Constraint Logic:
A Uniform Framework for Modeling Computation as Games

Erik D. Demaine[*] Robert A. Hearn[†]

Abstract

We introduce a simple game family, called Constraint Logic, where players reverse edges in a directed graph while satisfying vertex in-flow constraints. This game family can be interpreted in many different game-theoretic settings, ranging from zero-player automata to a more economic setting of team multiplayer games with hidden information. Each setting gives rise to a model of computation that we show corresponds to a classic complexity class. In this way we obtain a uniform framework for modeling various complexities of computation as games. Most surprising among our results is that a game with three players and a bounded amount of state can simulate any (infinite) Turing computation, making the game undecidable. Our framework also provides a more graphical, less formulaic viewpoint of computation. This graph model has been shown to be particularly appropriate for reducing to many existing combinatorial games and puzzles—such as Sokoban, Rush Hour, River Crossing, TipOver, the Warehouseman's Problem, pushing blocks, hinged-dissection reconfiguration, Amazons, and Konane (Hawaiian Checkers)—which have an intrinsically planar structure. Our framework makes it substantially easier to prove completeness of such games in their appropriate complexity classes.

1 Introduction

Games model computation. The idea that games serve as a useful model for certain kinds of computation is commonplace in complexity theory. The classic instance, highlighted in Papadimitriou's book [30, ch. 19] for example, is that PSPACE-complete computation can be modeled by a polynomial-length two-player game where the players take turns setting variables, one player wants to make a Boolean formula true, and the other player wants to make it false. But the connection runs much deeper than this classic instance. For example, even one-player puzzles, but with a polynomially unbounded number of moves, also model PSPACE-complete computation [25].

One goal of this paper is to show how a broad range of games—ranging from zero-player automata such as Conway's Game of Life, to a more economic setting of team multiplayer games with hidden information—model different complexities of computation—ranging from P-completeness to undecidability. Table 1 summarizes this correspondence. Most surprising among these results is that a game with just three players and a bounded amount of shared and hidden state models all decidable computation.

What is a game? It is important at this point to differentiate games from automata-based models of computation like Turing machines. The high-level idea is similar: at each step, some finite set of rules define a set of allowable moves which change the state in some predictable fashion. One key difference is

[*]MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, edemaine@mit.edu
[†]Neukom Institute for Computational Science, Dartmouth College, 6207 Moore Hall, Hanover, NH 03755, USA, robert.a.hearn@dartmouth.edu

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

Table 1: Game categories and their natural complexities. Constraint Logic is complete in each class.

in the size of the state: Turing machines have infinite tapes, so the state at any time is unbounded, while we always require the state of a game to be defined by a finite board position. We can, of course, bound the size of a Turing-machine tape, or use a circuit model of computation, but such restrictions limit the model of computation from general decidability. By contrast, we show that no such limitations exist for games.

Games are also naturally suited for the ideas of multiple players, teams, and hidden information. Each of these ideas has been grafted onto Turing-based models of computation—the most famous example being alternation [3], which corresponds to two players—although arguably these ideas come from games [32, 34]. Unlike the extensions of [32], we make the natural restriction that the moves of the game must cycle through the players in round-robin order. For any game, there is a standard decision question: does player X have a forced win from a given position?

A uniform framework. We develop one simple kind of game, *Constraint Logic*, which can be easily adapted to all different types of games: any number of players/teams, (polynomially) bounded and unbounded length, with and without hidden information. This approach naturally mimics Turing machines, which serve as a universal model of computation that can be easily restricted to model any complexity class in the classic time and space hierarchies. Such a uniform treatment of games has never been attained before.

Graphs, not formulae. In Constraint Logic, the game board is any undirected graph with weights of 1 or 2 on the edges and vertices. A board position is an orientation of this graph such that the total weight of edges directed into each vertex is at least that vertex's desire (weight). A move is just a reversal of an edge orientation that results in a valid board position. A computation is a sequence of moves; a computation accepts if it reverses a particular distinguished edge.

This model differs in several ways from traditional logical models of computation such as circuit or formula satisfiability. Constraint Logic is simple, being modeled directly around graphs. Circuits and formulae can be represented as graphs with sufficient augmentation (e.g., distinguishing variables from clauses/gates). But it is difficult to see the actual computation taking place in such a graph, whereas the dynamics of Constraint Logic defines the entire state in the orientation of the graph. This purely combinatorial view, without any explicit logical values, is much easier to work with in many cases.

Application to proving completeness. It is well-known that many one-player puzzles and two-player games are complete within their natural complexity class, as predicted by the middle two columns of Table 1. For example, Peg Solitaire is a bounded-length puzzle and NP-complete [43]; Sokoban and Rush Hour are unbounded-length puzzles and PSPACE-complete [6, 10, 13, 25]; Hex and Othello are bounded-length two-player games and PSPACE-complete [35, 27]; and generalized Chess, Checkers, and Go are unbounded-length two-player games and EXPTIME-complete [15, 38, 37]. For more examples, see the surveys [7, 14]. Most of these proofs are complicated, reducing from appropriate forms of formula satisfiability.

