Suppose, instead, that a signal arrives at C when one has not first arrived at A. By assumption, the variable feeding into A is then false, so the AND′ should not activate. In this case the internal switch gadget sends a signal towards E, but because M is not in the right state it bounces back, and is then switched to the exit at D. Thus, the gate has acknowledged the second input, without activating the output. The reverse process occurs when D is redirected in.

The OR′ gadget, shown in Figure 4-3(b), is significantly more complicated. This is because the gate must robustly handle the case where one input arrives and activates the output, and later the other input arrives. The gate needs to know that it has already been activated, and simply reply to the second activation gracefully. (Note the highlighted edge in Figure 4-3(b); this is an edge which has just reversed, and thus would be in the input edge set R_0.)

If an input arrives on edge A, the left switch gadget directs it up to the central OR vertex O, and then on to the output edge E. When the return signal arrives via F, the upper switch gadget S tries first one side and then the other. The edge left bouncing at M is in phase with this signal, which then propagates to B. The corresponding edge N is not bouncing, so when the signal from S arrives there it bounces back. Switch S has extra edges relative to the other switches; these create the proper signal phases. The entire process reverses when the signal is returned through B, first turning off the output, then returning via A. Since the gate is symmetric, a single input arriving at C first also behaves just as described, sending its return signal along D.

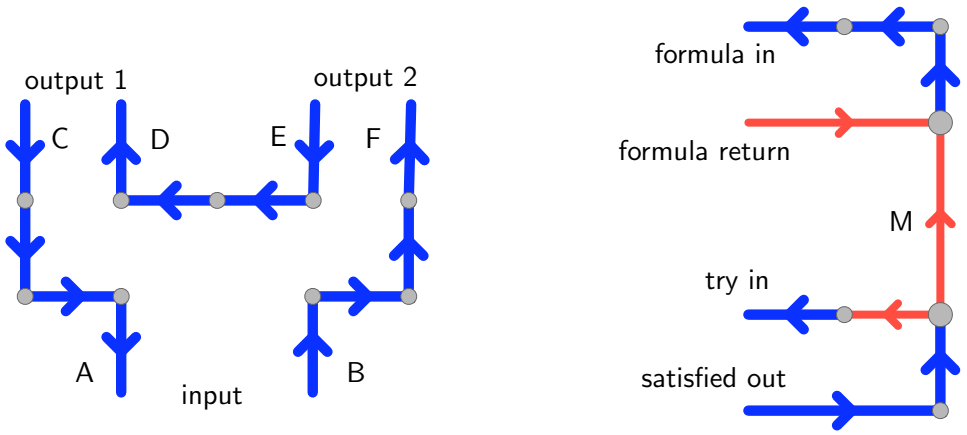
Suppose an input arrives at C after one has arrived at A and left at B, that is, when the gate is "on". Then when the signal reaches the OR vertex O it will propagate on towards M, and not towards E (because the path to E is already directed outward). But M will be directed away at this point, and the signal will bounce back, finally exiting at D with the right phase. Again, when the signal returns via D the reverse process sends it back to C. All of these sequences are shown explicitly in Appendix B.

The FANOUT′ gadget, shown in Figure 4-4(a), is straightforward. An input arriving on edge A is sent to outputs 1 and 2 in turn; then the return signal is sent back on edge B. The reverse sequence inactivates the gadget.

Remaining Connections. To start the computation, we attach a free edge terminator, as shown in Figure 3-4(a), to the leftmost quantifier's try in edge, and set that edge to reversing from G_0 to G_1. Similarly, we attach another free edge terminator to the leftmost quantifier's satisfied out edge; this is the target edge e which will reverse just when the QBF formula is true.

Finally, we must have a way to connect the rightmost quantifier and CNF outputs together, and feed the result into the rightmost quantifier's satisfied in input. This is done with the graph shown in Figure 4-4(b). The rightmost quantifier's try out edge connects to the try in edge here, and its satisfied in edge connects to the satisfied out edge here. The output of the CNF formula's final AND′ gadget connects to formula in, and its return output edge connects to formula return.

If the formula is ever satisfied by the currently-activated variable assignment from



(a) FANOUT′ gadget (b) CNF output gadget

Figure 4-4: Additional CNF gadgets.

all the quantifiers, then a signal will arrive at formula in and exit at formula return, leaving edge M bouncing every step. Then, when the last quantifier activates its try out, the signal arriving at try in will be in phase with M, propagating the signal on to the last quantifier's satisfied in input, where it will be processed as described above. If the formula is not satisfied, M will still be pointing up, and the try in signal will just bounce back into the last quantifier.

AND′ Ordering. As mentioned above, I assume that input 1 of an AND′ will always activate, if at all, before input 2. However, in a general quantified CNF formula it is not the case that the clauses need be satisfied in any predetermined order, if the variables are assigned in the order quantified. To solve this problem, we modify the circuit described above as follows. We protect input 2 of every AND′ a_1 in the original circuit with an additional AND′ a_2, so that the original input 2 signal now connects to a_2's input 1, and a_2's output connects to a_1's input 2.

Then, the rightmost quantifier's try out edge, instead of connecting directly to the merge gadget shown in Figure 4-4(b), first threads through every newly introduced AND′ input 2 pathway, and then from there connects to the merge gadget. (We make sure that the introduced pathways are the right length to satisfy the timing rule.) Thus, as the variable values are set, when an AND′ would have its second input arrive before the first in the original circuit, in the modified circuit the second input's activation is deferred until its first input has had a chance to arrive. This way, we can ensure that the inputs arrive in the right order, by threading the path from the final try out appropriately.

Theorem 3 *DCL is PSPACE-complete.*

Proof: Given a quantified Boolean formula F, we construct DCL graph G_0 and edge set R_0 as described above. The individual gadgets' correct operation is explicitly shown in Appendix B. The leftmost quantifier's satisfied out edge will eventually

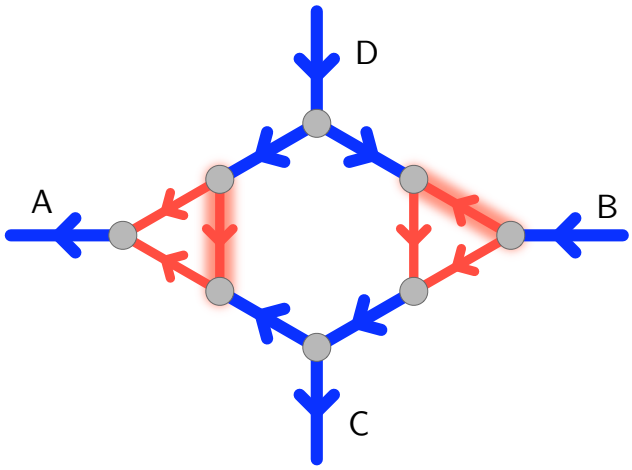


Figure 4-5: DCL crossover gadget.

reverse if and only if F is true. We convert the constructed graph to an equivalent AND/OR graph, via Lemma 2. This shows that DCL is PSPACE-hard.

DCL is also clearly in PSPACE: a simple, constant-space deterministic algorithm executes each deterministic step of the graph, and detects when the target edge reverses. □

4.2.2 Planar Graphs

The gadgets used above are non-planar, and additional edge crossings are necessary within the CNF network. For purposes of reductions to actual games, and also for physical implementation of DCL, it would be desirable to use only planar circuits.

The gadget shown in Figure 4-5 shows how to cross signals in DCL. (The gadget must be padded with extra edges to satisfy the timing constraints.) As shown in Appendix B, a signal arriving at edge A will exit via B, and likewise for C and D. The sequence A to B, C to D also works, as do reverses of all these sequences. However, one sequence which does not work as desired is the following: C to D, A to B. After a C to D traversal, a signal arriving at A exits at C instead of B.

This limitation will not matter for our application, however; all crossings in the above construction are of the form that one edge will always be activated first, and the second, if activated, will deactivate before the first. This may be verified for the crossings within the gadgets by examining the activation sequences in Appendix B. For the other crossings, within the CNF logic, the global pattern of activation ensures that a pathway is never deactivated until any pathways it has activated are first deactivated.

To see that a particular crossover input always arrives first when two eventually arrive within the CNF logic, note that there are two types of such crossings: variable outputs crossing on their way to OR′ inputs, and extra crossings created by the AND′-ordering pathway discussed above. (The OR′ outputs don't need to cross on their way

to the AND′ inputs, because in CNF this part of the network is just a tree.) In the first case, whenever both crossover inputs are active, the one from the earlier variable in the quantification sequence clearly must have arrived first. In the second case, the crossover input from the AND′-ordering pathway must always arrive after the input from the path it is crossing.

Theorem 4 *DCL is PSPACE-complete, even for planar graphs.*

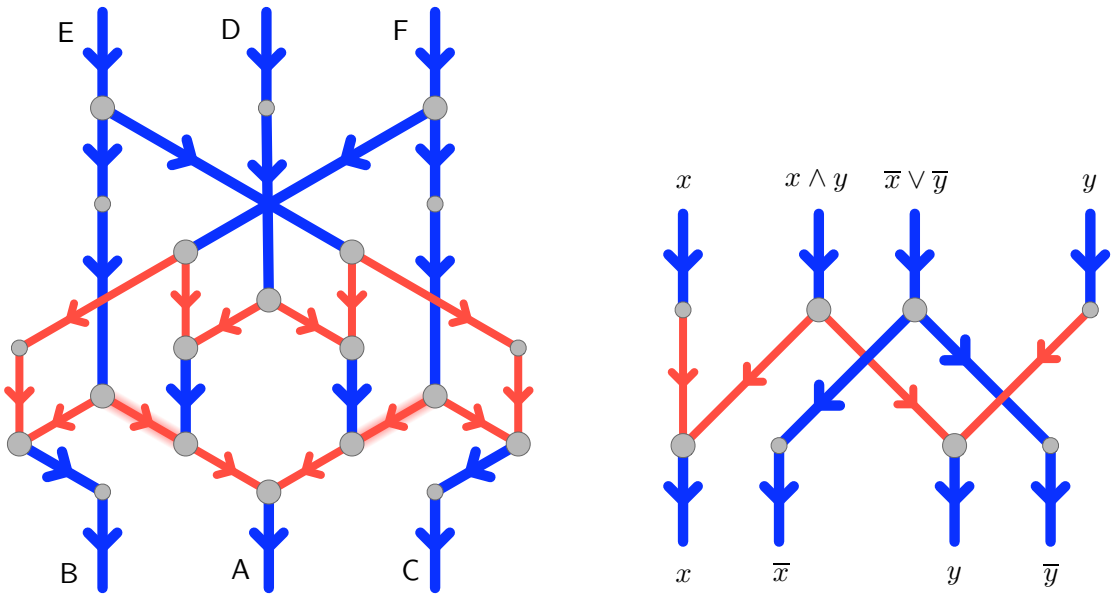
Proof: Everywhere edges cross in the original construction, we replace that crossing pair by a crossover gadget, suitably padded with extra edges to satisfy the timing requirements. We can easily ensure that it takes time 1 mod 4 to traverse a crossover in either direction; this guarantees that the gadget timing will be insensitive to the replacement. □

4.2.3 Efficient Reversible Computation

Ordinary computers are not reversible. As a result, the information losses that are constantly occurring in an ordinary computer result in an increase in entropy, manifested as heat. This is known as Landauer's Principle: every bit of lost information results in a dissipation of $kT \ln 2$ joules of energy [51] (where k is Boltzmann's constant). Thus, as computers perform more operations per second, they require more power, and dissipate more heat. This seems obvious. However, it is theoretically possible to build a reversible computer, in which all of the atomic steps, except for recording desired output, are reversible [2], and dissipate arbitrarily little heat. Reversible computing is an active area of research (see e.g. [27]), with many engineering challenges, and potential for dramatic increases in effective computing power over conventional computers.

Deterministic Constraint Logic represents a new style of reversible computation, which could potentially have practical application. It could be that the DCL deterministic edge-reversal rule is possible to implement effectively on a microscopic, perhaps molecular, level; certainly, the basic mechanism of switching edges between two states is suggestive of a simple physical system with two energy states.

As mentioned earlier, the construction showing DCL PSPACE-complete is not useful from a practical standpoint, because it involves an exponential slowdown in the reduction from Turing machine to QBF formula to DCL process. However, it is possible to build conventional kinds of reversible computing elements from DCL components. Figure 4-6(a) shows a Fredkin gate [29]. This gate has the property that a signal arriving on edge A, B, or C will be propagated to D, E, or F, respectively, but A and B in combination will activate D and F, and A and C in combination will activate D and E. Effectively, it is a "controlled switch" gate. It is also possible to directly implement a reversible dual-rail logic, as shown in Figure 4-6(b). Real computers built from such DCL gadgets would still require some localized non-reversible components, as would any usable reversible computer, which is why these gadgets were not the basis for the PSPACE-completeness proof.



(a) Fredkin gate

(b) Dual-rail logic

Figure 4-6: DCL reversible computation gadgets.

Chapter 5

One-Player Games (Puzzles)

A one-player game is a puzzle: one player makes a series of moves, trying to accomplish some goal. For example, in a sliding-block puzzle, the goal could be to get a particular block to a particular location. I will used the terms "puzzle" and "one-player game" interchangeably. For puzzles, the generic forced-win decision question—"does player X have a forced win?"—becomes "is this puzzle solvable?".

Constraint Logic. The one-player version of Constraint Logic is called Nondeterministic Constraint Logic (NCL). The rules are simply that on a turn the player may reverse any edge resulting in a legal configuration, and the decision question is whether a given edge may ever be reversed.

The unbounded version of NCL was inspired by Flake and Baum's "Generalized Rush Hour Logic" (GRL) [24], and in fact GRL incorporates gadgets with the same properties as my AND and OR vertices. GRL also requires a crossover gadget, however, which NCL does not—in the approach I take for showing PSPACE-completeness, it is possible to cross signals using merely AND and OR vertices. GRL uses a different notion of signal (dual-rail logic), for which this approach is not possible. The inherent flavor of GRL is also quite different from that of NCL; at a conceptual level, GRL requires inverters, and so is not monotone. However, formally it is the case that NCL is merely GRL reformulated as a graph problem, without a crossover gadget, and Flake and Baum deserve the credit for the original PSPACE-completeness proof. I take a different, simpler approach for showing PSPACE-completeness, reducing from QBF. Flake and Baum explicitly build a space-bounded, reversible computer.

Due to the simplicity of NCL, and the abundance of puzzles with reversible moves, it is often straightforward to find reductions showing various puzzles PSPACE-hard. This is the largest class of reductions presented in Part II.

Bounded NCL reverts essentially to Satisfiability (SAT), which is NP-complete. However, the NCL crossover gadget still works, which makes reductions from Bounded NCL to bounded puzzles much more straightforward than direct reductions from SAT. Planar SAT is also NP-complete [52], but that result is not generally useful for puzzle reductions. For Planar SAT, the graph corresponding to the formula is a planar bipartite graph, with vertex nodes and clause nodes, plus a loop connecting the vertex nodes. The clause nodes are not connected, however. In contrast, a Bounded

NCL graph corresponding to a Boolean formula feeds all the AND outputs into one final AND; reversing that final AND's output edge is possible just when the formula is satisfiable. Typically, this is a critical structure for puzzle reductions, because the victory condition is usually a local property (such as moving a black to a particular place) rather than a distributed property of the entire configuration. Thus, puzzle reductions typically require the construction of a crossover gadget, even though Planar SAT is NP-complete, and the Planarity result for NCL is thus stronger in a sense than that for SAT.

5.1 Bounded Games

Bounded one-player games are puzzles in which there is a polynomial bound (typically linear) on the number of moves that can be made. Usually there is some resource that is used up. For example, in a game of Sudoku, the grid eventually fills up with numbers, and then either the puzzle is solved or it is not. In Peg Solitaire, each jump removes one peg, until eventually no more jumps can be made. The nondeterminism of these games, plus the polynomial bound, means that they are in NP – a nondeterministically guessed solution can be checked for validity in polynomial time.

Bounded Nondeterministic Constraint Logic (Bounded NCL) is the form of Constraint Logic that corresponds to this type of puzzle. It is NP-complete. It is formally defined as follows:

BOUNDED NONDETERMINISTIC CONSTRAINT LOGIC (BOUNDED NCL)
INSTANCE: AND/OR constraint graph G, edge e in G.
QUESTION: Is there a sequence of moves on G that eventually reverses e, such that each edge is reversed at most once?

A Bounded NCL graph abstracts the essence of a bounded puzzle; it also serves as a concise model of polynomial-time-bounded nondeterministic computation.

5.1.1 NP-completeness

We reduce 3SAT (Section A.5.2) to Bounded NCL to show NP-hardness. Given an instance of 3SAT (a Boolean formula F in 3CNF), we construct an AND/OR constraint graph G with an edge e that can be eventually reversed just when F is satisfiable.

First we construct a general constraint graph G' corresponding to F, then we apply the conversion techniques described in Chapter 3 to transform G' into a strict AND/OR graph G. Constructing G' is straightforward. For each variable in F we have one CHOICE (red-red-red) vertex; for each OR in F we have an OR vertex; for each AND in F we have an AND vertex. At each CHOICE, one output corresponds to the negated form of the corresponding variable; the other corresponds to the negated form. The CHOICE outputs are connected to the OR inputs, using FANOUTs (which

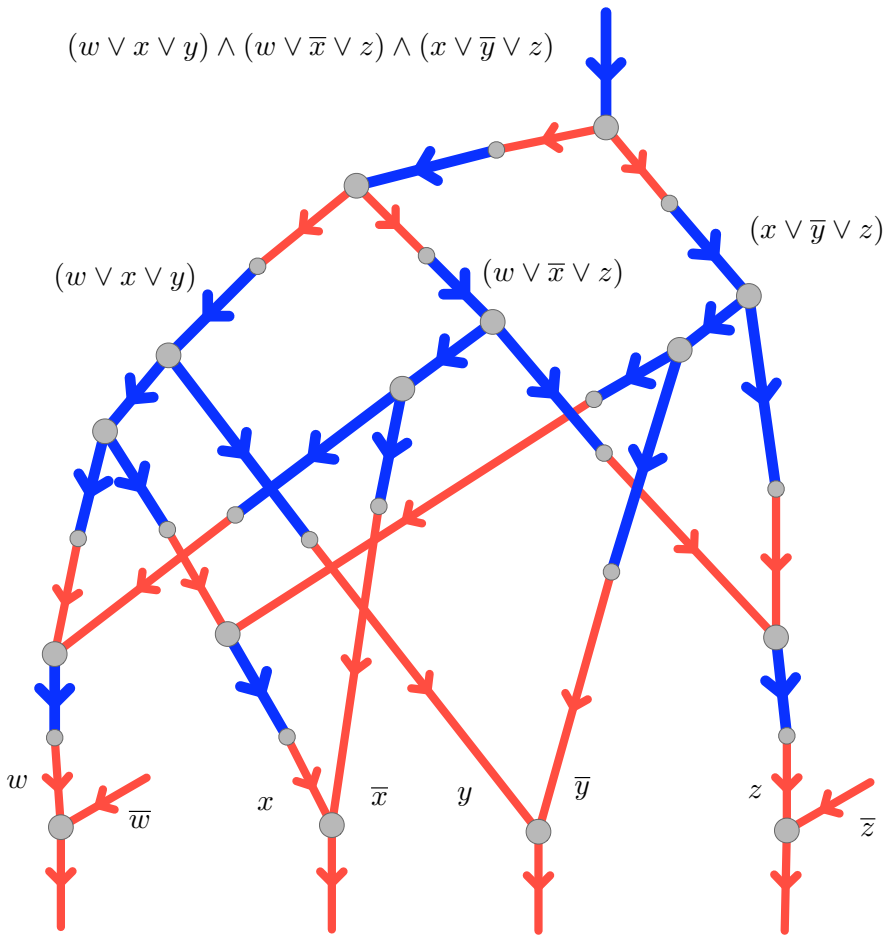


Figure 5-1: A constraint graph corresponding to the formula $(w \vee x \vee y) \wedge (w \vee \overline{x} \vee z) \wedge (x \vee \overline{y} \vee z)$. Edges corresponding to literals, clauses, and the entire formula are labeled.

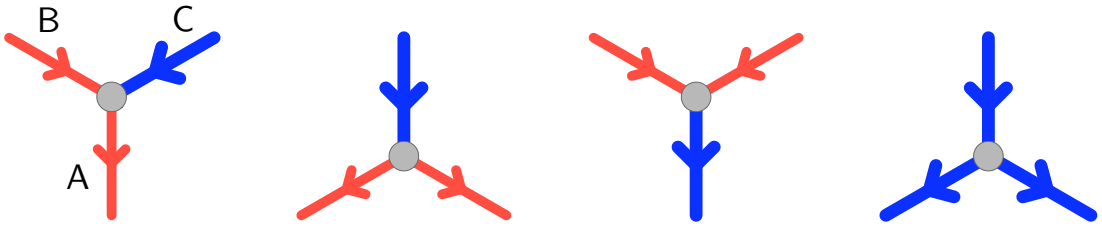
are the same as AND vertices) as needed. The outputs of the ORs are connected to the inputs of the ANDs. Finally, there will be one AND whose output corresponds to the truth of F. A sample graph representing a formula is shown in Figure 5-1.

Theorem 5 *Bounded NCL is NP-complete.*

Proof: Given an instance of 3SAT (a Boolean formula F in 3CNF), we construct graph G' as described above.

It is clear that if F is satisfiable, the CHOICE vertex edges may be reversed in correspondence with a satisfying assignment, such that the output edge may eventually be reversed. Similarly, if the output edge may be reversed, then a satisfying assignment may be read directly off the CHOICE vertex outputs.

Using the techniques described in Section 3.2, we can replace the CHOICE vertices, the terminal edges, and the red-blue vertices in G' with equivalent AND/OR constructions, so that we have an AND/OR graph G that can be solved just when F is satisfiable. Therefore, Bounded NCL is NP-hard.



(a) Unusual vertex (b) AND (c) FANOUT (d) OR

Figure 5-2: Distinct types of AND/OR vertex used in the Bounded NCL reduction.

(a) CHOICE (b) AND (c) FANOUT (d) OR

Figure 5-3: An equivalent set of vertices, better suited for reductions.

Bounded NCL is also clearly in NP. Since each edge can reverse at most once, there can only be polynomially many moves in any valid solution; therefore, we can guess a solution and verify it in polynomial time. □

5.1.2 Planar Graphs

For reductions to actual games, it is desirable to start from planar constraint graphs, instead of having to build a complicated crossover gadget for each new application. As mentioned above, the result that planar SAT is NP-complete is not useful to us for either constraint graphs or actual games. But it is possible to build a crossover gadget within NCL.

Theorem 6 *Bounded NCL is NP-complete, even for planar graphs.*

Proof: The crossover gadget is shown in Figure 5-8(a) (page 48). Rather than give an explicit proof of correctness here and then a more general proof for the unbounded case, we merely point out that the proof for the unbounded case, of Lemma 13, also applies to the bounded case; in the described sequences, no edge need ever reverse more than once. □

5.1.3 An Alternate Vertex Set

For reductions from (unbounded) NCL to actual puzzles, we only need consider two vertex types, AND and OR. However, for bounded NCL, there is effectively a symmetry

breaking. As mentioned, AND is effectively no longer the same type of vertex as FANOUT, even though it has the same constraints, because once either type of vertex has been used it cannot then be reversed for the other type of behavior. Effectively, there are now four types of vertex we must emulate for puzzle reductions, shown in Figure 5-2. In addition to AND, OR, and FANOUT, there is the unusual vertex in Figure 5-2(a). This has the property that initially, edge B may reverse, but if instead A reverses and then C reverses, A may no longer reverse. This set of constraints is not particularly natural, and an alternative vertex will do just as well, namely, the CHOICE vertex actually used in the reduction from 3SAT. The vertex in Figure 5-2(a) is only used inside the CHOICE conversion from Section 3.2, and inside the crossover construction in Figure 5-8(a). Figure 5-8(a) may be easily redrawn to use CHOICE vertices instead. Therefore, the set of vertices shown in Figure 5-3 is sufficient for reductions.

It will also turn out to be useful to reduce from graphs that have the property that only a single edge can initially reverse. For this, we will have to explicitly add loose edges and red-blue vertices to the gadget set to reduce from. Then, we simply take a single loose edge, and split it enough times to reach all the free CHOICE inputs in the reduction. Red-blue vertices generally add nothing to the difficulty of a reduction, because typically the difference between red and blue edges is manifested in reductions only inasmuch as AND and OR are different.

Theorem 7 *Theorem 6 remains true when the input graph uses the vertex types in Figure 5-3, as well as red-blue vertices, and a single loose edge, rather than a complete set of directional AND/OR vertices.*

Proof: As above. □

5.2 Unbounded Games

The material in this section is joint work with Erik Demaine [46, 47].

Unbounded one-player games are puzzles in which there is no restriction on the number of moves that can be made. Typically the moves are reversible. For example, in a sliding-block puzzle, the pieces may be slid around in the box indefinitely, and a block once slid can always be immediately slid back to its previous position. Since there is no polynomial bound on the number of moves required to solve the puzzle, it is no longer possible to verify a proposed solution in polynomial time – the solution could have exponentially many moves. Indeed, unbounded puzzles are often PSPACE-complete. It is clear that such puzzles can be solved in nondeterministic polynomial space (NPSPACE), by nondeterministically guessing a satisfying sequence of moves; the only state required is the current configuration and the current move. But Savitch's Theorem [71] says that PSPACE = NPSPACE, so these puzzles can also be solved using deterministic polynomial space.

Nondeterministic Constraint Logic (NCL) is the form of Constraint Logic that corresponds to this type of puzzle. It is PSPACE-complete. It is formally defined as follows:

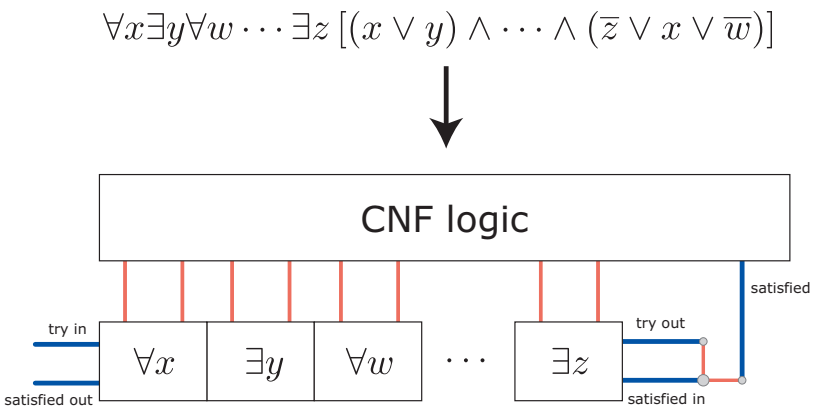


Figure 5-4: Schematic of the reduction from Quantified Boolean Formulas to NCL.

NONDETERMINISTIC CONSTRAINT LOGIC (NCL)
INSTANCE: AND/OR constraint graph G, edge e in G.
QUESTION: Is there a sequence of moves on G that eventually reverses e?

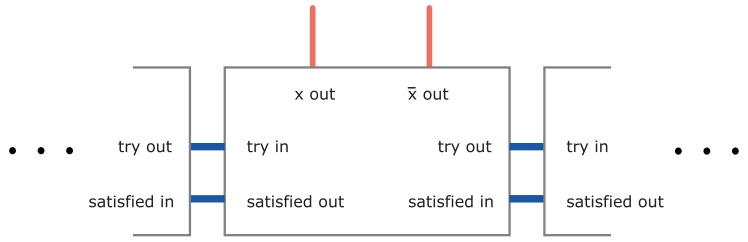
5.2.1 PSPACE-completeness

I show that NCL is PSPACE-hard by giving a reduction from Quantified Boolean Formulas (QBF). A simple argument then shows that configuration-to-edge is in PSPACE, and therefore PSPACE-complete. The PSPACE-completeness of the other two decision problems also follows simply.

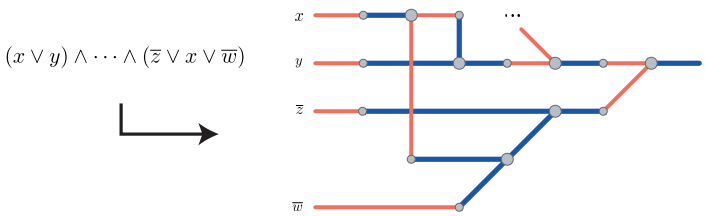
Reduction. First I give an overview of the reduction and the necessary gadgets; then I analyze the gadgets' properties. The reduction is illustrated schematically in Figure 5-4. We translate a given quantified Boolean formula F into an AND/OR constraint graph, so that a particular edge in the graph may be reversed if and only if F is true. The reduction is similar to the one for Deterministic Constraint Logic, in Chapter 4, but a bit simpler. Note that the constructed graph uses some red-blue vertices and some free edges, which can be converted to explicit AND/OR form using the techniques of Section 3.2.

One way to determine the truth of a quantified Boolean formula is as follows: Consider the initial quantifier in the formula. Assign its variable first to false and then to true, and for each assignment, recursively ask whether the remaining formula is true under that assignment. For an existential quantifier, return true if either assignment succeeds; for a universal quantifier, return true only if both assignments succeed. For the base case, all variables are assigned, and we only need to test whether the CNF formula is true under the current assignment.

This is essentially the approach used in the reduction. I define *quantifier gadgets*, which are connected together into a string, one per quantifier in the formula, as in Figure 5-5(a). The rightmost edges of one quantifier are identified with the leftmost edges of the next. (This is different from the corresponding reduction in Chapter 4, where the edges are distinct and join at a blue-blue vertex.) Each quantifier gadget



(a) Quantifier gadget connections



(b) Part of a CNF formula graph

Figure 5-5: QBF wiring.

outputs a pair of edges corresponding to a variable assignment. These edges feed into the *CNF network*, which corresponds to the unquantified formula. The output from the CNF network connects to the rightmost quantifier gadget; the output of the overall graph is the satisfied out edge from the leftmost quantifier gadget.

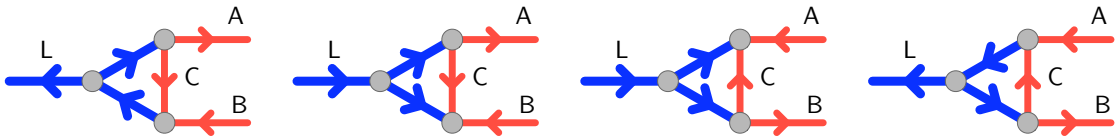
Quantifier Gadgets. When a quantifier gadget is *activated*, all quantifier gadgets to its left have fixed particular variable assignments, and only this quantifier gadget and those to the right are free to change their variable assignments. The activated quantifier gadget can declare itself *satisfied* if and only if the Boolean formula read from here to the right is true given the variable assignments on the left.

A quantifier gadget is activated by directing its try in edge inward. Its try out edge is enabled to be directed outward only if try in is directed inward, and its variable state is locked. A quantifier gadget may nondeterministically "choose" a variable assignment, and recursively "try" the rest of the formula under that assignment and those that are locked by quantifiers to its left. The variable assignment is represented by two output edges (x and $\overline{x}$), only one of which may be directed outward. For satisfied out to be directed outward, indicating that the formula from this quantifier on is currently satisfied, satisfied in must be directed inward.

I construct both existential and universal quantifier gadgets, described below, satisfying the above requirements.

Lemma 8 *A quantifier gadget's satisfied in edge may not be directed inward unless its try out edge is directed outward.*

Proof: By induction. The condition is explicitly satisfied in the construction for the rightmost quantifier gadget, and each quantifier gadget requires try in to be directed



(a) Locked, A active (b) Unlocked, A active (c) Unlocked, B active (d) Locked, B active

Figure 5-6: Latch gadget, transitioning from state A to state B.

inward before try out is directed outward, and requires satisfied in to be directed inward before satisfied out is directed outward. □

CNF Formula. In order to evaluate the formula for a particular variable assignment, we construct an AND/OR subgraph corresponding to the unquantified part of the formula, fed inputs from the variable gadgets, and feeding into the satisfied in edge of the rightmost quantifier gadget, as in Figure 5-4. The satisfied in edge of the rightmost quantifier gadget is further protected by an AND vertex, so it may be directed inward only if try out is directed outward and the formula is currently satisfied.

Because the formula is in conjunctive normal form, and we have edges representing both literal forms of each variable (true and false), we don't need an inverter for this construction. Part of such a graph is shown in Figure 5-5(b). (Also see Figure 5-1.)

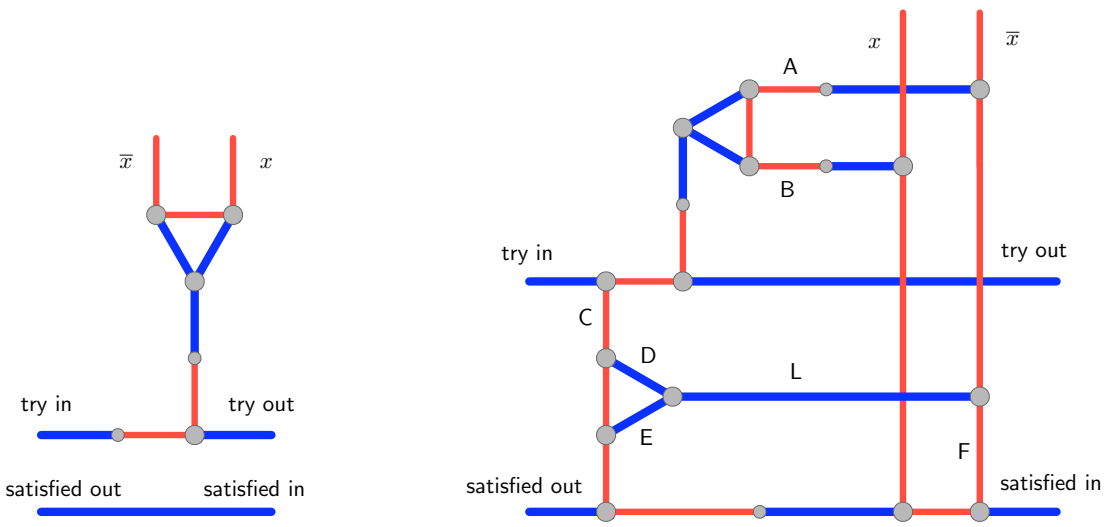
Lemma 9 *The satisfied out edge of a CNF subgraph may be directed outward if and only if its corresponding formula is satisfied by the variable assignments on its input edge orientations.*

Proof: Definition of AND and OR vertices, and the CNF construction described. □

Latch Gadget. Internally, the quantifier gadgets use *latch gadgets*, shown in Figure 5-6. This subgraph effectively stores a bit of information, whose state can be locked or unlocked. With edge L directed left, one of the other two OR edges must be directed inward, preventing its output red edge from pointing out. The orientation of edge C is fixed in this state. When L is directed inward, the other OR edges may be directed outward, and the red edges are free to reverse. Then when the latch is locked again, by directing L left, the state has been switched.

Existential Quantifier. An existential quantifier gadget (Figure 5-7(a)) uses a latch subgraph to represent its variable, and beyond this latch has the minimum structure needed to meet the definition of a quantifier gadget. If the formula is true under some assignment of an existentially quantified variable, then its quantifier gadget may lock the latch in the corresponding state, enabling try out to activate, and recursively receive the satisfied in signal. Receiving the satisfied in signal simultaneously passes on the satisfied out signal to the quantifier on the left.

Here we exploit the nondeterminism in the model to choose the correct variable assignment.



(a) Existential quantifier

(b) Universal quantifier

Figure 5-7: Quantifier gadgets.

Universal Quantifier. A universal quantifier gadget is more complicated (Figure 5-7(b)). It may only direct satisfied out leftward if the formula is true under both variable assignments. Again we use a latch for the variable state; this time we split the variable outputs, so they can be used internally. In addition, we use a latch internally, as a memory bit to record that one variable assignment has been successfully tried. With this bit set, if the other assignment is then successfully tried, satisfied out is allowed to point out.

Lemma 10 *A universal quantifier gadget may direct its satisfied out edge outward if and only if at one time its satisfied in edge is directed inward while its variable state is locked in the false ($\overline{x}$) assignment, and at a later time the satisfied in edge is directed inward while its variable state is locked in the true (x) assignment, with try in directed inward throughout.*

Proof: First I argue that, with try in directed outward, edge E must point right. The try out edge must be directed inward in this case, so by Lemma 8, satisfied in must be directed outward. As a consequence, F and thus L must point right. On the other hand, C must point up and thus D must point left. Therefore, E is forced to point right in order to satisfy its OR vertex.

