Toward a General Complexity Theory of Motion Planning: Characterizing Which Gadgets Make Games Hard

Erik D. Demaine
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
edemaine@mit.edu

Dylan H. Hendrickson
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
dylanhen@mit.edu

Jayson Lynch
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, USA
jaysonl@mit.edu

Abstract

We begin a general theory for characterizing the computational complexity of motion planning of robot(s) through a graph of "gadgets," where each gadget has its own state defining a set of allowed traversals which in turn modify the gadget's state. We study two general families of such gadgets within this theory, one which naturally leads to motion planning problems with polynomially bounded solutions, and another which leads to polynomially unbounded (potentially exponential) solutions. We also study a range of competitive game-theoretic scenarios, from one player controlling one robot to teams of players each controlling their own robot and racing to achieve their team's goal. Under certain restrictions on these gadgets, we fully characterize the complexity of bounded 1-player motion planning (NL vs. NP-complete), unbounded 1-player motion planning (NL vs. PSPACE-complete), and bounded 2-player motion planning (P vs. PSPACE-complete), and we partially characterize the complexity of unbounded 2-player motion planning (P vs. EXPTIME-complete), bounded 2-team motion planning (P vs. NEXPTIME-complete), and unbounded 2-team motion planning (P vs. undecidable). These results can be seen as an alternative to Constraint Logic (which has already proved useful as a basis for hardness reductions), providing a wide variety of agent-based gadgets, any one of which suffices to prove a problem hard.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases motion planning, computational complexity, NP, PSPACE, EXP, NEXP, undecidability, games

Digital Object Identifier 10.4230/LIPIcs.ITCS.2020.62

Acknowledgements This work grew out of an open problem session from MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.890) from Fall 2014.

1 Introduction

Most hardness proofs are based on ***gadgets*** — local fragments, each often representing corresponding fragments of the input instance, that combine to form the overall reduction. Garey and Johnson [10] called gadgets "basic units" and the overall technique "local replacement proofs". The search for a hardness reduction usually starts by experimenting with small candidate gadgets, seeing how they behave, and repeating until amassing a sufficient collection of gadgets to prove hardness.

This approach leads to a natural question: what gadget sets suffice to prove hardness? There are many possible answers to this question, depending on the precise meaning of "gadget" and the style of problem considered. Schaefer [17] characterized the complexity of all ***Boolean constraint satisfiability*** gadgets, with a dichotomy between polynomial problems (e.g., 2SAT, Horn SAT, dual-Horn SAT, XOR SAT) and NP-complete problems (e.g., 3SAT, 1-in-3SAT, NAE 3SAT). At STOC'97, Khanna, Sudan, Trevisan, and Williamson [13] refined this result to characterize ***approximability*** of constraint satisfaction problems, forking into polynomial, APX-complete, Poly-APX-complete, Nearest-Codeword-complete, and Min-Horn-Deletion-complete. Introduced at CCC'08, ***Constraint Logic*** [7, 11] proves sufficiency of small sets of gadgets on directed graphs that always satisfy one local rule (weighted in-degree at least 2), in many game types (1-player, 2-player, and team games, both polynomially bounded and unbounded), although the exact minimal sets of required gadgets remain unknown.

The aforementioned general techniques naturally model "global" moves that can be made anywhere at any time (while satisfying the constraints). Nonetheless, the techniques have been successful at proving hardness for problems where moves must be made local to an agent/robot that traverses the instance. For single-player agent-based problems, the ***doors-and-buttons*** framework (described in [9] and improved by [19] and [18]) is a good example of classifying a universe of abstract motion planning problems which can then be applied. In addition, the ***door gadget*** used to prove Lemmings [20] and various Nintendo games [2] PSPACE-complete served as a primary example of the form of gadget we wanted to generalize.

In this paper, we analyze which gadgets suffice for hardness in a general ***semi-static motion planning problem*** where one or more agents/robots traverse a given environment, which only changes in response to the agent's actions, from given start location(s) to given goal location(s). We study a very general model of gadget, where the gadget changes state when it gets traversed by an agent according to a general transition function, enabling and/or disabling certain traversals in the future. We study this model from the traditional single-robot (one-player) perspective, extending our initial work on this case [6], as well as from the perspective of two robots or teams of robots competing to reach their respective goals. We also analyze natural settings where the number of moves is polynomially bounded, because each gadget can be traversed only a bounded number of times, or more general settings where gadgets can be re-used many times and thus the number of moves can be exponential in the environment complexity. In each case, we partially or fully characterize which gadgets suffice to make the motion planning problem hard (NP-hard, PSPACE-hard, EXPTIME-hard, NEXPTIME-hard, or RE-hard, depending on the scenario), and conversely which gadgets result in a polynomially solvable problem (NL or P). Table 1 summarizes our results.