One motivation for developing our game model of computation is that it is much closer in flavor to many existing games, making the appropriate form of Constraint Logic a natural problem to reduce from. We have successfully developed such reductions for many different games: bounded-length one-player TipOver [19]; unbounded-length one-player sliding blocks, Sokoban, Rush Hour, the Warehouseman's Problem, and sliding tokens [25], River Crossing [21], Push-2-F [8], and hinged-dissection reconfiguration [20]; and bounded-length two-player Amazons, Konane (Hawaiian Checkers), and Cross Purposes [23]. In the cases of Sokoban, Rush Hour, and the Warehouseman's Problem, which were proved PSPACE-complete previously, our proofs based on Constraint Logic are simpler and (except for Rush Hour) establish hardness for weaker forms of the games. The complexities of many of the other games were open problems for several years; for example, sliding blocks was posed by Martin Gardner 40 years earlier [16].

This paper proves the formal underpinnings necessary for the reductions listed above to actually establish the desired form of completeness. Only the unbounded-length one-player case was established previously [25]; the other cases were conjectured in the papers above. We expect our framework of Constraint Logic to offer simple completeness proofs for many more games and puzzles. In particular, with our newly developed frameworks for zero-player games, two-player unbounded-length games, and team multiplayer games with hidden information, we expect a new wealth of applications.

The Constraint Logic framework provides a host of tools for making it easy to prove completeness of a game, as described in Section 3. The proof only needs to show how to implement two kinds of constraint graph vertices, one representing a kind of AND computation, and another representing a kind of OR computation. (In fact, even simpler forms of these vertices suffice to establish completeness.) The proof also needs to be able to construct edges that connect two vertices. Fortunately, the Constraint Logic framework can guarantee that the graph is planar, so the game-specific proof does not need to worry about crossing edges.[1] In contrast, crossover gadgets are the most complicated part of most previous completeness proofs for two-dimensional games; with constraint graphs, they come for free so are unnecessary to build.

As an illustrative example of the power of the Constraint Logic framework, Figure 1 shows the entire construction required to prove that sliding-block puzzles are PSPACE-complete. Thus the solution to this 40-year-old problem becomes almost trivial.

Outside of games, others have used the unbounded-length one-player form of Constraint Logic from [25] to establish the complexity of airport planning [26], steel slab stacking [28], finding paths between graph colorings [2], and morphing parallel graph drawings [41]. Recently we discovered an application of zero-player Constraint Logic to evolutionary graph theory [9].

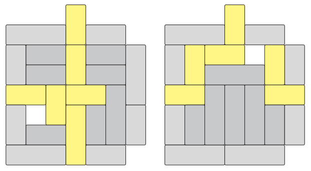


Figure 1: Constraint Logic gadgets showing PSPACE-completeness of sliding-block puzzles.

Related work. We draw inspiration from two main previous papers.

As mentioned above, the unbounded one-player case was considered previously in [25]; that paper, in turn, was inspired by the Rush Hour Logic of Flake and Baum [13]. The present work represents a culmination of generalizations to more players, bounded length, and hidden information.

The idea of team multiplayer games with hidden information comes from a paper by Peterson and Reif [32]. That paper also tries to make many connections between games and models of computation. In particular, they claim a result similar to ours: that certain team multiplayer games with hidden information are undecidable. Unfortunately, some of their results along these lines are flawed, as described in Section 7; in particular, their game TEAM-PEEK is in fact decidable. Nonetheless, we use several of their ideas.

2 Overview

This section gives a more detailed summary of our results and their context and importance, as well as providing an overview of the paper's structure.

Constraint Logic framework. Section 3 gives a formal definition of the Constraint Logic framework, defining constraint graphs and describing some of their properties, including the generality and utility of the complexity results for planar constraint graphs. The following sections then define specific Constraint Logics, which have additional restrictions appropriate to the corresponding game types.

Zero-player games. We begin our study of game categories in Section 4 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 [17]. More generally, the class of zero-player games corresponds naturally to ordinary computers, or deterministic space-bounded Turing machines—the kinds of computational tools we have available in the real world (at least until quantum computers develop further).

A bounded zero-player game is essentially a simulation that can run for only linear time. This style of computation is captured by *Boolean circuits*; determining their value is P-complete. We define a bounded zero-player Constraint Logic, and prove P-completeness by correspondence to Boolean circuits.

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 good examples. Because there is no longer a linear bound on the number of moves, it is not generally possible to determine the outcome in polynomial time. Indeed, Life has been shown to be Turing-universal [1, 44, 36] on an infinite grid. In other words, 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.[2] The canonical PSPACE-complete formula problem is Quantified Boolean Formula Satisfiability (QBF). We show that QBF corresponds to a natural unbounded zero-player Constraint Logic, called Deterministic Constraint Logic.

Deterministic Constraint Logic also represents a new style of reversible, monotone computation that potentially could be physically built, and could have significant advantages over conventional digital logic.

One-player games. Section 5 gives our results for one-player games. 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,

[1]There is one exception: in the bounded zero-player setting, no crossover gadget exists unless $NC^3 = P$.

[2]This result is not mentioned explicitly in the cited works, but it follows directly, at least from [36].

the goal could be to get a particular block to a particular location. We use 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?".

Bounded one-player games have a polynomial bound (typically linear) on the number of moves that can be made. Typically each move uses up some bounded resource. 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. The canonical NP-complete problem is Satisfiability (SAT). We show that SAT corresponds to the natural bounded one-player Constraint Logic. As in other game categories, reductions from this game are often dramatically simplified because of planarity. Planar SAT is also NP-complete [29], but that result is not generally useful for puzzle reductions.[3]

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. Because 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 [39] says that PSPACE = NPSPACE, so these puzzles can also be solved using deterministic polynomial space. The natural form of Constraint Logic for this type of puzzle corresponds to QBF, and is PSPACE-complete. It was developed in [24, 25]; the present paper extends the notion of Constraint Logic to other kinds of games.