Suppose that try in is directed inward, the variable is locked in the false state (edge A points right), and satisfied in is directed inward. These conditions allow the internal latch to be unlocked, by directing edge L left. With the latch unlocked, edge E is free to point left. The latch may then lock again, leaving E pointing left (because C may now point down, allowing D to point right). Now, the entire edge reversal sequence that occurred between directing try out outward and unlocking the internal latch may be reversed. After try out has deactivated, the variable may be unlocked, and change state. Then, suppose that satisfied in activates with the variable locked

in the true state (edge B points right). This condition, along with edge E pointing left, is both necessary and sufficient to direct satisfied out outward. □

The behavior of both types of quantifiers is captured with the following property:

Lemma 11 *A quantifier gadget may direct its satisfied out edge out if and only if its try in edge is directed in, and the formula read from the corresponding quantifier to the right is true given the variable assignments that are fixed by the quantifier gadgets to the left.*

Proof: By induction. By Lemmas 8 and 10, if a quantifier gadget's satisfied in edge is directed inward and the above condition is inductively assumed, then its satisfied out edge may be directed outward only if the condition is true for this quantifier gadget as well. For the rightmost quantifier gadget, the precondition is explicitly satisfied by Lemma 9 and the construction in Figure 5-4. □

Theorem 12 *NCL is PSPACE-complete.*

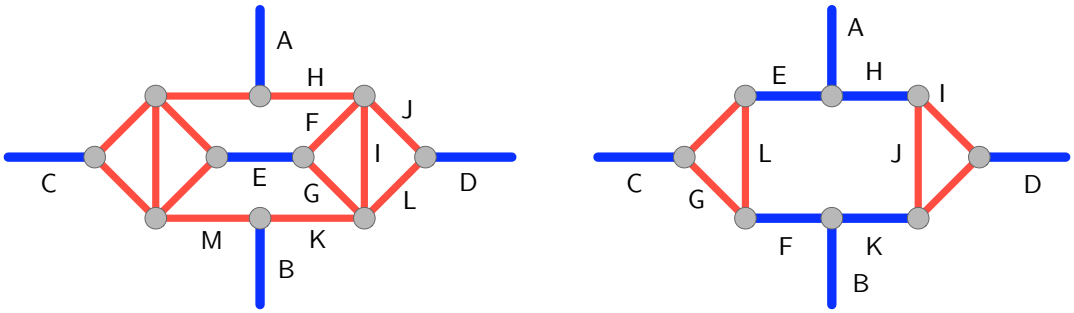
Proof: The graph is easily seen to have a legal configuration with the quantifier try in edges all directed leftward; this is the input graph G. The leftmost quantifier's try in edge may freely be directed rightward to activate the quantifier. By Lemma 11, the satisfied out edge of the leftmost quantifier gadget may be directed leftward if and only if F is true. Therefore, deciding whether that edge may reverse also decides the QBF problem, so NCL is PSPACE-hard.

NCL is in PSPACE because the state of the constraint graph can be described in a linear number of bits, specifying the direction of each edge, and the list of possible moves from any state can be computed in polynomial time. Thus we can nondeterministically traverse the state space, at each step nondeterministically choosing a move to make, and maintaining the current state but not the previously visited states. Savitch's Theorem [71] says that this NPSPACE algorithm can be converted into a PSPACE algorithm. □

5.2.2 Planar Graphs

The result obtained in the previous section used particular constraint graphs, which turn out to be nonplanar. Thus, reductions from NCL to other problems must provide a way to encode arbitrary graph connections into their particular structure. For 2D motion-planning kinds of problems, such a reduction would typically require some kind of crossover gadget. Crossover gadgets are a common requirement in complexity results for these kinds of problems, and can be among the most difficult gadgets to design. For example, the crossover gadget used in the proof that Sokoban is PSPACE-complete [13] is quite intricate. A crossover gadget is also among those used in the Rush Hour proof [24].

In this section I show that any AND/OR constraint graph can be translated into an equivalent planar AND/OR constraint graph, obviating the need for crossover gadgets in reductions from NCL.



(a) Crossover (b) Half-crossover

Figure 5-8: Planar crossover gadgets.

Figure 5-8(a) illustrates the reduction. In addition to AND and OR vertices, this subgraph contains red-red-red-red vertices; these need any two edges to be directed inward to satisfy the inflow constraint of 2. (Next I will show how to substitute AND/OR subgraphs for these vertices.)

Lemma 13 *In a crossover subgraph, each of the edges A and B may face outward if and only if the other faces inward, and each of the edges C and D may face outward if and only if the other faces inward.*

Proof: I show that edge B can face down if and only if A does, and D can face right if and only if C does. Then by symmetry, the reverse relationships also hold.

Suppose A faces up, and assume without loss of generality that E faces left. Then so do F, G, and H. Because H and F face left, I faces up. Because G and I face up, K faces right, so B must face up. Next, suppose D faces right, and assume without loss of generality that I faces down. Then J and F must face right, and therefore so must E. An identical argument shows that if E faces right, then so does C.

Suppose A faces down. Then H may face right, I may face down, and K may face left (because E and D may not face away from each other). Symmetrically, M may face right; therefore B may face down. Next, suppose D faces left, and assume without loss of generality that B faces up. Then J and L may face left, and K may face right. Therefore G and I may face up. Because I and J may face up, F may face left; therefore, E may face left. An identical argument shows that C may also face left. □

Next, I show how to represent the degree-4 vertices in Figure 5-8(a) with equivalent AND/OR subgraphs. The necessary subgraph is shown in Figure 5-8(b). This is the same subgraph as Deterministic Constraint Logic crossover gadget (Figure 4-5), but the different rules mean that a more complex crossover is necessary for NCL. Note that red-blue vertices are necessary when substituting this subgraph into the previous one; the terminal edges in Figure 5-8(b) are blue, but it replaces red-red-red-red vertices. We must be careful to keep the graph planar when performing the red-blue reduction shown in Figure 3-3. But this is easy; we pair up edges A and D, and edges B and C.

Lemma 14 *In a half-crossover gadget, at least two of the edges A, B, C, and D must face inward; any two may face outward.*

Proof: Suppose that three edges face outward. Without loss of generality, assume that they include A and C. Then E and F must face left. This forces H to face left and I and J to face up; then D must face left and K must face right. But then B must face up, contradicting the assumption.

Next we must show that any two edges may face outward. We already showed how to face A and C outward. A and B may face outward if C and D face inward: we may face G and L down, F and K right, I and J up, and H and E left, satisfying all vertex constraints. Also, C and D may face outward if A and B face inward; the obvious orientations satisfy all the constraints. By symmetry, all of the other cases are also possible. □

The crossover subgraph has blue free edges; what if we need to cross red edges, or a red and a blue edge? For crossing red edges, we may attach red-blue conversion subgraphs to the crossover subgraph in two pairs, as we did for the half-crossover. We may avoid having to cross a red edge and a blue edge, as follows: replace one of the blue edges with a blue-red-blue edge sequence, using a dual red-blue conversion subgraph. Then the original blue edge may be effectively crossed by crossing two red edges instead.

Theorem 15 *NCL is PSPACE-complete, even for planar graphs.*

Proof: Lemmas 13 and 14. Any crossing edge pairs may be replaced by the above constructions; a crossing edge may be reversed if and only if a corresponding crossover edge (e.g., A or C) may be reversed. We must also specify configurations in the replacement graph corresponding to source configurations, but this is easy: pick any legal configuration of the crossover subgraphs with matching crossover edges. □

5.2.3 Protected OR Graphs

So far I have shown that NCL is PSPACE-complete for planar AND/OR constraint graphs. It is useful to make the conditions required for PSPACE-completeness still weaker; this will make the puzzle reductions in Chapter 9 simpler.

I call an OR vertex *protected* if there are two of its edges that, due to global constraints, cannot simultaneously be directed inward. Intuitively, graphs with only protected ORs are easier to reduce to another problem domain, since the corresponding OR gadgets need not function correctly in all the cases that a true OR must. We can simulate an OR vertex with a subgraph all of whose OR vertices are protected, as shown in Figure 5-9.

Lemma 16 *Edges A, B, and C in Figure 5-9 satisfy the same constraints as an OR vertex; all ORs in this subgraph are protected.*

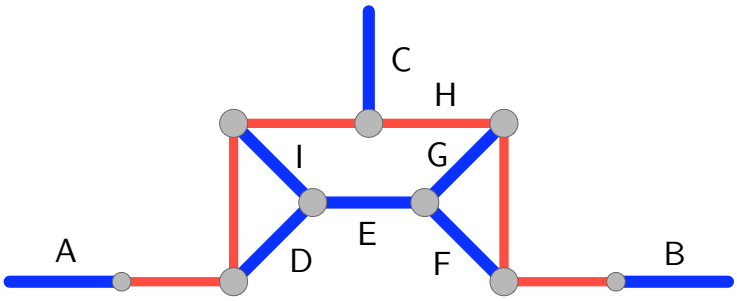


Figure 5-9: OR vertex made with protected OR vertices.

Proof: Suppose that edges A and B are directed outward. Then D and F must be directed away from E. Assume without loss of generality that E points left. Then so must G; this forces H right and C down, as required. Then, by pointing A, D, and E right, we can direct G right, H left, and C up. Symmetrically, we can direct A and C out, and B in.

The two OR vertices shown in the subgraph are protected: edges I and D cannot both be directed inward, due to the red edge they both touch; similarly, G and F cannot both be directed inward. The red-blue conversion subgraph (Figure 3-3) we need for the two red-blue vertices also contains an OR vertex, but this is also protected. □

Theorem 17 *Theorem 15 remains valid even when all of the OR vertices in the input graph are protected.*

Proof: Lemma 16. Any OR vertex may be replaced by the above construction; an OR edge may be reversed if and only if a corresponding subgraph edge (A, B, or C) may be reversed. We must also specify configurations in the replacement graph corresponding to source configurations: pick any legal configuration of the subgraphs with matching edges. □

5.2.4 Configuration-to-Configuration Problem

Occasionally it will be desirable to reduce NCL to a problem in which the goal is to achieve some particular total state, rather than an isolated partial state. For example, in many sliding-block puzzles the goal is to move a particular piece to a particular place, but in others it is to reach some given complete configuration. One version of such a problem is the Warehouseman's Problem (Section 9.3).

For such problems, I show that an additional variant of NCL is hard.

Theorem 18 *Theorem 17 remains valid when the decision question is whether there is a sequence of moves on G that reaches a new state G', rather than reverses an edge e.*

Proof: Instead of terminating satisfied out in Figure 5-4 with a free-edge terminator, instead attach a latch gadget (Figure 5-6), with free-edge terminators on its loose red edges. Then, it's possible to reach the initial state modified so that only the latch state is reversed just when it's possible to reverse satisfied out: first reverse satisfied out (by solving the QBF problem), unlocking the latch; then reverse the latch state; then undo all the moves that reversed satisfied out. □

Chapter 6

Two-Player Games

With two-player games, we are finally in territory familiar to classical game theory and Combinatorial Game Theory. Two-player, perfect-information games are also the richest source of existing hardness results for games. In a two-player game, players alternate making moves, each trying to achieve some particular objective. The standard decision question is "does player X have a forced win from this position?".

The earliest hardness results for two-player games were PSPACE-completeness results for bounded games, beginning with Generalized Hex [22], and continuing with several two-player versions of known NP-complete problems [72]. Later, when the notion of alternating computation was developed [8], there were tools to show unbounded two-player games EXPTIME-complete. Chess [26], Go [68], and Checkers [70] then fell in quick succession to these techniques.

The connection between two-player games and computation is quite manifest. Just as adding the concept of nondeterminism to deterministic computation creates a new useful model of computation, adding an extra degree of nondeterminism leads to the concept of *alternating nondeterminism*, or *alternation*, [8] discussed in Appendix A. Indeed, up to this point it is clear that adding an extra degree of nondeterminism is like adding an extra player in a game, and seems to raise the computational complexity of the game, or the computational power of the model of computation. Unfortunately this process does not generalize in the obvious way: simply adding extra players beyond two does not alter the situation in any fundamental way, from a computational complexity standpoint. In later chapters we will find other ways to add computational power to games.

Alternation raises the complexity of bounded games from the one-player complexity of NP-complete to PSPACE-complete, and of unbounded games from the one-player complexity of PSPACE-complete to EXPTIME-complete [81]. Since it is not known whether P = NP or even PSPACE, with two-player games we finally reach games that are provably intractable: P $\neq$ EXPTIME [40]. In each case there is a natural game played on a Boolean formula which is complete for the appropriate class. For bounded games the game is equivalent to the Quantified Boolean Formulas problem: the "existential" and "universal" players take turns choosing assignments of successive variables. The unbounded games are similar, except that variable assignments can be changed back and forth multiple times.

Constraint Logic. The two-player version of Constraint Logic, Two-Player Constraint Logic (2CL), is defined as might be expected. To create different moves for the two players, Black and White, we label each constraint graph edge as either Black or White. (This is independent of the red/blue coloration, which is simply a shorthand for edge weight.) Black (White) is allowed to reverse only Black (White) edges on his move. Each player has a target edge he is trying to reverse.

6.1 Bounded Games

Bounded two-player games are games in which there is a polynomial bound (typically linear) on the number of moves that can be made. As with bounded puzzles, usually there is some resource that is used up. In Hex, for example, each move fills a space on the board, and when all the spaces are full, the game must be over. Similarly, in Amazons, on each move an amazon must remove one of the spaces from the board. In Konane, each move removes at least one stone. Et cetera. When the resource is exhausted, the game cannot continue.

Deciding such games can be no harder than PSPACE, because a Turing machine using space polynomial in the board size can perform a depth-first search of the entire game tree, determining the winner. In general these games are also PSPACE-hard. The canonical PSPACE-complete game is simply Quantified Boolean Formulas (QBF). The question "does there exist an x, such that for all y, there exists a z, such that ... formula F is true" is equivalent to the question of whether the first player can win the following formula game: "Players take turn assigning truth values to a sequence of variables. When they are finished, player one wins if formula F is true; otherwise, player two wins."

Bounded Two-Player Constraint Logic is the natural form of Constraint Logic that corresponds to this type of game. It is formally defined as follows:

BOUNDED TWO-PLAYER CONSTRAINT LOGIC (BOUNDED 2CL)
INSTANCE: AND/OR constraint graph G, partition of the edges of G into sets B and W, and edges $e_B \in B$, $e_W \in W$.
QUESTION: Does White have a forced win in the following game? Players White and Black alternately make moves on G, or pass. White (Black) may only reverse edges in W (B). Each edge may be reversed at most once. White (Black) wins (and the game ends) if he ever reverses e_W (e_B).

One remark about this definition is in order. In Combinatorial Game Theory, it is normal to define the loser as the first player unable to move. Games are thus about maximizing one's number of available moves. This definition would work perfectly well for 2CL, rather than using target edges to determine the winner; the hardness reduction would not be substantially altered, and the definition would seem to be a bit more concise. However, the definition above is more consistent with the other varieties of Constraint Logic. Always, the goal is to reverse a given edge.

6.1.1 PSPACE-completeness

Schaefer [72] showed that many variants of the basic QBF problem are also PSPACE-complete. It will be most convenient to reduce one of these variants to Bounded 2CL to show PSPACE-hardness, rather than reducing directly from the standard form of QBF:

G_{pos}(POS CNF)
INSTANCE: Monotone CNF formula A (that is, a CNF formula in which there are no negated variables).
QUESTION: Does Player I have a forced win in the following game? Players I and II alternate choosing some variable of A which has not yet been chosen. The game ends after all variables of A have been chosen. Player I wins in and only if A is true when all variables chosen by Player I are set to true and all variables chosen by II are set to false.

The reduction from G_{pos}(POS CNF) to Bounded 2CL is similar to the reduction from SAT to Bounded NCL in Section 5.1. There, the single player is allowed to choose a variable assignment via a set of CHOICE vertices. All we need do to adapt this reduction is replace the CHOICE vertices with *variable* vertices, such that if White plays first in a variable vertex the variable is true, and if Black plays first the variable is false. Then, we attach White's variable vertex outputs to the CNF formula inputs as before; Black's variable outputs are unused. The CNF formula consists entirely of White edges. Black is given enough extra edges to ensure that he will not run out of moves before White. White's target edge is the formula output, and Black's is an arbitrary edge that is arranged to never be reversible. A sample game graph corresponding to a formula game is shown in Figure 6-1; compare to Figure 5-1. (The extra Black edges are not shown.) Note that the edges are shown filled with the color that controls them.

The game breaks down into two phases. In the first phase, players alternate playing in variable vertices, until all have been played in. Then, White will win if he has chosen a set of variables satisfying the formula. Since the formula is monotone, it is exactly the variables assigned to be true, that is, the ones White chose, that determine whether the formula is satisfied. Black's play is irrelevant after this.

Theorem 19 *Bounded 2CL is PSPACE-complete.*

Proof: Reduction from G_{pos}(POS CNF), as described above. If Player I can win the formula game, then White can win the corresponding Bounded 2CL game, by playing the formula game on the edges, and then reversing the necessary remaining edges to reach the target edge. If Player I can't win the formula game, then White can't play so as to make a set of variables true which will satisfy the formula, and thus he can't reverse the target edge. Neither player can benefit from playing outside the variable vertices until all variables have been selected, because this can only allow the opponent to select an extra variable.

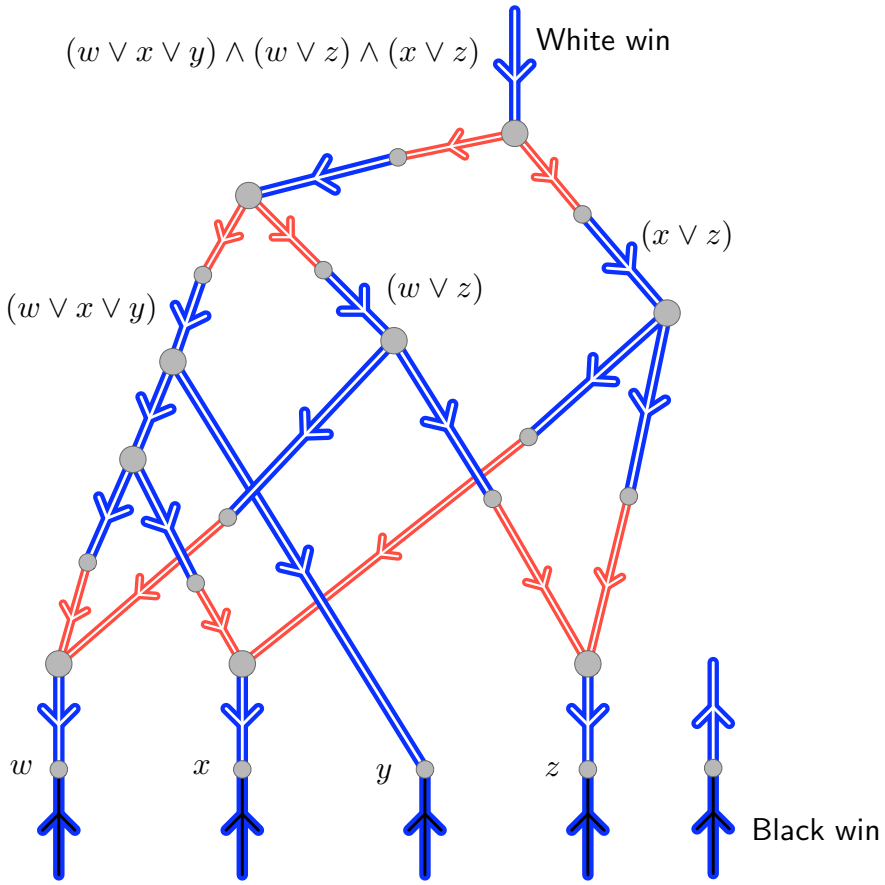
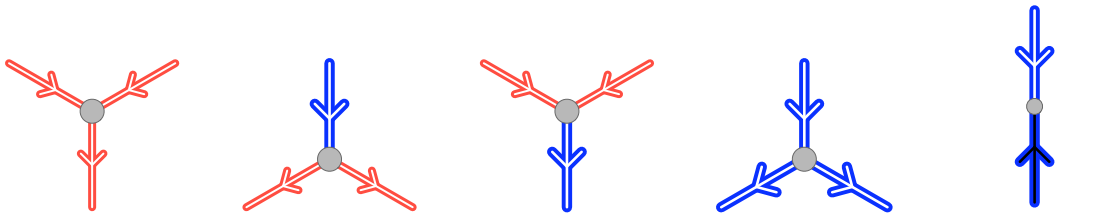


Figure 6-1: A constraint graph corresponding to the G_{pos}(POS CNF) formula game $(w \vee x \vee y) \wedge (w \vee z) \wedge (x \vee z)$. Edges corresponding to variables, clauses, and the entire formula are labeled.



(a) CHOICE (b) AND (c) FANOUT (d) OR (e) Variable

Figure 6-2: A sufficient set of vertex types for reductions from Bounded 2CL.

The construction shown in Figure 6-1 is not an AND/OR graph; we must still convert it to an equivalent one. The standard conversion techniques from Section 3.2 work here to handle the red-blue vertices, blue-blue vertices, and free edges. The color of the newly-introduced edges is irrelevant.

This shows that Bounded 2CL is PSPACE-hard. It is also clearly in PSPACE: since the game can only last as many moves as there are edges, a simple depth-first traversal of the game tree suffices to determine the winner from any position. □

6.1.2 Planar Graphs

As was the case with one-player games, strengthening Theorem 19 to apply to planar graphs will make reductions from Bounded 2CL to actual games much more convenient. Indeed, the above reduction is almost trivial, and the true benefit of using Bounded 2CL for game reductions, rather than simply using one of the many QBF variants, is that when reducing from planar Bounded 2CL one does not have to build a crossover gadget. The bounded two-player games addressed in Part II have relatively straightforward reductions for this reason. The complexity of Amazons remained open for several years, despite some effort by the game complexity community.

Theorem 20 *Bounded 2CL is PSPACE-complete, even for planar graphs.*

Proof: The crossover gadget presented in Section 5.2.2 is again sufficient. Note that no Black edges ever cross; therefore, all crossovers are monochrome, and essentially one-player crossovers. Thus, Theorem 15 is directly applicable. □

6.1.3 An Alternate Vertex Set

As was the case for Bounded Nondeterministic Constraint Logic, allowing each edge to reverse only once breaks the symmetry of the AND vertex, and there are effectively three types of AND vertex that must be implemented when reducing from Bounded NCL to a target game. Additionally, there are several more possible kinds of vertex to consider, when differing color possibilities are taken into account. Again, it is more convenient to use a slightly different, smaller vertex set than all of the possibilities allowed in two-color, bounded AND/OR graphs. In addition to the vertex types used

in Section 5.1.3, we only need one vertex allowing player interaction, the variable vertex used in Section 6.1.1.

These are the vertex types I will use to reduce to bounded two-player games in Chapter 10.

Theorem 21 *Theorem 20 remains true when the input graph uses the vertex types in Figure 6-2, rather than a complete set of directional AND/OR vertices.*

Proof: Apart from variable vertices, the reduction in Section 6.1.1 uses the same types of vertex as that in Section 5.1.1. □

6.2 Unbounded Games

Unbounded two-player games are games in which there is no restriction on the number of moves that can be made. Typically (but not always) the moves are reversible. Examples include the classic games Chess, Checkers, and Go. Some moves in each of these games are not reversible: pawn movement, castling, capturing, and promoting, in Chess; normal piece movement, capturing, and kinging in Checkers; and, actually, every move in Go. Go is an interesting case, because at first sight it appears to be a bounded game: every move places a stone, and when the board is full the game is over. However, capturing removes stones from the board, and reopens the spaces they occupied. Each of these games is EXPTIME-complete [26, 70, 68].[1]

Indeed, there are "proofs" that Go is PSPACE-complete, including one by Papadimitriou [59, pages 462–469]. Papadimitriou does not make the mistake of thinking Go is a bounded game, however; instead, he considers a modified version which is bounded. In fact, Go's peculiarities, combined with the extreme simplicity of the rules, make it worthy of extra study from a complexity standpoint.

The removal of a polynomial bound on the length of the game means that it is no longer possible to perform a complete search of the game tree using polynomial space, so the PSPACE upper bound no longer applies. In general, two-player games of perfect information are EXPTIME-complete [81].

Two-Player Constraint Logic (2CL) is the form of Constraint Logic that corresponds to this type of game. It is formally defined as follows:

TWO-PLAYER CONSTRAINT LOGIC (2CL)
INSTANCE: AND/OR constraint graph G, partition of the edges of G into sets B and W, and edges $e_B \in B$, $e_W \in W$.
QUESTION: Does White have a forced win in the following game? Players White and Black alternately make moves on G. White (Black) may only reverse edges in W (B). White (Black) wins if he ever reverses e_W (e_B).

[1] For Go, the result is only for Japanese rules. See Section 6.3.

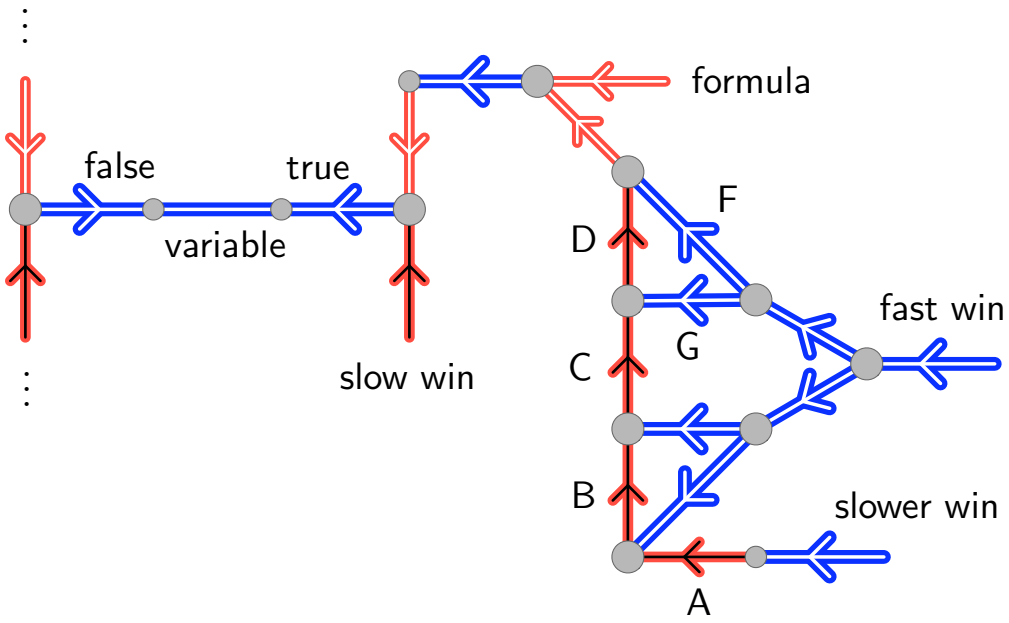


Figure 6-3: Reduction from G_6 to 2CL.

6.2.1 EXPTIME-completeness

To show that 2CL is EXPTIME-hard, we reduce from one of the several Boolean formula games shown EXPTIME-complete by Stockmeyer and Chandra [81]:

G_6
INSTANCE: CNF Boolean formula F in variables $X \cup Y$, $(X \cup Y)$ assignment α.
QUESTION: Does Player I have a forced win in the following game? Players I and II take turns. Player I (II) moves by changing at most one variable in X (Y); passing is allowed. Player I wins if F ever becomes true.

Note that there is no provision for Player II to ever win; the most he can hope to accomplish is a draw, by preventing Player I from ever winning. But this will not matter to us, because the relevant decision question for 2CL is simply whether White can force a win.

Reduction. The essential elements of the reduction from G_6 to 2CL are shown in Figure 6-3. This figure shows a White variable gadget and associated circuitry; a Black variable gadget is identical except that the edge marked variable is then black instead of white. The left side of the gadget is omitted; it is the same as the right side. The state of the variable depends on whether the variable edge is directed left or right, enabling White to reverse either the false or the true edge (and thus lock the variable edge into place).

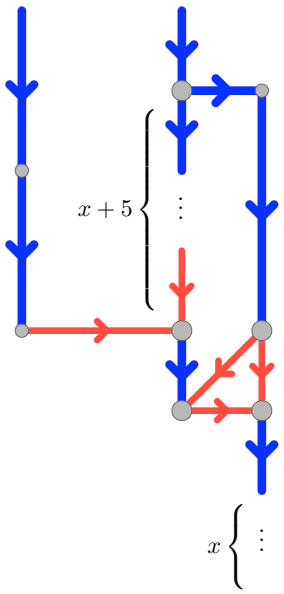


Figure 6-4: Path-length-equalizer gadget.

The basic idea of the reduction is the same as for Bounded 2CL: the players should play a formula game on the variables, and then if White can win the formula game, he can then reverse a sequence of edges leading into the formula, ending in his target edge. In this case, however, the reduction is not so straightforward, because the variables are not fixed once chosen; there is no natural mechanism in 2CL for transitioning from the variable-selection phase to the formula-satisfying phase. That is what the rest of the circuitry is for. (Also note that unlike the bounded case, the formula need not be monotone.)

White has the option, whenever he wishes, of *locking* any variable in its current state, without having to give up a turn, as follows. First, he moves on some true or false edge. This threatens to reach an edge F in four more moves, enabling White to reach a fast win pathway leading quickly to his target edge. Black's only way to prevent F from reversing is to first reverse D. But this would just enable White to immediately reverse G, reaching the target edge even sooner. First, Black must reverse A, then B, then C, and finally D; otherwise, White will be able to reverse one of the blue edges leading to the fast win. This sequence takes four moves. Therefore, Black must respond to White's true or false move with the corresponding A move, and then it is White's turn again.

The lengths of the pathways slow win, slower win, and fast win are detailed below in the proof. The pathways labeled formula feed into the formula vertices, culminating in White's target edge. It will be necessary to ensure that regardless of how the formula is satisfied, it always requires exactly the same number of edge reversals beginning with the formula input edges. The first step to achieving this is to note that the formula may be in CNF. Thus, every clause must have one variable satisfied, so it seems we are well on our way. However, there is a problem. Generally, a variable must

pass through an arbitrary number of FANOUTs on its way into the clauses. This means that if it takes x reversals from a given variable gadget to a usage of that variable in a clause, it will take less than $2x$ reversals to reach two uses of the variable, and we cannot know in advance how many variables will be re-used in different clauses. The solution to this problem is to use a path-length-equalizer gadget, shown in Figure 6-4. This gadget has the property that if it takes x reversals from some arbitrary starting point before entering the gadget, then it takes $x + 6$ reversals to reverse either of the topmost output edges, or $2x + 12$ reversals to reverse both of them. By using a chain of n such gadgets whenever a variable is used n times in the formula, we can ensure that it always takes the same number of moves to activate any variable instance in a clause, and thus that it always takes the same number of moves to activate the formula output.

Theorem 22 *2CL is EXPTIME-complete.*

Proof: Given an instance of G_6, we construct a corresponding constraint graph as described above.

Suppose White can win the formula game. Then, also suppose White plays to mimic the formula game, by reversing the corresponding variable edges up until reversing the last one that lets him win the formula game. Then Black must also play to mimic the formula game: his only other option is to reverse one of the edges A to D, but any of these lead to a White win.

Now, then, assume it is White's turn, and either he has already won the formula game, or he will win it with one more variable move. He proceeds to lock all the variables except possibly for the one remaining he needs to change, one by one. As described above, Black has no choice but to respond to each such move. Finally, White changes the remaining variable, if needed. Then, on succeeding turns, he proceeds to activate the needed pathways through the formula and on to the target edge. With all the variables locked, Black cannot interfere. Instead, Black can try to activate one of the slow win pathways enabled during variable locking. However, the path lengths are arranged such that it will take Black one move longer to win on such a pathway than it will take White to win by satisfying the formula.

Suppose instead that White cannot win the formula game. He can accomplish nothing by playing on variables forever; eventually, he must lock one. Black must reply to each lock. If White locks all the variables, then Black will win, because he can follow a slow win pathway to victory, but White cannot reach his target edge at the end of the formula, and Black's slow win pathway is faster than White's slower win pathway. However, White may try to cheat by locking all the Black variables, and then continuing to change his own variables. But in this case Black can still win, because if White takes the time to change more than one variable after locking any variable, Black's slow win pathway will be faster than White's formula activation.

Thus, White can win the 2CL game if and only if he can win the corresponding G_6 game, and 2CL is EXPTIME-hard. Also, 2CL is easily seen to be in EXPTIME: the complete game tree has exponentially many positions, and thus can be searched in exponential time, labeling each position as a win, loss, or draw depending on the

labels of its children. (A draw is possible when optimal play loops.) Therefore, 2CL is EXPTIME-complete. □

6.2.2 Planar Graphs

As before, 2CL remains EXPTIME-complete when the graph is planar. In principle, this should enable much simpler reductions to actual games. The existing Chess, Checkers, and Go hardness results are all quite complicated, and I had hoped to re-derive some of them more simply using 2CL. I have not done so yet, however. Enforcing the necessary constraints in a two-player game gadget is much more difficult than in a one-player game.

Theorem 23 *2CL is EXPTIME-complete, even for planar graphs.*

Proof: The crossover gadget presented in Section 5.2.2 is again sufficient. Note that no Black edges ever cross; therefore, all crossovers are monochrome, and essentially one-player crossovers. Thus, Theorem 15 is directly applicable. It is possible that introduction of crossover gadgets can change some path lengths from variables through the formula; however, it is easy to pad all the variable pathways to correct for this, because it takes a fixed number of reversals to traverse a crossover gadget in each direction. □

6.3 No-Repeat Games

One interesting result deserves to be mentioned here. Robson [69] shows that if the condition that no previous position may ever be recreated is added to two-player games, then the general complexity rises from EXPTIME-complete to EXPSPACE-complete. The intuition is that it requires an exponential amount of space even to determine the legal moves from a position, because the game history leading up to a position could be exponentially long.

In fact, some of the EXPTIME-complete formula games in [81] automatically become EXPSPACE-complete when this modification is made, and as a result, no-repeat versions of Chess and Checkers are EXPSPACE-complete. However, the result does not apply to the game G_6, which was used to show 2CL EXPTIME-complete; nor does it apply to Go, which was shown EXPTIME-hard by a reduction from G_6. In fact, Go is actually played with this rule in many places; in Go it is called the "superko" rule. The complexity of Go with superko is an interesting problem which is still unresolved. Actually, both the lower and the upper bounds of Robson's EXPTIME-completeness proof [68] break when the superko rule is added. Superko is not used in Japan; it is used in the U.S., China, and other places.

It is arguably a bit unnatural in a game for the set of legal moves to not be determinable from the current position. Of course, if the position is defined to include the game history then this is not a problem, but then the position grows over time, which is also against the spirit of generalized combinatorial games.

There is a tantalizing connection here to a kind of imperfect information, which is also connected to the idea of an additional player. A useful perspective is that in a two-person game, there is an "existential" player and a "universal" player. No-repeat games are almost like two-person games with an extra, "super-universal" player added. This player can remember, secretly, one game position. Then, if that position is ever recreated, whoever recreated it loses. In principle, this approach seems capable of resolving the above problems. All ordinary moves are legal, whether they are repeating or not, but in actual play repeating moves are losing because the super-universal player can nondeterministically guess in advance which position will be repeated. However, this idea seems difficult to formalize usefully; in particular, it is not clear how to formulate an appropriate decision question so that the super-universal player doesn't effectively team up with the universal player and against the existential one. But this seems an interesting path for further exploration.

Notwithstanding the above concerns, a no-repeat version of two-player Constraint Logic ought to be EXPSPACE-complete. A reduction from game G_3 from [81], for example, would do the trick, but I do not have one yet.

Chapter 7

Team Games

It turns out that adding players beyond two to a game does not increase the complexity of the standard decision question, "does player X have a forced win?". We might as well assume that all the other players team up to beat X, in which case we effectively have a two-player game again. If we generalize the notion of the decision question somewhat, we do obtain new kinds of games. In a *team game*, there are still two "sides", but each side can have multiple players, and the decision question is whether *team* X has a forced win. A team wins if any of its players wins.

Team games with perfect information are still just two-player games in disguise, however, because again all the players on a team can cooperate and play as if they were a single player. However, when there is not perfect information, then team games turn out to be different from two-player games.[1] We could think of a team in this case as a player with a peculiar kind of mental limitation—on alternate turns he forgets some aspects of his situation, and remembers others.