1.1 Gadget Model and Motion-Planning Games

In general, we model a ***gadget*** as consisting of a finite number of ***locations*** (entrances/exits) and a finite number of ***states***; see Figure 1 for two examples. We may also consider a family of gadgets parameterized by the problem size. In this case we restrict the number of locations and states to be polynomial in the size of the problem. Each state s of the gadget defines a labeled directed graph on the locations, where a directed edge (a, b) with label s' means that a robot can enter the gadget at location a and exit at location b, and that such a traversal forcibly changes the state of the gadget to s'. Equivalently, a gadget is specified

	1-Player Game	2-Player Game	Team Game
Polynomially Bounded (DAG gadgets)	**NL** vs. **NP-complete**: full characterization [§5]	**P** vs. **PSPACE-complete**: full characterization [§6]	**P** vs. **NEXPTIME-complete**: full characterization [§7]
Polynomially Unbounded (reversible, deterministic gadgets)	**NL** vs. **PSPACE-complete**: full characterization [§2] **Planar**: equivalent [§2.3]	**P** vs. **EXPTIME-complete**: partial characterization [§3]	**P** vs. **RE-complete** (⇒ **Undecidable**): partial characterization [§4]

Table 1 Summary of our results for k-tunnel gadgets (with additional constraints listed in the left column). A "full characterization" means that we give an easily checkable condition on the allowed gadget set that determines the complexity of the corresponding motion planning problem; a "partial characterization" means that we give two easily checkable conditions on the allowed gadget set, one for the easy class and one for the hard class, each of which suffices to establish the complexity of the corresponding motion planning problem.

by its ***transition graph***,[1] a directed graph whose vertices are state/location pairs, where a directed edge from (s, a) to (s', b) represents that the robot can traverse the gadget from a to b if it is in state s, and that such traversal will change the gadget's state to s'. Gadgets are ***local*** in the sense that traversing a gadget does not change the state of any other gadgets.

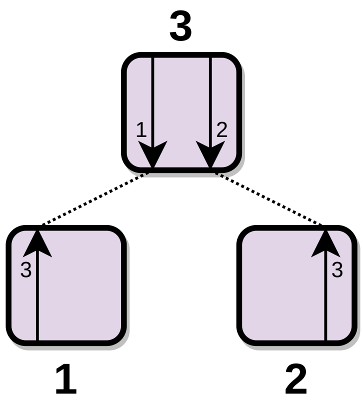
A ***system of gadgets*** consists of gadgets, their initial states, and a ***connection graph*** on the gadgets' locations. [2] If two locations a, b of two gadgets (possibly the same gadget) are connected by a path in the connection graph, then the robot can traverse freely between a and b (outside the gadgets). (Equivalently, we can think of locations a and b as being identified, effectively contracting connected components of the connection graph.) These are all the ways that the robot can move: exterior to gadgets using the connection graph, and traversing gadgets according to their current states.

We define a general family of ***motion planning*** problems involving one or more robots, each with their own start and goal location, in a system of gadgets. In a ***one-player game***, we are given a system of gadgets, a single robot that starts at a specified start location, and we want to decide whether there is a sequence of moves that brings the robot to a specified goal location. (This problem is perhaps the most common setting for robot motion planning.)

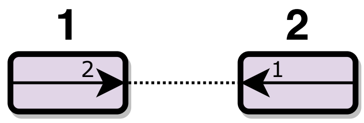
In a ***two-player game***, we are given a system of gadgets and the start and goal locations of two robots, and two players alternate moving their own robot by traversing a single gadget (entering at a location reachable from the robot's current location via the connection graph). Both players have complete information about the locations of the robots, the locations of the gadgets, and the states of the gadgets. Here we count gadget traversals as costing one move, and view movement in the connection graph as instantaneous/free. The goal is to decide whether the first player has a ***forced win***, that is, their robot can reach their goal location before the second player's does, no matter how the second player responds to the first player's moves. In a ***team game***, there are more than two robots, each controlled by a

[1] In [6], the transition graph is called the "state space", but we feel that "transition graph" more clearly captures the automaton nature of transitions, which are discrete and directed.

[2] In [6], locations could only be matched to exactly one other location and a 'branching hallway' gadget was introduced to fulfill the need of the connection graph.