Two-player games. With two-player games (Section 6), we are finally in territory familiar to both classical (economic) 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 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* [3]. 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.

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 gets used up by each move. In Hex, for example, each move fills a space on the board, and when all the spaces are full, the game must be over. The earliest hardness results for two-player games were PSPACE-completeness results for bounded games, beginning with Generalized Hex [11], and continuing with several two-player versions of known NP-complete problems [40]. The canonical PSPACE-complete game is simply QBF. QBF is equivalent to the question of whether the first player can win the following formula game: "Players take turns assigning truth values to a sequence of variables. When they are finished, player one wins if formula F is true; otherwise, player two wins." We show that the natural bounded two-player Constraint Logic corresponds directly to a variant formulation of QBF.

Unbounded two-player games have 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. Each of these games is EXPTIME-complete [15, 38, 37].[4] Thus, we finally reach games that are provably intractable [18]. These reductions are from an unbounded formula game that is similar to QBF, except that

[3]In Planar SAT, the graph corresponding to the formula is a planar bipartite graph, with variable nodes and clause nodes, plus a cycle connecting the variable nodes. The clause nodes are not connected, however. In contrast, a constraint 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 feature is a critical structure for puzzle reductions, because the victory condition is usually a local property (such as moving a block to a particular place) rather than a distributed property of the entire configuration.

[4]For Go, the result is only for Japanese rules.

variable assignments can be changed back and forth multiple times [42]. We show that one version of this formula game is equivalent to the natural unbounded two-player Constraint Logic.

Multiplayer games. In Section 7 we consider the generalization to multiplayer games. It turns out that naïvely adding players beyond two 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. But when there is hidden information, team games turn out to be different from two-player games. Therefore, we only consider team games of imperfect information, referring to them simply as "team 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 either to make the bid, or, if on defense, to set the other team. The hand ends after all the cards have been played. Focusing on a given hand also removes the random element from the game, making it potentially suitable for study within the present framework. Peterson and Reif [32] showed that bounded team games of private information are NEXPTIME-complete in general, by a reduction from Dependency Quantified Boolean Formulas (DQBF). We show that DQBF corresponds directly to the natural bounded team imperfect-information Constraint Logic.

In general, team games with private information are undecidable, even with three players. This result was claimed by Peterson and Reif in 1979 [32]. However, as mentioned above, there are several problems with the proof, which we address in Section 7.2. We use an improved version of a team computation game given in [32] to prove that a natural unbounded team Constraint Logic is indeed undecidable, thus showing that games are as powerful a computational model as general Turing machines.[5]

Undecidable games using bounded space—finite physical resources—at first seem counterintuitive. Such games have only finitely many configurations, so eventually a position must repeat. Yet somehow the game's state must effectively encode the contents of an unboundedly long Turing-machine tape!

The resolution to the apparent paradox is that, though the position will eventually repeat, the players will not know when it has repeated. To play optimally, they must remember the entire game history. It might seem, therefore, that the unbounded state has simply been transferred from the machine to the players' memory. But there is a big difference: the players' resources are not part of the problem statement. The players may be viewed simply as a source of nondeterminism. Thus we arrive at the fundamental difference of computation through games: *a game computation is a manipulation of finite resources*.

3 Constraint Logics in General

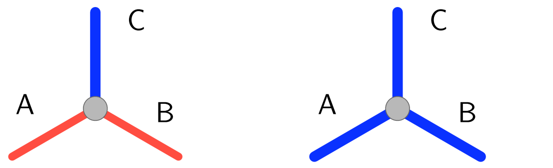
The general model of games we develop is based on the idea of a *constraint graph*; the rules defining legal moves on such graphs are called *Constraint Logic*. In later sections the graphs and the rules will be specialized to produce zero-player, one-player, two-player, etc. games. A game played on 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.

A *constraint graph* is a directed graph with edge weights among $\{1, 2\}$. 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 that results in a legal configuration. Generally, in any game, the goal will be to reverse a given edge by executing a sequence of (legal) 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 moves is forced. For the bounded games, each edge may only reverse once.

[5]Some results in the field of interactive proof systems can also be interpreted as showing that there are undecidable games played on finite boards [4, 12]. However, these games occur in a probabilistic setting, and are thus outside the notion of game considered here.

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. An *AND vertex* (Figure 2(a)) has minimum inflow constraint 2 and incident edge weights of 1, 1, and 2. It 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. An *OR vertex* (Figure 2(b)) has minimum inflow constraint 2 and incident edge weights of 2, 2, and 2. It 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.



(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 2: 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. Blue edges are thicker than red edges 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.

For some of the game categories, there can be many subtypes 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 cases there are alternate vertex "basis sets" that enable simpler reductions to other problems than do the complete set of ANDs and ORs. See [22] for details.

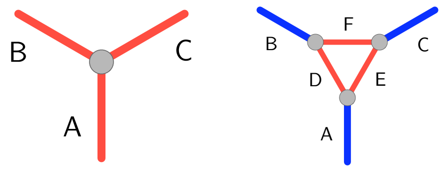
Directionality; Fanout. As implied above, although it is natural to think of AND and OR vertices as having inputs and outputs, there is nothing enforcing this interpretation. A sequence of edge reversals could first direct both red edges into an AND vertex, and then direct its blue edge outward; in this case, we 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.