Therefore, we will only consider team games of imperfect information in this chapter, and I will sometimes simply refer to them simply as "team games". Such games, as with two-player imperfect-information games, were first studied from a complexity standpoint by Peterson and Reif [61]. The general result is that bounded team games are NEXPTIME-complete, and unbounded games are undecidable. However, there are several technical problems with the original undecidability result; in fact, the game claimed undecidable, called TEAM-PEEK, is trivially decidable in at least one instance implicitly claimed undecidable. I show how to fix the problems, confirming that indeed team games of imperfect information are undecidable.

The fact that there are undecidable games using bounded space—when actually played, finite physical resources—at first seems counterintuitive and bizarre. There are only finitely many configurations in such a game. Eventually, the position must

[1] Adding imperfect information to a two-player, unbounded game does create a new kind of game, intermediate in complexity between two-player perfect-information games and team games with imperfect information [64]; such games can be 2EXPTIME-complete (complete in doubly-exponential time) to decide. There was not time to develop the natural version of Constraint Logic for this class of games in this thesis, but I have every expectation that it will be 2EXPTIME-complete. [64], [61], and [60] also introduce "blindfold" and "hierarchical" games, which correspond to yet more complexity classes.

repeat. Yet, somehow the state of the game must effectively encode the contents of an unboundedly long Turing-machine tape! How can this be? These issues are discussed in Chapter 8.

Constraint Logic. The natural team, private-information version of Constraint Logic assigns to each player a set of edges he can control, and a set of edges whose orientation he can see. As always, each player has a target edge he must reverse to win. To enable a simpler reduction to the unbounded form of team Constraint Logic, I allow each player to reverse up to some given constant k edges on his turn, rather than just one, and leave the case of $k = 1$ as an open problem.

7.1 Bounded Games

Bounded team games of imperfect information include card games such as Bridge. Here we can consider one hand to be a game, with the goal being to either make the bid, or, if on defense, to set the other team. Focusing on a given hand also removes the random element from the game, making it potentially suitable for study within the present framework. However, it might still be difficult to formulate appropriately. Bounded Team Private Constraint Logic (Bounded TPCL) starts from a configuration known to all players; the private information arises as a result of some moves not being visible to all players. These attributes do not apply to Bridge directly, but some sort of reduction may be possible.

Peterson and Reif [61] showed that bounded team games of private information are NEXPTIME-complete in general, by a reduction from Dependency Quantified Boolean Formulas (DQBF).

BOUNDED TEAM PRIVATE CONSTRAINT LOGIC (BOUNDED TPCL)

INSTANCE: AND/OR constraint graph G; integer N; for $i \in \{1 \ldots N\}$: sets $E_i \subset V_i \subset G$; edges $e_i \in E_i$; partition of $\{1 \ldots N\}$ into nonempty sets W and B.
QUESTION: Does White have a forced win in the following game? Players $1 \ldots N$ take turns in that order. Player i only sees the orientation of the edges in V_i, and moves by reversing an edge in E_i which has not previously reversed; a move must be known legal based on V_i. White (Black) wins if Player $i \in W$ (B) ever reverses edge e_i.

Conjecture 1 *Bounded TPCL is NEXPTIME-complete.*

7.2 Unbounded Games

In general, team games of private information are undecidable. This result was claimed by Peterson and Reif in 1979 [61]. However, as mentioned above, there are several problems with the proof, which I address in Section 7.2.1. Strangely, the

result also seems to be not very well known. I worked in the area of game complexity for several years, in collaboration with experts in the field, before stumbling across it. Part of the problem may be that the authors seem to consider the result of secondary importance to the other results in [61]. Indeed, immediately after showing their particular game undecidable, the authors remark

> *In order to eliminate, by restriction, the over-generality of MPA_k-TMs, we considered several interesting variants.* [61, page 355]

The rest of the paper then considers those variants. From my perspective, however, the fact that there are undecidable space-bounded games is fundamental to the viewpoint that games are an interesting model of computation. It both shows that games are as powerful as general Turing machines, and highlights the essential difference from the Turing-machine foundation of theoretical computer science, namely that *a game computation is a manipulation of finite resources.* Thus, this seems to be a result of some significance.

It might seem that the concept of an unbounded-length team game of private information is getting rather far from the intuitive notion of game. However, individually each of these attributes is common in games. There is at least one actual game that fits this category, called Rengo Kriegspiel. This is a team, blindfold version of Go. (See Chapter 11 for details.) I have personally played this game on a few occasions, and it is intriguing to think that it's possible I have played the hardest game in the world, which cannot even in principle be played perfectly by any algorithm. Again, the important difference here from other undecidable problems, such as the Post Correspondence Problem (PCP), is that Rengo Kriegspiel is a game with bounded space; there are a fixed number of positions in any given game. Thus, the game can *actually* be played; PCP, though a puzzle of a sort, cannot be played in the real world without infinite resources. This theme will be developed further in Chapter 8.

Team, private Constraint Logic is defined as follows. Note the addition of the parameter k relative to the bounded case. This is, an admittedly, an extra generalization to make a reduction easier; nonetheless, it is a reasonable generalization, and all other Constraint Logic games in this thesis are naturally restricted versions of this game.

TEAM PRIVATE CONSTRAINT LOGIC (TPCL)
INSTANCE: AND/OR constraint graph G; integer N; for $i \in \{1 \ldots N\}$: sets $E_i \subset V_i \subset G$; edges $e_i \in E_i$; partition of $1 \ldots N$ into nonempty sets W and B; integer k.
QUESTION: Does White have a forced win in the following game? Players $1 \ldots N$ take turns in that order. Player i only sees the orientation of the edges in V_i, and moves by reversing up to k edges in E_i; a move must be known legal based on V_i. White (Black) wins if Player $i \in W$ (B) ever reverses edge e_i.

Before showing this game undecidable, I discuss the earlier results of Peterson and Reif [61].

7.2.1 Incorrectness of Existing Results

Peterson and Reif [61] claim that a particular space-bounded game with alternating turns, TEAM-PEEK, is undecidable. There are several problems with this claim. First, the game as defined is trivially decidable in at least one case implicitly claimed undecidable.

TEAM-PEEK. TEAM-PEEK [61] is a generalization of Stockmeyer and Chandra's game Peek, which is EXPTIME-complete [81]. Peek is played with a box containing horizontal plates which slide in and out. Each plate has holes drilled in particular locations, as do the top and bottom of the box. The two players stand on opposite sides of the box. Each player controls a subset of the plates; his plates have handles attached on his side. On a player's turn he may slide one plate to one of two positions, either fully in or partially out. The game ends when a hole is opened up from the top of the box to the bottom, through the stack of plates, and the winner is the last player to move. This is a physical description of what is actually a game played on variables of a Boolean formula in DNF (in [81], called G_4). The connection is that the variables are represented by plates, and the disjunctive clauses by holes in the top and bottom of the box.

TEAM-PEEK adds the concepts of teams and private information. There are two teams, one on each side of the box. Each player controls a subset of the plates with handles on his team's side. Some plates' positions are private to their controlling player; no other player can tell which position those plates are in. Another change from Peek—a critical one, as it turns out—is that on his turn a player may move any subset of his plates in or out, and not just one. Again, this is merely a physical way of describing a game played on variables of a Boolean formula in DNF.

No proof is offered that TEAM-PEEK is undecidable. Instead, what is said is the following [61, page 356]:

> *We now cite the obvious complexities of the outcome problem for these versions of PEEK. (All are based on the plates representing clauses of formulas, and so on. Our results follow simply from [8].) ... TEAM-PEEK is undecidable with two or more players on team A.*

In a later version of the paper [60], a proof is given for the complexities of a variety of PEEK games. This version of TEAM-PEEK is not specifically among them. However, the original paper is referred to [60, page 982]:

> *Peterson and Reif [61] use a TEAM-PRIVATE-PEEK game which is not hierarchical to prove the following undecidability result.*
>
> *THEOREM 4.3.3. (See [61].) In general, a TEAM-PRIVATE-PEEK [game] can be undecidable with two or more players on Team T_1.*

Note that in [60], the original TEAM-PEEK is called "non-hierarchical TEAM-PRIVATE-PEEK". A game is considered *hierarchical* if the team of preference (the team the decision question refers to) is structured so that all information visible to Player i on that team is also visible to Player $i-1$.

The best indication of what the "obvious" reduction should look like for the original TEAM-PEEK is the reduction used here for hierarchical versions of TEAM-PRIVATE-PEEK. In fact, the structure of the proof is such that merely removing the restriction that the game be hierarchical yields, formally, a "proof" that TEAM-PEEK is undecidable. In that proof, the assumption is made that all players on the team of preference (the "existential" team) play first, and the single player on the the other ("universal") team plays next.

So in that case, following the logic of [60], TEAM-PEEK ought to be undecidable. But at least in that particular case, TEAM-PEEK is not only decidable; it is in NP. The initial state of the plates is known to all players. Suppose that the existential team can win on their first set of turns, before the universal player moves. This is easily determinable in nondeterministic polynomial time. Then, of course, they would do so. Suppose instead they cannot win on their first set of turns. But then they can never win, because the universal player will simply leave his plates where they are, declining to move any of them, and the existential players will be faced with the same situation on their next set of turns. Therefore, whether the existential team can win can be decided in NP.

So where is the problem? There are multiple problems. The immediate problem here is actually a simple mismatch between what was proven and what was claimed, not only for TEAM-PEEK, but for all the varieties of PEEK considered in [60]. In the original, two-player, perfect-information version of Peek, the players may move up to one plate on their turn. Peterson and Reif changed this rule to allow moving an arbitrary subset of the plates, and this is what leads to the trivial solution above; note that it would no longer work if at most one plate could be moved per turn. And, in fact, the formula games that [60] derive results for actually allow each player to change at most one variable per turn! Then the statement is made that these formula games are in direct correspondence with the previously-defined TEAM-PEEK games. But all those games allow players to move an arbitrary subset of plates.

Round-Robin Play. If this were the only mistake, it would be a trivial, if unfortunate, error in definition; perhaps the games suitably modified have the claimed complexities? However, there are other problems. In particular, in the formula game reduction, which is from multiplayer alternating Turing machines of various types to corresponding formula games, the assumption is made that that the players take turns in a given order every round. This assumption is not supported. And in fact, the corresponding property does not hold of the multiplayer alternating Turing machines, so there is a missing reduction which is not obvious.

Such machines are set up in correspondence with a general notion of multiplayer game. Part of their description of such games is as follows: [61, pages 349–350]

We take a very general view of games of incomplete information so as to include as many variants as possible. Some important facts about these games are listed below.

1. *Players need not take turns in a round-robin fashion. The rules will dictate whose turn is next.*

   ...

5. *A player may not know how many turns were taken by other players between its turns.*

These properties are reflected in the behavior of the "MPA$_k$" Turing machines used to support the claim in [61] that TEAM-PEEK is undecidable, and used in the formula game reduction in [60]. In those machines, each state includes information about which "player" is to play next; the other players are not aware of what happens between their turns.

Reif has confirmed in a personal communication regarding round-robin play in TEAM-PEEK [63] that "it looks like therefore the players do not play round robin". In fact it is not even clear how to define a version of TEAM-PEEK without round-robin play.

Indeed, the intention to include as many variants as possible in the notion of game is laudable, but by so broadening the notion of game the actual results are correspondingly weakened. In "normal" games, players *do* take turns in order, and *are* aware of what goes on between their turns. Such games are *excluded* from, rather than included in, the domain of applicability of Peterson and Reif's team game results.

Let us examine the undecidable game used to show MPA$_k$ machines undecidable for bounded space. (It is the fact that these machines are undecidable for bounded space which is the origin of the "obvious" result that TEAM-PEEK is undecidable.)

The version of this game given in [60] is as follows:

Given a D-TM M ... The game will be based on having each of the $\exists$-players find a sequence of configurations of the D-TM M which on an input ω lead to acceptance. Accordingly, upon request each $\exists$-player will give the $\forall$-player the next character of its sequence of configurations. Each $\exists$-player does this secretly from the other $\exists$-player. The configuration will be of the form $\#C_0\#C_1\#\ldots\#C_m\#$, where C_0 is the initial configuration of M on the input, and C_m is an accepting configuration of M.

The $\forall$-player will choose to verify the sequences in one of the following ways.

1. *Check one of the first configurations against the input, and ensure that it is in the correct form. Check the last configuration for accepting state. This implies that the input tape is private to the $\forall$-player.*
2. *Check the last configuration for accepting state.*
3. *Check that the $\exists$-players are giving the same sequence by alternating turns between them.*

4. *Run one of the players ahead of the next #, and check its C_i against the other $\exists$-player's C_{i-1} for proper head motion, state change, and tape cells changes, in correspondence with the transition rules, etc.*

The key to the game is that the universal player only needs a fixed amount of memory to check that the computation histories are valid. With only one existential player, the universal player would have to remember an entire M configuration to validate the incoming characters, but with the ability to nondeterministically run one existential player ahead, and check against the other existential player, this is not necessary.

As the game is described, it is clear that the existential team can guarantee a win if and only if M accepts ω. If either existential player deviates from producing an accepting computation history, then the universal player can nondeterministically detect the deviation, and win. It is implicit in the definition of the game, however, that the universal player chooses, on each of his turns, which existential player is to play next, and the other existential player cannot know how many turns have elapsed before he gets to play again.

For suppose instead that play does go round robin. Then we must assume that on the universal player's turn, he announces which existential player is to make a computation-history move this turn; the other one effectively passes on his turn. But then each existential player knows exactly where in the computation history the other one is, and whichever player is behind knows he cannot be checked for validity, and is at liberty to generate a bogus computation history. It is the very information about how many turns the other existential player has had that must be kept private for the game to work properly.

Therefore, it seems that there is no reasonable support in either [61] or [60] for the claim that TEAM-PEEK, or any bounded game where the players take turns in sequence, is undecidable. And in fact the missing step in the formula game proofs in [60], justifying the assumption that play goes round robin, calls into question all of those results, and not merely TEAM-PEEK undecidability.

7.2.2 Undecidability

To solve the above problems, I introduce a slightly more elaborate computation game, in which the players take successive turns, and which I show to be undecidable. I reduce this game to a formula game, and the formula game to TPCL. I am rather explicit with this chain of reductions, so as to avoid the subtle errors apparent in [60].

The new computation game will be similar to the above game, but each existential player will be required to produce successive symbols from two identical, independent computation histories, A and B; on each turn, the universal player will select which history each player should produce a symbol from, privately from the other player. Then, for any game history experienced by each existential player, it is always possible that his symbols are being checked for validity against the other player's, because

one of the other existential player's histories could always be retarded by one configuration. The fact that the other player has produced the same number of symbols as the current player does not give him any useful information, because he does not know the relative advancement of the other player's two histories.

TEAM COMPUTATION GAME
INSTANCE: Finite set of $\exists$-*options* O, Turing machine S with fixed tape length k, and with tape symbols $\Gamma \supset (O \cup \{\mathsf{A}, \mathsf{B}\})$.
QUESTION: Does the existential team have a forced win in the following game? Players $\forall$ (universal), $\exists_1$, and $\exists_2$ (existential) take turns in that order, beginning with $\forall$. S's tape is initially set empty. On $\exists_i$'s turn, he makes a move from O. On $\forall$'s turn, he takes the following steps:

1. If not the first turn, record $\exists_1$'s and $\exists_2$'s moves in particular reserved cells of S's tape.

2. Simulate S using its current tape state as input, either until it terminates, or for k steps. If S accepts, $\forall$ wins the game. If S rejects, $\forall$ loses the game. Otherwise, leave the current contents of the tape as the next turn's input.

3. Make a move $(x, y) \in \{\mathsf{A}, \mathsf{B}\}\times\{\mathsf{A}, \mathsf{B}\}$, and record this move in particular reserved cells of S's tape.

The state of S's tape is always private to $\forall$. Also, $\exists_1$ sees only the value of x, and $\exists_2$ sees only the value of y. The existential players also do not see each other's moves. The existential team wins if either existential player wins.

Theorem 24 *TEAM COMPUTATION GAME is undecidable.*

Proof: We reduce from acceptance of a Turing machine on an empty input, which is undecidable. Given a TM M, we construct TM S as above so that when it is run, it verifies that the moves from the existential players form valid computation histories, with each successive character following in the selected history, A or B. It needs no nondeterminism to do this; all the necessary nondeterminism by $\forall$ is in the moves (x, y). The $\exists$-options O are the tape alphabet of $M \cup \#$.

S maintains several state variables on its tape that are re-used the next time it is run. First, it detects when both existential players are simultaneously beginning new configurations (by making move #), for each of the four history pairs $\{\mathsf{A}, \mathsf{B}\}\times\{\mathsf{A}, \mathsf{B}\}$. Using this information, it maintains state that keeps track of when the configurations match. Configurations partially match for a history pair when either both are beginning new configurations, or both partially matched on the previous time step, and both histories just produced the same symbol. Configurations exactly match when they partially matched on the previous time step and both histories just began new configurations (with #).

S also keeps track of whether one existential player has had one of its histories advanced exactly one configuration relative to one of the other player's histories.[2] It does this by remembering that two configurations exactly matched, and since then only one history of the pair has advanced, until finally it produced a #. If one history in a history pair is advanced exactly one configuration, then this state continues as long as each history in the pair is advanced on the same turn. In this state, the histories may be checked against each other, to verify proper head motion, change of state, etc., by only remembering (on preserved tape cells) a finite number of characters from each history. S is designed to reject whenever this check fails, or whenever two histories exactly match and nonmatching characters are generated, and to accept when one computation history completes a configuration which is accepting for M. All of these computations may be performed in a constant number of steps; we use this number for k.

For any game history of A/B requests seen by $\exists_1$ ($\exists_2$), there is always some possible history of requests seen by $\exists_2$ ($\exists_1$) such that either $\exists_1$ ($\exists_2$) is on the first configuration (which must be empty), or $\exists_2$ ($\exists_1$) may have one of its histories exactly one configuration behind the currently requested history. Therefore, correct histories must always be generated to avoid losing.[3] Also, if correct accepting histories are generated, then the existential team will win, and thus the existential team can guarantee a win if and only if M accepts the empty string. □

Next I define a team game played on Boolean formulas, and reduce TEAM COMPUTATION GAME to this formula game. Traditionally one defines a formula game in a form for which it is easy to prove a hardness result, then reduces to another formula game with a cleaner definition and nicer properties. In this case, however, our formula game will only serve as an intermediate step on the way to a Constraint Logic game, so no effort is made to define the simplest possible team formula game. On the contrary, the structure of the game is chosen to as to enable the simplest possible reduction to a Constraint Logic game.

The reduction from TEAM COMPUTATION GAME works by creating formulas that simulate the steps of Turing machine S.

TEAM FORMULA GAME
INSTANCE: Sets of Boolean variables X, X', Y_1, Y_2; Boolean variables $h_1, h_2 \in X$; and Boolean formulas $F(X, X', Y_1, Y_2)$, $F'(X, X')$, and $G(X)$, where F implies $\overline{F'}$.

[2]The number of steps into the history does not have to be exactly one configuration ahead; because M is deterministic, if the configurations exactly matched then one can be used to check the other's successor.

[3]Note that this fact depends on the nondeterminism of $\forall$ on each move. If instead $\forall$ followed a strategy of always advancing the same history pair, until it nondeterministically decided to check one against the other by switching histories on one side, the existential players could again gain information enabling them to cheat. This is a further difference from the original computation game from [61], where such a strategy is used; the key here is that $\forall$ is always able to detect when the histories happen to be nondeterministically aligned, and does not have to arrange for them to be aligned in advance by some strategy that the existential players could potentially take advantage of.

QUESTION: Does White have a forced win in the following game? The steps taken on each turn repeat in the following order:

1. B sets variables X to any values. If F and G are then true, Black wins.
2. If F is false, White wins. Otherwise, W_1 does nothing.
3. W_2 does nothing.
4. B sets variables X' to any values.
5. If F' is false, White wins. W_1 sets variables Y_1 to any values.
6. W_2 sets variables Y_2 to any values.

B sees the state of all the variables; W_i only sees the state of variables Y_i and h_i.

Theorem 25 *TEAM FORMULA GAME is undecidable.*

Proof: Given an instance of TEAM COMPUTATION GAME, we create the necessary variables and formulas as follows.

F will verify that B has effectively run TM S for k steps, by setting X to correspond to a valid non-rejecting computation history for it. (This can be done straightforwardly with $O(k^2)$ variables; see, for example, [10].) F also verifies that the values of Y_i are equal to particular variables in X, and that a set of "input" variables $I \subset X$ are equal to corresponding variables X'. X' thus represents the output of the previous run of S.

G is true when the copies of the Y_i in X represent an illegal white move (see below), or when X corresponds to an accepting computation history for S.

F' is true when the values X' equal those of a set of "output" variables $O \subset X$. These include variables representing the output of the run of S, and also h_1, h_2. We can assume without loss of generality here that S always changes its tape on a run. (We can easily create additional tape cells and states in S to ensure this if necessary, without affecting the simulation.) As a result, F implies $\overline{F'}$, as required; the values of X' cannot simultaneously equal those of the input and the output variables in X.

$\forall$'s move $(x, y) \in \{\mathsf{A}, \mathsf{B}\} \times \{\mathsf{A}, \mathsf{B}\}$ is represented by the assignments to history-selecting variables h_1 and h_2; false represents A and true B. The $\exists$-options O correspond to the Y_i; each element of O has one variable in Y_i, so that W_i must move by setting one of the Y_i to true and the rest to false.

Then, it is clear that the rules of TEAM FORMULA GAME force the players effectively to play the given TEAM COMPUTATION GAME. $\square$

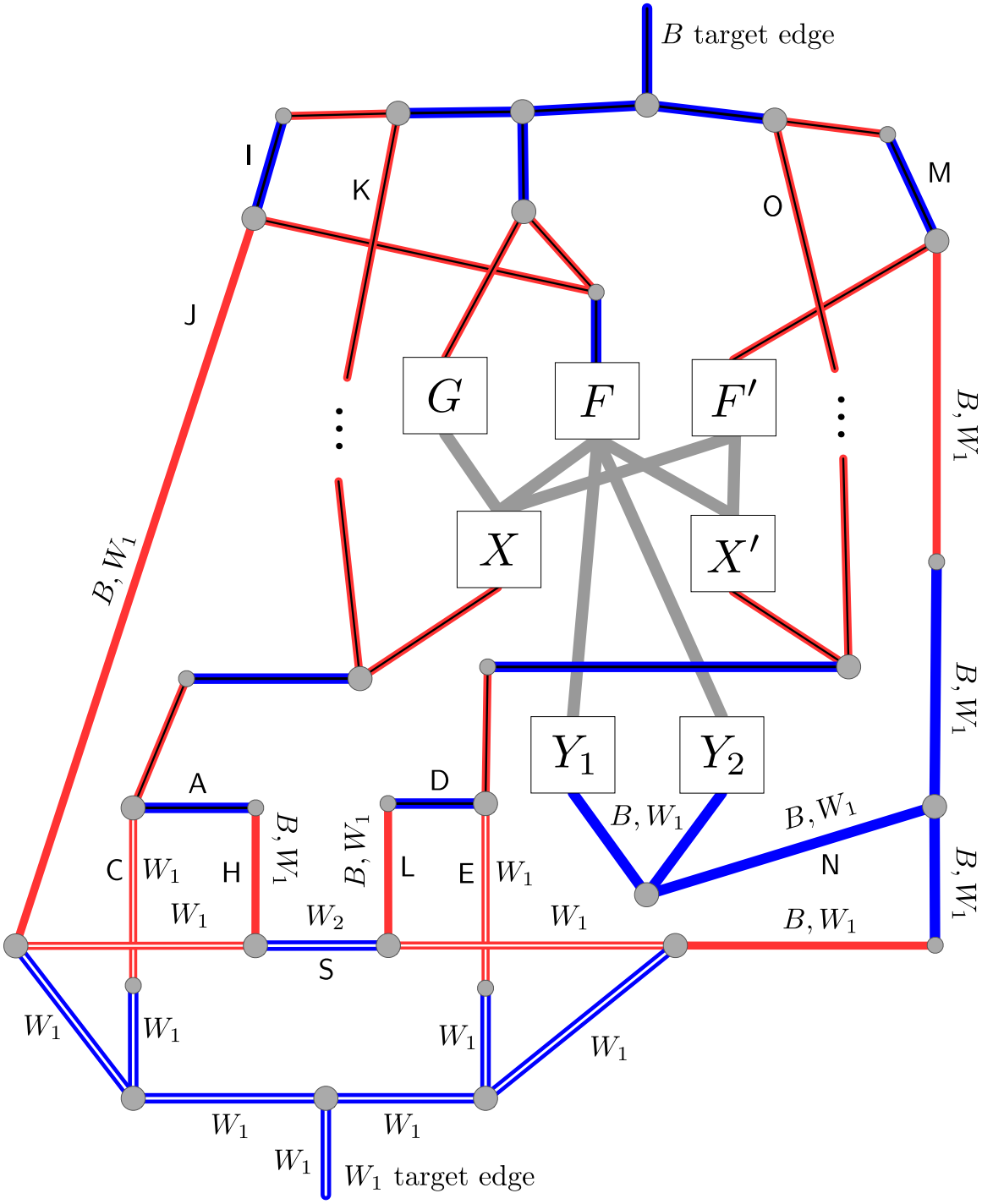
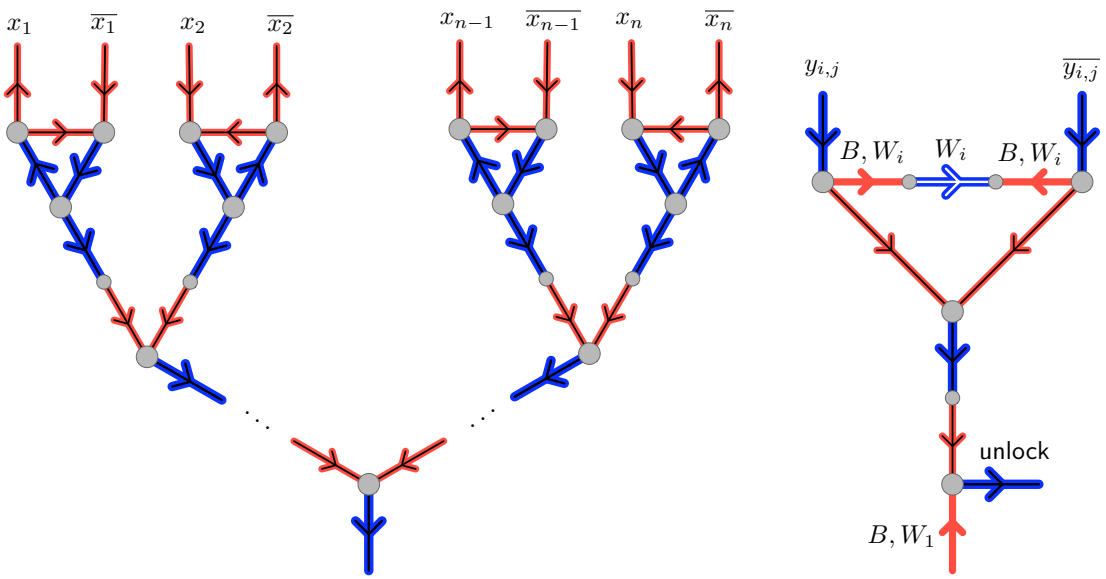


Figure 7-1: Reduction from TEAM FORMULA GAME to TPCL. White edges and multiplayer edges are labeled with their controlling player(s); all other edges are black. Thick gray lines represent bundles of black edges.



(a) Latches locking black variables

(b) White variable

Figure 7-2: Additional gadgets for TPCL reduction.

TPCL Reduction. Finally, we are ready to complete the undecidability reduction for TPCL. The overall reduction from TEAM FORMULA GAME is shown in Figure 7-1. Before proving its correctness, we first examine the subcomponents represented by boxes in the figure.

The F, F', and G boxes represent AND/OR subgraphs that implement the corresponding Boolean functions, as in earlier chapters. Their inputs come from outputs of the variable-set boxes. All these edges are black.

The boxes X and X' represent the corresponding variable sets. The incoming edge at the bottom of each box unlocks their values, by a series of latch gadgets (as in Section 5.2.1), shown in Figure 7-2(a). When the input edge is directed upward, the variable assignment may be freely changed; when it is directed down, the assignment is fixed.

The boxes Y_1 and Y_2 represent the white variables. An individual white variable for Player W_i is shown in Figure 7-2(b). B may activate the appropriate top output edge at any time; however, doing so also enables the bottom output edge controlled jointly by B and W_1. If B wants to prevent W_1 from directing this edge down, he must direct unlock right; but then the black output edges are forced down, allowing W_i to freely change the central variable edge. The unlock edges are left loose (shorthand for using a free edge terminator); the bottom edges are ORed together to form the single output edge for each box in Figure 7-1 (still jointly controlled by B and W_1). Note that for variables controlled by W_2, W_1 can know whether the variable is unlocked without knowing what its assignment is.

We will also consider some properties of the "switch" edge S before delving into the proof. This edge is what forces the alternation of the two types of B-W_1-W_2 move

sequences in TEAM FORMULA GAME. When S points left, B is free to direct the connecting edges so as to unlock variables X. But if B leaves edge A pointing left at the end of his turn, then W_1 can immediately win, starting with edge C. (We skip label B to avoid confusion with the B player label.) Similarly, if S points right, B can unlock the variables in X', but if he leaves edge D pointing right, then W_1 can win beginning with edge E. Later we will see that W_2 must reverse S each turn, forcing distinct actions from B for each direction.

Theorem 26 *TPCL is undecidable.*

Proof: Given an instance of TEAM FORMULA GAME, we construct a TPCL graph as described above. B sees the states of all edges; W_i sees only the states of the edges he controls, those immediately adjacent (so that he knows what moves are legal), and the edge in X corresponding to variable h_i.

We will consider each step of TEAM FORMULA GAME in turn, showing that each step must be mirrored in the TPCL game. Suppose that initially, S points left.

1. B may set variables X to any values by unlocking their controlling latches, beginning with edge H. He may also direct the edges corresponding to the current values of X, Y_1, Y_2, and X' into formula networks F, F', and G, but he may not change the values of X', because their latches must be locked if S points left. If these moves enable him to satisfy formulas F and G, then he wins. Otherwise, if F is true, he may direct edge I upward. He must finish by redirecting A right, thus locking the X variables; otherwise, W_1 could then win as described above. Also, B may leave the states of Y_1 and Y_2 locked.

   B does not have time to both follow the above steps and direct edge K upward within k moves; the pathway from H through "..." to K has $k-3$ edges.

   Also, if F is true then M must point down at the end of B's turn, because F and F' cannot simultaneously be true.

2. If F is false, then I must point down. This will enable W_1 to win, beginning with edge J (because S still points left). Also, if H still points up, W_1 may direct it down, unlocking S; as above, A must point right. Otherwise W_1 has nothing useful to do. He may direct the bottom edges of the Y_1 variables downward, but nothing is accomplished by this, because S points left.

3. On this step W_2 has nothing useful to do but direct S right, which he must do. Otherwise...

4. If S still points left, then B can win, by activating the long series of edges leading to K; I already points up, so unlike in step 1, he has time for this.

   Otherwise, B can now set variables X' to any values, by unlocking their latches, beginning with edge L. If G was not true in step 1, then it cannot be true now, because X has not changed, so B cannot win that way. If F' is true, then he may direct edge M upward. Also, at this point B should unlock Y_1 and Y_2, by

directing his output edges back in and activating the unlock edges in the white variable gadgets. This forces I down, because F depends on the Y_i.

As in step 1, B cannot win by activating edge O, because he does not have time to both follow the above steps and reach O within k moves. (Note that M must point down at the beginning of this turn; see step 1.)

5. If any variable of Y_1 or Y_2 is still locked, W_1 can win by activating the pathway through N. Also, if F' is false then M must point down; this lets W_1 win. (In both cases, note that S points right.) Otherwise, W_1 may now set Y_1 to any values.

6. W_2 may now set Y_2 to any values. Also, W_2 must now direct S left again. If he does not, then on B's next turn he can win by activating O.

Thus, all players are both enabled and required to mimic the given TEAM FORMULA GAME at each step, and so the White team can win the TPCL game if and only if it can win the TEAM FORMULA GAME. □

7.2.3 Planar Graphs

As before, this result holds even when the TPCL graph is planar.

Theorem 27 *TPCL is undecidable, even for planar graphs.*

Proof: All crossings in Figure 7-1 involve only edges controlled by a single player; we can replace these with crossover gadgets from Section 5.2.2 without changing the game. □

Chapter 8

Summary of Part I

In this chapter I step back and take a broader perspective on the results in the preceding chapters. We may view the family of Constraint Logic games, taken as a whole, as a *hierarchy of complete problems*; this idea is developed in Section 8.1. I return to the overall theme of games as computation, this time from a more philosophical and speculative perspective, in Section 8.2. There I address the apparently nonsensical result from Chapter 7 that play in a game of fixed physical size can emulate a Turing machine with an infinite tape.

8.1 Hierarchies of Complete Problems

Galil [31] proposed the notion of a *hierarchy of complete problems*, and gave several examples. The concept is based on the observation that there are many problems known complete for the classes

$$\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE},$$

but at the time, in 1976, there were no examples in the literature of quintuples of complete problems, one for each class, such that each problem was a restricted version of the next. Galil presents several such hierarchies, from domains of graph theory, automata theory, theorem proving, and games.[1]

Galil's definition of a hierarchy of complete problems is specific to those particular complexity classes. However, it seems reasonable to apply the same concept more broadly. In the current case, the family of Constraint Logic games forms what could be considered to be a two-dimensional hierarchy of complete problems. The complexities of the games, ranging from zero player to team games horizontally and bounded vs. unbounded vertically, stand in the following relation (see Figure 1-1):

$$\begin{array}{ccccccc}
\mathrm{PSPACE} & \subseteq & \mathrm{PSPACE} & \subseteq & \mathrm{EXPTIME} & \subseteq & \text{r.e.} \\
\cup\!| & & \cup\!| & & \cup\!| & & \cup\!| \\
\mathrm{P} & \subseteq & \mathrm{NP} & \subseteq & \mathrm{PSPACE} & \subseteq & \mathrm{NEXPTIME}
\end{array}$$

[1] These games are all zero-player games, or simulations, in our sense of "game". They are loosely based on Conway's Game of Life.

Furthermore, in each case a Constraint Logic game is a restricted version of the game to its right or top. Starting with Team Private Constraint Logic, restricting the number of players to be two, the set of private edges to be empty, k (number of moves per turn) to 1, and requiring that the players control disjoint sets of edges yields Two-Player Constraint Logic. Restricting further so that one player has no edges to control[2] yields Nondeterministic Constraint Logic. Restricting further so that the sequence of moves is forced gives Deterministic Constraint Logic. (Technically this last step is not a proper restriction, but we can suppose that there is a general move-order constraint in the other games which is taken to allow any order by default.)

Similarly, adding the restriction that each edge may reverse at most once turns each of those games into their bounded counterparts. Finally, requiring that the graph be planar changes the complexity from P-complete to NC^3-easy in the case of Bounded Deterministic Constraint Logic.

8.2 Games, Physics, and Computation

In Section 7.2, on team games with private information, I showed that there are space-bounded games that are undecidable. At first, this fact seems counterintuitive and bizarre. There are only finitely many positions in the game, and yet somehow the state of the game must effectively encode an arbitrarily long Turing machine tape. How can this be? Eventually the state would have to repeat.

The short answer is that yes, the position must eventually repeat, but some of the players will not know when it is repeating. The entire history of the game is relevant to correct play, and of course the history grows with each move. So in a sense, the infinite tape has merely been shoved under the rug. However, the important point is that the infinite space has been *taken out of the definition of the problem.* Perhaps these games can only be played perfectly by players with infinite memories; perhaps not. That question depends on the nature of the players; perhaps they have access to some non-algorithmic means of generating moves. In any case, the composition and capabilities of the players are not part of the definition of the problem—of the finite computing machine which is a game. A player is simply a black box which is fed game state and generates moves.

Compare the situation to the notion of a nondeterministic Turing machine. The conventional view is that a nondeterministic Turing machine is allowed to "guess" the right choice at each step of the computation. There is no question or issue of how the guess is made. Yet, one speaks of nondeterministic computations being "performed" by the machine. It is allowed access to a non-algorithmic resource to perform its computation. Nondeterministic computers may or may not be "magical" relative to ordinary Turing machines; it is unknown whether P = NP. However, one kind of magic they definitely cannot perform is to turn finite space into infinite space. But a team game "computer", on the other hand, *can* perform this kind of magic, using

[2]Technically each player must have a target edge, but it is easy to construct instances where one player effectively can do nothing, and the question is whether the other player can win. These are effectively Nondeterministic Constraint Logic games.

only a slight generalization of the notion of nondeterminism.