(a) The locking 2-toggle gadget (L2T). In the top state 3, you can traverse either tunnel going down, which blocks off the other tunnel until you reverse the initial traversal.



(b) The 1-toggle gadget. Traversing the tunnel reverses the direction that it can be traversed.

Figure 1 Basic gadgets that can be simulated by any interacting-k-tunnel reversible deterministic gadget, as shown in Section 2.1.

single player, and the robots/players are partitioned into two teams; the goal of each team is to get any one of its player's robot to their goal location. Crucially, after a team game begins, each player has only partial information of the current gadgets' states: they can only see the state of the gadgets reachable by their robot via the connection graph.

We also define ***planar motion planning***. In this case, the cyclic order of locations on a gadget is specified, and the system of gadgets must be embedded in the plane without intersections. Specifically, construct the following graph from a system of gadgets: replace each gadget with a wheel graph, which has a cycle of vertices corresponding to the locations on the gadget in the appropriate order, and a central vertex connected to each location. Connect locations on these wheels with edges according to the connection graph. The system of gadgets is ***planar*** if this graph is planar. In planar motion planning, we restrict the problem to planar systems of gadgets. Note that this allows rotations and reflections of gadgets, but no other permutation of their locations. In some contexts, one may want to disallow reflections of gadgets, which corresponds to imposing a handedness constraint on the planar embedding of each wheel.

1.2 Gadget Types

We define different subclasses of gadgets that naturally model motion planning where the number of moves is either polynomially bounded or unbounded (potentially exponential).

In both cases, we require that the various states of a gadget differ only in their orientations of the possible traversals. More precisely, a ***k-tunnel*** gadget has $2k$ locations, paired in a perfect matching whose pairs are called ***tunnels***, such that each state defines which direction(s) each tunnel can be traversed. All of the gadgets we consider in this paper are k-tunnel.

In the polynomial case, we focus on "DAG" gadgets. First define the ***state-transition (multi)graph*** of a gadget to have a vertex for each state, and a directed edge from s to s' for each possible traversal of the gadget in state s that leads to state s'. (This graph can be obtained from the transition graph by combining together all vertices with the same state.)

Then a gadget is a ***DAG*** if its state-transition graph is a directed acyclic graph. Such gadgets naturally lead to polynomially bounded motion planning, as every gadget traversal consumes potential within that gadget, as measured by the state (e.g., in a topological ordering of the state-transition graph). The total number of traversals is thus bounded by the total number of states in all gadgets in the system. (It is not enough to require that the transition graph be acyclic, because the robot can use the connection graph and other gadgets to reach other locations of this gadget in between traversals.)

In the polynomially unbounded case, we focus on gadgets that are "deterministic" and "reversible". A gadget is ***deterministic*** if its transition graph has maximum out-degree ≤ 1; i.e., a robot entering the gadget at some location in some state can exit at only one location and in only one state. A gadget is ***reversible*** if its transition graph has the reverse of every edge, i.e., it is the bidirectional version of an undirected graph. Thus a robot can immediately undo any gadget traversal.[3] Together, determinism and reversibility are equivalent to requiring that the transition graph is the bidirectional version of a matching.

We also consider ***planar motion planning*** problems with a ***planar*** system of gadgets, where the gadgets and connections are drawn in the plane without crossings. Planar gadgets are drawn as small regions (say, disks) with their locations as points in a fixed clockwise order along their boundary. A single gadget type thus corresponds to multiple planar gadget types, depending on the choice of the clockwise order of locations. Connections are drawn as paths connecting the points corresponding to the endpoint locations, without crossing gadget interiors or other connections.

The gadget model described above is an extension of the model introduced in [6], which characterized 2-state deterministic reversible k-tunnel gadgets that make for PSPACE-complete one-player games (polynomially unbounded), and showed that this characterization is the same when restricting to planar systems of gadgets. This prior result corresponds to the 2-state special case of our result in the bottom-left cell of Table 1. In this paper, we generalize that characterization to gadgets with arbitrarily many states, and generalize to 2-player games, team games, and (polynomially bounded) DAG k-tunnel gadgets.

1.3 Our Characterizations

In each type of motion planning problem where we obtain a full characterization of easy vs. hard gadget sets (bounded one-player, bounded two-player, bounded team, and unbounded one-player), we identify a class of gadgets such that motion planning with any single gadget in that class is hard, while motion planning with any collection of gadgets not in the class is easy. Thus, we do not see a difference in hardness between one and multiple gadget types; it is not possible for two "easy" gadgets to combine into a hard motion planning problem. This result is in surprising contrast to Constraint Logic where multiple gadgets were required for hardness in any setting.