3.1 Constraint Graph Conversion Techniques

In the reductions that follow, 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(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.



(a) CHOICE vertex

(b) Equivalent AND/OR subgraph

Figure 3: CHOICE vertex conversion.

The AND/OR subgraph shown in Figure 3(b) has the same constraints on its A, B, and C edges as the CHOICE vertex does. The replacement subgraph may not be substituted directly for a CHOICE vertex, however, because its terminal edges are blue, instead of red. This brings us 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 cannot 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 vertex. The conversion is shown in Figure 4. 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.

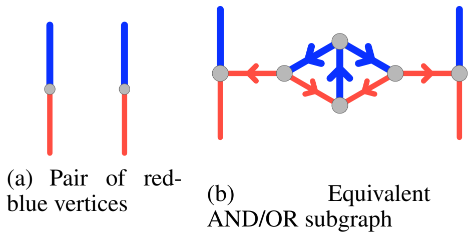
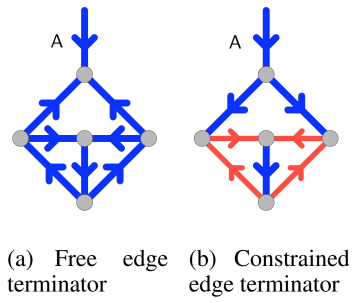

(a) Pair of red-blue vertices (b) Equivalent AND/OR subgraph

Figure 4: Red-blue vertex conversion. Red-blue vertices, which have an inflow constraint of 1 instead of 2, are drawn smaller than other vertices.

It will occasionally be useful to use blue-blue 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).

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 5(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 5(b) forces A to point down.



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

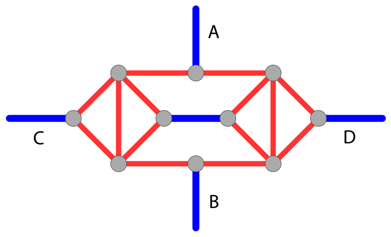
Figure 5: Terminating loose edges.

3.2 Planarity

All of the following complexity results, except for the bounded zero-player case, apply even when the constraint graphs are required to be planar. The basic crossover gadgets that enable this are given in [25] for the unbounded one-player case, and are reproduced in Figures 6(a) and 6(b). In addition to AND and OR vertices, Figure 6(a) contains red-red-red-red vertices; these need any two edges to be directed inward to satisfy the inflow constraint of 2. The "half-crossover gadget" in Figure 6(b) may be substituted in for each red-red-red-red vertex, using the red-blue conversion described earlier.

The relevant property of the crossover gadget is that 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.

The applications to Constraint Logics other than Unbounded NCL are generally straightforward. For example, for Bounded NCL, no edge ever need reverse more than once during a crossing sequence. For games with more than one player, the crossings are always arranged so that only edges controlled by a single player cross, reverting effectively to the one-player case. The one case that must be treated slightly differently is Unbounded Deterministic Constraint Logic; we refer the reader to [22] for details.



(a) Crossover

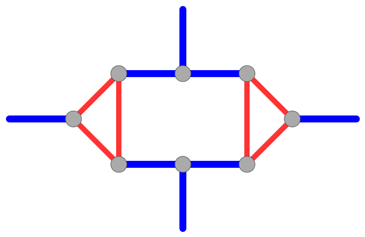

(b) Half-crossover

Figure 6: Planar crossover gadgets.

4 Zero-Player Games

The Constraint Logic formalism does not restrict the set of moves available on a constraint graph to a unique next move from any given configuration. To obtain a deterministic version, we must further constrain the legal moves. Rather than propose a rule that 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 idea is that each vertex should allow "signals" to "flow" through it if possible. For example, if both red edges reverse inward at an AND vertex, then the next move reverses the blue edge.

4.1 Bounded Games

In the bounded case, we start with a constraint graph G_0 and a set R_0 of unreversible edges. At each step i, the game reverses all edges that are reversible and have not been reversed before:

$$\begin{aligned} R_{i+1} &= \{e \mid \text{reversing edge } e \text{ is a legal move in } G_i \text{ and } e \notin R_0 \cup R_1 \cup \cdots \cup R_i\}, \\ G_{i+1} &= G_i \text{ with edges in } R_{i+1} \text{ reversed.} \end{aligned}$$

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, because 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?

Theorem 1 *Bounded DCL is P-complete.*

Proof: Given a monotone Boolean circuit C (a Boolean circuit with no NOT gates), 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 3.1. For the input nodes, we use terminators as in Figures 5(a) and 5(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. □

Note that the monotone Boolean circuit value problem is in NC^3 if the circuit is required to be planar; thus, we cannot show P-completeness for planar graphs in this one case.

4.2 Unbounded Games

The definition of Deterministic Constraint Logic (DCL) is more complicated in the unbounded case, where edges may reverse more than once. The existing rule would no longer flow signals through vertices: 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, leading 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 possible reversals away from the vertex pointed to by the edge.

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,

$$\begin{aligned} R_{i+1} &= \{e \mid e \text{ points to } v \text{ in } G_i, \text{ and either } e \in R_i \text{ or } v \in F(G_i, R_i) \text{ but not both}\}, \\ G_{i+1} &= G_i \text{ with edges in } R_{i+1} \text{ reversed.} \end{aligned}$$

The effect of this rule is that signals flow through constraint graphs as desired, but when a signal reaches a vertex that it cannot "activate", or "flow through", it instead "bounces". For AND/OR graphs, bouncing can happen only when a single red edge reverses into an AND vertex and the other red edge is directed away.