Whether these games can be played perfectly in the real world—and thus, whether we can actually perform arbitrary computations with finite physical resources—is a question of physics, not of computer science. And it is not immediately obvious that the answer must be no.

Others have explored various possibilities for squeezing unusual kinds of computation out of physical reality. There have been many proposals for how to solve NP-complete problems in polynomial time; Aaronson [1] offers a good survey. One such idea, which works if one subscribes to Everett's relative-state interpretation of quantum mechanics [23] (popularly called "many worlds"), is as follows. Say you want to solve an instance of SAT, which is NP-complete. You need to find a variable assignment which satisfies a Boolean formula with n variables. Then you can proceed as follows: guess a random variable assignment, and if it doesn't happen to satisfy the formula, kill yourself. Now, in the only realities you survive to experience you will have "solved" the problem in polynomial time.[3] Aaronson has termed this approach "anthropic computing".

Apart from the possibly metaphysical question of whether there would indeed always be a "you" that survived this "computation", there is the annoying practical problem that those around you would almost certainly experience your death, instead of your successful efficient computation. There is a way around this problem, however. Suppose that, instead of killing yourself, you destroy the entire universe. Then, effectively, the entire universe is cooperating in your computation, and nobody will ever experience you failing and killing yourself. A related idea was explored in the science-fiction story "Doomsday Device", by John Gribbin [37]. In that story a powerful particle accelerator seemingly fails to operate, for no good reason. Then a physicist realizes that if it were to work, it would effectively destroy the entire universe, by initiating a transition from a cosmological false vacuum state to a lower-energy vacuum state. In fact, the accelerator *has* worked; the only realities the characters experience involve highly unlikely equipment failures. (Whether such a false vacuum collapse is actually possible is an interesting question [82].) We can imagine incorporating such a particle accelerator in a computing machine. I would like to propose the term "doomsday computation" for any kind of computation in which the existence of the universe might depend on the output of the computation. Clearly doomsday computation is a special case of anthropic computation.

However, neither approach seems to offer the ability to perform *arbitrary* computations. Other approaches considered in [1] might do better: "time-travel computing", which works by sending bits along closed timelike curves (CTCs), can solve PSPACE-complete problems in polynomial time.

[3] To avoid the problem with what happens when there is no satisfying assignment, Aaronson proposes you instead kill yourself with probability $1-2^{-2n}$ if you don't guess a satisfying assignment. Then if you survive without having guessed an assignment, it is almost certain that there is no satisfying assignment. This step is not strictly necessary, however. There would always be some reality in which you somehow avoided killing yourself; perhaps your suicide machine of choice failed to operate in some highly improbable way. Of course, for the technique to work at all, such a failure must be very improbable.

Perhaps there is some way to generalize some such "weird physics" kind of computation to enable perfect game play. The basic idea of anthropic computation seems appropriate: filter out the realities in which you lose, post-selecting worlds in which you win. But directly applied, as in the SAT example above, this only works for bounded one-player puzzles. Computing with CTCs gets you to PSPACE, which is suggestive of solving a two-player, bounded-length game, or a one-player, unbounded-length puzzle. Perhaps just one step more is all that is needed to create a perfect team game player, and thus a physically finite, but computationally universal, computer.

Part II

Games in Particular

Part II of this thesis applies the results of Part I to particular games and puzzles, to prove them hard. The simplicity of many of the reductions strengthens the view that Constraint Logic is a general game model of computation. Each reduction may be viewed as the construction of a kind of computer, using the physics provided by the game components at hand. Especially useful is the fact that the hardness results for Constraint Logic hold even when the graphs are planar. Traditionally some sort of crossover gadget has often been required for game and puzzle hardness proofs, and these are often among the most difficult gadgets to design.

For all of these results, it must be borne in mind that it is the *generalized* version of a game that is shown hard. For example, Amazons is typically played on a 10×10 board. But it is meaningless to discuss the complexity of a problem for a fixed input size; it is Amazons played on an $n \times n$ board that is shown PSPACE-complete. The 'P' in 'PSPACE' must be polynomial in something.

I give new hardness results for eight games: TipOver, sliding-block puzzles, sliding-coin puzzles, plank puzzles, hinged polygon dissections, Amazons, Konane, and Cross Purposes. Among these, sliding-block puzzles, Amazons, and Konane had been well-known open problems receiving study for some time.

I strengthen the existing hardness results for two games, the Warehouseman's Problem and Sokoban, and I give a simpler hardness proof than the extant one for Rush Hour, and show that a triangular version of Rush Hour is also hard. I also mention some additional results for which I was not the primary contributor.

Part II concludes with a list of interesting open problems.

Chapter 9

One-Player Games (Puzzles)

In this chapter I present several new results for one-player games.

9.1 TipOver

TipOver™ is a puzzle in which the goal is to navigate a layout of vertical crates, tipping some over to reach others, so as to eventually reach a target crate. Crates can only tip into empty space, and you can't jump over empty space to reach other crates. The challenge is to tip the crates in the right directions and the right order.

TipOver originated as an online puzzle created by James Stephens, called the "The Kung Fu Packing Crate Maze" [78]. Now it also exists in physical form (shown in Figure 9-1, produced by ThinkFun, the makers of Rush Hour and other puzzles. Like Rush Hour, TipOver comes with a board and a set of pieces, and 40 challenge cards, each with a different puzzle layout.

The standard TipOver puzzles are laid out on a 6×6 grid, but the puzzle naturally generalizes to $n \times n$ layouts. This is a bounded-move puzzle—each crate can only tip over once. Therefore, it is a candidate for a Bounded NCL reduction. I give a reduction showing that TipOver is NP-complete. (See also [41].)

Figure 9-1: TipOver puzzle.

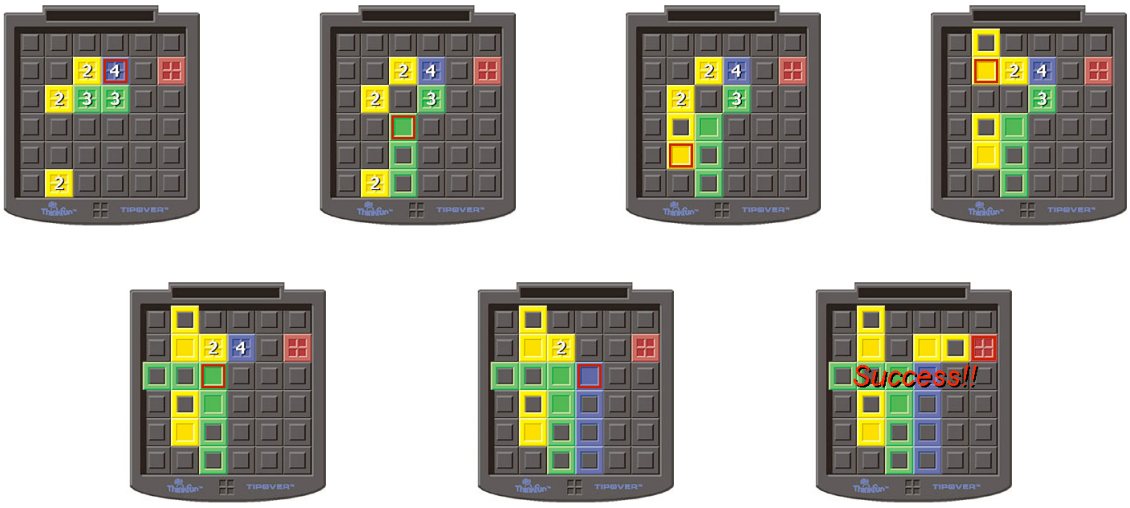


Figure 9-2: A sample TipOver puzzle and its solution.

Rules. In its starting configuration, a TipOver puzzle has several vertical crates of various heights ($1 \times 1 \times h$) arranged on a grid, with a "tipper"—representing a person navigating the layout—standing on a particular starting crate. There is a unique red crate, $1 \times 1 \times 1$, elsewhere on the grid; the goal is to move the tipper to this target red crate.

The tipper can tip over any vertical crate that it is standing on, in any of the four compass directions, provided that there is enough empty space within the grid for that crate to fall unobstructed and lie flat. The tipper is nimble enough to land safely on the newly-fallen crate. The tipper can also walk, or climb, along the tops of any crates that are directly adjacent, even when they have different heights. However, the tipper is not allowed to jump empty space to reach another crate. It can't even jump to a diagonally neighboring crate; the crates must be touching.

A sample puzzle and its solution are shown in Figure 9-2. The first layout is the initial configuration, with the tipper's location marked with a red square outline, and the height of each vertical crate indicated. In each successive step, one more crate has been tipped over.

9.1.1 NP-completeness

We reduce Bounded Planar NCL (Section 5.1.2) to TipOver to show NP-hardness. Given an instance of Bounded Planar NCL, we construct a TipOver puzzle that can be solved just when the target edge can be reversed. We need to build AND, OR, FANOUT, and CHOICE gadgets, as in Section 5.1.3, and show how to wire them together. We also need to build a single loose edge, but this just corresponds to the tipper's starting point.

All of our gadgets will be built with initially vertical, height-two crates. The mapping from constraint graph properties to TipOver properties is that an edge can be reversed just when a corresponding TipOver region is reachable by the tipper.

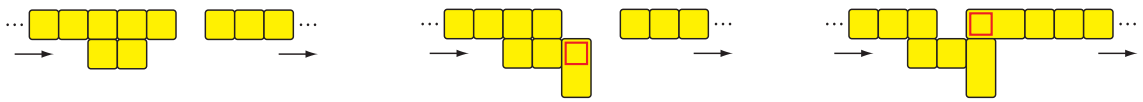
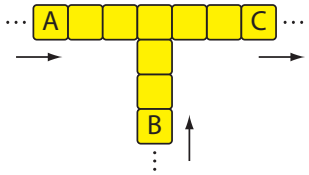
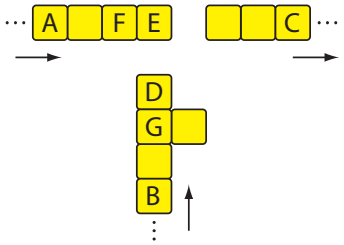


Figure 9-3: A wire that must be initially traversed from left to right. All crates are height two.



(a) OR gadget. If the tipper can reach either A or B, then it can reach C.



(b) AND gadget. If the tipper can reach both A and B, then it can reach C.

Figure 9-4: TipOver AND and OR gadgets.

One-way Gadget. We will need an auxiliary gadget that can only be traversed by the tipper in one direction initially. The gadget, and the sequence of steps in a left-right traversal, are shown in Figure 9-3. Once it's been so traversed, a one-way gadget can be used as an ordinary wire. But if it is first approached from the right, there's no way to bridge the gap and reach the left side.

We will attach one-way gadgets to the inputs and outputs of each of the following gadgets.

OR / FANOUT Gadget. A simple intersection, protected with one-way gadgets, serves as an OR, shown in Figure 9-4(a).

Lemma 28 *The construction in Figure 9-4(a) satisfies the same constraints as a Bounded NCL* OR *vertex, with* A *and* B *corresponding to the input edges, and* C *corresponding to the output edge.*

Proof: The tipper can clearly reach C if and only if it can reach either A or B. Since the output is protected with a one-way gadget, the tipper cannot reach C by any other means. □

Clearly, changing the direction of the one-way gadget protecting input A turns it input an output, and turns an OR gadget into a FANOUT gadget with the input at B.

AND Gadget. AND is a bit more complicated. The construction is shown in Figure 9-4(b). This time the tipper must be able to exit to the right only if it can independently enter from the left and from the bottom. This means that, at a minimum, it will have to enter from one side, tip some crates, retrace its path, and enter from the other side. Actually, the needed sequence will be a bit longer than that.

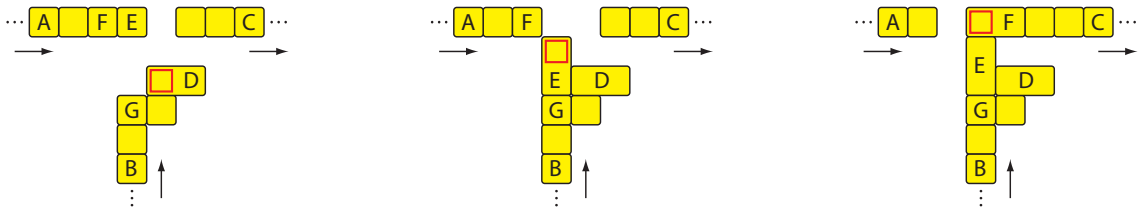


Figure 9-5: How to use the AND gadget.

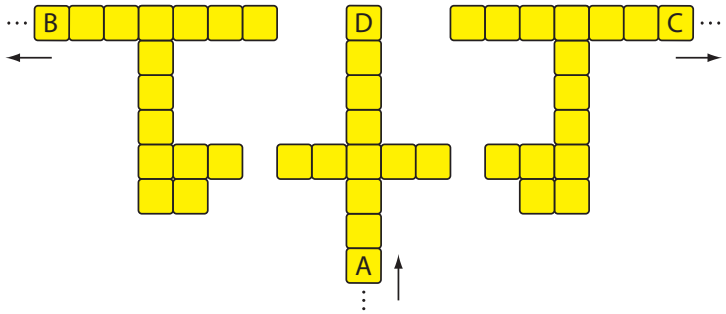


Figure 9-6: TipOver CHOICE gadget. If the tipper can reach A, then it can reach B or C, but not both.

Lemma 29 *The construction in Figure 9-4(b) satisfies the same constraints as a Bounded NCL AND vertex, with A and B corresponding to the input edges, and C corresponding to the output edge.*

Proof: We need to show that the tipper can reach C if and only if it can first reach A and B. First, note that F is the only crate that can possibly be tipped so as to reach C; no other crate will do. If the tipper is only able to enter from A, and not from B, it can never reach C. The only thing that can be accomplished is to tip crate F down, so as to reach B from the wrong direction. But this doesn't accomplish anything, because once F has been tipped down it can never be tipped right, and C can never be reached. Suppose, instead, the tipper can enter from B, but not from A. Then again, it can reach A from the wrong direction, by tipping crate D right and crate G up. But again, nothing is accomplished by this, because now crate E can't be gotten out of the way without stranding the tipper.

Now suppose the tipper can reach both A and B. Then the following sequence (shown in Figure 9-5) lets it reach C. First the tipper enters from B, and tips crate D right. Then it retraces its steps along the bottom input, and enters this time from A. Now it tips crate E down, connecting back to B. From here it can again exit via the bottom, return to A, and finally tip crate F right, reaching C. The right side winds up connected to the bottom input, so that the tipper can still return to its starting point as needed from later in the puzzle. □

CHOICE Gadget. Finally, we need a CHOICE gadget. This is shown in Figure 9-6.

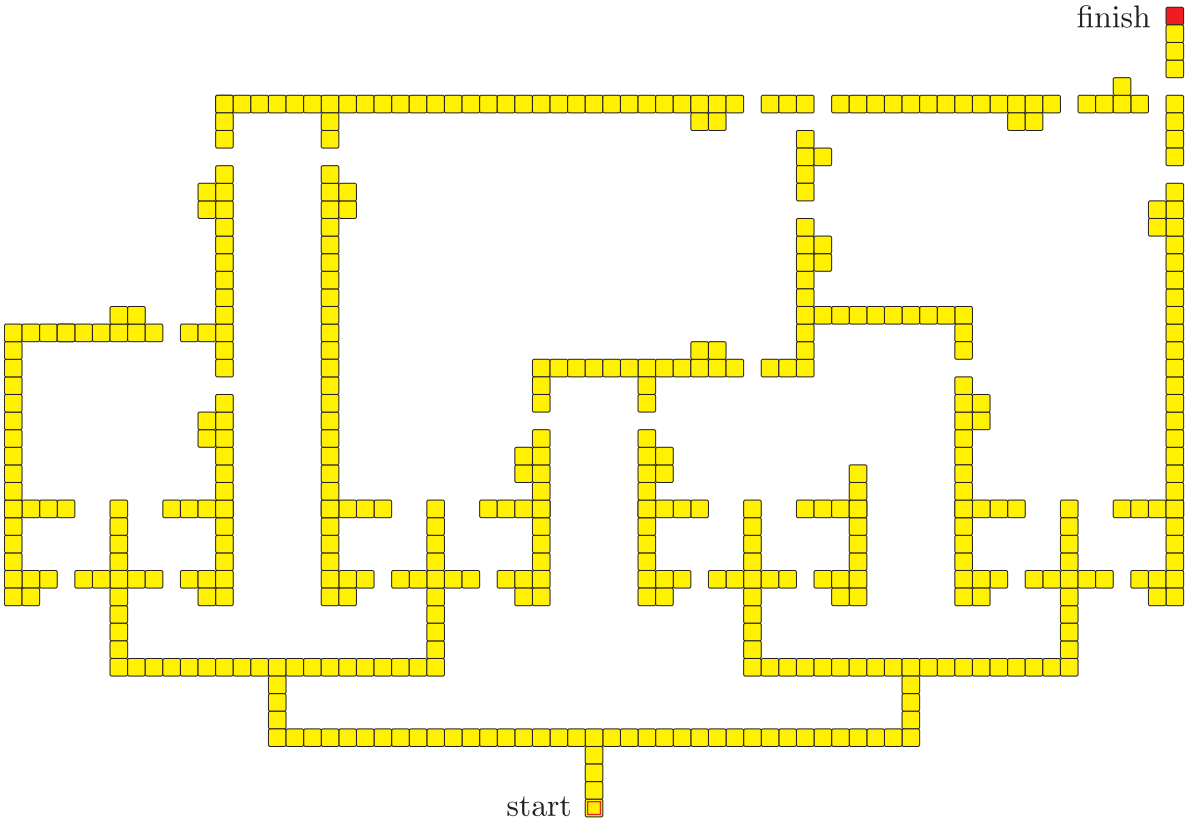


Figure 9-7: TipOver puzzle for a simple constraint graph.

Lemma 30 *The construction in Figure 9-6 satisfies the same constraints as a Bounded NCL* CHOICE *vertex, with* A *corresponding to the input edge, and* B *and* C *corresponding to the output edges.*

Proof: Because of the built-in one-way gadgets, the only way the tipper can exit the gadget is by irreversibly tipping D either left or right. It may then reconnect to A by using the appropriate one-way pathway, but it can never reach the other side. □

Theorem 31 *TipOver is NP-complete.*

Proof: Given a bounded planar constraint graph made of AND, OR, FANOUT, CHOICE, and red-blue vertices, and with a single edge which may initially reverse, we construct a corresponding TipOver puzzle, as described above. The wiring connecting the gadgets together is simply a chain of vertical, height-2 crates. The tipper starts on some gadget input corresponding to the location of the single loose edge, and can reach the target crate just when the target edge in the constraint graph may be reversed. Therefore, TipOver is NP-hard.

TipOver is clearly in NP: there are only a linear number of crates that may tip over, and therefore a potential solution may be verified in polynomial time. □

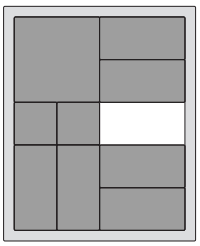


Figure 9-8: Dad's Puzzle.

9.2 Sliding-Block Puzzles

This section is joint work with Erik Demaine [46, 43, 47].

Sliding-block puzzles have long fascinated aficionados of recreational mathematics. From the infamous 15 puzzle [75] associated with Sam Loyd to the latest whimsical variants such as Rush Hour^{TM}, these puzzles seem to offer a maximum of complexity for a minimum of space.

In the usual kind of sliding-block puzzle, one is given a box containing a set of rectangular pieces, and the goal is to slide the blocks around so that a particular piece winds up in a particular place. A popular example is Dad's Puzzle, shown in Figure 9-8; it takes 59 moves to slide the large square to the bottom left.

Effectively, the complexity of determining whether a given sliding-block puzzle is solvable was an open problem for nearly 40 years. Martin Gardner devoted his February, 1964 Mathematical Games column to sliding-block puzzles. This is what he had to say [32]:

> *These puzzles are very much in want of a theory. Short of trial and error, no one knows how to determine if a given state is obtainable from another given state, and if it is obtainable, no one knows how to find the minimum chain of moves for achieving the desired state.*

The computational complexity of sliding-block puzzles was considered explicitly by Spirakis and Yap in 1983 [77]; they showed that determining whether there is a solution to a given puzzle is NP-hard, and conjectured that it is PSPACE-complete. However, the form of the problem they considered was somewhat different from that framed here. In their version, the goal is to reach a given total configuration, rather than just moving a given piece to a given place, and there was no restriction on the sizes of blocks allowed. This problem was shown PSPACE-complete shortly afterwards, by Hopcroft, Schwartz, and Sharir [48], and renamed the "warehouseman's problem". (This problem is discussed in Section 9.3.)

This left the appropriate form of the decision question for actual sliding-block puzzles open until Demaine and I showed it PSPACE-complete[1] in 2002 [46], based on the earlier result that the related puzzle Rush Hour is PSPACE-complete [24].

[1]The problem may be posed either combinatorially, as considered here, or geometrically. Here we consider only discrete moves, as appropriate for generalized combinatorial games; if continuous block movements are allowed, then the result is only that sliding-block puzzles are PSPACE-hard, and the upper bound is not addressed.

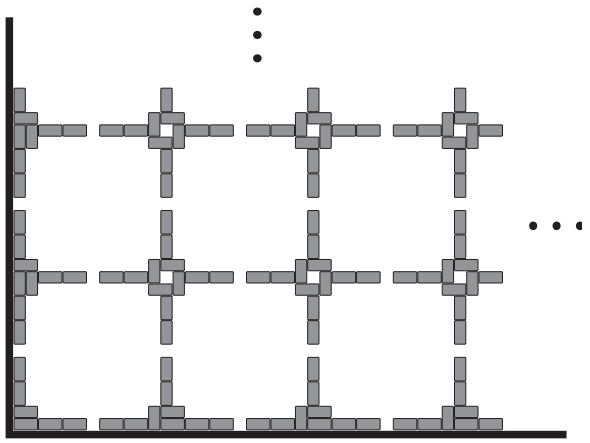


Figure 9-9: Sliding Blocks layout.

The *Sliding Blocks* problem is defined as follows: given a configuration of rectangles (*blocks*) of constant sizes in a rectangular 2-dimensional box, can the blocks be translated and rotated, without intersection among the objects, so as to move a particular block?

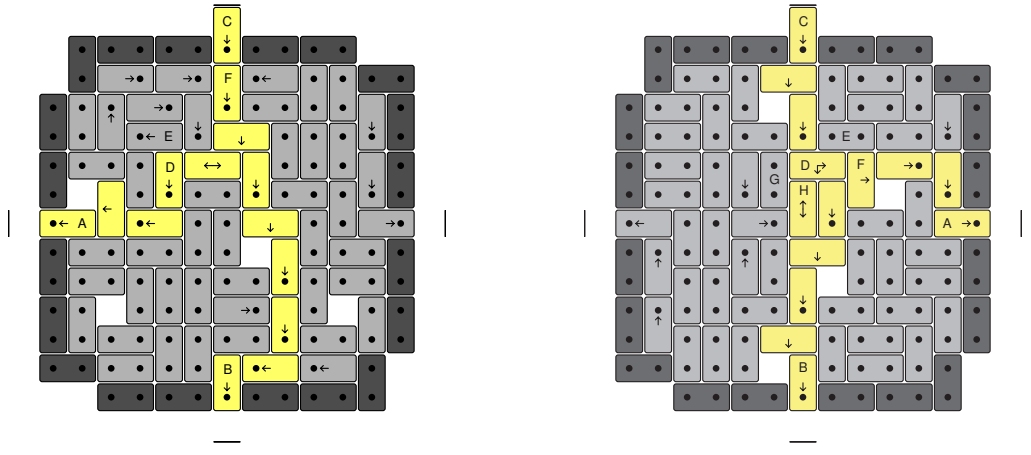
9.2.1 PSPACE-completeness

I give a reduction from planar Nondeterministic Constraint Logic (NCL) showing that Sliding Blocks is PSPACE-hard even when all the blocks are 1×2 rectangles (dominoes). (Somewhat simpler constructions are possible if larger blocks are allowed.) In contrast, there is a simple polynomial-time algorithm for 1×1 blocks; thus, the results are in some sense tight.

Sliding Blocks Layout. We fill the box with a regular grid of gate gadgets, within a "cell wall" construction as shown in Figure 9-9. The internal construction of the gates is such that none of the cell-wall blocks may move, thus providing overall integrity to the configuration.

AND and OR Vertices. We construct NCL AND and protected OR (Section 5.2.3) vertex gadgets out of dominoes, in Figures 9-10(a) and 9-10(b). Each figure provides the bulk of an inductive proof of its own correctness, in the form of annotations. A dot indicates a square that is always occupied; the arrows indicate the possible positions a block can be in. For example, in Figure 9-10(b), block D may occupy its initial position, the position one unit to the right, or the position one unit down (but not, as we will see, the position one unit down and one unit right).

For each vertex gadget, if we inductively assume for each block that its surrounding annotations are correct, its own correctness will then follow, except for a few cases noted below. The annotations were generated by a computer search of all reachable configurations, but are easy to verify by inspection.



(a) AND (b) Protected OR

Figure 9-10: Sliding Blocks vertex gadgets.

In each diagram, we assume that the cell-wall blocks (dark colored) may not move outward; we then need to show they may not move inward. The light-colored ("trigger") blocks are the ones whose motion serves to satisfy the vertex constraints; the medium-colored blocks are fillers. Some of them may move, but none may move in such a way as to disrupt the vertices' correct operation.

The short lines outside the vertex ports indicate constraints due to adjoining vertices; none of the "port" blocks may move entirely out of its vertex. For it to do so, the adjoining vertex would have to permit a port block to move entirely inside the vertex, but in each diagram the annotations show this is not possible. Note that the port blocks are shared between adjoining vertices, as are the cell-wall blocks. For example, if we were to place a protected OR above an AND, its bottom port block would be the same as the AND's top port block.

A protruding port block corresponds to an inward-directed edge; a retracted block corresponds to an outward-directed edge. Signals propagate by moving "holes" forward. Sliding a block *out* of a vertex gadget thus corresponds to directing an edge *in* to a graph vertex.

Lemma 32 *The construction in Figure 9-10(a) satisfies the same constraints as an NCL AND vertex, with A and B corresponding to the AND red edges, and C to the blue edge.*

Proof: We need to show that block C may move down if and only if block A first moves left and block B first moves down.

First, observe that this motion is possible. The trigger blocks may each shift one unit in an appropriate direction, so as to free block C.

The annotations in this case serve as a complete proof of their own correctness, with one exception. Block D appears as though it might be able to slide upward, because block E may slide left, yet D has no upward arrow. However, for E to slide

left, F must first slide down, but this requires that D be first be slid down. So when E slides left, D is not in a position to fill the space it vacates.

Given the annotations' correctness, it is easy to see that it is not possible for C to move down unless A moves left and B moves down. □

Lemma 33 *The construction in Figure 9-10(b) satisfies the same constraints as an NCL protected* OR *vertex, with A and B corresponding to the protected edges.*

Proof: We need to show that block C may move down if and only if block A first moves right, or block B first moves down.

First, observe that these motions are possible. If A moves right, D may move right, releasing the blocks above it. If B moves down, the entire central column may also move down.

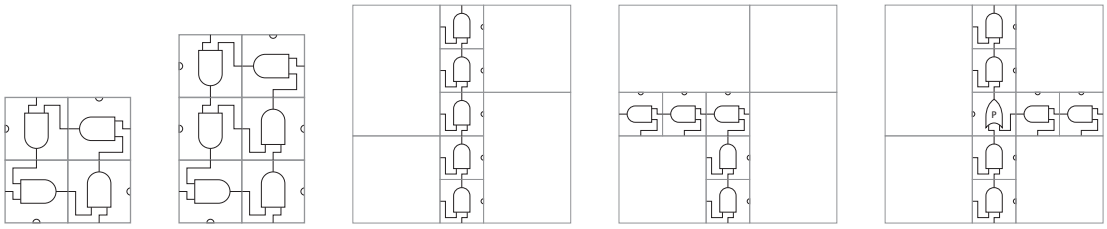
The annotations again provide the bulk of the proof of their own correctness. In this case there are three exceptions. Block E looks as if it might be able to move down, because D may move down and F may move right. However, D may only move down if B moves down, and F may only move right if A moves right. Because this is a protected OR, we are guaranteed that this cannot happen: the vertex will be used only in graphs such that at most one of A and B can slide out at a time. Likewise, G could move right if D were moved right while H were moved down, but again those possibilities are mutually exclusive. Finally, D could move both down and right one unit, but again this would require A and B to both slide out.

Given the annotations' correctness, it is easy to see that it is not possible for C to move down unless A moves right or B moves down. □

Graphs. Now that we have AND and protected OR gates made out of sliding-blocks configurations, we must next connect them together into arbitrary planar graphs. First, note that the box wall constrains the facing port blocks of the vertices adjacent to it to be retracted (see Figure 9-9). This does not present a problem, however, as I will show. The unused ports of both the AND and protected OR vertices are unconstrained; they may be slid in or out with no effect on the vertices. Figures 9-11(a) and 9-11(b) show how to make (2×2)-vertex and (2×3)-vertex "filler" blocks out of ANDs. (We use conventional "and" and "or" icons to denote the vertex gadgets.) Because none of the ANDs need ever activate, all the exterior ports of these blocks are unconstrained. (The unused ports are drawn as semicircles.)

We may use these filler blocks to build (5 × 5)-vertex blocks corresponding to "straight" and "turn" wiring elements (Figures 9-11(c) and 9-11(d)). Because the filler blocks may supply the missing inputs to the ANDs, the "output" of one of these blocks may activate (slide in) if and only if the "input" is active (slid out). Also, we may "wrap" the AND and protected OR vertices in 5 × 5 "shells", as shown for protected OR in Figure 9-11(e). (Note that "left turn" is the same as "right turn"; switching the roles of input and output results in the same constraints.)

We use these 5×5 blocks to fill the layout; we may line the edges of the layout with unconstrained ports. The straight and turn blocks provide the necessary flexibility to



(a) 2 × 2 filler (b) 2 × 3 filler (c) 5 × 5 straight (d) 5 × 5 turn (e) 5 × 5 protected OR

Figure 9-11: Sliding Blocks wiring.

construct any planar graph, by letting us extend the vertex edges around the layout as needed.

Theorem 34 *Sliding Blocks is PSPACE-complete, even for* 1×2 *blocks.*

Proof: Reduction from planar NCL for protected-OR graphs, by the construction described. A port block of a particular vertex gadget may move if and only if the corresponding NCL graph edge may be reversed.

Sliding Blocks is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 12. □

9.3 The Warehouseman's Problem

This section is joint work with Erik Demaine [46, 47].

As mentioned in Section 9.2, The *Warehouseman's Problem* is a particular formulation of a kind of sliding-block problem in which the blocks are not required to have a fixed size, and the goal is to put each block at a specified final position. Hopcroft, Schwartz, and Sharir [48] showed the Warehouseman's Problem PSPACE-hard in 1984.

Their construction critically requires that some blocks have dimensions that are proportional to the box dimensions. Using Nondeterministic Constraint Logic, we can strengthen (and greatly simplify) the result: it is PSPACE-complete to achieve a specified total configuration, even when the blocks are all 1×2.

9.3.1 PSPACE-completeness

Theorem 35 *The Warehouseman's Problem is PSPACE-hard, even for* 1×2 *blocks.*

Proof: As in Section 9.2, but using Theorem 18, which shows determining whether a given total configuration may be reached from a given AND/OR graph is PSPACE-hard. The graph initial and desired configurations correspond to two block configurations; the second is reachable from the first if and only if the NCL problem has a solution. □

If we restrict the block motions to unit translations (as appropriate when viewing the problem as a generalized combinatorial game), then the problem is also in PSPACE, as in Theorem 12.

9.4 Sliding-Coin Puzzles

This section is joint work with Erik Demaine [47].

Sliding-block puzzles have an obvious complexity about them, so it is no surprise that they are PSPACE-complete. What is more surprising is that there are PSPACE-complete sliding-coin puzzles. For sliding-block puzzles, if the blocks are all 1×1, as in the 15 puzzle, the puzzles become easy—it is the fact that one block can be dependent on the positions of two other blocks for the ability to move in a particular direction that makes it possible to build complex puzzles and gadgets. In a typical sliding-coin puzzle, a coin is like a 1×1 block; it only needs one other coin to move out of the way for it to be able to move and take its place. Indeed, many forms of sliding-coin puzzle have been shown to be efficiently solvable [17].

But it turns out that adding a simple constraint to the motion of the coins leads to a very natural problem which is PSPACE-complete.

The *Sliding Tokens* problem is defined as follows. It is played on an undirected graph with tokens placed on some of the vertices. A *legal configuration* of the graph is a token placement such that no adjacent vertices both have tokens. (That is, the tokens form an independent set of vertices.) A move is made by sliding a token from one vertex to an adjacent one, along an edge, such that the resulting configuration is legal. Given an initial configuration, is is possible to move a given token?

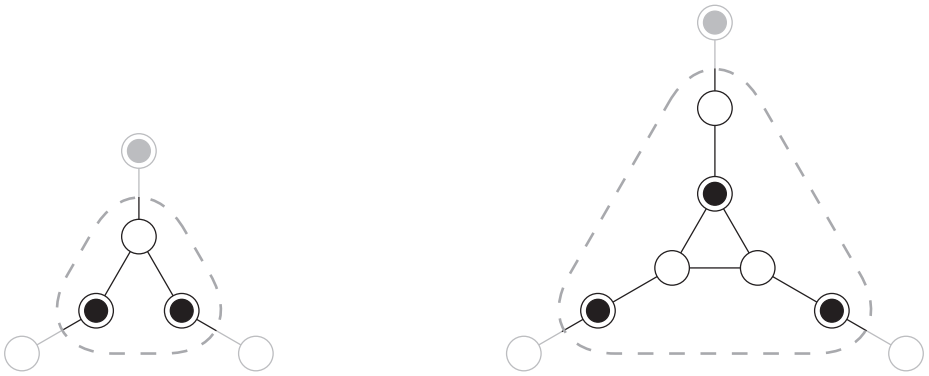
Note that this problem is essentially a dynamic, puzzle version of the Independent Set problem, which is NP-complete [34]. Similarly, the natural two-player-game version of Independent Set, called Kayles, is also PSPACE-complete [34]. Just as many NP-complete problems become PSPACE-complete when turned into two-player games [72], it is also natural to expect that they become PSPACE-complete when turned into dynamic puzzles.

Finally, from a more computational perspective, sliding-token graphs also superficially resemble Petri nets.

9.4.1 PSPACE-completeness

I give a reduction from planar Nondeterministic Constraint Logic showing that this problem is PSPACE-complete.

AND and OR Vertices. We construct NCL AND and OR vertex gadgets out of sliding-token subgraphs, in Figures 9-12(a) and 9-12(b). The edges that cross the dotted-line gadget borders are "port" edges. A token on an outer port-edge vertex represents an inward-directed NCL edge, and vice-versa. Given an AND/OR graph and configuration, we construct a corresponding sliding-token graph, by joining to-



(a) AND (b) OR

Figure 9-12: Sliding Tokens vertex gadgets.

gether AND and OR vertex gadgets at their shared port edges, placing the port tokens appropriately.

Theorem 36 *Sliding Tokens is PSPACE-complete..*

Proof: First, observe that no port token may ever leave its port edge. Choosing a particular port edge A, if we inductively assume that this condition holds for all other port edges, then there is never a legal move outside A for its token another port token would have to leave its own edge first.

The AND gadget clearly satisfies the same constraints as an NCL AND vertex; the upper token can slide in just when both lower tokens are slid out. Likewise, the upper token in the OR gadget can slide in when either lower token is slid out the internal token can then slide to one side or the other to make room. It thus satisfies the same constraints as an NCL AND vertex.

Sliding Tokens is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 12. □

9.5 Plank Puzzles

A plank puzzle is a puzzle in which the goal is to cross a crocodile-infested swamp, using only wooden planks supported by tree stumps.

Plank puzzles were invented by UK maze enthusiast Andrea Gilbert. Like TipOver, they originated as a popular online puzzle applet [35]; now there is also a physical version, sold by ThinkFun as River Crossing^{TM}. Like Rush Hour and TipOver, the puzzle comes with a set of challenge cards, each with a different layout. Also like Rush Hour, and unlike TipOver, plank puzzles are unbounded games; there is no resource that is used up as the game is played. (By contrast, in TipOver, the number of vertical crates must decrease by one each turn.) I give a reduction from Nondeterministic Constraint Logic showing that plank puzzles are PSPACE-complete. (See also [43].)

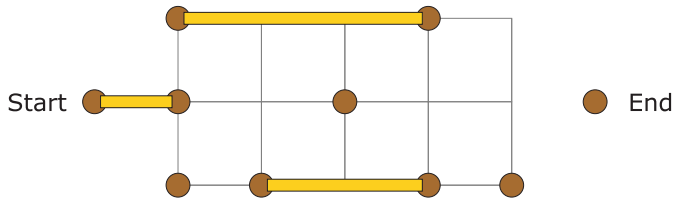
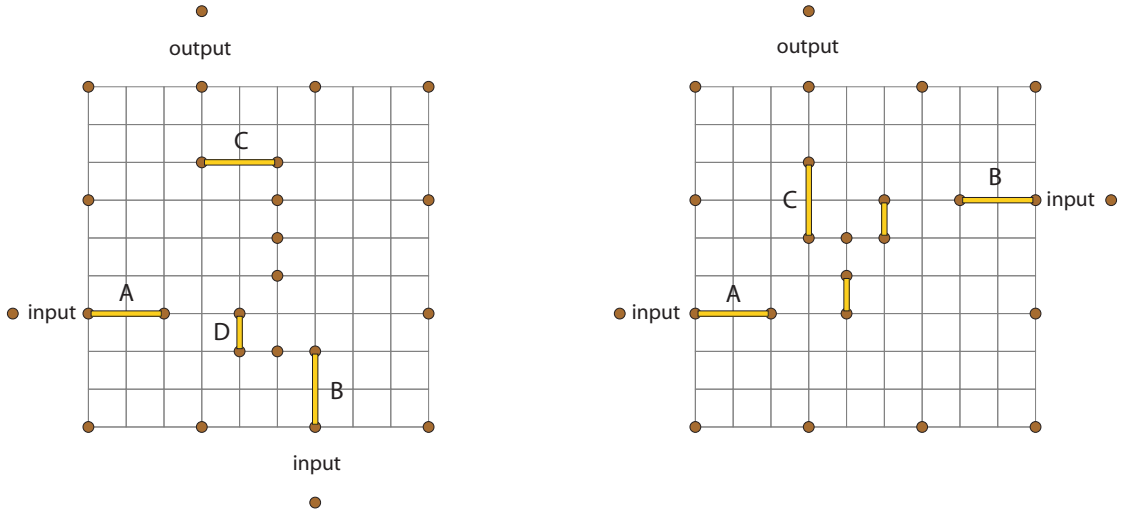


Figure 9-13: A plank puzzle.



(a) AND (b) OR

Figure 9-14: Plank-puzzle AND and OR vertices.

Rules. The game board is an $n \times n$ grid, with *stumps* at some intersections, and *planks* arranged between some pairs of stumps, along the grid lines. The goal is to go from one given stump to another. You can pick up planks, and put them down between other stumps separated by exactly the plank length. You are not allowed to cross planks over each other, or over intervening stumps, and you can carry only one plank at a time.

A sample plank puzzle is shown in Figure 9-13. The solution begins as follows: walk across the length-1 plank; pick it up; lay it down to the south; walk across it; pick it up again; lay it down to the east; walk across it again; pick it up again; walk across the length-2 plank; lay the length-1 plank down to the east; ...

9.5.1 PSPACE-completeness

I give a reduction from Nondeterministic Constraint Logic. We need AND and OR vertices, and a way to wire them together to create a plank puzzle corresponding to any given AND/OR graph.

The constraint graph edge orientations are represented by the positions of "port planks" at each vertex interface; moving a port plank into a vertex gadget enables it to operate appropriately, and prevents it from being used in the paired vertex gadget.

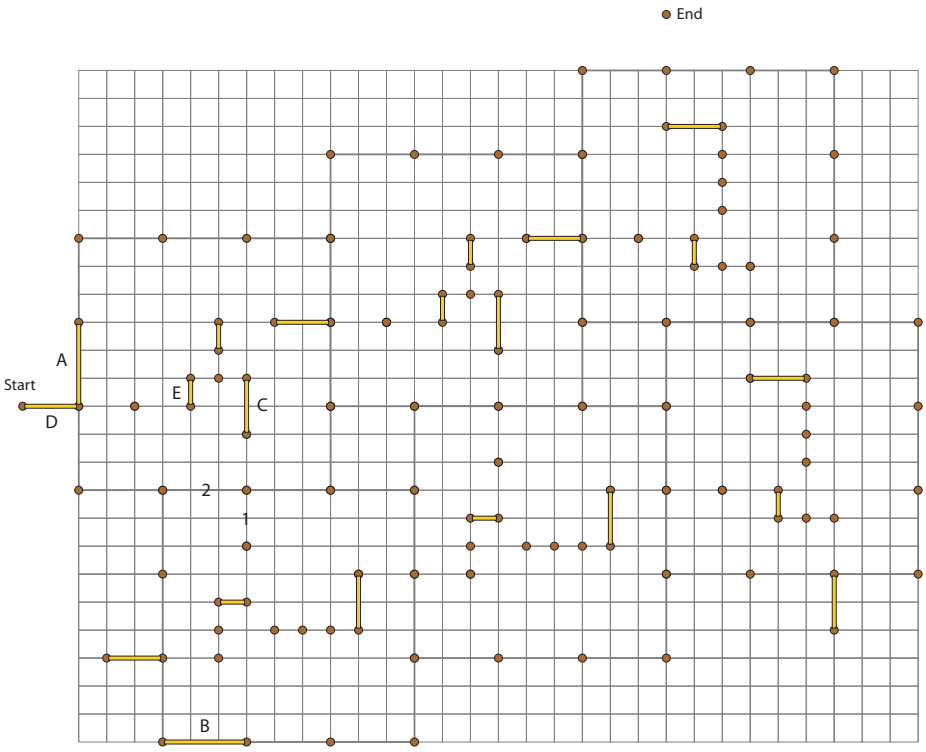


Figure 9-15: A plank puzzle made from an AND/OR graph.

AND vertex. The plank-puzzle AND vertex is shown in Figure 9-14(a). The length-2 planks serve as the input and output ports. (Of course, the gate may be operated in any direction.) Both of its input port planks (A and B) are present, and thus activated; this enables you to move its output port plank (C) outside the gate. Suppose you are standing at the left end of plank A. First walk across this plank, pick it up, and lay it down in front of you, to reach plank D. With D you can reach plank B. With B and D, you can reach C, and escape the gate. At the end of this operation A and B are trapped inside the vertex, inaccessible to the adjoining vertex gadgets.

The operation is fully reversible, since the legal moves in plank puzzles are reversible.

OR vertex. The OR vertex is shown in Figure 9-14(b). In this case, starting at either A or B will let you move the output plank C outside the vertex, by way of internal length-1 plank(s), trapping the starting plank inside the vertex.

Constraint Graphs. To complete the construction, we must have a way to wire these gates together into large puzzle circuits. Once you have activated an AND gate, you're stuck standing on its output plank—now what?

Figure 9-15 shows a puzzle made from six gates. For reference, the equivalent constraint graph is shown in Figure 9-16. The gates are arranged on a staggered grid, in order to make matching inputs and outputs line up. The port planks are shared between adjoining gates. Notice that two length-3 planks have been added to the

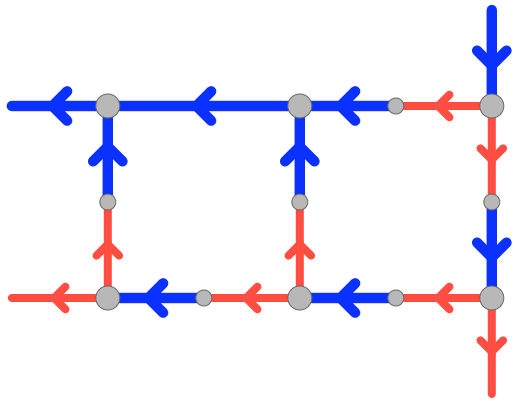


Figure 9-16: The equivalent constraint graph for Figure 9-15.

puzzle. These are the key to moving around between the gates. If you are standing on one of these planks, you can walk along the edges of the gates, by repeatedly laying the plank in front of you, walking across it, then picking it up. This will let you get to any port of any of the gates. By using both the length-3 planks, you can alternately place one in front of the other, until you reach the next port you want to exit from. Then you can leave one length-3 plank there, and use the remaining one to reach the desired port entrance.

However, you can't get inside any of the gates using just a length-3 plank, because there are no interior stumps exactly three grid units from a border stump.

To create arbitrary planar constraint graphs, we can use the same techniques used in Section 9.2 to build large "straight" and "turn" blocks out of 5×5 blocks of vertex gadgets.

Theorem 37 *Plank puzzles are PSPACE-complete.*

Proof: Reduction from planar NCL, by the construction described. A given stump may be reached if and only if the corresponding NCL graph edge may be reversed.

Plank puzzles are in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 12. □

9.6 Sokoban

This section is joint work with Erik Demaine [46, 47].

In the pushing-blocks puzzle *Sokoban*, one is given a configuration of 1×1 blocks, and a set of target positions. One of the blocks is distinguished as the *pusher*. A move consists of moving the pusher a single unit either vertically or horizontally; if a block occupies the pusher's destination, then that block is pushed into the adjoining space, providing it is empty. Otherwise, the move is prohibited. Some blocks are *barriers*, which may not be pushed. The goal is to make a sequence of moves such that there is a (non-pusher) block in each target position.

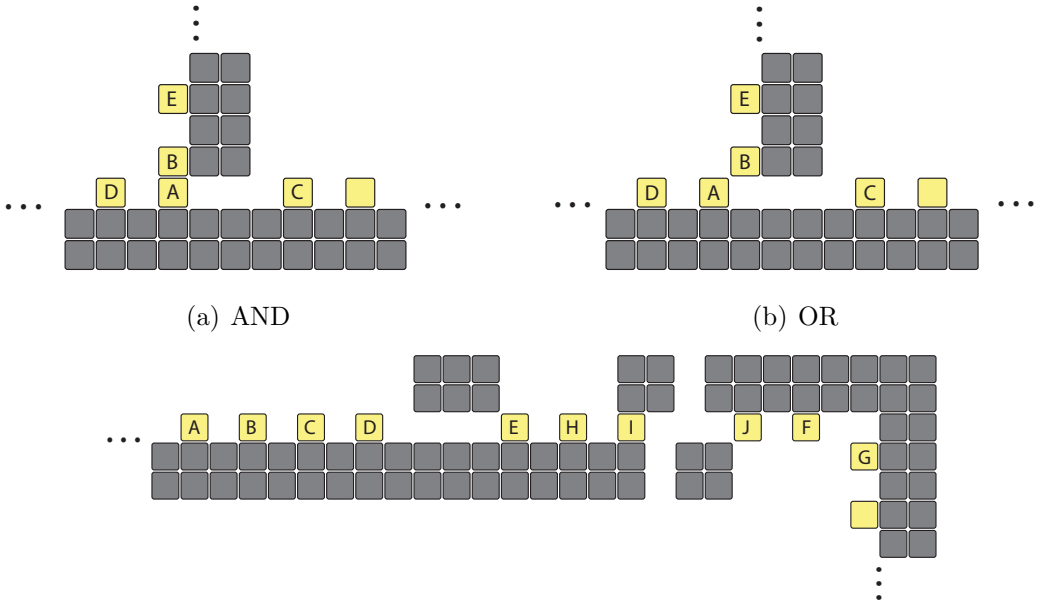
Culberson [13] proved that Sokoban is PSPACE-complete, by showing how to construct a Sokoban position corresponding to a space-bounded Turing machine. Using Nondeterministic Constraint Logic, I give an alternate proof. Our result applies even if there are no barriers allowed in the Sokoban position, thus strengthening Culberson's result.

9.6.1 PSPACE-completeness

Unrecoverable Configurations. The idea of an *unrecoverable configuration* is central to Culberson's proof, and it will be central to our proof as well. We construct our Sokoban instance so that if the puzzle is solvable, then the original configuration may be restored from any solved state by reversing all the pushes. Then any push which may not be reversed leads to an unrecoverable configuration. For example, in the partial configuration in Figure 9-17(a), if block A is pushed left, it will be irretrievably stuck next to block D; there is no way to position the pusher so as to move it again. We may speak of such a move as being prohibited, or impossible, in the sense that no solution to the puzzle can include such a move, even though it is technically legal.

AND and OR Vertices. We construct NCL AND and OR vertex gadgets out of partial Sokoban positions, in Figure 9-17. (The pusher is not shown.) The dark-colored blocks in the figures, though unmovable, are not barriers; they are simply blocks that cannot be moved by the pusher because of their configuration. The yellow (light-colored) "trigger" blocks are the ones whose motion serves to satisfy the vertex constraints. In each vertex, blocks A and B represent outward-directed edges; block C represents an inward-directed edge. A and C switch state by moving left one unit; B switches state by moving up one unit. We assume that the pusher may freely move to any empty space surrounding a vertex. We also assume that block D in Figure 9-17(a) may not reversibly move left more than one unit. Later, I show how to arrange both of these conditions.

Lemma 38 *The construction in Figure 9-17(a) satisfies the same constraints as an NCL AND vertex, with A and B corresponding to the AND red edges, and C to the blue edge.*



(a) AND

(b) OR

(c) Utility gadgets

Figure 9-17: Sokoban gadgets.

Proof: We need to show that C may move left if and only if A first moves left, and B first moves up. For this to happen, D must first move left, and E must first move up; otherwise pushing A or B would lead to an unrecoverable configuration. Having first pushed D and E out of the way, we may then push A left, B up, and C left. However, if we push C left without first pushing A left and B up, then we will be left in an unrecoverable configuration; there will be no way to get the pusher into the empty space left of C to push it right again. (Here we use the fact that D can only move left one unit.) □

Lemma 39 *The construction in Figure 9-17(b) satisfies the same constraints as an NCL* OR *vertex.*

Proof: We need to show that C may move left if and only if A first moves left, or B first moves up.

As before, D or E must first move out of the way to allow A or B to move. Then, if A moves left, C may be pushed left; the gap opened up by moving A lets the pusher get back in to restore C later. Similarly for B.

However, if we push C left without first pushing A left or B up, then, as in Lemma 38, we will be left in an unrecoverable configuration. □

Graphs. We have shown how to make AND and OR vertices, but we must still show how to connect them up into arbitrary planar graphs. The remaining gadgets we shall need are illustrated in Figure 9-17(c).

The basic idea is to connect the vertices together with alternating sequences of blocks placed against a double-thick wall, as in the left of Figure 9-17(c). Observe that for block A to move right, first D must move right, then C, then B, then finally A, otherwise two blocks will wind up stuck together. Then, to move block D left again, the reverse sequence must occur. Such movement sequences serve to propagate activation from one vertex to the next.

We may switch the "parity" of such strings, by interposing an appropriate group of six blocks: E must move right for D to, then D must move back left for E to. We may turn corners: for F to move right, G must first move down. Finally, we may "flip" a string over, to match a required orientation at the next vertex, or to allow a turn in a desired direction: for H to move right, I must move right at least two spaces; this requires that J first move right.

We satisfy the requirement that block D in Figure 9-17(a) may not reversibly move left more than one unit by protecting the corresponding edge of every AND with a turn; observe that in Figure 9-17(c), block F may not reversibly move right more than one unit. The flip gadget solves our one remaining problem: how to position the pusher freely wherever it is needed. Observe that it is always possible for the pusher to cross a string through a flip gadget. (After moving J right, we may actually move I *three* spaces right.) If we simply place at least one flip along each wire, then the pusher can get to any side of any vertex.

Theorem 40 *Sokoban is PSPACE-complete, even if no barriers are allowed.*

Proof: Reduction from planar Nondeterministic Constraint Logic. Given a planar AND/OR graph, we build a Sokoban puzzle as described above, corresponding to the initial graph configuration. We place a target at every position that would be occupied by a block in the Sokoban configuration corresponding to the target graph configuration. Since NCL is inherently reversible, and our construction emulates NCL, then the solution configuration must also be reversible, as required for the unrecoverable configuration constraints.

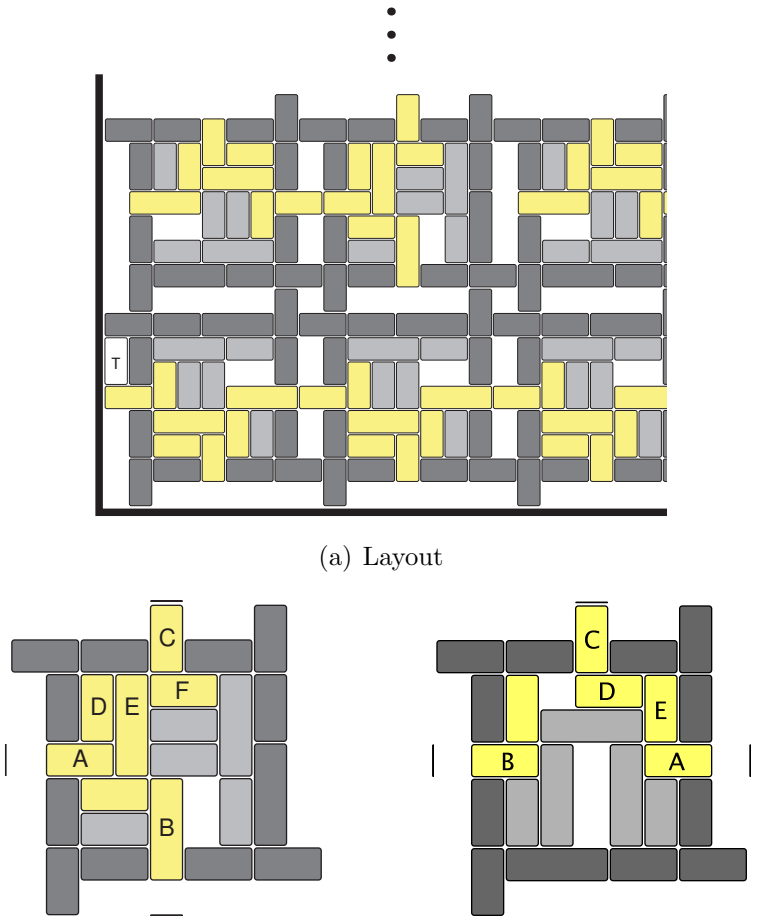
Sokoban is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 12. □

9.7 Rush Hour

This section is joint work with Erik Demaine [46, 47].

In the puzzle *Rush Hour*, one is given a sliding-block configuration with the additional restriction that each block is constrained to move only horizontally or vertically on a grid. The goal is to move a particular block to a particular location at the edge of the grid. In the commercial version of the puzzle, the grid is 6×6, the blocks are all 1×2 or 1×3 ("cars" and "trucks"), and each block constraint direction is the same as its lengthwise orientation.

Flake and Baum [24] showed that the generalized problem is PSPACE-complete, by showing how to build a kind of reversible computer from Rush Hour gadgets that



(a) Layout

(b) AND

(c) Protected OR

Figure 9-18: Rush Hour layout and vertex gadgets.

work like Constraint Logic AND and OR vertices, as well as a crossover gadget. (Their construction was the basis for the development of Constraint Logic.) Tromp [83, 84] strengthened their result by showing that Rush Hour is PSPACE-complete even if the blocks are all 1×2.

Here I give a simpler construction showing that Rush Hour is PSPACE-complete, again using the traditional 1×2 and 1×3 blocks which must slide lengthwise. We only need an AND and a protected OR (Section 5.2.3), which turns out to be easier to build than OR; because of the generic crossover construction (Section 5.2.2), we don't need a crossover gadget. (We also don't need the miscellaneous wiring gadgets used in [24].)

9.7.1 PSPACE-completeness

Rush Hour Layout. We tile the grid with our vertex gadgets, as shown in Figure 9-19(a). One block (T) is the target, which must be moved to the bottom left corner; it is released when a particular port block slides into a vertex.

Dark-colored blocks represent the "cell walls", which unlike in our sliding-blocks construction are not shared. They are arranged so that they may not move at all. Yellow (light-colored) blocks are "trigger" blocks, whose motion serves to satisfy the vertex constraints. Medium-gray blocks are fillers; some of them may move, but they don't disrupt the vertices' operation.

As in the sliding-blocks construction (Section 9.2), edges are directed inward by sliding blocks out of the vertex gadgets; edges are directed outward by sliding blocks in. The layout ensures that no port block may ever slide out into an adjacent vertex; this helps keep the cell walls fixed.

Lemma 41 *The construction in Figure 9-19(b) satisfies the same constraints as an NCL* AND *vertex, with* A *and* B *corresponding to the* AND *red edges, and* C *to the blue edge.*

Proof: We need to show that C may move down if and only if A first moves left and B first moves down.

Moving A left and B down allows D and E to slide down, freeing F, which releases C. The filler blocks on the right ensure that F may only move left; thus, the inputs are required to move to release the output. □

Lemma 42 *The construction in Figure 9-19(c) satisfies the same constraints as an NCL protected* OR *vertex, with* A *and* B *corresponding to the protected edges.*

Proof: We need to show that C may move down if either A first moves left or B first moves right.

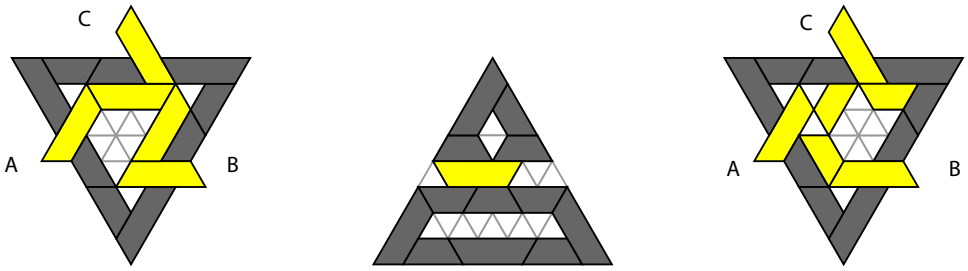
If either A or B slides out, this allows D to slide out of the way of C, as required. Note that we are using the protected OR property: if A were to move right, E down, D right, C down, and B left, we could not then slide A left, even though the OR property should allow this; E would keep A blocked. But in a protected OR, we are guaranteed that A and B will not simultaneously be slid out. □

Graphs. We may use the same constructions here we used for sliding-blocks layouts: 5×5 blocks of Rush Hour vertex gadgets serve to build all the wiring necessary to construct arbitrary planar graphs (Figure 9-11).

In the special case of arranging for the target block to reach its destination, this will not quite suffice; however, we may direct the relevant signal to the bottom left of the grid, and then remove the bottom two rows of vertices from the bottommost 5×5 blocks; these can have no effect on the graph. The resulting configuration, shown in Figure 9-19(a), allows the target block to be released properly.

Theorem 43 *Rush Hour is PSPACE-complete.*

Proof: Reduction from planar Nondeterministic Constraint Logic with protected OR vertices, by the construction described. The output port block of a particular vertex



(a) AND vertex (b) Connector (c) OR vertex

Figure 9-19: Triagonal Slide-Out gadgets.

may move if and only if the corresponding NCL graph edge may be reversed. We direct this signal to the lower left of the grid, where it may release the target block.

Rush Hour is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 12. □

9.7.2 Generalized Problem Bounds

We may consider the more general *Constrained Sliding Block* problem, where blocks need not be 1×2 or 1×3, and may have a constraint direction independent of their dimension. In this context, the existing Rush Hour results do not yet provide a tight bound; the complexity of the problem for 1×1 blocks has not been addressed.

Deciding whether a block may move at all is in P: e.g, we may do a breadth-first search for a movable block that would ultimately enable the target block to move, beginning with the blocks obstructing the target block. Since no block need ever move more than once to free a dependent block, it is safe to terminate the search at already-visited blocks.

Therefore, a straightforward application of our technique cannot show this problem hard; however, the complexity of moving a given block to a given position is not obvious.

Tromp and Cilibrasi [84] provide some empirical indications that minimum-length solutions for 1×1 Rush Hour may grow exponentially with puzzle size.

9.8 Triangular Rush Hour

Rush Hour has also inspired a triangular variant, called Triagonal Slide-Out, playable as an online applet [38]. The rules are the same as for Rush Hour (Section 9.7), except that the game is played on a triangular, rather than square, grid. 40 puzzles are available on the website.

Nondeterministic Constraint Logic gadgets showing Triagonal Slide-Out PSPACE-hard are shown in Figures 9-19 and 9-20. A and B cars represent inactive (outward-directed) input edges; C represents an inactive (inward-directed) output edge. I omit the proof of correctness.

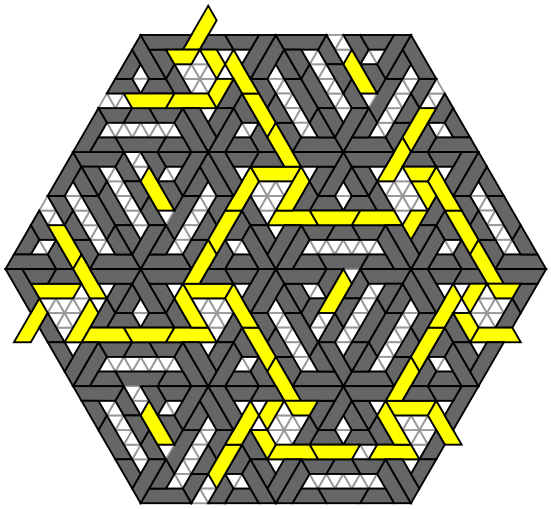


Figure 9-20: How the gadgets are connected together.

One interesting feature of this triangular variant is that while it seems very difficult to build gadgets showing 1×1 Rush Hour hard, it might be much easier to show Triagonal Slide-Out with unit triangular cars hard, because for a unit triangular car to slide one unit, two triangular spaces must be empty: the one it is moving into, and the intervening space with the opposite parity.

9.9 Hinged Polygon Dissections

This section is joint work with Erik Demaine and Greg Frederickson [42].

Properly speaking, this section is not about a generalized combinatorial game, because the problems are geometrical and continuous. Nonetheless, it is a further application of Nondeterministic Constraint Logic. I will simply define the problems and state the results here; I refer the reader to [42] for the detailed reductions.

A hinged dissection of a polygon is a dissection with a set of hinges connecting the pieces, so that they may kinematically reach a variety of configurations in the plane. Hinged polygon dissections have been a staple of recreational mathematics for at least the last century. One well known dissection is shown in Figure 9-21; this is Dudeney's [19] hinged dissection of a triangle to a square. Recently there has been an entire book [28] dedicated to hinged dissections.

The most basic problem is, given two configurations of a hinged polygon dissection, is it kinematically possible to go from one to the other? This problem and others are formalized as follows.

Terminology. We define a *piece* as an instance of a polygon. A *dissection* is a set of pieces. A *configuration* is an embedding of a dissection in the plane, such that only the boundaries of pieces may overlap. The *shape* of a configuration is the set of points, including the boundaries of pieces, that it occupies.

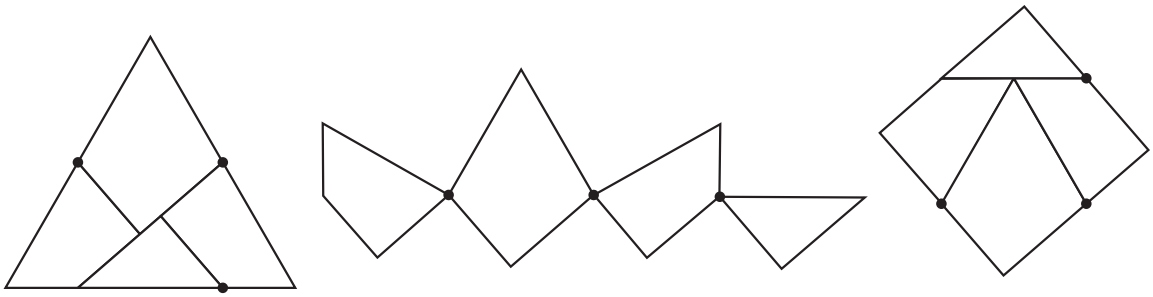


Figure 9-21: Dudeney's hinged triangle-to-square dissection.

A *hinge point* of a piece is a given point on the boundary of the piece. A *hinge* for a set of pieces is a set of hinge points, one for each piece in the set. Given a dissection and a set of hinges, two pieces are *hinge-connected* if either they share a hinge or there is another piece in the set to which both are hinge-connected. A *hinging* of a dissection is a set of hinges such that all pieces in the dissection are hinge-connected.

A *hinged configuration* of a dissection and a hinging is a configuration that, for each hinge, collocates all hinge points of that hinge. A *kinematic solution* of a dissection, a hinging, and two hinged configurations is a continuous path from one configuration to the other through the space of hinged configurations. A *hinged dissection* of two polygons is a dissection and a hinging such that there are hinged configurations of the same shape as the polygons and there is a kinematic solution for the dissection, hinging, and hinged configurations. A hinged dissection of more than two polygons is similarly defined.

Decision questions. All of the hardness results are for problems of the form "is there a kinematic solution to..."; the difference is what we're given in each case. We are always given a dissection. We may or may not be given a hinging. We may be given two configurations, one configuration and one shape, or two shapes. The shapes may or may not be required to be convex. The pieces may or may not be required to be convex. We will always require that the configurations form polygonal shapes.

For each question, all of the information we are not given will form the desired answer. For example, if we are given a dissection, a hinging, and two shapes, the question is whether there exist satisfying configurations A and B forming the shapes, and a kinematic solution S from A to B.

The brief statement of our results is that if we are given a hinging, then all such questions are PSPACE-hard; if we are not given a hinging, but are given two configurations, then determining whether there is a hinging admitting a kinematic solution is PSPACE-hard.

One obvious question we do not address is whether, given polygons A and B, there is a hinged dissection of $\{A, B\}$. But it is an open question whether such a solution exists for any pair of equal-area polygons [20]; thus, it is conceivable that the answer is always *yes*. Therefore, we can say nothing about the complexity of this decision question.

9.10 Additional Results

In this section I mention some additional results, for which I was not the primary contributor.

9.10.1 Push-2F

This section is joint work with Erik Demaine and Michael Hoffman [18].

Push-2F is a kind of pushing-block puzzle similar to the classic Sokoban. The puzzle consists of unit square blocks on an integer lattice; some of the blocks are movable. A mobile "pusher" block may move horizontally and vertically in order to reach a specified goal position, and may push up to two blocks at once. Unlike Sokoban, the goal is merely to get the pusher to a target location. This simpler goal makes constructing gadgets more difficult; there is less relevant state to work with.

We give a reduction from Nondeterministic Constraint Logic showing Push-2F is PSPACE-complete.

9.10.2 Dyson Telescope Game

This section is joint work with Erik Demaine, Martin Demaine, Rudolf Fleischer, Timo von Oertzen [15]. It is included as an example of a problem which seemingly should have yielded to an attack with Nondeterministic Constraint Logic, but did not. It was eventually proven PSPACE-complete by a fairly complex reduction from Quantified Boolean Formulas.

The Dyson Telescope game is a computer game, originally developed by the Dyson company to advertise a vacuum cleaner called the "Telescope".

The goal of the game is to maneuver a ball on a square grid from a starting position to a goal position, by extending and retracting telescopes on the grid. In addition to the ball, the grid contains a number of *telescopes*, each pointing in a given direction (up, right, down, left), and able to extend a given number of spaces.

Each telescope can be in either an *extended* or a *retracted* state. Initially, all telescopes are retracted. A move is made by changing the state of a telescope. If a telescope extends, it will expand in its direction until it reaches its full length, or until it is blocked by another telescope. If the telescope extends through a space the ball occupies, it pushes the ball to the next space, unless this next space is occupied. In that case, the telescope is also considered blocked and will stop. If a telescope retracts, it retracts all the way until it occupies only its base space. If the telescope end touches the ball when retracting, it pulls the ball with it.

We prove that it is PSPACE-complete to determine whether a given problem instance has a series of telescope movements that moves the ball from a starting position to a goal position.

Chapter 10

Two-Player Games

In this chapter I present three new results for two-player games: Amazons, Konane, and Cross Purposes. Amazons is a relatively new game, less than 20 years old, but it has received a considerable amount of study. Konane is a very old game, hundreds of years old at least; it has received some study, but not as much as Amazons. And Cross Purposes is a brand new game; I believe this thesis is the first work to address the problem.

10.1 Amazons

Amazons was invented by Walter Zamkauskas in 1988. Both human and computer opponents are available for Internet play, and there have been several tournaments, both for humans and for computers.

Amazons has several properties which make it interesting for theoretical study. Like Go, its endgames naturally separate into independent subgames; these have been studied using combinatorial game theory [3, 76]. Amazons has a very large number of moves available from a typical position, even more than in Go. This makes straightforward search algorithms impractical for computer play. As a result, computer programs need to incorporate more high-level knowledge of Amazons strategy [56, 54].

Buro [6] showed that playing an Amazons endgame optimally is NP-complete, leaving the complexity of the general game open.[1] I show that generalized Amazons is PSPACE-complete [44, 45], by a reduction from bounded planar Two-Player Constraint Logic (2CL).

[1] Furtak, Kiyomi, Uno, and Buro independently showed Amazons to be PSPACE-complete at the same time as the author [30]. Curiously, [30] already contains two different PSPACE-completeness proofs: one reduces from Hex, and the other from Generalized Geography. The paper is the result of the collaboration of two groups which had also solved the problem independently, then discovered each other. Thus, after remaining an open problem for many years, the complexity of Amazons was solved independently and virtually simultaneously by three different groups, using three completely different approaches, each of which leverages different aspects of the game to construct gadgets.

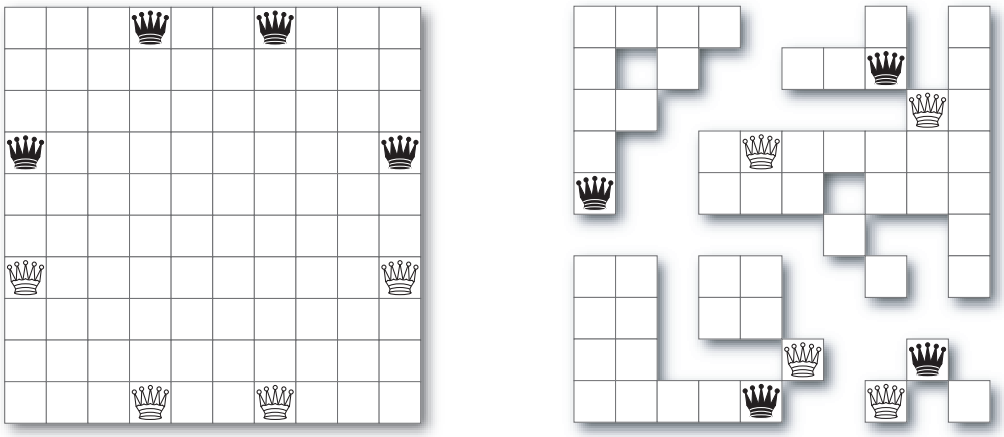


Figure 10-1: Amazons start position and typical endgame position.

Amazons Rules. Amazons is normally played on a 10×10 board. The standard starting position, and a typical endgame position, are shown in Figure 10-1. (I indicate burned squares by removing them from the figures, rather than marking them with tokens.) Each player has four *amazons*, which are immortal chess queens. White plays first, and play alternates. On each turn a player must first move an amazon, like a chess queen, and then fire an *arrow* from that amazon. The arrow also moves like a chess queen. The square that the arrow lands on is burned off the board; no amazon or arrow may move onto or across a burned square. There is no capturing. The first player who cannot move loses.

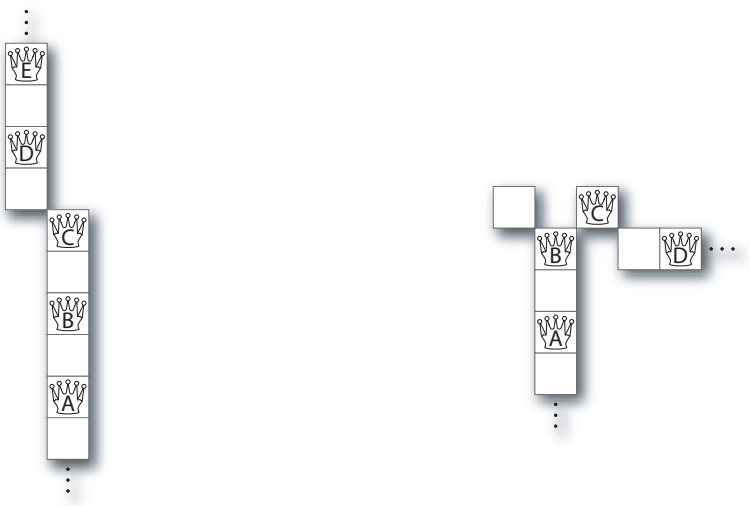
Amazons is a game of mobility and control, like Chess, and of territory, like Go. The strategy involves constraining the mobility of the opponent's amazons, and attempting to secure large isolated areas for one's own amazons. In the endgame shown in Figure 10-1, Black has access to 23 spaces, and with proper play can make 23 moves; White can also make 23 moves. Thus from this position, the player to move will lose.