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: 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.

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?

Theorem 2 *DCL is PSPACE-complete.*

Proof idea: Reduction from QBF. The reduction is rather elaborate; see [22] for details. □

5 One-Player Games

The one-player version of Constraint Logic is called Nondeterministic Constraint Logic (NCL). The rules are simply that on a turn the player reverses a single edge that results in a legal configuration. The goal is to reverse a particular edge.

5.1 Bounded Games

Bounded Nondeterministic Constraint Logic (Bounded NCL) 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?

Theorem 3 *Bounded NCL is NP-complete.*

Proof of Theorem 3: We reduce 3SAT 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 Section 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 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 7.

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.

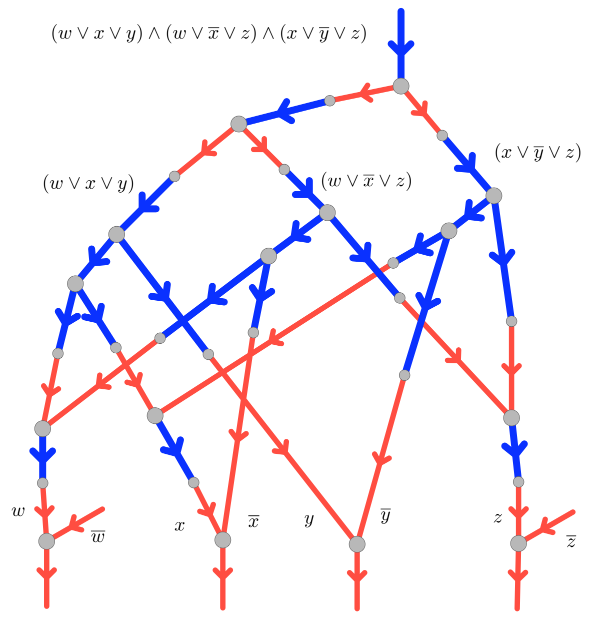

$(w \vee x \vee y) \wedge (w \vee \overline{x} \vee z) \wedge (x \vee \overline{y} \vee z)$
$(w \vee x \vee y)$
$(w \vee \overline{x} \vee z)$
$(x \vee \overline{y} \vee z)$
w
$\overline{w}$
x
$\overline{x}$
y
$\overline{y}$
z
$\overline{z}$

Figure 7: 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.

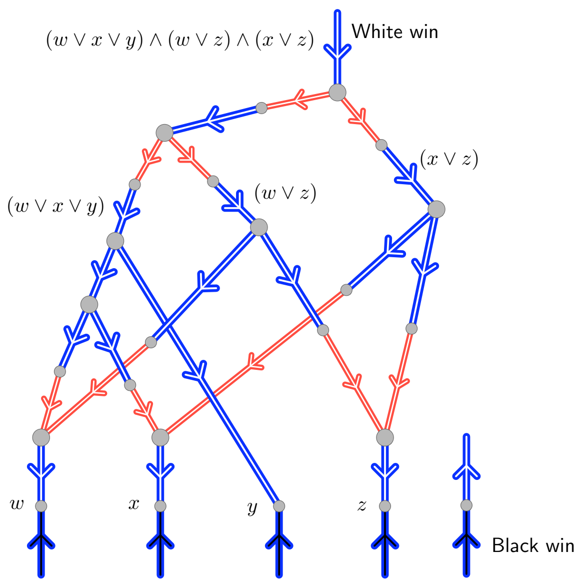


Figure 8: A constraint graph corresponding to the G_{pos}(POS CNF) formula game $(w \vee x \vee y) \wedge (w \vee z) \wedge (x \vee z)$. Edges corresponding to variables, clauses, and the entire formula are labeled.

Using the techniques described in Section 3.1, 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.

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. □

5.2 Unbounded Games

Nondeterministic Constraint Logic (NCL), formally defined as follows, is shown PSPACE-complete in [25].

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?

6 Two-Player Games

The two-player version of Constraint Logic, Two-Player Constraint Logic (2CL), is defined as follows. 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. As before, a move must reverse exactly one edge and result in a valid configuration. Each player has a target edge he is trying to reverse.

6.1 Bounded Games

The bounded case permits each edge to reverse at most once:[6]

[6]We can assume without loss of generality that the game ends with no winner if a player is unable to move. 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

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. 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).

We reduce Schaefer's game G_{pos}(POS CNF), a variant of QBF [40], to Bounded 2CL.

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 if 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.

Theorem 4 *Bounded 2CL is PSPACE-complete.*

Proof: 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 8; compare to Figure 7. (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.

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 cannot win the formula game, then White cannot play so as to make a set of variables true which will satisfy the formula, and thus he cannot 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.

The construction shown in Figure 8 is not an AND/OR graph; we must still convert it to an equivalent one. The standard conversion techniques from Section 3.1 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. □

6.2 Unbounded Games

The unbounded case simply removes the restriction of edges reversing at most once:

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).

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 given definition is more consistent with the other varieties of Constraint Logic: always, the goal is to reverse a given edge.