10.1.1 PSPACE-completeness

I reduce from the alternate vertex set for planar bounded 2CL, in Section 6.1.3. This requires AND, OR, FANOUT, CHOICE, and variable gadgets.

Basic Wiring. Signals propagate along *wires*, which will be necessary to connect the vertex gadets. Figure 10-2(a) shows the construction of a wire. Suppose that amazon A is able to move down one square and shoot down. This enables amazon B to likewise move down one and shoot down; C may now do the same. This is the basic method of signal propagation. When an amazon moves backward (in the direction of input, away from the direction of output) and shoots backward, I will say that it has *retreated*.

Figure 10-2(a) illustrates two additional useful features. After C retreats, D may



(a) Wire, parity, flow limiter (b) Turn, one way

Figure 10-2: Amazons wiring gadgets.

retreat, freeing up E. The result is that the position of the wire has been shifted by one in the horizontal direction. Also, no matter how much space is freed up feeding into the wire, D and E may still only retreat one square, because D is forced to shoot into the space vacated by C.

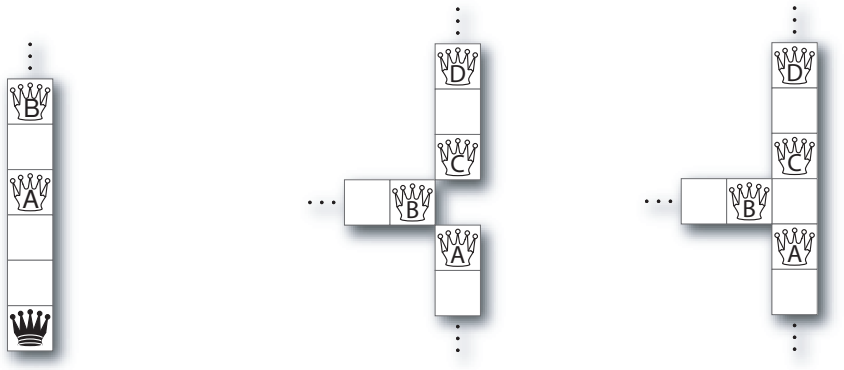
Figure 10-2(b) shows how to turn corners. Suppose A, then B may retreat. Then C may retreat, shooting up and left; D may then retreat. This gadget also has another useful property: signals may only flow through it in one direction. Suppose D has moved and shot right. C may then move down and right, and shoot right. B may then move up and right, but it can only shoot into the square it just vacated. Thus, A is not able to move up and shoot up.

By combining the horizontal parity-shifting in Figure 10-2(a) with turns, we may direct a signal anywhere we wish. Using the unidirectional and flow-limiting properties of these gadgets, we can ensure that signals may never back up into outputs, and that inputs may never retreat more than a single space.

Variable, AND, OR, CHOICE. The *variable* gadget is shown in Figure 10-3(a). If White moves first in a variable, he can move A down, and shoot down, allowing B to later retreat. If Black moves first, he can move up and shoot up, preventing B from ever retreating.

The AND and OR gadgets are shown in Figures 10-3(b) and 10-3(c). In each, A and B are the inputs, and D is the output. Note that, because the inputs are protected with flow limiters (Figure 10-2(a)), no input may retreat more than one square; otherwise the AND might incorrectly activate.

In an AND gadget, no amazon may usefully move until at least one input retreats. If B retreats, then a space is opened up, but C is unable to retreat there; similarly if just A retreats. But if both inputs retreat, then C may move down and left, and



(a) Variable (b) AND (c) OR, CHOICE

Figure 10-3: Amazons logic gadgets.

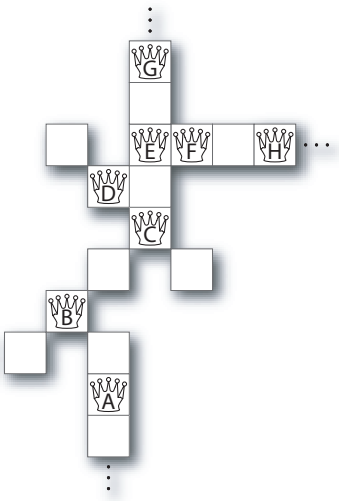


Figure 10-4: Amazons FANOUT gadget.

shoot down and right, allowing D to retreat.

Similarly, in an OR gadget, amazon D may retreat if and only if either A or B first retreats.

The existing OR gadget also suffices as a CHOICE gadget, if we reinterpret the bottom input as an output: if if B retreats, then either C or A, but not both, may retreat.

FANOUT. Implementing a FANOUT in Amazons is a bit trickier. The gadget shown in Figure 10-4 accomplishes this. A is the input; G and H are the outputs. First, observe that until A retreats, there are no useful moves to be made. C, D, and F may not move without shooting back into the square they left. A, B, and E may move one unit and shoot two, but nothing is accomplished by this. But if A retreats, then the following sequence is enabled: B down and right, shoot down; C down and left two,

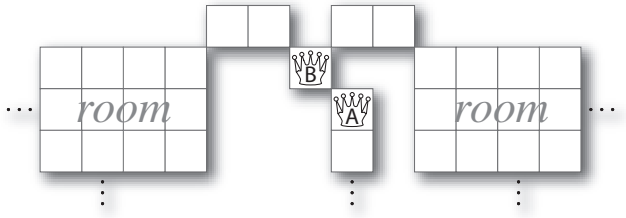


Figure 10-5: Amazons victory gadget.

shoot down and left; D up and left, shoot down and right three; E down two, shoot down and left; F down and left, shoot left. This frees up space for G and H to retreat, as required.

Winning. We will have an AND gadget whose output may be activated only if the white target edge in the 2CL game can be reversed; we need to arrange for White to win if he can activate this AND. We feed this output signal into a *victory* gadget, shown in Figure 10-5. There are two large rooms available. The sizes are equal, and such that if White can claim both of them, he will win, but if he can claim only one of them, then Black will win; we give Black an additional room with a single Amazon in it with enough moves to ensure this property.

If B moves before A has retreated, then it must shoot so as to block access to one room or the other; it may then enter and claim the accessible room. If A first retreats, then B may move up and left, and shoot down and right two, leaving the way clear to enter and claim the left room, then back out and enter and claim the right room.

Theorem 44 *Amazons is PSPACE-complete.*

Proof: Given a bounded planar 2CL graph of the form described in Section 6.1.3, we construct a corresponding Amazons position, as described above. The reduction may be done in polynomial time: if there are k variables and l clauses, then there need be no more than $(kl)^2$ crossover gadgets to connect each variable to each clause it occurs in; all other aspects of the reduction are equally obviously polynomial.

As described, White can win the Amazons game if and only if he can win the corresponding 2CL game, so Amazons is PSPACE-hard. Since the game must end after a polynomial number of moves, it is possible to perform a search of all possible move sequences using polynomial space, thus determining the winner. Therefore, Amazons is also in PSPACE, and thus PSPACE-complete. □

10.2 Konane

Konane is an ancient Hawaiian game, with a long history. Captain Cook documented the game in 1778, noting that at the time it was played on a 14×17 board. Other sizes were also used, ranging from 8×8 to 13×20. The game was usually played with

pieces of basalt and coral, on stone boards with indentations to hold the pieces. King Kamehameha the Great was said to be an expert player; the game was also popular among all classes of Hawaiians.

More recently, Konane has been the subject of combinatorial game-theoretic analysis [21, 7]. Like Amazons, its endgames break into independent games whose values may be computed and summed. However, as of this writing, even $1 \times n$ Konane has not been completely solved, so it is no surprise that complicated positions can arise. I show the general problem to be PSPACE-complete.

Konane Rules. Konane is played on a rectangular board, which is initially filled with black and white stones in a checkerboard pattern. To begin the game, two adjacent stones in the middle of the board or in a corner are removed. Then, the players take turns making moves. Moves are made as in peg solitaire—indeed, Konane may be thought of as a kind of two-player peg solitaire. A player moves a stone of his color by jumping it over a horizontally or vertically adjacent stone of the opposite color, into an empty space. Stones so jumped are captured, and removed from play. A stone may make multiple successive jumps in a single move, as long as they are in a straight line; no turns are allowed within a single move. The first player unable to move wins.

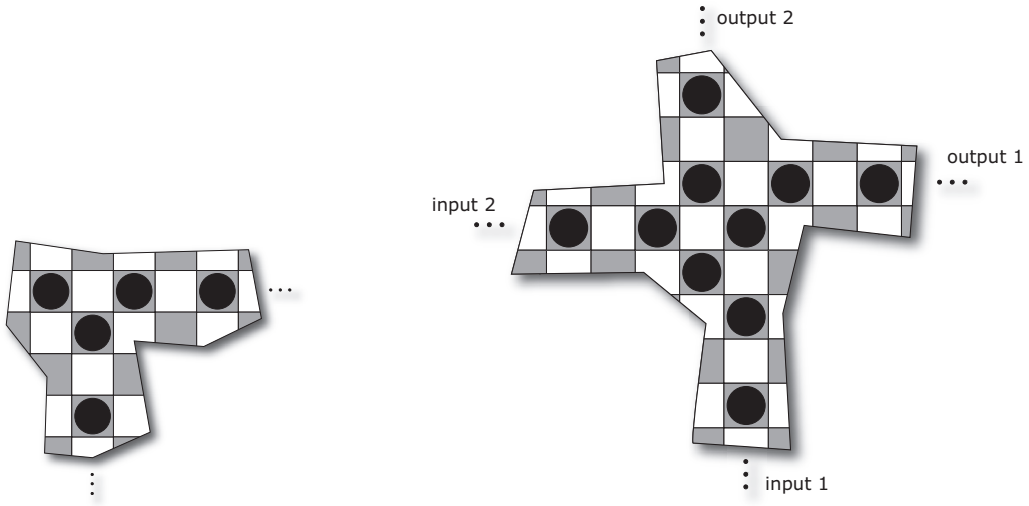
10.2.1 PSPACE-completeness

The Konane reduction is similar to the Amazons reduction; the Konane gadgets are somewhat simpler. As before, the reduction is from the alternate vertex set for planar bounded 2CL, in Section 6.1.3. Therefore, we need AND, OR, FANOUT, CHOICE, and variable gadgets.

Also as in the Amazons reduction, if White can win the Constraint Logic game then he can reach a large supply of extra moves, enabling him to win. Black is supplied with enough extra moves of his own to win otherwise.

Basic Wiring. Wiring is needed to connect the vertex gadgets together. A Konane wire is simply a string of alternating black stones and empty spaces. By capturing the black stones, a white stone traverses the wire. Note that in the Amazons reduction, signals propagate by Amazons moving backwards; in Konane, signals propagate by stones moving forwards, capturing opposing stones.

Turns are enabled by adjoining wires as shown in Figure 10-6(a); at the end of one wire, the white stone comes to rest at the beginning of another, protected from capture by being interposed between two black stones. If the white stone tried to traverse the turn in the other direction, it would not be so protected, and Black could capture it. Thus, as in the Amazons reduction, the turn is also a one-way device, and we assume that gadget entrances and exits are protected by turns to ensure that signals can only flow in the proper directions.



(a) Wire, turn (b) Conditional

Figure 10-6: Konane wiring gadgets.

Conditional Gadget. A single gadget serves the purpose of AND, FANOUT, and positional parity adjustment. It has two input/output pathways, with the property that the second one may only be used if the first one has already been used. This *conditional gadget* is shown in Figure 10-6(b); the individual uses are outlined below.

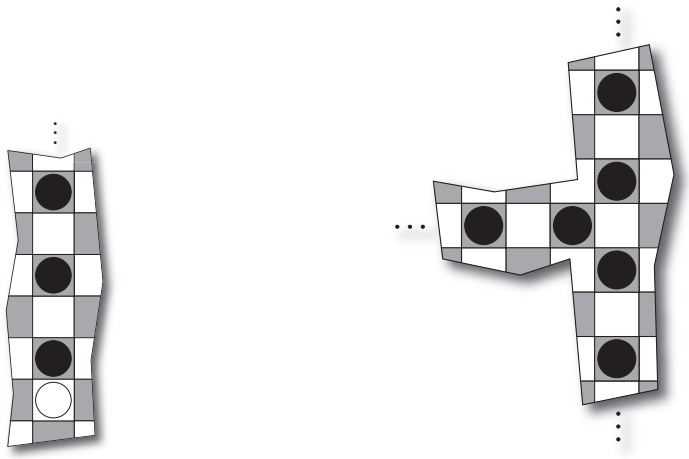
Observe that a white stone arriving at input 1 may only leave via output 1, and likewise for input 2 and output 2. However, if White attempts to use pathway 2 before pathway 1 has been used, Black can capture him in the middle of the turn. But if pathway 1 has been used, the stone Black needs to make this capture is no longer there, and pathway 2 opens up.

FANOUT, Parity. If we place a white stone within the wire feeding input 2 of a conditional gadget, then both outputs may activate if input 1 activates. This splits the signal arriving at input 1.

If we don't use output 1, then this FANOUT configuration also serves to propagate a signal from input 1 to output 2, with altered positional parity. This enables us to match signal parities as needed at the gadget inputs and outputs.

Variable, AND, OR, CHOICE. The variable gadget consists of a white stone at the end of a wire, as in Figure 10-7(a). If White moves first in a variable, he can traverse the wire, landing safely at an adjoining turn. If Black moves first, he can capture the white stone and prevent White from ever traversing the wire.

The AND gadget is a conditional gadget with output 1 unused. By the properties of the conditional gadget, a white stone may exit output 2 only if white stones have arrived at both inputs. The OR gadget is shown in Figure 10-7(b). The inputs are on the bottom and left; the output is on the top. Clearly, a white stone arriving via either input may leave via the output.



(a) Variable (b) OR, CHOICE

Figure 10-7: Konane variable, OR, and CHOICE gadgets.

As was the case with Amazons, the OR gadget also suffices to implement CHOICE, if we relabel the bottom input as an output: a white stone arriving along the left input may exit via either the top or the bottom.

Winning. We will have an AND gadget whose output may be activated just when White can win the given Constraint Logic game. We feed this signal into a long series of turns, providing White with enough extra moves to win if he can reach them. Black is provided with his own series of turns, made of white wires, with a single black stone protected at the end of one of them, enabling Black to win if White cannot activate the final AND.

Theorem 45 *Konane is PSPACE-complete.*

Proof: Given a bounded planar 2CL graph of the form described in Section 6.1.3, we construct a corresponding Konane position, as described above. As in the Amazons construction, the reduction is clearly polynomial. Also as in Amazons, White may reach his supply of extra moves just when he can win the Constraint Logic game.

Therefore, a player may win the Konane game if and only if he may win the corresponding Constraint Logic game, and Konane is PSPACE-hard. As before, Konane is clearly also in PSPACE, and therefore PSPACE-complete. □

10.3 Cross Purposes

Cross Purposes was invented by Michael Albert, and named by Richard Guy, at the Games at Dalhousie III workshop, in 2004. It was introduced to the author by Michael Albert at the 2005 BIRS Combinatorial Game Theory Workshop. Cross Purposes is a kind of two-player version of the popular puzzle Tipover, which is NP-complete

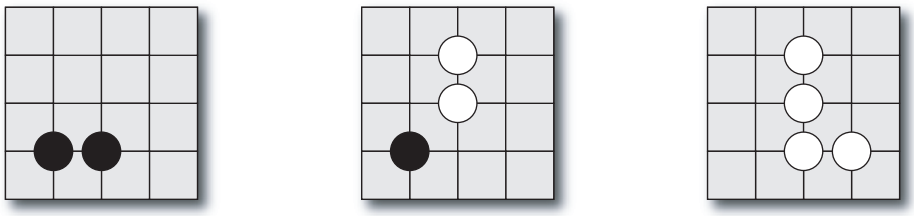


Figure 10-8: An initial Cross Purposes configuration, and two moves.

(Section 9.1; [41]). From the perspective of Combinatorial Game Theory [4, 9], in which game positions have values that are a generalization of numbers, Cross Purposes is fascinating because its positions can easily represent many interesting combinatorial game values.

Cross Purposes Rules. Cross Purposes is played on the intersections of a Go board, with black and white stones. In the initial configuration, there are some black stones already on the board. A move consists of replacing a black stone with a pair of white stones, placed in a row either directly above, below, to the left, or to the right of the black stone; the spaces so occupied must be vacant for the move to be made. See Figure 10-8. The idea is that a stack of crates, represented by a black stone, has been tipped over to lie flat. Using this idea, we describe a move as *tipping* a black stone in a given direction.

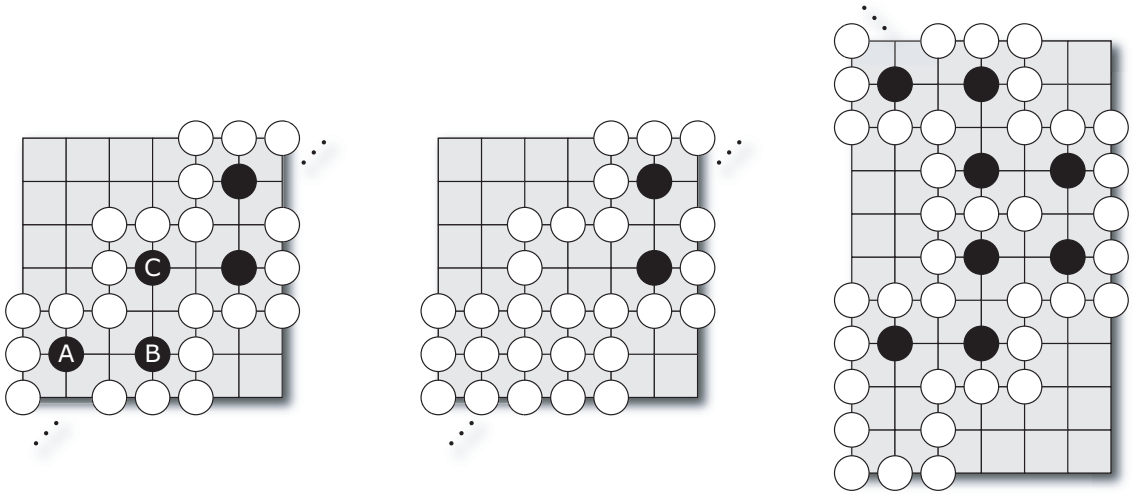
The players are called *Vertical* and *Horizontal*. Vertical moves first, and play alternates. Vertical may only move vertically, up or down; Horizontal may only move horizontally, left or right. All the black stones are available to each player to be tipped, subject to the availability of empty space. The first player unable to move loses.

I give a reduction from planar Bounded Two-Player Constraint Logic showing that Cross-Purposese is PSPACE-complete.

10.3.1 PSPACE-completeness

The Cross Purposes construction largely follows those used for Amazons and Konane. To reduce from Bounded Two-Player Constraint Logic, we need AND, OR, FANOUT, CHOICE, and variable gadgets, and a way to wire them together into arbitrary graphs.

One new challenge in constructing the gadgets is that each player may only directly move either horizontally or vertically, but not both. Yet, for formula game gadgets to work, one player must be able to direct signals two dimensionally. We solve this problem by restricting the moves of Horizontal so that, after the variable selection phase, his possible moves are constrained so as to force him to cooperate in Vertical's signal propagation. (We assume that the number of variables is even, so that it will be Vertical's move after the variable selection phase.) An additional challenge is that a single move can only empty a single square, enabling at most one more move to be made, so it is not obviously possible to split a signal. Again, we use the interaction



(a) Wire (b) Wire, after three moves (c) Turn, free input

Figure 10-9: Cross Purposes wiring.

of the two players to solve this problem.

We do not need a supply of extra moves at the end, as used for Amazons and Konane; instead, if Vertical can win the formula game, and correspondingly activate the final AND gadget, then Horizontal will have no move available, and lose. Otherwise, Vertical will run out of moves first, and lose.

Basic Wiring. We need wiring gadgets to connect the vertex gadgets together into arbitrary graphs. Signals flow diagonally, within surrounding corridors of white stones. A *wire* is shown in Figure 10-9(a). Suppose that Vertical tips stone A down, and suppose that Horizontal has no other moves available on the board. Then his only move is to tip B left. This then enables Vertical to tip C down. The result of this sequence is shown in Figure 10-9(b).

The turn gadget is shown in Figure 10-9(c); its operation is self-evident. Also shown in Figure 10-9(c) is a *free input* for Vertical: he may begin to activate this wire at any time. We will need free inputs in a couple of later gadgets.

Conditional Gadget. As with Konane (Section 10.2), a single *conditional gadget*, shown in Figure 10-10, serves the role of FANOUT, parity adjustment, and AND. A signal arriving along input 1 may only leave via output 1, and likewise for input 2 and output 2; these pathways are ordinary turns embedded in the larger gadget. However, if Vertical attempts to use pathway 2 before pathway 1 has been used, then after he tips stone A down, Horizontal can tip stone B left, and Vertical will then have no local move. But if pathway 1 has already been used, stone B is blocked from this move by the white stones left behind by tipping C down, and Horizontal has no choice but to tip stone D right, allowing Vertical to continue propagating the signal along pathway 2.

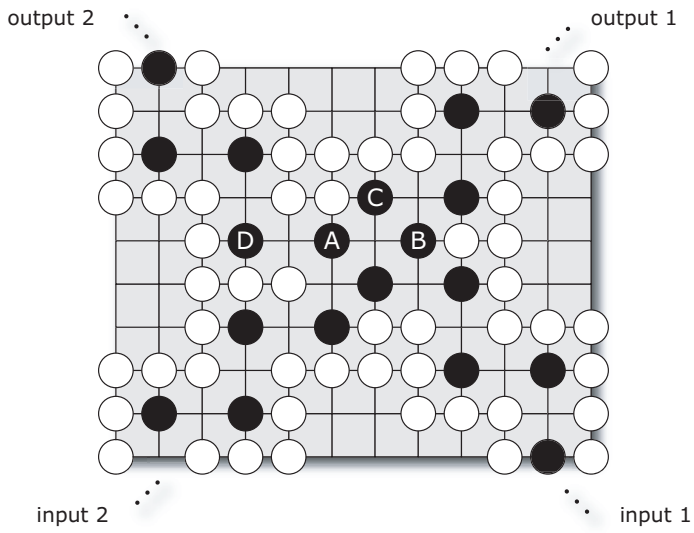


Figure 10-10: Cross Purposes conditional gadget.

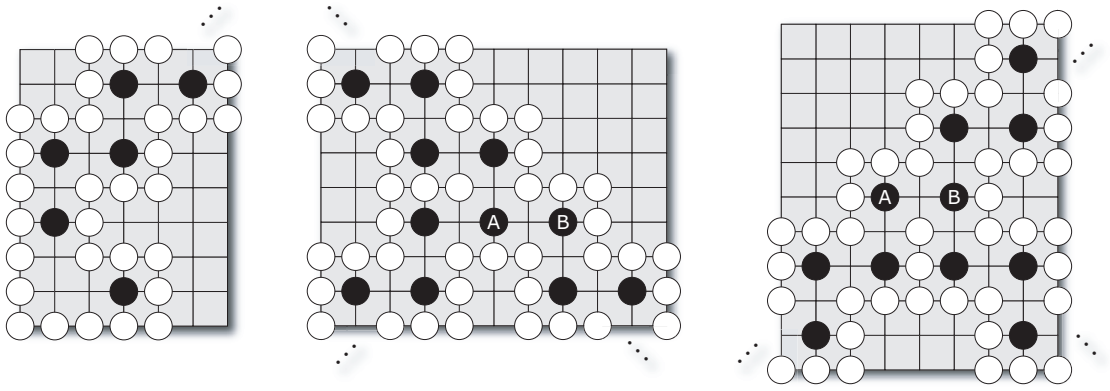
FANOUT, Parity, AND. As with Konane, if we give Vertical a free input to the wire feeding input 2 of a conditional gadget, then both outputs may activate if input 1 activates. This splits the signal arriving at input 1.

If we don't use output 1, then this FANOUT configuration also serves to propagate a signal from input 1 to output 2, with altered positional parity. This enables us to match signal parities as needed at the gadget inputs and outputs. We must be careful with not using outputs, since we need to ensure that Vertical has no free moves anywhere in the construction; unlike in the constructions for Amazons and Konane, in Cross Purposes, there is no extra pool of moves at the end, and every available move within the layout counts. However, blocking an output is easy to arrange; we just terminate the wire so that Horizontal has the last move in it. Then Vertical gains nothing by using that output.

The AND gadget is a conditional gadget with output 1 unused. By the properties of the conditional gadget, output 2 may activate only if both inputs have activated.

Variable, OR, CHOICE. The *variable* gadget is shown in Figure 10-11(a). If Vertical moves first in a variable, he can begin to propagate a signal along the output wire. If Horizontal moves first, he will tip the bottom stone to block Vertical from activating the signal.

The OR gadget is shown in Figure 10-11(b). The inputs are on the bottom; the output is on the top. Whether Vertical activates the left or the right input, Horizontal will be forced to tip stone A either left or right, allowing Vertical to activate the output. Here we must again be careful with available moves. Suppose Vertical has activated the left input, and the output, of an OR. Now what happens if he later activates the right input? After he tips stone B down, Horizontal will have no move; he will already have tipped stone A left. This would give Vertical the last move even



(a) Variable (b) OR (c) CHOICE

Figure 10-11: Cross Purposes variable, OR, and CHOICE gadgets.

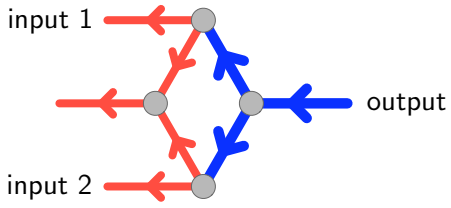


Figure 10-12: Protected OR.

if he were unable to activate the final AND gadget; therefore, we must prevent this from happening. We will show how to do so after describing the CHOICE gadget.

As with Amazons and Konane, the existing OR gadget suffices to implement CHOICE, if we reinterpret it. This time the gadget must be rotated. The rotated version is shown in Figure 10-11(c). The input is on the left, and the outputs are on the right. When Vertical activates the input, and tips stone A down, Horizontal must tip stone B left. Vertical may then choose to propagate the signal to either the top or the bottom output; either choice blocks the other.

Protecting the OR Inputs. As mentioned above, we must ensure that only one input of an OR is ever able to activate, to prevent giving Vertical extra moves. We do so with the graph shown in Figure 10-12. Vertical is given a free input to a choice gadget, whose output combines with one of the two OR input signals in an AND gadget. Since only one choice output can activate, only one AND output, and thus one OR input, can activate. Inspection of the relevant gadgets shows that Vertical has no extra moves in this construction; for every move he can make, Horizontal has a response. (This construction is analogous to the protected OR used in Nondeterministic Constraint Logic (Section 5.2.3), and should logically be added to the Two-Player Constraint Logic formalism.)

Winning. We will have an AND gadget whose output may be activated only if the White player can win the corresponding Constraint Logic game. We terminate its output wire with Vertical having the final move. If he can reach this output, Horizontal will have no moves left, and lose. If he cannot, then since Horizontal has a move in reply to every Vertical move within all of the gadgets, Vertical will eventually run out of moves, and lose.

Theorem 46 *Cross Purposes is PSPACE-complete.*

Proof: Given a planar Bounded Two-Player Constraint Logic graph, we construct a corresponding Cross Purposes position, as described above. The reduction is clearly polynomial. Vertical may activate a particular AND output, and thus gain the last move, just when he can win the Constraint Logic game.

Therefore, and Cross Purposes is PSPACE-hard. As with Amazons and Konane, Cross Purposes is clearly also in PSPACE, and therefore PSPACE-complete. □

Chapter 11

Open Problems

In this chapter I list some games whose complexity appears to be unknown, or to have some interesting aspect which is open. In general I do not formally define each game.

Lunar Lockout. Lunar Lockout^{TM} is a puzzle made by ThinkFun. It is a kind of sliding-block puzzle, with only 1×1 blocks (robots). However, the robots, when slid, are required to slide until they reach another block, where they stop. A robot cannot slide if there is not another one in its path to stop it. The goal is to get a specified robot a to specified place. This is an unbounded puzzle, and so potentially PSPACE-complete. But the fact that the blocks are 1×1 makes it very difficult to build gadgets, as is also the case with 1×1 Rush Hour, below. However, there is some hope for a reduction from Nondeterministic Constraint Logic to Lunar Lockout; the challenges with 1×1 Rush Hour appear to be more severe.

There is a paper by Hartline and Libeskind-Hadas [39] purporting to show that Lunar Lockout is PSPACE-complete. However, the authors changed the nature of the game dramatically by adding the notion of fixed blocks. These are not in the actual game, and not part of the natural generalization, which would just be to play on an $n \times n$ board. The result is valid for the problem as defined, which is worthy of study, but it is misleading to refer to it as Lunar Lockout. It is also much easier to construct gadgets when fixed blocks are added; a big part of the challenge in Lunar Lockout is making any sort of structure at all which is stable.

1×1 **Rush Hour.** In Section 9.7, I raised the problem of Rush Hour with all the blocks 1×1. There is some empirical evidence due to Tromp and Cilibrasi [84] that the minimum solution length for these puzzles grows exponentially with the puzzle size, so it looks as if the puzzle could be hard. However, merely moving a single car is in P, so a direct application of Nondeterministic Constraint Logic will not work. This problem seems to straddle the line between easy and hard problems, and as a result is very tantalizing.

Subway Shuffle. Subway Shuffle is a generalization of 1×1 Rush Hour along a different dimension. It is played on a graph with colored edges (subway lines) and

colored tokens (subway cars) occupying some of the vertices. A token may slide from one vertex to another if the destination is empty, and the edge color matches the token color. The challenge is again to get a given token to a given location. As with 1×1 Rush Hour, it is in P to merely move a token, so again a straightforward application of Nondeterministic Constraint Logic will not work.

The extra freedom available in constructing configurations, due to the relaxation of the grid graph constraint, and the addition of arbitrarily many colors, ought to make it easier to build some sort of gadget in Subway Shuffle than it is in 1×1 Rush Hour. Therefore, it is even more frustrating and tantalizing that no hardness proof has been found.

Phutball. Phutball [4] is a game played on a go board, with one black stone, the ball, initially placed on the center of the board. On his turn, a player may either place a white stone, or make a series of jumps. A jump is made by jumping the ball over a contiguous line of stones, horizontally, vertically, or diagonally, into an empty space. The white stones jumped are removed before the next jump. A player may make as many jumps as he wishes on a single turn. The game is won by jumping the ball onto or over the goal line. Left's goal line is the right edge of the board, and Right's is the left edge of the board.

This game has the unusual property that it is NP-complete merely to determine whether there is a single-move win [16]! The complexity of determining the winner from an arbitrary position is unknown. As an unbounded, two-player game of perfect information, it could be as hard as EXPTIME-complete. But the nature of the game makes it extremely difficult to construct any sort of stable gadget.

Retrograde Chess. Given two configurations of chess pieces in a generalized $n \times n$ board, is it possible to play from one configuration to the other if the players cooperate? This problem is known to be NP-hard [5]; is it PSPACE-complete?

Dots and Boxes. This well-known game has an open complexity. A generalized version is known to be NP-hard [4], but as a two-player, bounded game, by all rights it should be PSPACE-complete.

Minesweeper. Minesweeper is a computer game in which the player attempts to clear a grid of mines. Clicking on a square loses if the square contains a mine; otherwise, the surrounding eight squares are exposed. When a square is exposed, it displays either a mine, or a number from one to eight, indicating how many mines are orthogonally or diagonally adjacent. By clever reasoning, a player tries to deduce which unexposed squares are safe, and plays on them to expose them and gain information.

Richard Kaye showed that Minsweeper is NP-complete in 2000 [50]. This result drew quite a lot of attention, as it tangibly connected theoretical computer science to a game everybody had played.

However, Minesweeper is *not* NP-complete!

Of course, I have to clarify that statement. What I mean is, the "natural decision question" about Minesweeper is not NP-complete. Kaye asked a different question, and showed that it was NP-complete. In this thesis, and more generally, I argue, the natural decision question for any puzzle is, "from a given (legal) position, is the puzzle solvable?". But Kaye asked instead, "given a position (partial set of exposed squares), is there a consistent placement for the mines?" This is of course an interesting question as well, but by saying that a given puzzle is NP-complete, the decision question ought to be whether the puzzle is solvable.

I have a result that Minesweeper is actually coNP-hard, and if we can assume that we are given a valid (consistent) configuration in the decision question, then it is coNP-complete. Unfortunately I did not have time to finish writing up this result for this thesis; it will appear later. The reduction is from TAUTOLOGY.

Another interesting decision question about Minesweeper is, from a given position, is the probability of winning by careful play greater than p? This seems like it could conceivably be PSPACE-complete.

Go. Go with Japanese rules has been "solved", from a complexity standpoint, for 23 years: it is EXPTIME-complete [68]. (Newer results have shown restricted configurations PSPACE-hard [12, 86].) It is rather remarkable, therefore, that the complexity of Go with the addition of the superko rule, as used for example in China and in the USA, is still unresolved. In fact, both the upper *and* the lower bounds of Robson's EXPTIME-compleness proof fail when superko is added! All that is known is that it is PSPACE-hard [53], and in EXPSPACE [69].

Robson, and others who have studied the problem (notably John Tromp), are evidently of the opinion that go with superko is probably in EXPTIME. However, as an unbounded, no-repeat, two-player game, it "ought" to be EXPSPACE-complete. But this may be a case where there is effectively some special structure in the game that makes it easier. The EXPTIME-hardness result builds variable gadgets out of sets of kos (basic repeating patterns). If all dynamic state is to be encoded in kos, then the problem is in fact in EXPTIME, because it is an instance of Generalized Geography on an undirected graph, which is polynomial in the input size. The input size is exponential in this case; it is the space of possible board configurations.

But it may be possible to build gadgets that are not merely sets of kos. It is very difficult to do so; Go is "almost" in PSPACE, because in normal play moves are not reversible, and it is only through capture that there is the possibility of the repeating patterns necessary for a harder complexity.

Bridge. A hand of bridge is a bounded team game with private information. Therefore, determining the winner could potentially be as hard as NEXPTIME. A reduction from bounded Team Private Constraint Logic would appear to be difficult, for two reasons. First, in bounded TPCL, the private information resides in the moves selected; the initial state is known. But in Bridge, the private information resides in

the initial disposition of the cards. Second, there is no natural geometric structure to exploit in Bridge, as there is in a typical board game.

Rengo Kriegspiel. This is a kind of team, blindfold go. Four players sit facing away from each other, two Black, two White. A referee sees the actual game board in the center. The players take turns attempting to make moves. Only the referee sees the attempted moves. The referee announces when a move is illegal, or when the team's own stone is already on the intersection, and when captures are made. (He also removes captured stones from all players' boards.) Players gradually gain knowledge of the position as the game progresses and the board fills up.

This is an unbounded team game with private information, and therefore could potentially be undecidable.[1] It would appear to be extremely difficult to engineer a reduction from Team Private Constraint Logic to Rengo Kriegspiel, showing undecidability, but perhaps it is possible.

[1]Technically, Rengo Kriegspiel as played in the USA should be assumed to use a superko rule by default. This prevents the global pattern from ever repeating, and thus bounds the length of the game, making it decidable. However, we can consider the game without superko.

Chapter 12

Summary of Part II

In Part II of this thesis I have shown very many games and puzzles hard. Some of the proofs were difficult, especially as the proof technique was being developed, but some were very easy, once the proof technique was in place. For example, it took about half an hour to show Konane PSPACE-complete. Yet, in spite of a fair amount of study by combinatorial game theorists, and an important cultural history, no prior complexity results about Konane were known.

It is this kind of result that demonstrates the utility of Constraint Logic. A very large part of the work of reductions has already been done, and often one can simply select the kind of Constraint Logic appropriate for the problem at hand, and the gadgets will almost make themselves.