To show that 2CL is EXPTIME-hard, we reduce from one of the several Boolean formula games shown EXPTIME-complete by Stockmeyer and Chandra [42]:

> ***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 Player II has no winning condition, so the most he can hope for is a draw, by preventing Player I from ever winning. This does not affect our decision question of whether White can force a win.

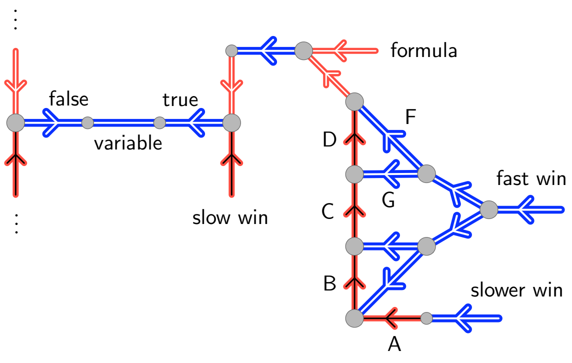


Figure 9: Reduction from G_6 to 2CL.

Theorem 5 *2CL is EXPTIME-complete.*

Proof: The essential elements of the reduction from G_6 to 2CL are shown in Figure 9. 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).

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. 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 10. 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

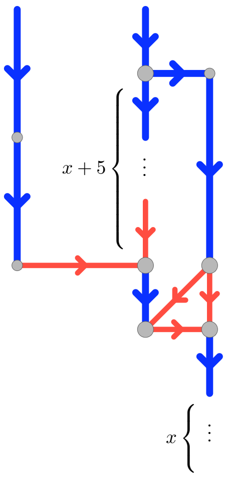


Figure 10: Path-length-equalizer gadget.

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.

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. $\square$

7 Team Games

The natural team private-information Constraint Logic assigns to each player a set of edges he can reverse, and a set of edges whose orientation he can see, in addition to the target edge he aims to reverse.

7.1 Bounded Games

Bounded Team Private Constraint Logic is formally defined as follows:

BOUNDED TEAM PRIVATE CONSTRAINT LOGIC (BOUNDED TPCL)
INSTANCE: AND/OR constraint graph G; integer N; for $i \in \{1 \dots N\}$: sets $E_i \subset V_i \subset G$; edges $e_i \in E_i$; partition of $\{1 \dots N\}$ into nonempty sets W and B.
QUESTION: Does White have a forced win in the following game? Players $1 \dots N$ 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.

Theorem 6 *Bounded TPCL is NEXPTIME-complete.*

Proof idea: The proof is a reduction from the Dependency QBF problem introduced in [32]. $\square$

7.2 Unbounded Games

To enable a simpler reduction to an unbounded form of the team Constraint Logic, we 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.

TEAM PRIVATE CONSTRAINT LOGIC (TPCL)
INSTANCE: AND/OR constraint graph G; integer N; for $i \in \{1 \dots N\}$: sets $E_i \subset V_i \subset G$; edges $e_i \in E_i$; partition of $1 \dots N$ into nonempty sets W and B; integer k.
QUESTION: Does White have a forced win in the following game? Players $1 \dots N$ 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, we discuss the earlier results of Peterson and Reif [32].

TEAM-PEEK. Peterson and Reif [32] state that a particular space-bounded game with alternating turns, TEAM-PEEK, is undecidable. (TEAM-PEEK is a team version of Stockmeyer and Chandra's EXPTIME-complete game PEEK [42].) There are two problems with this claim. First, there is a simple mistake in the definition, making the undecidability claim false. To see this, consider the following formal statement of TEAM-PEEK, which is equivalent to the more physical version described in [32]:

TEAM-PEEK
INSTANCE: DNF Boolean formula F in variables S, integer N; for $i \in \{1 \dots N\}$: sets $X_i \subset V_i \subset S$; partition of $1 \dots N$ into nonempty sets W and B; S assignment α.
QUESTION: Does White have a forced win in the following game? Players $1 \dots N$ take turns in that order. Player i only knows the truth assignment to the variables in V_i, and moves by changing the truth assignment of any subset of the variables X_i. White (Black) wins if F is ever true after a move by Player $i \in W$ (B).

We show that TEAM-PEEK is decidable when White has two or more players and Black has one player (contrary to [32, Theorem 5]). Whatever the turn order, the White players will wind up playing in sequence. Now it is easy to tell whether White can win before Black's first move, so assume that they cannot. Then, either White can win immediately regardless of Black's move, which is also easy to determine, or they cannot. Suppose they cannot. Then, they cannot have a forced win at all, because whatever moves they make in sequence on any pair of turns, there is always some move Black could have just made that prevented a win.

The basic problem with the game definition is that allowing a player to change any or all of his variables in a single turn, instead of at most one variable as in PEEK, prevents threats and thus forcing moves. Thus the standard machinery of building Turing-machine-acceptance reductions to formula games breaks down.

Round-robin play. The second problem has to do with the order in which players move. TEAM-PEEK's definition follows the natural form of game play in which players take turns in round robin. However, the problem developed in [32, page 355] for a reduction to TEAM-PEEK does not have this property:

Given a TM M ... The game ... will be based on having each of the $\exists$-players find a sequence of configurations of M which on an input that leads to acceptance. Hence, each $\exists$-player will give to the $\forall$-player on request the next character of its sequence of configurations (secretly from the other). Each $\exists$-player does this secretly from the other $\exists$-player. The configuration will [be] in the form: $\#C_0\#C_1\# \dots \#C_m\#$, 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: ...

The $\forall$-player verifies the sequences by ensuring that the initial configurations match the input, that the final sequences are accepting, and that the transitions are valid. The existential team wins if a player generates a valid accepting history; the universal player wins if it detects an invalid history. The key is that the validity of the transitions can be checked with only a fixed amount of memory, by running one of the players ahead to the next # symbol, and stepping through both histories symbol by symbol.

It is implicit in the definition of the game 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.

Reif has confirmed in a personal communication regarding round-robin play in TEAM-PEEK [33] that "it looks like therefore the players do not play round robin".[7] Indeed, in the general definition of game in [32], it is stated that "Players need not take turns in a round-robin fashion. The rules will dictate whose turn is next", and that "A player may not know how many turns were taken by other players between its turns." These notions deviate from (and hence exclude) the intuitive notion of a game, where players take turns in order and are aware of what happens between their turns. Therefore we work to strengthen the approach to apply to this natural form of game.

Undecidability. To solve the above problems, we introduce a somewhat more elaborate computation game, in which the players take successive turns, and which we show to be undecidable. We reduce this game to a formula game, and the formula game to TPCL.

The new computation game will be similar to the above computation 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 (or the history could be checked against the input). 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.

We show that TPCL is undecidable by a series of reductions beginning with the following computation game.

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 7 *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 #).

[7]Both the mistake in definition described above and the turn-order problem also apply to all of the TEAM-PEEK variants defined in [31].

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.[8] 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.[9] 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. $\square$

Next we 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, 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.

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 $\overline{F'}$.
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 8 *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, [5].) 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.

[8]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.

[9]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 [32], 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.

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. □

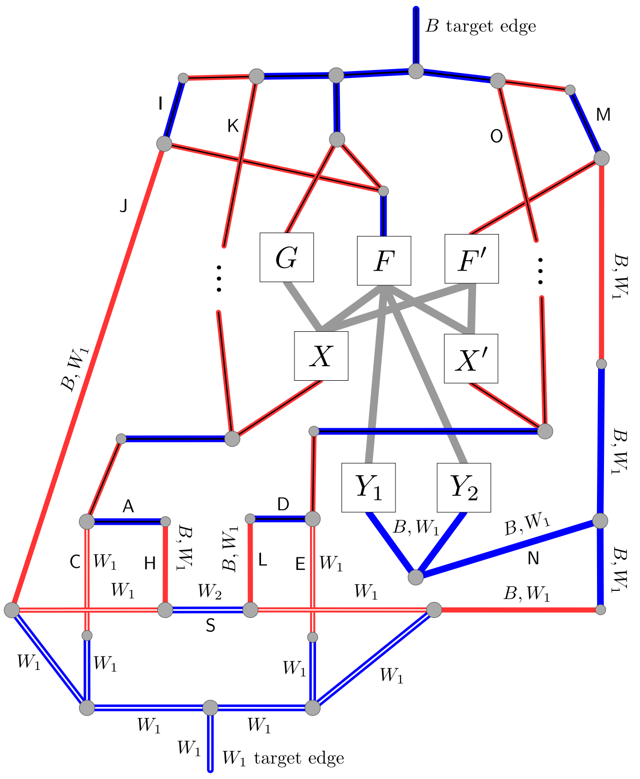


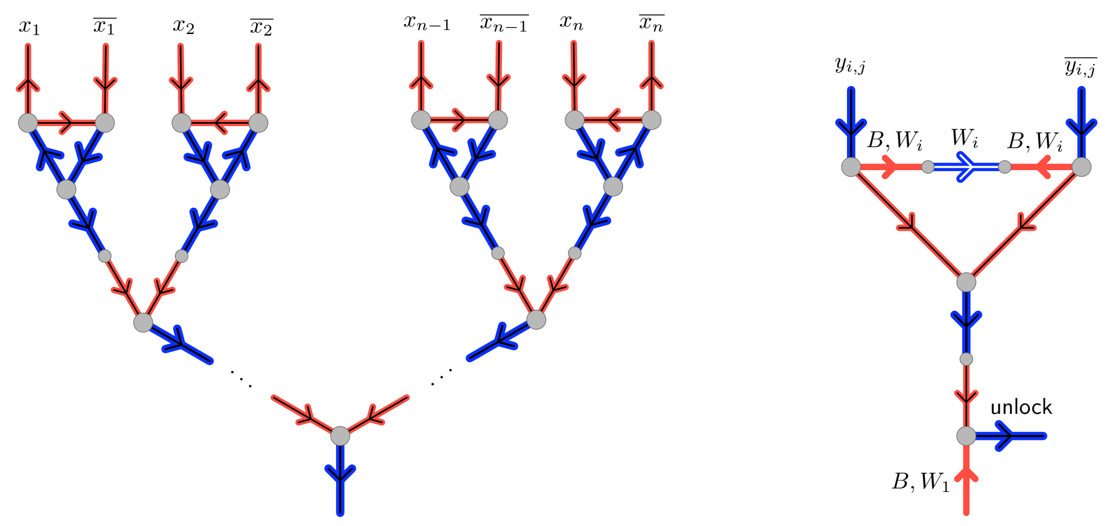
Figure 11: Reduction from TEAM FORMULA GAME to TPCL. White edges and multiplayer edges are labeled with their controlling player(s); all other edges are black. Thick gray lines represent bundles of black edges.