Most individual game complexity results are not particularly important. There are no game results in this thesis that are surprising, except for the undecidable version of Constraint Logic, and that is an abstract game. But taken as a whole, they demonstrate that the essential nature of games is captured effectively by the notion of Constraint Logic. Furthermore, they lend credence to the idea that a game is always as hard as it "can" be: a bounded two-player game (without any trivial simplifying structure) ought to be PSPACE-complete, for example. Every additional result adds weight to this hypothesis.

Chapter 13

Conclusions

In this section I summarize the contributions I have made in this thesis, and sketch some directions for future research.

13.1 Contributions

In this thesis I have made four important contributions.

First, I have demonstrated a simple, uniform game framework, Constraint Logic, which concisely captures the concept of generalized combinatorial game. A Constraint Logic game consists simply of a sequence of edge reversals in a directed graph, subject to simple constraints. There are natural versions of Constraint Logic for zero-, one-, two-player, and team games, both in bounded- and unbounded-length versions.

Second, I have demonstrated that each of these kinds of Constraint Logic game corresponds to a distinct complexity class, or equivalently, to a distinct kind of resource-bounded computation, ranging from P-complete bounded, zero-player games, through PSPACE-complete unbounded puzzles, and up to undecidable (r.e.-complete) team games. This correspondence is proven by means of seven distinct reductions from Boolean formula games complete for the appropriate class to the corresponding Constraint Logic game. For the undecidable team games, I also demonstrated that the existing Boolean formula game in the literature was in fact decidable, and independently derived a formula game which is actually undecidable.

Third, I have shown that the Constraint Logic game framework makes hardness proofs for actual games and puzzles significantly easier. I have provided very many new proofs, mostly for problems which were either known to be open or previously unaddressed. In a few cases I rederived much simpler hardness proofs than the ones in the literature for games already known to be hard, thus explicitly demonstrating how much more concise and compact reductions from Constraint Logic can be, compared to conventional techniques (such as reducing directly from Satisfiability, Quantified Boolean Formulas, etc.) One key feature of Constraint Logic games that often makes such reductions straightforward is that the hardness results for Constraint Logic apply even when the graphs are planar, across the spectrum of Constraint Logic games (with the single exception of bounded, zero-player Constraint Logic). This means that there

is no need to build "crossover" gadgets—often the most difficult component—in the actual game and puzzle reductions.

Finally, I have made more manifest the deep connection between the notions of game and of computation. Games are a natural generalization of conventional, deterministic computation. But a key difference from ordinary computation is that in a (generalized combinatorial) game, one is always dealing with a finite spatial resource. Any generalized combinatorial game can actually be played, physically, in the real world. But Turing machines, by contrast, are only idealized computers. We can never build a real Turing machine, because we can't make an infinite tape.

The linchpin in this argument for games as computation is the undecidability result for team games with private information. Perfect play in such games is in direct correspondence with arbitrary computation on a Turing machine with an infinite tape. Yet, there are only a finite number of positions in the game. Thus, games represent a fundamentally distinct kind of computation.

13.2 Future Work

One direction for future work is obviously to apply the results here to additional games and puzzles, to show them hard. I listed some candidates in Chapter 11. Some of those games may yield to an application of Constraint Logic; others may not.

It is the ones which will not that are ultimately more interesting, such as 1×1 Rush Hour, and Subway Shuffle, which seem to lie right on the border between east and hard problems. By attempting to hit them with the hammer of Constraint Logic, and observing how they fail to break, more can be learned about the mathematical nature of games.

More generally, there is still the question of, what is a game? I declined to pin down precisely what I meant by a game in Chapter 2, on the grounds that previous statements that all "reasonable" games were no harder than X turned out to change when the concept of game was extended. However, now that we have reached undecidable games, is there anywhere left to go? I think so.

For one thing, there are higher degrees of undecidability. What if we play a game on an infinite board? Deterministic computations on an infinite Turing machine tape are recursively enumerable. Is there any kind of game we can conceive of which is not even recursively enumerable? Such a game would correspond to a *hypercomputation* [11], a "computation" beyond what Turing machines are capable of.

But there is still interesting space to explore in the relatively more pedestrian classes of PSPACE, EXPTIME, EXPSPACE, etc. In particular, it is a bit unsatisfying that one needs to add the concepts of teams and of private information to move beyond ordinary two-player games. Isn't there some way to just add another kind of player? In general, an extra player represents an extra source of nondeterminism, and computational power. The notion of private information is merely another way of introducing an extra source of nondeterminism. In fact, one can explicitly add private information to a one-player game as well; it then becomes, effectively, a two-player game, because we might as well assume an adversary is choosing the private

information. But what is the right way to look at private information in a two or more player game that makes it look like another player?

Similarly, the notion of disallowing repetitions in a game is using a kind of private information: the relevant history of the game is not present in the current configuration. Is there a way to translate that kind of hidden information, or nondeterminism, into another kind of player? It is these kinds of question which continue to fascinate me.

Appendix A

Computational Complexity Reference

This appendix serves as a refresher on computability and complexity, and as a "cheat sheet" for the complexity classes used in this thesis. For more thorough references see, for example, [55], [74], [59], or [49].

The fundamental model of computation used in computer science is the Turing machine. We begin with a definition of Turing machines. Note that there are generally irrelevant differences in the precise form different authors use for defining Turing machines.

A.1 Basic Definitions

Turing Machines. Informally, a Turing machine is a deterministic computing device which has a finite number of states, and a one-way infinite tape in which each tape cell contains a symbol from a finite alphabet. The machine has a scanning head which is positioned over some tape cell. On a time step, the machine writes a new symbol in the currently-scanned cell, moves the head left or right, and enters a new state, all as a function of the current state and the current symbol scanned. If the transition function is ever undefined, the machine halts, and either accepts or rejects the computation, based on whether it halted in the special accepting state.

Formally, a Turing machine is a 6-tuple $(Q, \Gamma, \delta, q_0, b, q_{accept})$ where

Q is a finite set of *states*,
Γ is a finite set of *tape symbols*,
$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, a partial function, is the *transition function*,
$q_0 \in Q$ is the *initial state*,
$b \in \Gamma$ is the *blank symbol*, and
$q_{accept} \in Q$ is the *accepting state*.

A *configuration* is a string in $(Q \cup \Gamma)^*$ containing exactly one symbol from Q. A configuration represents the contents of the tape, the current state, and the current

head position at a particular time; the symbol to the right of the state symbol in the string is the symbol currently scanned by the head. The rest of the tape is empty (filled with blank symbols); configurations identical except for trailing blanks are considered equivalent.

The *next-state relation* $\vdash$ relates configurations separated by one step of the Turing machine. $a \vdash b$ when the head motion, state change, and symbol change following the previous head position from a to b correspond to the transition specified in δ. $\vdash^*$ is the transitive and reflexive closure of $\vdash$.

A Turing machine *halts* on input $\omega \in \Gamma^*$ in configuration x if $q_0\omega \vdash^* x$ and $\neg\exists y\ x \vdash y$. It *accepts* input ω if it halts in configuration x for some x which contains q_{accept}; it *rejects* ω if it halts but does not accept.

A Turing machine M *computes function* f_M if M halts on input ω in configuration $q_{accept}x$, where $x = f_M(\omega)$, for all ω.

Languages. A *language* is a set of strings over some alphabet. The language $\{w \mid M \text{ accepts } w\}$ that a Turing machine M accepts is denoted $L(M)$. If some Turing machine accepts a language L, then L is *Turing-recognizable* (also called *recursively enumerable, or r.e.*).

A language corresponds to a *decision problem*—given a string w, is w in the language?

Decidability. If a Turing machine M halts for every input, then it *decides* its language $L(M)$, and is called a *decider*. If some Turing machine decides a language L, then L is *decidable* (also called *recursive*); otherwise, L is *undecidable*. Note that a Turing machine M which computes a function f_M must be a decider.

One example of an undecidable language is the formal language corresponding to the decision problem, "Given a Turing machine M and input ω, does halt on input ω?". This is called the *halting problem*. A string in the actual language would consist of encodings of M and ω according to some rule.

Complexity. A Turing machine *uses time t on input ω* if it halts on input ω in t steps: $\omega \vdash c_1 \vdash \ldots \vdash c_t$. The *time complexity* of a Turing machine M which is a decider is a function $t(n) =$ the maximum number of steps M uses on any input of length n.

The *time complexity class* TIME($t(n)$) is the set of languages decided by some Turing machine with time complexity in $O(t(n))$.

Space complexity is defined similarly. A Turing machine *uses space s on input ω* if it halts on input ω using configurations with maximum length s (not counting trailing blanks). The *space complexity* of a Turing machine M which is a decider is a function $f(n) =$ the maximum space M uses on any input of length n.

The *space complexity class* SPACE($f(n)$) is the set of languages decided by some Turing machine with space complexity in $O(f(n))$.

We are now ready to define some commonly used complexity classes:

$$\begin{aligned} \text{P} &= \bigcup_k \text{TIME}(n^k), \\ \text{PSPACE} &= \bigcup_k \text{SPACE}(n^k), \\ \text{EXPTIME} &= \bigcup_k \text{TIME}(2^{n^k}) \end{aligned}$$

are the classes of languages decidable in, respectively, polynomial time, polynomial space, and exponential time. Another important class, NP, will have to wait for Section A.2 for definition.

Reducibility. A language L is polynomial-time reducible to language L' if there is a Turing machine M with polynomial time complexity such that $\omega \in L \iff f_M(\omega) \in L'$. That is, membership of a string in L may be tested by computing a polynomial-time function of the string and testing the result in L'.

Completeness. A language L is *hard* for a complexity class X (abbreviated X-*hard*) if every language $L' \in X$ is polynomial-time reducible to L. A language L is *complete* for a complexity class X (abbreviated X-*complete*) if $L \in X$ and L is X-hard.

Intuitively, the languages that are X-complete are the "hardest" languages in X to decide. For example, if every language in PSPACE can be reduced to a language L in polynomial time, then L must be at least as hard as any other language in PSPACE to decide, because one can always translate such a problem in to a membership test for L. The notion of polynomial-time reducibility is used, because a function that can be computed in polynomial time is considered a "reasonable" function.[1]

Note that if a language L is X-hard and L is polynomial-time reducible to language L', then L' is also X-hard. This fact is the basis for most hardness proofs.

A.2 Generalizations of Turing Machines

The basic one-tape, deterministic Turing machine, as defined above, can be enhanced in various ways. For example, one could imagine a Turing machine with multiple read-write tapes, instead of just one. Are such machines more powerful than the basic machine? In this case, any multitape Turing machine M has an equivalent single-tape machine M' that accepts the same language, with at most a quadratic slowdown. Relative to the above complexity classes, they are the same.

[1]However, this definition is only appropriate for classes harder than P, because any language in P is polynomial-time reducible to any other language in P. To define P-completeness appropriately, we need the notion of *log-space reducibility*, which I will not define. See, e.g., [74] for details.

Nondeterminism. One kind of enhancement that seems to increase the power is *nondeterminism*. A *nondeterministic* Turing machine is defined similarly to a deterministic one, except that the transition function δ is allowed to be multivalued:

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{L,R\}}.$$

That is, a nondeterministic transition function specifies an arbitrary set of possible transitions. The above definition of acceptance still works, but the meaning has changed: a nondeterministic Turing machine accepts input ω if there is *any* accepting computation history $q_0\omega \vdash^* x$. Thus, a nondeterministic computer is allowed to nondeterministically "guess" the sequence of transitions needed to accept its input. If there is no such sequence, then it rejects.

Whether nondeterminism actually increases the power of Turing machines is a very important unresolved question [73].

Nondeterministic Complexity. By analogy with the above definitions, we can define time- and space- complexity classes for nondeterministic Turing machines.

The *nondeterministic time complexity class* NTIME($t(n)$) is the set of languages decided by some nondeterministic Turing machine with time complexity in $O(t(n))$. The *nondeterministic space complexity class* NSPACE($f(n)$) is the set of languages decided by some nondeterministic Turing machine with space complexity in $O(f(n))$.

We may now define some additional complexity classes:

$$\begin{aligned} \text{NP} &= \bigcup_k \text{NTIME}(n^k), \\ \text{NPSPACE} &= \bigcup_k \text{NSPACE}(n^k) \end{aligned}$$

are the classes of languages decidable in, respectively, nondeterministic polynomial time and nondeterministic polynomial space.

The relationship between P and NP is unknown. Clearly P $\subseteq$ NP, but is NP strictly larger? That is, are there problems that can be solved in efficiently—in polynomial time—using a nondeterministic computer, but that can't be solved efficiently using a deterministic computer? We can't actually build nondeterministic computers, so the question may seem academic, but many important problems are known to be NP-complete [34], so if P $\neq$ NP, then there is no efficient deterministic algorithm for solving them.

However, it *is* known that PSPACE = NPSPACE [71]. More generally, NSPACE($f(n)$) $\subseteq$ SPACE($f^2(n)$). Nondeterminism thus does not increase the power of space-bounded computation beyond at most a quadratic savings.

In relation to the concept of games and puzzles, a nondeterministic computation is similar to a puzzle: if the right moves to solve the "computation puzzle" may be found, then the computation nondeterministically accepts. We can't build an actual nondeterministic computer, but we *can* build and solve puzzles. A perfect puzzle solver is performing a nondeterministic computation.

Alternation. Chandra, Kozen, and Stockmeyer [8] have extended the concept of nondeterminism to that of *alternation*. Essentially, the idea is to add the notion of universal, as well as existential, quantification. A nondeterministic Turing machine accepts existentially: it accepts if there exists an accepting computation history. In an alternating Turing machine, the states are divided into *existential* states and *universal* states. A machine accepts starting from a configuration in an existential state if *any* transition from the transition function leads to acceptance; it accepts starting from a configuration in a universal state if *all* possible transitions lead to acceptance.

Alternating time- and space-complexity classes ATIME($t(n)$) and ASPACE($f(n)$) are defined as above, and AP and APSPACE are defined analogously to P and PSPACE (or NP and NPSPACE). It turns out that AP = PSPACE, and APSPACE = EXPTIME. Thus, alternating time is as powerful as deterministic space, and alternating space is as powerful as exponential deterministic time.

But what does alternation mean, intuitively? The best way to think of an alternating computation is as a two-player game. One player, the existential one, is trying to win the game (accept the computation) by choosing a winning move (transition); the other player, the universal one, is trying to win (reject the computation) by finding a move (transition) from which the existential player can't win. And in fact, the concept of alternation, and the results mentioned above, have been very useful in the field of game complexity.

Again, we can't build an alternating computer, but we can play actual two-player games; a perfect game player is performing an alternating computation.

Multiplayer Alternation. Building on the notion of Alternation, Peterson and Reif [61] introduced *multiplayer alternation*. It turns out that simply adding new computational "players", continuing the idea that an extra degree of nondeterminism adds computational power, is not sufficient here. Instead, a multiplayer computation is like a *team* game, with multiple players on a team, and with the additional notion of *private information*. The game analogy is that in some games, not all information is public to all players. (Many card games, for example, have this property.) The concept is added to Turing machines by having multiple read-write tapes, with the transition function from some states not allowed to depend on the contents of some tapes. Multiplayer alternation is explored Chapters 7 and 8.

Multiplayer alternating machines turn out to be extremely powerful—so powerful, in fact, that MPA-PSPACE, the class of languages decidable in multiplayer alternating polynomial space, is all the decidable languages.

This is a remarkable fact. A multiplayer alternating Turing machine can do in a bounded amount of space what a deterministic Turing machine can do with an *infinite* tape. Again, we can't build actual multiplayer alternating computers. But if we lived in a world containing perfect game players, we could do arbitrary computations with finite physical resources.

A.3 Relationship of Complexity Classes

The containment relationships of the classes mentioned above are as follows:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} = \mathrm{NPSPACE} \subseteq \mathrm{EXPTIME} \subsetneq \text{recursive} \subsetneq \text{r.e.}$$

All of the containments are believed to be strict, but beyond the above relations, the only strict containment known among those classes is $\mathrm{P} \subsetneq \mathrm{EXPTIME}$. $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ is the most famous unknown relation, but it is not even known whether P = PSPACE.

A.4 List of Complexity Classes Used in this Thesis

The following classes are listed in order of increasing containment; that is, $\mathrm{L} \subseteq \mathrm{NL} \subseteq \mathrm{NC}^3 \ldots$, with the exception that the relationship between NP and coNP is unknown. (However, either NP = coNP, or neither contains the other.)

L $= \mathrm{SPACE}(\log n)$.

NL $= \mathrm{NSPACE}(\log n)$.

NC^3 see, e.g., [74] for definition.

P $= \bigcup_k \mathrm{TIME}(n^k)$ = languages decidable in polynomial time.

NP $= \bigcup_k \mathrm{NTIME}(n^k)$ = languages decidable in nondeterministic polynomial time.

coNP $= \{L \mid \overline{L} \in \mathrm{NP}\}$ = languages whose complements are decidable in nondeterministic polynomial time. ($\omega \in \overline{L} \iff \omega \notin L$.)

PSPACE $= \bigcup_k \mathrm{SPACE}(n^k)$ = languages decidable in polynomial space.

NPSPACE $= \bigcup_k \mathrm{NSPACE}(n^k)$ = languages decidable in nondeterministic polynomial space = PSPACE.

EXPTIME $= \bigcup_k \mathrm{TIME}(2^{n^k})$ = languages decidable in exponential time.

NEXPTIME $= \bigcup_k \mathrm{NTIME}(2^{n^k})$ = languages decidable in nondeterministic exponential time.

EXPSPACE $= \bigcup_k \mathrm{SPACE}(2^{n^k})$ = languages decidable in exponential space.

NEXPSPACE $= \bigcup_k \mathrm{NSPACE}(2^{n^k})$ = languages decidable in nondeterministic exponential space = EXPSPACE.

2EXPTIME $= \bigcup_k \mathrm{TIME}(2^{2^{n^k}})$ = languages decidable in doubly exponential time.

Recursive = decidable languages.

Recursively Enumerable (r.e.) = Turing-recognizable languages.

A.5 Formula Games.

A game played on a Boolean formula is often the canonical complete problem for a complexity class. Boolean Satisfiability (SAT), the first problem shown to be NP-complete [10], can be viewed as a puzzle in which the moves are to choose variable assignments. Quantified Boolean Formulas (QBF), which is PSPACE-complete, essentially turns this puzzle into a two-player game, where the players alternate choosing variable assignments. There are formula games for EXPTIME, EXPSPACE, and other classes, as well.

Here I will define Boolean formulas, and the basic formula games SAT and QBF. Other formula games are defined in the text as they are needed.

A.5.1 Boolean Formulas.

A *Boolean variable* is a variable which can have the value *true* or *false*. A *Boolean operation* is one of AND ($\wedge$), OR ($\vee$), or NOT ($\neg$). A *Boolean formula* is either a Boolean variable, or one of the expressions $(\phi \wedge \psi)$, $(\phi \vee \psi)$, and $\neg\phi$, where ϕ and ψ are Boolean formulas.

$(\phi \wedge \psi)$ is true if ϕ and ψ are both true, and false otherwise. $(\phi \vee \psi)$ is true if either ϕ or ψ is true, and false otherwise. $\neg\phi$ is true if ϕ is false, and false otherwise.

A *literal* is a variable x or its negation $\neg x$, abbreviated $\overline{x}$.

A *monotone formula* is a formula which does not contain $\neg$. Monotone formulas have the property that if the value of any contained variable is changed from false to true, the value of the formula can never change from true to false.

A *quantified variable* is either $\forall x$ or $\exists x$, for variable x.

A *quantified Boolean formula* is either a Boolean formula, or a quantified Boolean formula preceded by a quantified variable.

$\forall x\, \phi$ is true if ϕ is true both when x is assigned to false and when it is assigned to true, and $\exists x\, \phi$ is true if ϕ is true when x is assigned either to false or to true.

A.5.2 Satisfiability (SAT).

The Boolean formula satisfiability problem is NP-complete, and is almost invariably the problem of choice to reduce to another problem to show that problem NP-hard. It is defined as follows:

Satisfiability (SAT)
INSTANCE: Boolean formula ϕ.
QUESTION: Is there an assignment to the variables of ϕ such that ϕ is true?

Equivalently, SAT could be defined as the question of whether a given quantified Boolean formula which uses only existential quantifiers is true.

The process of choosing a satisfying variable assignment can be viewed as solving a kind of puzzle.

A.5.3 Quantified Boolean Formulas (QBF).

Quantified Boolean Formulas is PSPACE-complete, and is almost invariably the problem of choice to reduce to another problem to show that problem PSPACE-hard. It is defined as follows:

Quantified Boolean Formulas (QBF)
INSTANCE: Quantified Boolean formula ϕ.
QUESTION: Is ϕ true?

The truth of a quantified Boolean formula corresponds to the winner of a two-person game. This is easiest to see in the case where the quantifiers strictly alternate between $\exists$ and $\forall$, as in $\exists x \, \forall y \, \exists z \, \ldots \phi$. Then, we may say that the $\exists$ player can win the formula game if he can choose a value for x such that for any value the $\forall$ player chooses for y, the $\exists$ player can choose a value for z, such that $\ldots \phi$ is true.

This correspondence may also be understood in terms of the previously mentioned result that AP = PSPACE: a two-player game of polynomially-bounded length is an alternating computation that can be carried out in polynomial time.

Appendix B

Deterministic Constraint Logic Activation Sequences

In this appendix I present the explicit activation sequences for several DCL gadgets described in Section 4.2. Refer to that section for complete descriptions of the gadgets' intended behaviors, and of the deterministic rule used. As mentioned there, all of the gadgets used are designed on the assumption that signals will only arrive at their inputs at some time 0 mod 4 (so that the first internal edge reversal occurs at time 1 mod 4), and also that signals will activate output edges only at times 0 mod 4. This makes it possible to know what the internal state of the gadgets is when inputs arrive, because any persistent activity in a gadget will repeat every two or four steps.

Switch Gadget. This gadget is used internally in many of the other gadgets. In Figure B-1 I show all the steps in its activation sequence. When input arrives at A, an output signal is sent first to B, then, when that signal has returned to the switch, on to C, then to B again, and finally back to A. In some cases the extra activation of B is useful; in the other cases, it is redundant but not harmful.

Existential Quantifier This gadget uses a switch to "try" both possible variable assignments. The connected CNF circuitry follows a protocol by which a variable state can be asserted by activating one pathway; a return pathway will then activate back to the quantifier. When the quantifier is done using that assignment, it can de-assert it by following the return pathway backwards; activation will then proceed back into the gadget along the original assertion output edge.

In Figure B-2 I show the activation sequence. Only every fourth time step is shown; in between these steps the internal switch is operating as above. Possible activation of the satisfied in / satisfied out pathway is not shown, but when it occurs it clearly preserves the necessary timing.

Universal Quantifier. The universal quantifier is similar to the existential quantifier; it also uses a switch to try both variable assignments. However, if the assignment to x = false succeeds, the gadget sets an internal latch to remember this fact. Then,

if x = true also succeeds, the latch enables satisfied out to be directed out. Finally, the switch tries x = false again; this resets the latch. This sequence is shown in Figures B-3 and B-4. Again, only every fourth time step is shown. Only the "forward" operation of the gadget is shown; deactivation follows an inverse sequence.

If the assignment to x = false fails, and x = true succeeds, then the unset latch state causes the x = true success to simply bounce back. This sequence is shown in Figure B-5.

AND′. The AND′ gadget must respond to two different circumstances: first, input 1 arrives, and then input 2 later arrives (or not); and second, input 2 arrives when input 1 has not arrived. The first case is shown in Figure B-6, the second in Figure B-7. In each case only every fourth time step is shown, and the reverse, deactivating, sequences are not shown.

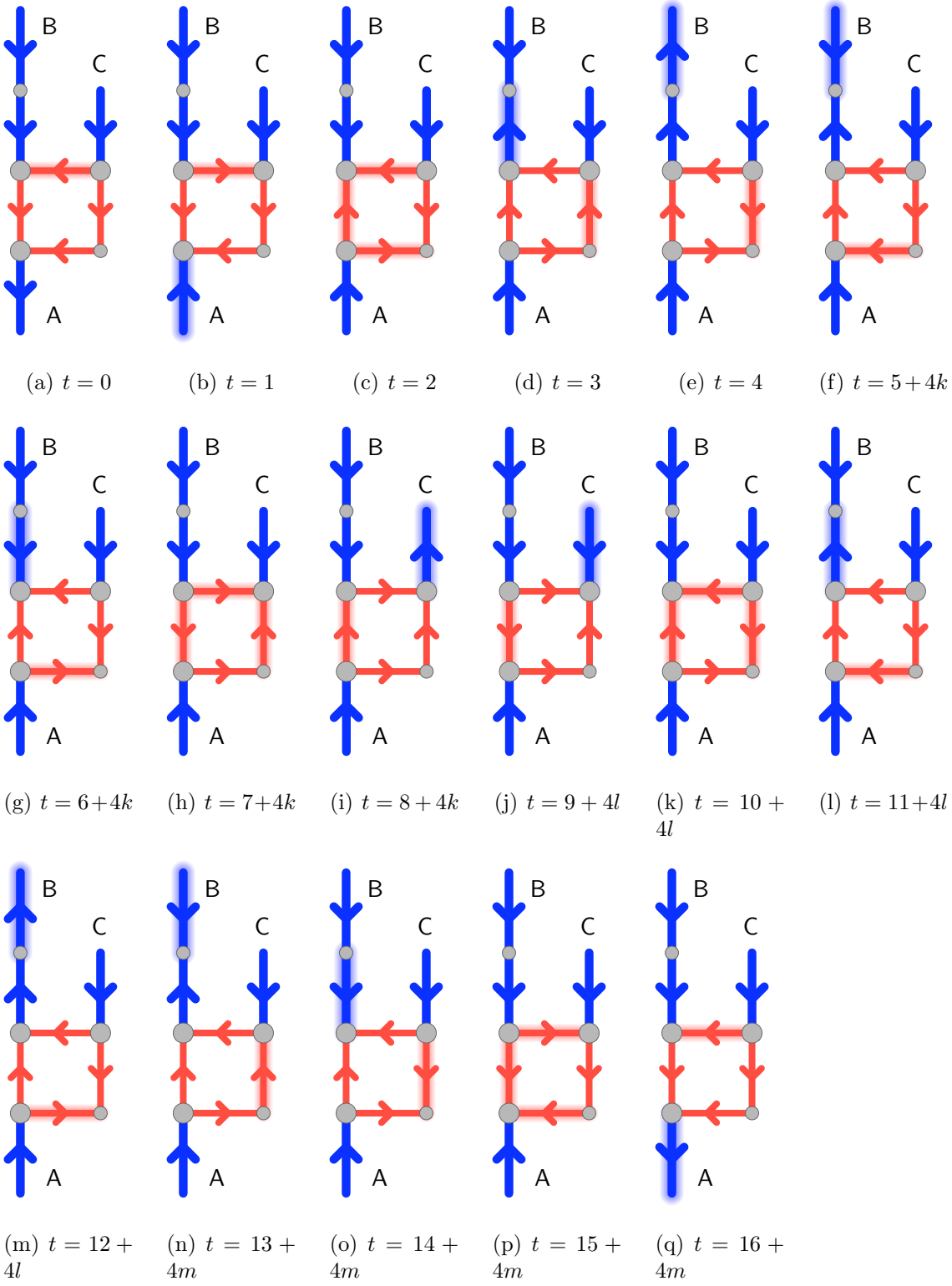
OR′. The OR′ is complicated for two reasons. First, it must activate when either input activates, but whichever has activated, if the other input then arrives, it must simply bounce back cleanly (because the output is already activated). Second, the internal switch required is more complicated than the basic switch. The basic switch may be described as following the sequence ABCBA; the switch used in the OR′ would correspondingly follow the sequence ABC.

Two sequences are shown. First, in Figures B-8 and B-9, an activation sequence beginning with input 1 is shown. Part of the deactivating sequence is shown as well, because it is not the reverse of the forward sequence (due to the modified switch). The activation sequence beginning with input 2 is similar, but the "extra search step" taken by the internal switch occurs during the forward rather than the reverse activation sequence in this case.

Second, Figure B-10 shows the activation sequence when the OR′ is already active, and the other input arrives. (In this case the operation is symmetric with respect to the two inputs.) The second input is propagated directly to its return path.

FANOUT′, CNF Output Gadget. The correct operation of these gadgets (shown in Figure 4-4) is obvious.

Crossover Gadget. The steps involved in crossover activation are shown in Figure B-11. The reverse sequence deactivates the crossover. The sequence shown has a C-D traversal following an A-B traversal. C-D can also occur in isolation (but not followed by A-B); note that after the A-B traversal (and at the same time mod 4), the gadget is in a vertically symmetric state to the original one.

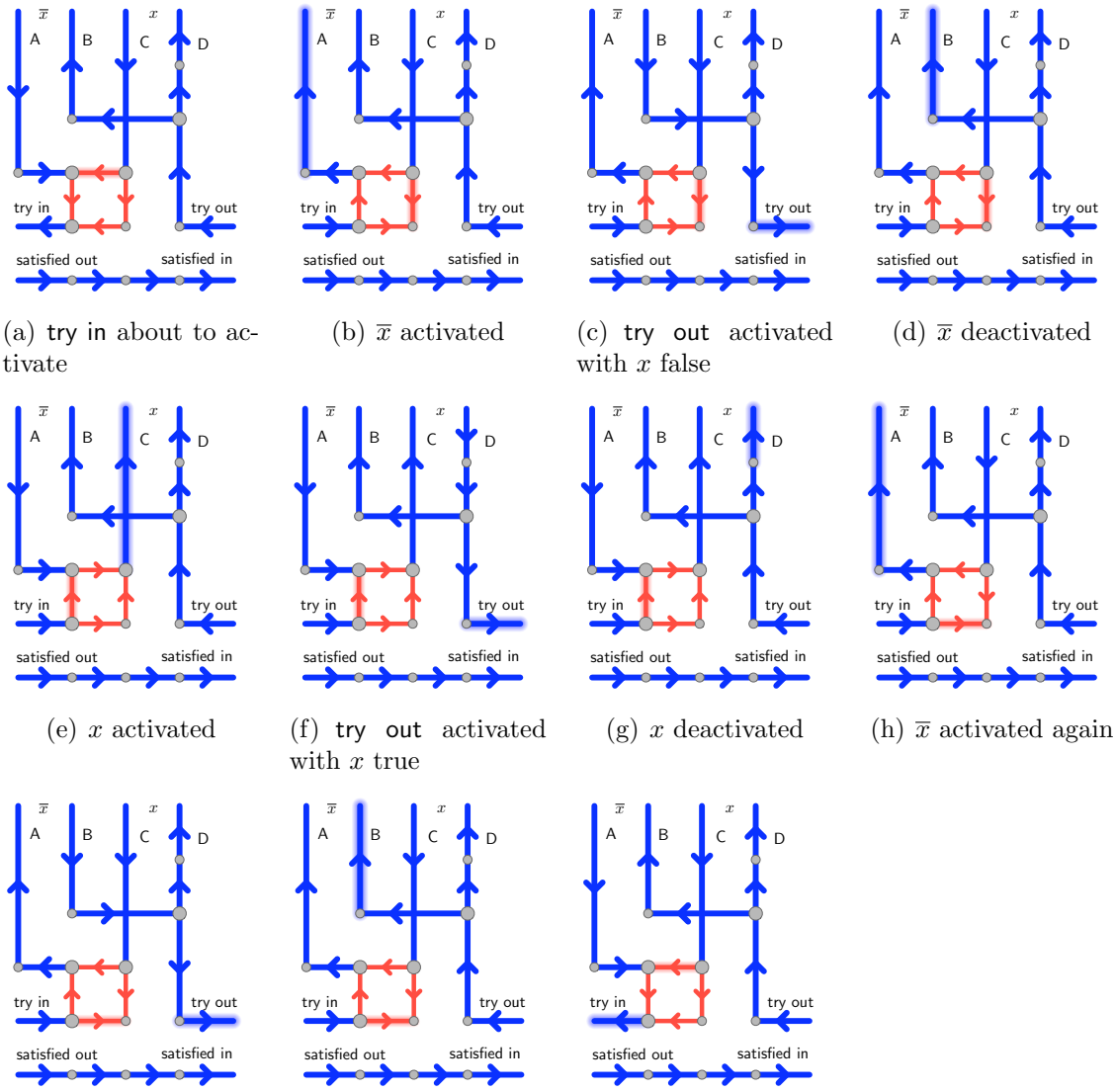


(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$ (e) $t = 4$ (f) $t = 5+4k$

(g) $t = 6+4k$ (h) $t = 7+4k$ (i) $t = 8+4k$ (j) $t = 9+4l$ (k) $t = 10+4l$ (l) $t = 11+4l$

(m) $t = 12+4l$ (n) $t = 13+4m$ (o) $t = 14+4m$ (p) $t = 15+4m$ (q) $t = 16+4m$

Figure B-1: Switch gadget steps. $0 \leq k \leq l \leq m$.



(a) try in about to activate

(b) $\overline{x}$ activated

(c) try out activated with x false

(d) $\overline{x}$ deactivated

(e) x activated

(f) try out activated with x true

(g) x deactivated

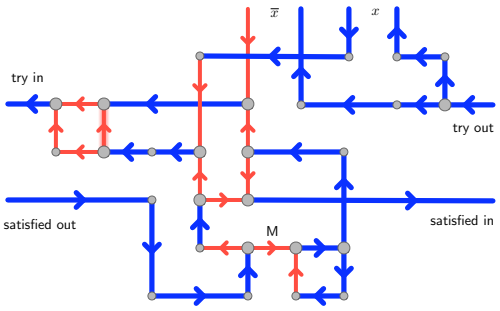
(h) $\overline{x}$ activated again

(i) try out activated with x false

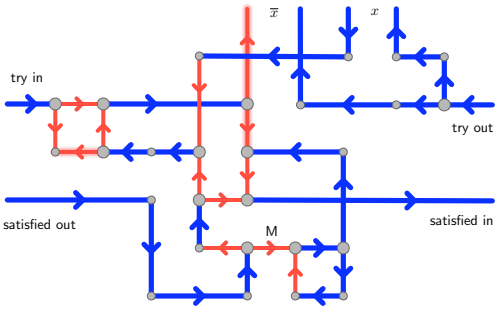
(j) $\overline{x}$ deactivated

(k) try in deactivated

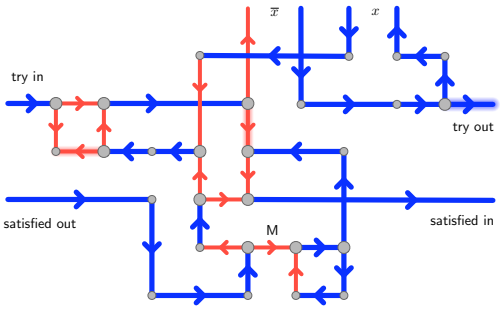
Figure B-2: Existential quantifier steps. Every fourth step is shown. Between steps (b) and (c), (d) and (e), (e) and (f), (g) and (h), (h) and (i), and (j) and (k), a signal is propagated into and out of the connecting CNF circuitry. Between steps (c) and (d), (f) and (g), and (i) and (j), a signal is propagated through to the quantifier to the right, and possibly through satisfied in / satisfied out and back. All inputs are guaranteed to arrive at times 0 mod 4.



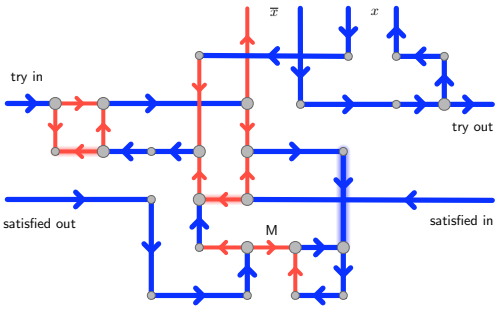
(a) try in about to activate



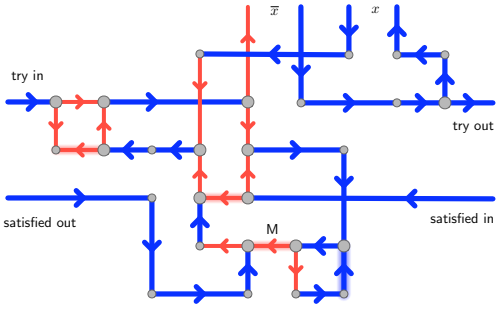
(b) $\overline{x}$ activated



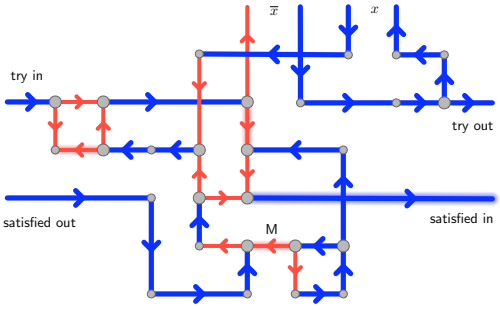
(c) try out activated with x false



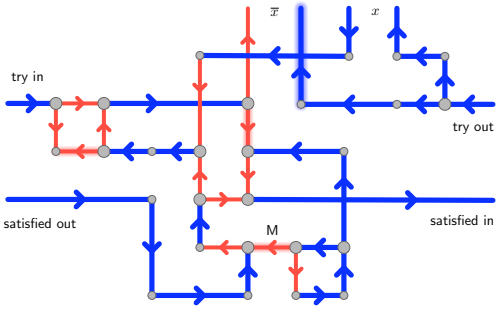
(d) satisfied in activated with x false



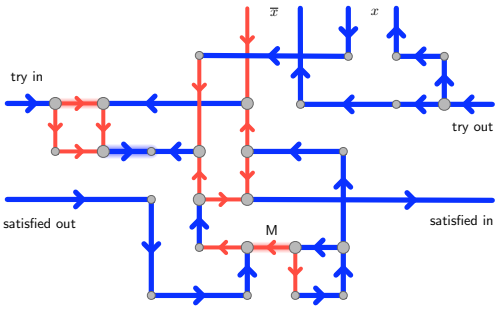
(e) Internal latch M activated



(f) satisfied in deactivated

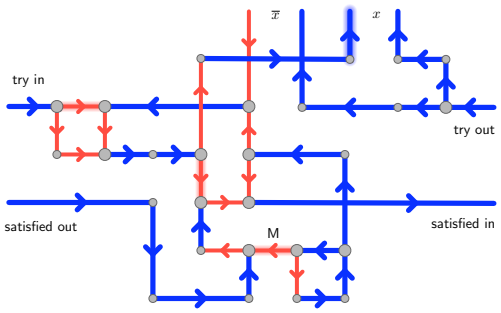


(g) $\overline{x}$ deactivated

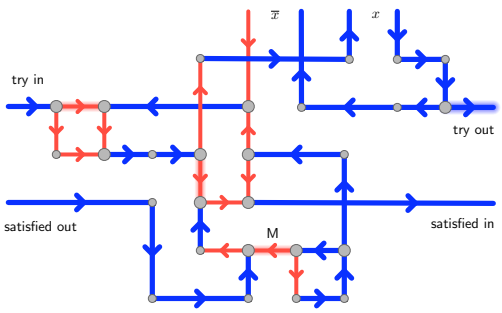


(h) Switch trying second branch

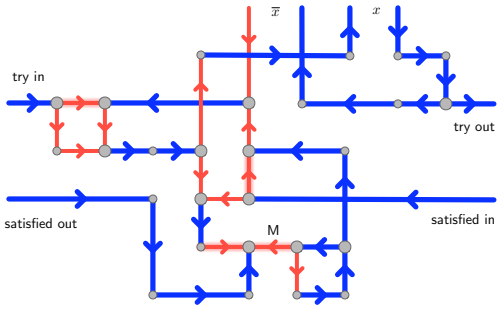
Figure B-3: Universal quantifier steps, part one. Every fourth step is shown.



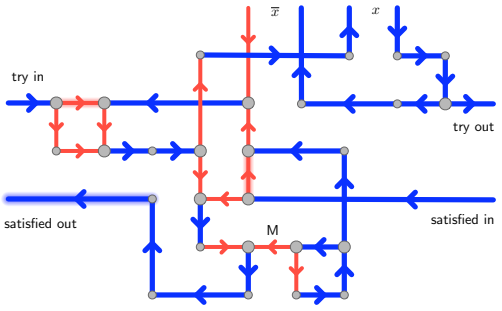
(a) x activated



(b) try out activated with x true

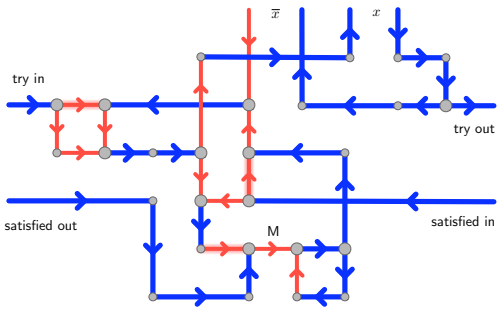


(c) satisfied in activated with x true and latch M set

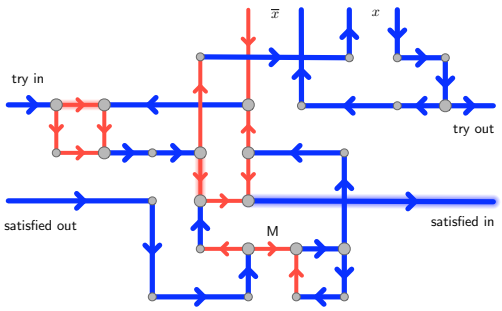


(d) satisfied out activated

Figure B-4: Universal quantifier steps, part two (continuation of part one).

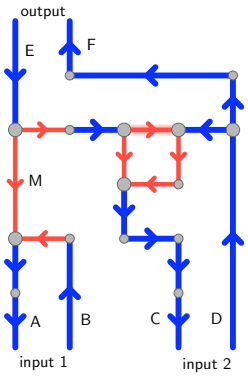


(a) satisfied in activated with x true and latch M not set

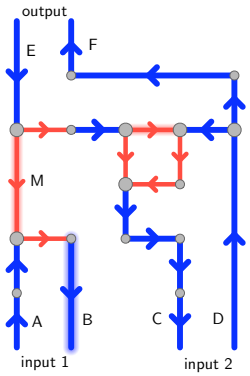


(b) satisfied in deactivated

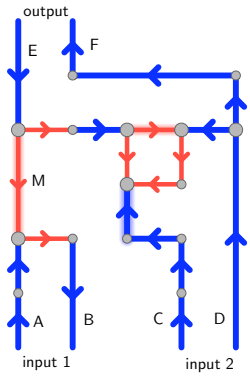
Figure B-5: Universal quantifier steps, part three. These two steps replace the last two in Figure B-4, in the case where the x false assignment did not succeed and set latch M.



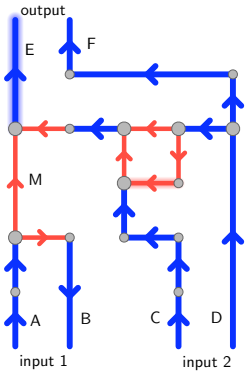
(a) input 1 about to activate



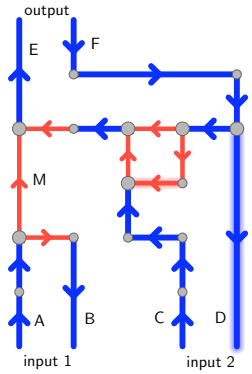
(b) input 1 return path activated; latch M set



(c) input 2 activated

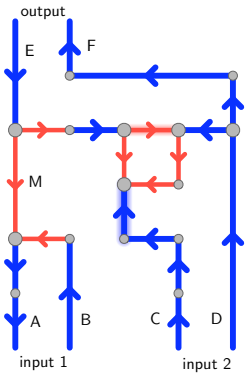


(d) output activated

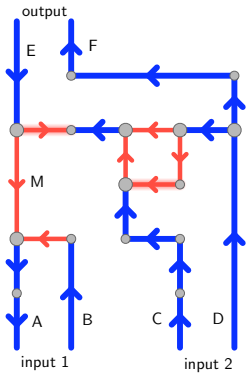


(e) input 2 return path activated

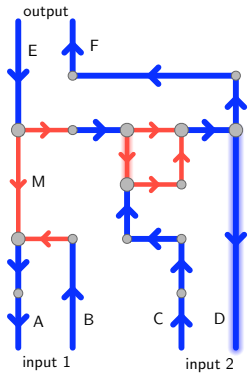
Figure B-6: AND′ steps, in the case when both inputs activate in sequence. Every fourth step is shown.



(a) input 2 activated

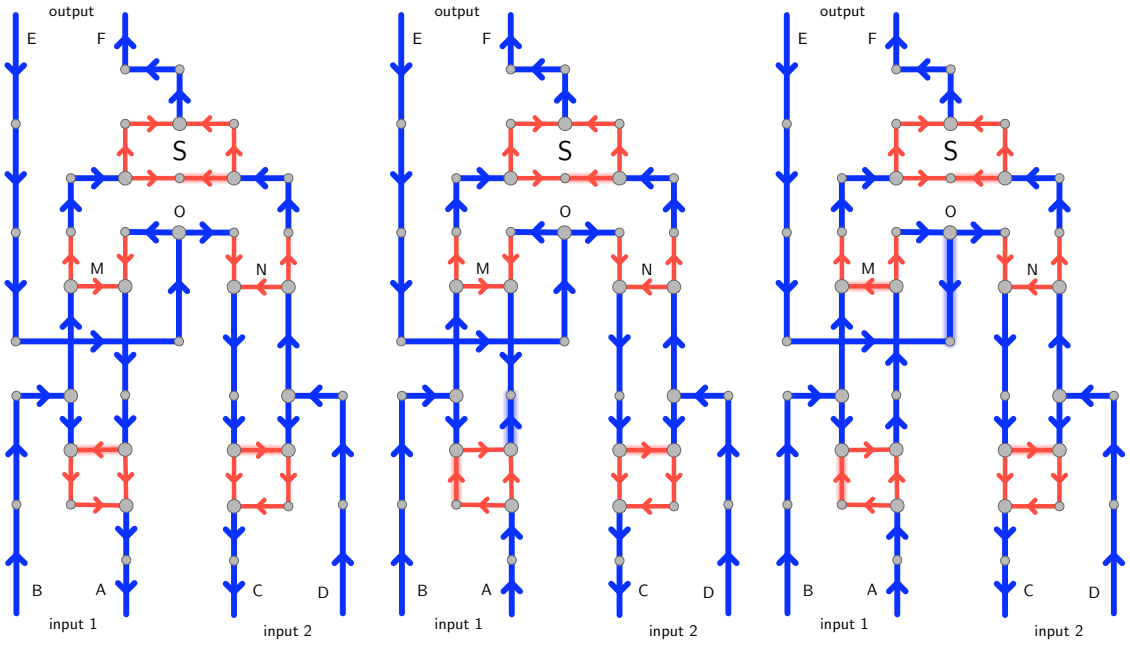


(b) Without latch M set, signal bounces without activating output

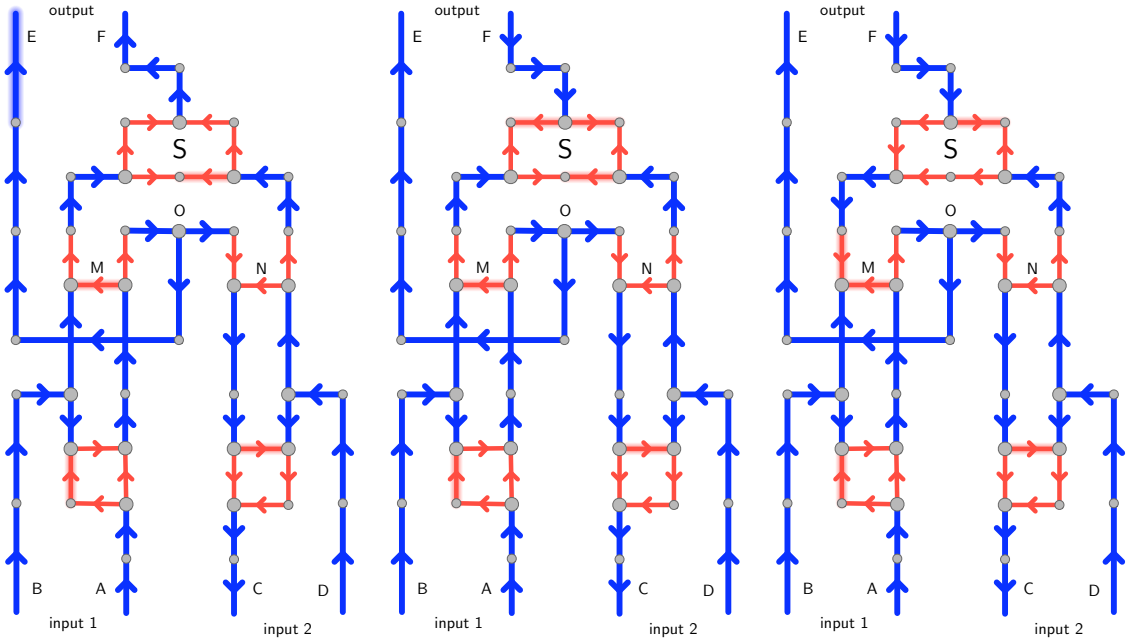


(c) input 2 return path activated

Figure B-7: AND′ steps, in the case when input 2 activates without input 1 first activating. Every fourth step is shown.

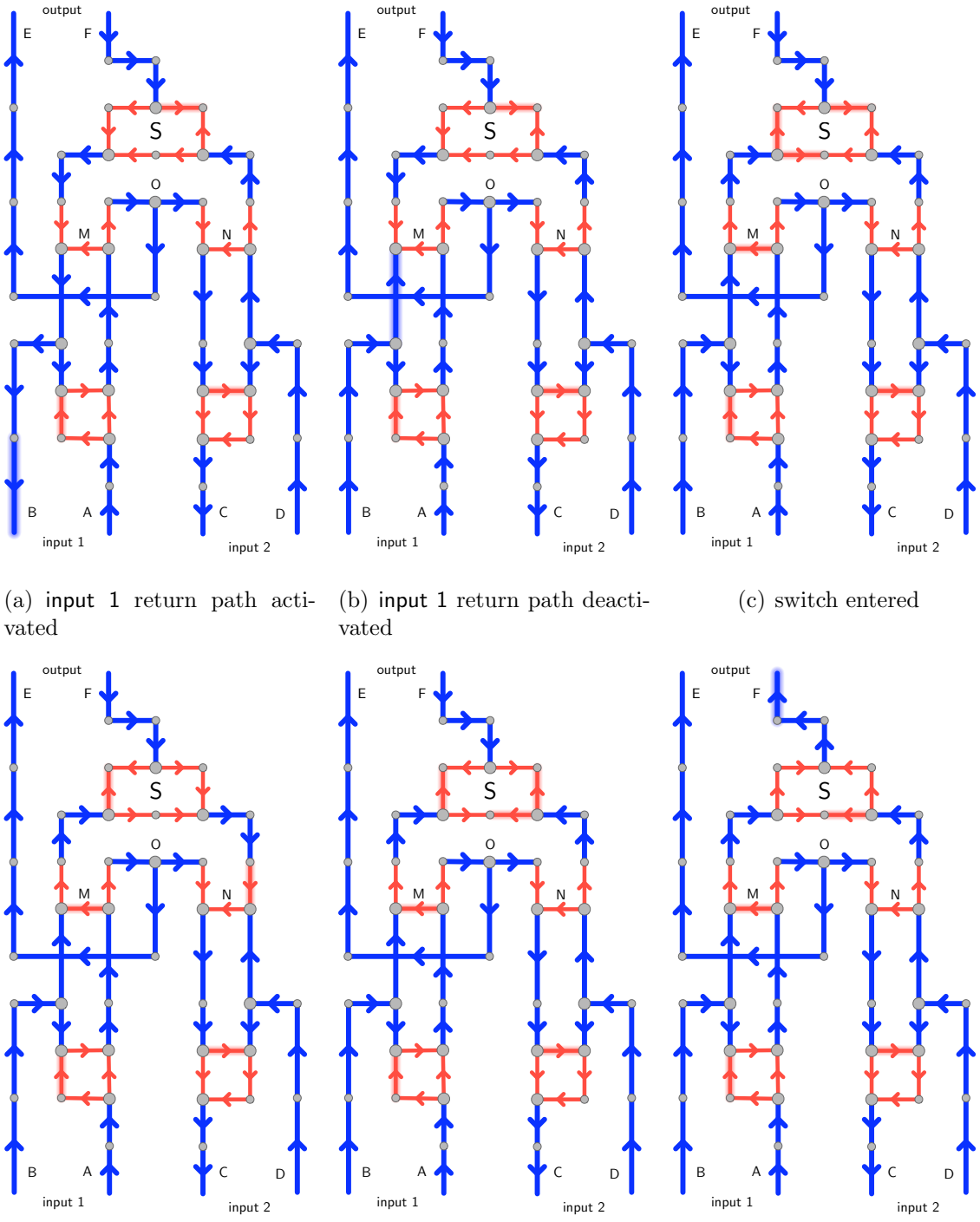


(a) input 1 about to activate (b) signal propagating (c) internal OR activated



(d) output activated (e) output return path activated (f) switch successfully tries left side

Figure B-8: OR′ steps, part one. Every fourth step is shown. If input 2 is activated instead, the sequence will be slightly different; the switch will first try the left side, and then the right.



(a) input 1 return path activated

(b) input 1 return path deactivated

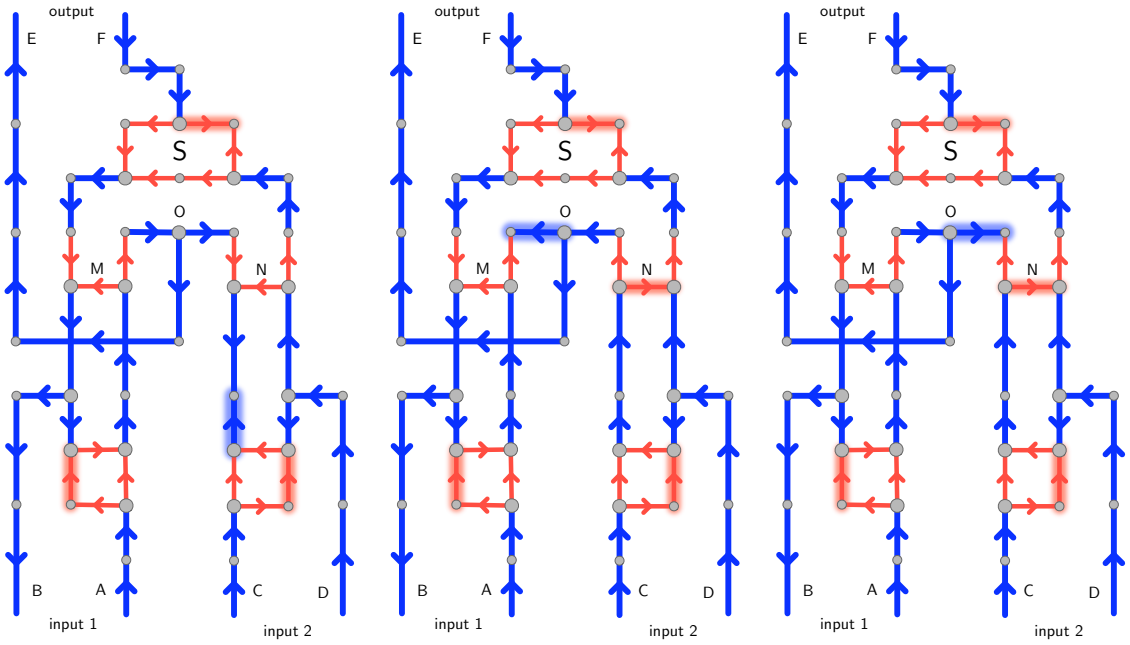
(c) switch entered

(d) switch tries right side

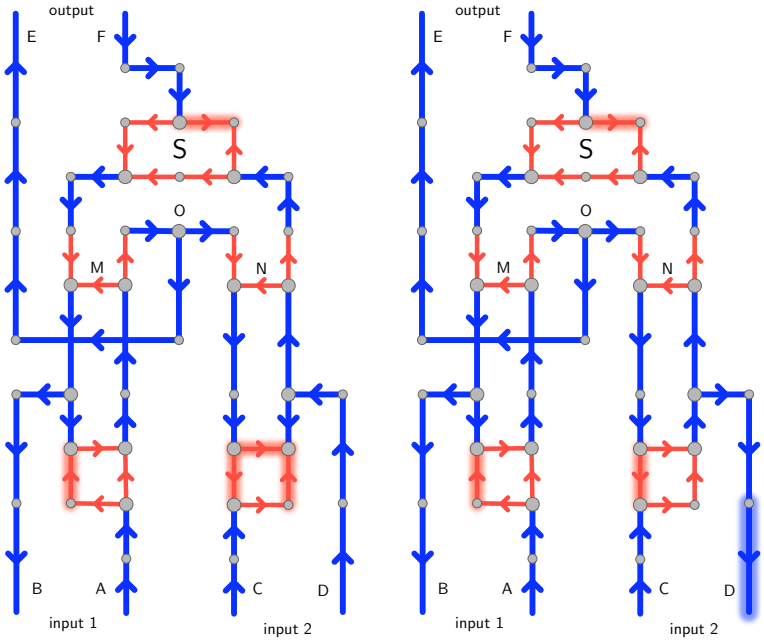
(e) back into switch

(f) output return path deactivated

Figure B-9: OR′ steps, part two. Every fourth step is shown. If input 2 was activated instead, the sequence will be slightly different; the switch send the return signal out earlier.

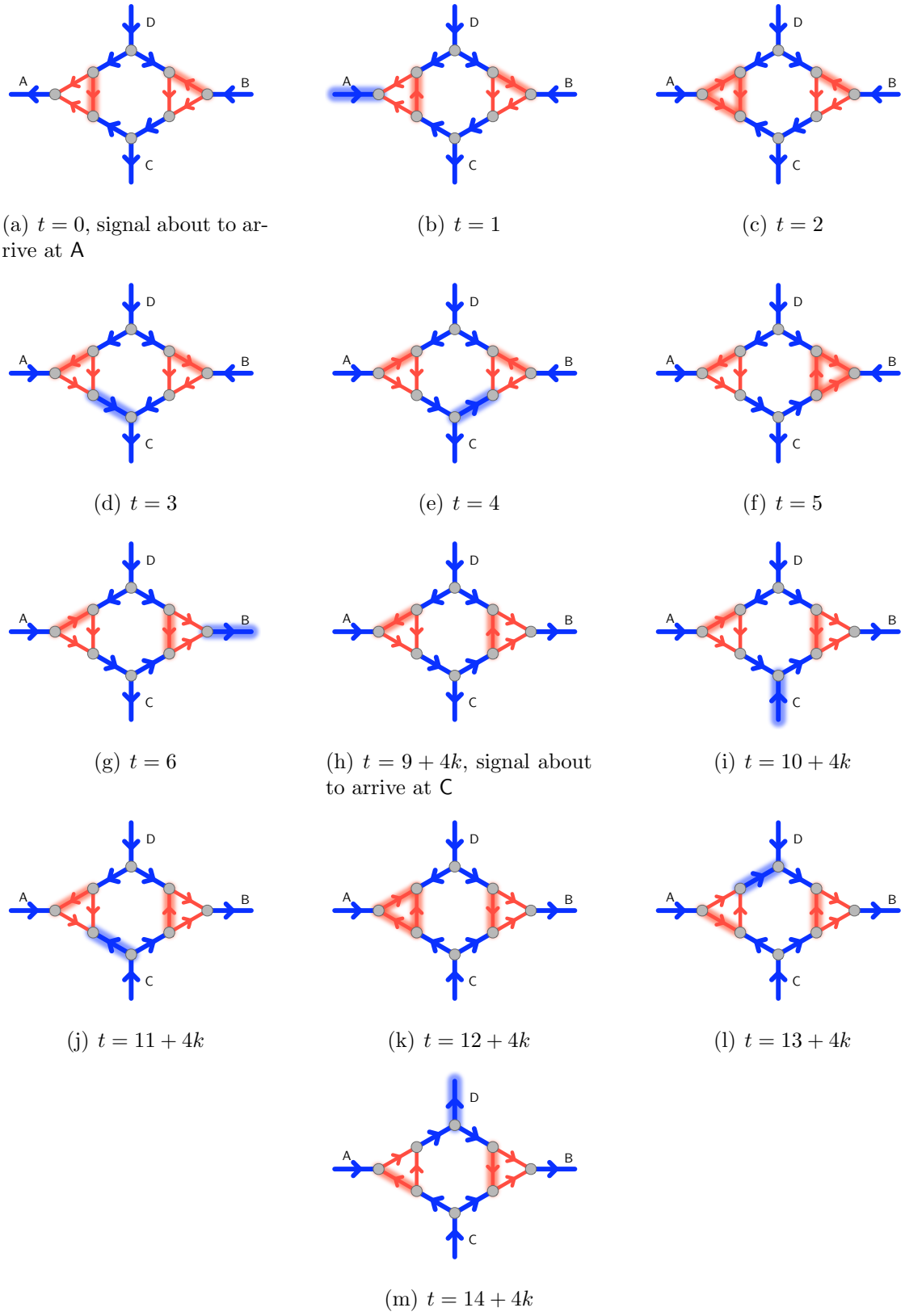


(a) input 2 activated (b) signal propagating (c) signal bounces



(d) input switch redirects signal (e) input 2 return path activated

Figure B-10: OR′ steps, part three. Every fourth step is shown. input 2 arrives when the gate is already activated, and is cleanly propagated on to its return path. Deactivation follows the reverse sequence.



(a) $t = 0$, signal about to arrive at A

(b) $t = 1$

(c) $t = 2$

(d) $t = 3$

(e) $t = 4$

(f) $t = 5$

(g) $t = 6$

(h) $t = 9 + 4k$, signal about to arrive at C

(i) $t = 10 + 4k$

(j) $t = 11 + 4k$

(k) $t = 12 + 4k$

(l) $t = 13 + 4k$

(m) $t = 14 + 4k$

Figure B-11: Crossover gadget steps. The padding edges required to enter and exit at times 0 mod 4 are omitted; as a result, there is a gap between steps (g) and (h).

Bibliography

[1] Scott Aaronson. NP-complete problems and physical reality. *SIGACT News*, 36(1):30–52, 2005.

[2] Charles H. Bennett. Logical reversibility of computation. *IBM J. Res. & Dev.*, 17:525–532, 1973.

[3] Elwyn R. Berlekamp. Sums of $N \times 2$ Amazons. In *Lecture Notes - Monograph Series*, volume 35, pages 1–34. Institute of Mathematical Statistics, 2000.

[4] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways*. A. K. Peters, Ltd., Wellesley, MA, second edition, 2001–2004.

[5] Hans Bodlaender. Re: Is retrograde chess NP-hard? Usenet posting to `rec.games.abstract`, March 16 2001.

[6] Michael Buro. Simple Amazons endgames and their connection to Hamilton circuits in cubic subgrid graphs. In *Proceedings of the 2nd International Conference on Computers and Games*, Lecture Notes in Computer Science, Hamamatsu, Japan, October 2000.

[7] Alice Chan and Alice Tsai. $1 \times n$ konane: a summary of results. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 331–339, 2002.

[8] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[9] John Horton Conway. *On Numbers and Games*. A. K. Peters, Ltd., second edition, 2000.

[10] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.

[11] Paolo Cotogno. Hypercomputation and the physical church-turing thesis. *Brit. J. Philosophy of Science*, 54:181–223, 2003.

[12] Marcel Crâsmaru and John Tromp. Ladders are pspace-complete. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games*, volume 2063 of *Lecture Notes in Computer Science*, pages 241–249. Springer, 2000.

[13] J. C. Culberson. Sokoban is PSPACE-complete. In *Proceedings International Conference on Fun with Algorithms (FUN98)*, pages 65–76, Waterloo, Ontario, Canada, June 1998. Carleton Scientific.

[14] Konstantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *Electronic Colloquium on Computational Complexity (ECCC)*, (115), 2005.

[15] Erik Demaine, Martin Demaine, Rudolf Fleischer, Robert Hearn, and Timo von Oertzen. The complexity of dyson telescopes. Manuscript, February 2005.

[16] Erik D. Demaine, Martin L. Demaine, and David Eppstein. Phutball endgames are NP-hard. In R. J. Nowakowski, editor, *More Games of No Chance*. Cambridge University Press, 2002.

[17] Erik D. Demaine, Martin L. Demaine, and Helena A. Verrill. Coin-moving puzzles. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 405–431. Cambridge University Press, 2002. Collection of papers from the MSRI Combinatorial Game Theory Research Workshop, Berkeley, California, July 24–28, 2000.

[18] Erik D. Demaine, Robert A. Hearn, and Michael Hoffmann. Push-2-f is pspace-complete. In *Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG 2002)*, pages 31–35, Lethbridge, Alberta, Canada, August 12–14 2002.

[19] Henry Ernest Dudeney. *The Canterbury Puzzles and Other Curious Problems*. W. Heinemann, London, 1907.

[20] David Eppstein. Hinged kite mirror dissection. ACM Computing Research Repository, June 2001.

[21] Michael D. Ernst. Playing Konane mathematically: A combinatorial game-theoretic analysis. *UMAP Journal*, 16(2):95–121, Spring 1995.

[22] Shimon Even and Robert E. Tarjan. A combinatorial problem which is complete in polynomial space. *Journal of the Association for Computing Machinery*, 23(4):710–719, 1976.

[23] Hugh Everett. Relative state formulation of quantum mechanics. *Rev. Mod. Phys.*, 29:454–462, 1957.

[24] Gary William Flake and Eric B. Baum. *Rush Hour* is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1–2):895–911, January 2002.

[25] Aviezri Fraenkel. Combinatorial games : Selected bibliography with a succinct gourmet introduction. In R. J. Nowakowski, editor, *Games of No Chance*, pages 493–537. MSRI Publications, Cambridge University Press, 1996.

[26] 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, 31:199–214, 1981.

[27] Michael P. Frank. Approaching the physical limits of computing. In *35th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2005), 18-21 May 2005, Calgary, Canada*, pages 168–185, 2005.

[28] Greg N. Frederickson. *Hinged Dissections: Swinging and Twisting*. Cambridge University Press, 2002.

[29] E. Fredkin and T. Toffoli. Conservative logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[30] Timothy Furtak, Masashi Kiyomi, Takeaki Uno, and Michael Buro. Generalized amazons is PSPACE-complete. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 132–137. Professional Book Center, 2005.

[31] Zvi Galil. Hierarchies of complete problems. *Acta Inf.*, 6:77–88, 1976.

[32] Martin Gardner. The hypnotic fascination of sliding-block puzzles. *Scientific American*, 210:122–130, 1964.

[33] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, October 1970.

[34] 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, 1979.

[35] Andrea Gilbert. Plank puzzles, 2000. `http://www.clickmazes.com/planks/ixplanks.htm`.

[36] L. M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, 1977.

[37] John Gribbin. Doomsday device. *Analog Science Fiction/Science Fact*, 105(2):120–125, Feb 1985.

[38] Markus Gtz. Triagonal slide-out, 2005. `http://www.markus-goetz.de/puzzle/java/0021/crh.html`.

[39] Jeffrey R. Hartline and Ran Libeskind-Hadas. The computational complexity of motion planning. *SIAM Review*, 45:543–557, 2003.

[40] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematics Society*, 117:285–306, 1965.

[41] Robert Hearn. Tipover is NP-complete. *Mathematical Intelligencer*, 2006. To appear.

[42] Robert Hearn, Erik Demaine, and Greg Frederickson. Hinged dissection of polygons is hard. In *Proc. 15th Canad. Conf. Comput. Geom.*, pages 98–102, 2003.

[43] Robert A. Hearn. The complexity of sliding block puzzles and plank puzzles. In *Tribute to a Mathemagician*, pages 173–183. A K Peters, 2004.

[44] Robert A. Hearn. Amazons is PSPACE-complete. Manuscript, February 2005. `http://www.arXiv.org/abs/cs.CC/0008025`.

[45] Robert A. Hearn. Amazons, Konane, and Cross Purposes are PSPACE-complete. In R. J. Nowakowski, editor, *Games of No Chance 3*, 2006. Submitted.

[46] Robert A. Hearn and Erik D. Demaine. The nondeterministic constraint logic model of computation: Reductions and applications. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *ICALP*, volume 2380 of *Lecture Notes in Computer Science*, pages 401–413. Springer, 2002.

[47] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, October 2005. Special issue "Game Theory Meets Theoretical Computer Science".

[48] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects: PSPACE-hardness of the 'Warehouseman's Problem'. *International Journal of Robotics Research*, 3(4):76–88, 1984.

[49] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, second edition, 2000.

[50] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer*, 22(2):9–15, 2000.

[51] R. Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. & Dev.*, 5:183, 1961.

[52] David Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[53] David Lichtenstein and Michael Sipser. GO is polynomial-space hard. *Journal of the Association for Computing Machinery*, 27(2):393–401, April 1980.

[54] Jens Lieberum. An evaluation function for the game of Amazons. *Theoretical Computer Science*, 349(2):230–244, December 2005. Special issue "Advances in Computer Games".

[55] Marvin Minsky. *Computation: Finite and Infinite Machines*. Prentice Hall, Englewoods Cliffs, 1967.

[56] Martin Müller and Theodore Tegos. Experiments in computer Amazons. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 243–257. Cambridge University Press, 2002.

[57] John F. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36:48–49, 1950.

[58] Christos H. Papadimitriou. Games against nature. *J. Comput. Syst. Sci.*, 31(2):288–301, 1985.

[59] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1993.

[60] Gary Peterson, John Reif, and Salman Azhar. Lower bounds for multiplayer non-cooperative games of incomplete information. *Computers and Mathematics with Applications*, 41:957–992, Apr 2001.

[61] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *FOCS*, pages 348–363. IEEE, 1979.

[62] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

[63] John Reif, 2006. Personal communication.

[64] John H. Reif. Universal games of incomplete information. In *STOC '79: Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 288–308, New York, NY, USA, 1979. ACM Press.

[65] Stefan Reisch. Hex ist PSPACE-vollständig (Hex is PSPACE-complete). *Acta Informatica*, 15:167–191, 1981.

[66] Paul Rendell. Turing universality of the game of life. In *Collision-based computing*, pages 513–539, London, UK, 2002. Springer-Verlag.

[67] Edward Robertson and Ian Munro. NP-completeness, puzzles and games. *Utilitas Mathematica*, 13:99–116, 1978.

[68] J. M. Robson. The complexity of Go. In *Proceedings of the IFIP 9th World Computer Congress on Information Processing*, pages 413–417, 1983.

[69] J. M. Robson. Combinatorial games with exponential space complete decision problems. In *Proceedings of the Mathematical Foundations of Computer Science 1984*, pages 498–506, London, UK, 1984. Springer-Verlag.

[70] J. M. Robson. N by N Checkers is EXPTIME complete. *SIAM Journal on Computing*, 13(2):252–267, May 1984.

[71] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[72] Thomas J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16:185–225, 1978.

[73] Michael Sipser. The history and status of the P versus NP question. In *STOC '92: Proceedings of the Twenty Fourth Annual ACM Symposium on Theory of Computing*, pages 603–618. ACM, 1992.

[74] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, second edition, 2005.

[75] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle*. Slocum Puzzle Foundation, 2006.

[76] Raymond Georg Snatzke. New results of exhaustive search in the game Amazons. *Theor. Comput. Sci.*, 313(3):499–509, 2004.

[77] Paul Spirakis and Chee Yap. On the combinatorial complexity of motion coordination. Report 76, Computer Science Department, New York University, 1983.

[78] James W. Stephens. The kung fu packing crate maze, 2003. `http://www.puzzlebeast.com/crate/index.html`.

[79] L. Stockmeyer and A. Meyer. Word problems requiring exponential time: Preliminary report. In *Proceedings of 5th ACM Symposium on Theory of Computing*, pages 1–9, 1973.

[80] L. J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, October 1976.

[81] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979.

[82] Max Tegmark and Nick Bostrom. Is a doomsday catastrophe likely? *Nature*, 438:754, Dec 2005.

[83] John Tromp. On size 2 Rush Hour logic. Manuscript, December 2000. `http://turing.wins.uva.nl/~peter/teaching/tromprh.ps`.

[84] John Tromp and Rudy Cilibrasi. Limits of Rush Hour Logic complexity. Manuscript, June 2004. `http://www.cwi.nl/~tromp/rh.ps`.

[85] Robert T. Wainwright. Life is universal! In *WSC '74: Proceedings of the 7th conference on Winter simulation*, pages 449–459, New York, NY, USA, 1974. ACM Press.

[86] David Wolfe. Go endgames are PSPACE-hard. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 125–136. Cambridge University Press, 2002.

[87] Honghua Yang. An NC algorithm for the general planar monotone circuit value problem. In *Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing*, pages 196–203, 1991.