TPCL Reduction. Finally, we are ready to complete the undecidability reduction for TPCL. The overall reduction from TEAM FORMULA GAME is shown in Figure 11. 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 [25]), shown in Figure 12(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 12(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



(a) Latches locking black variables (b) White variable

Figure 12: Additional gadgets for TPCL reduction.

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 11 (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 9 *TPCL is undecidable, even with $N = 3$ players.*

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. □

References

[1] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways*. A. K. Peters, Ltd., Wellesley, MA, second edition, 2001–2004.

[2] Paul Bonsma and Luis Cereceda. Finding paths between graph colourings: PSPACE-completeness and super-polynomial distances. In *Proceedings of the 32nd International Symposium on Mathematical Foundations of Computer Science*, 2007. To appear.

[3] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[4] Anne Condon and Richard J. Lipton. On the complexity of space bounded interactive proofs (extended abstract). In *FOCS*, pages 462–467. IEEE, 1989.

[5] 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.

[6] 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.

[7] Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *Proceedings of the 26th Symposium on Mathematical Foundations in Computer Science*, volume 2136 of *Lecture Notes in Computer Science*, pages 18–32, Marianske Lazne, Czech Republic, August 2001. Full paper available at http://www.arXiv.org/abs/cs.CC/0106019.

[8] 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.

[9] Erik D. Demaine, Robert A. Hearn, and Erez Lieberman. Evolving populations and their computational ability. Manuscript in preparation, 2007.

[10] Dorit Dor and Uri Zwick. SOKOBAN and other motion planning problems. *Computational Geometry: Theory and Applications*, 13(4):215–228, 1999.

[11] 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.

[12] Uriel Feige and Adi Shamir. Multi-oracle interactive protocols with constant space verifiers. *J. Comput. Syst. Sci.*, 44(2):259–271, 1992.

[13] 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.

[14] 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.

[15] 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.

[16] Martin Gardner. The hypnotic fascination of sliding-block puzzles. *Scientific American*, 210:122–130, 1964.

[17] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, October 1970.

[18] Juris Hartmanis and Richard E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematics Society*, 117:285–306, 1965.

[19] Robert Hearn. TipOver is NP-complete. *Mathematical Intelligencer*, 28(3):10–14, 2006.

[20] Robert Hearn, Erik Demaine, and Greg Frederickson. Hinged dissection of polygons is hard. In *Proc. 15th Canad. Conf. Comput. Geom.*, pages 98–102, 2003.

[21] Robert A. Hearn. The complexity of sliding block puzzles and plank puzzles. In *Tribute to a Mathemagician*, pages 173–183. A K Peters, 2004.

[22] Robert A. Hearn. *Games, Puzzles, and Computation*. PhD dissertation, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, May 2006. http://www.swiss.ai.mit.edu/~bob/hearn-thesis-final.pdf.

[23] Robert A. Hearn. Amazons, Konane, and Cross Purposes are PSPACE-complete. In R. J. Nowakowski, editor, *Games of No Chance 3*, 2007. To appear.

[24] Robert A. Hearn and Erik D. Demaine. The nondeterministic constraint logic model of computation: Reductions and applications. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming (ICALP 2002)*, volume 2380 of *Lecture Notes in Computer Science*, pages 401–413, Malaga, Spain, July 2002.

[25] 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".

[26] Malte Helmert. New complexity results for classical planning benchmarks. In *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS 2006)*, pages 52–61, 2006.

[27] Shigeki Iwata and Takumi Kasai. The Othello game on an $n \times n$ board is PSPACE-complete. *Theoretical Computer Science*, 123:329–340, 1994.

[28] F. G. König, M. E. Lübbecke, R. H. Möhring, G. Schäfer, and I. Spenke. Solutions to real-world instances of PSPACE-complete stacking. Preprint 2007/4, Institut für Mathematik, Technische Universtät Berlin, 2007.

[29] David Lichtenstein. Planar formulae and their uses. *SIAM J. Comput.*, 11(2):329–343, 1982.

[30] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[31] 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.

[32] Gary L. Peterson and John H. Reif. Multiple-person alternation. In *FOCS*, pages 348–363. IEEE, 1979.

[33] John Reif, 2006. Personal communication.

[34] 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.

[35] Stefan Reisch. Hex ist PSPACE-vollständig (Hex is PSPACE-complete). *Acta Informatica*, 15:167–191, 1981.

[36] Paul Rendell. Turing universality of the game of life. In *Collision-based computing*, pages 513–539, London, UK, 2002. Springer-Verlag.

[37] J. M. Robson. The complexity of Go. In *Proceedings of the IFIP 9th World Computer Congress on Information Processing*, pages 413–417, 1983.

[38] J. M. Robson. N by N Checkers is EXPTIME complete. *SIAM Journal on Computing*, 13(2):252–267, May 1984.

[39] Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[40] Thomas J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16:185–225, 1978.

[41] Michael John Spriggs. *Morphing Parallel Graph Drawings*. PhD dissertation, University of Waterloo, May 2007. http://uwspace.uwaterloo.ca/handle/10012/3083.

[42] Larry J. Stockmeyer and Ashok K. Chandra. Provably difficult combinatorial games. *SIAM Journal on Computing*, 8(2):151–174, 1979.

[43] Ryuhei Uehara and Shigeki Iwata. Generalized Hi-Q is NP-complete. *Transactions of the IEICE*, E73:270–273, February 1990.

[44] 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.