For one-player motion planning, the key property of a gadget is ***interacting tunnels***: the traversal of some tunnel must affect the traversability of some other tunnel in the same gadget.[4] In the unbounded case (Section 2), we show that any such gadget (that is also reversible and deterministic) can be used to simulate two specific gadgets, the "locking 2-toggle" and "1-toggle" (shown in Figure 1), which together suffice to prove PSPACE-hardness.

[3] This notion is different than the sense of "reversible" in reversible computing, which would mean that we could derive which move to undo from the current state; here the undoing move only needs to be an option.

[4] This is roughly what [6] calls 'non-trivial' gadgets.

This argument involves surprisingly little case analysis, in contrast to the prior work in this area [6], which simply enumerated and analyzed all 2-state gadgets. On the other hand, we show that any fixed collection of gadgets without interacting tunnels reduces (via a shortcutting argument) to graph traversal, which can be solved in NL. We furthermore show that this dichotomy still holds for 1-player planar motion planning (Section 2.3). In the bounded case (Section 5), we examine the naturally bounded class of DAG gadgets. We again obtain a somewhat more complicated full characterization, which mostly depends on the existence of interacting tunnels.

For two-player motion planning, it turns out that interacting tunnels are not required for hardness. In the bounded case (Section 6), we show that PSPACE-completeness holds for any DAG gadget that is ***nontrivial***, i.e., has at least one transition in some state. We show that any nontrivial DAG gadget can simulate one of two one-tunnel gadgets, "single-use unidirectional edge" or "single-use bidirectional edge", and surprisingly either suffices to prove PSPACE-completeness. A single use-use edge is a transition in a gadget such that after taking that transition, there are no further transitions between the two associated locations. Obviously, two-player motion planning with trivial gadgets is in P: the robots can only traverse the connection graph, and one merely needs to see which is closer to their goal. In the unbounded case (Section 3), we show that any gadget with interacting tunnels suffices for EXPTIME-completeness, and it remains an open problem whether some weaker condition suffices.

For team motion planning, interacting tunnels are again not required for hardness. In the bounded case (Section 7), we show that NEXPTIME-completeness holds for any nontrivial DAG gadget, again by showing that any single-use edge gadget suffices. In the unbounded case (Section 4), we again show that any gadget with interacting tunnels suffices for undecidability, and it remains an open problem whether some weaker condition suffices.

Armed with the general framework of this paper, it should be much easier to prove hardness of most games that involve motion planning of robots in an environment with nontrivial local state. You simply need to pick a gadget that is hard according to our characterization (with the matching boundedness and number of players/teams), draw a single figure of how to build that gadget within the game of interest, and check that it is possible to connect these gadgets together. While this paper focuses on general theory building, we return to possible applications in Section 8.

2 1-Player Unbounded Motion Planning

In this section, we study reversible, deterministic gadgets, extending the work in [6] which only considered gadgets with two or fewer states. Here we give a complete categorization as either in NL or PSPACE-complete for reversible, deterministic gadget. For the NL half of the characterization, Theorem 2 below shows that 1-player motion planning problems with non-interacting-k-tunnel gadgets is in NL. For the PSPACE-completeness half of the characterization, we introduce a new base gadget, the ***locking 2-toggle (L2T)*** shown in Figure 1a. In Section 2.1 we show that all interacting-k-tunnel reversible deterministic gadgets are able to simulate the locking 2-toggle. Then in Section 2.2 we show that 1-player motion planning with locking 2-toggles is PSPACE-complete by simulating Nondeterministic Constraint Logic. Section 2.2 shows how to adapt the construction to show these gadgets remain PSPACE-hard even for the planar 1-player motion planning problem.

▶ **Lemma 1.** *1-player motion planning with any set of gadgets is in PSPACE.*

Proof. This was shown in [6], but included here for convenience. A configuration of the system of gadgets consists of the state of each gadget and the location of the robot, and has polynomial length. The algorithm that repeatedly nondeterministically picks a legal transition, and updates the configuration based on it, accepting when the robot reaches the goal location, decides the reachability problem in nondeterministic polynomial space. By Savitch's theorem, the problem is in PSPACE. ◀

▶ **Theorem 2.** *1-player motion planning with any k-tunnel gadget that does not have interacting tunnels is in NL.*

Proof. We first show that if a system of such gadgets has a solution, then it has a solution which visits each location at most once. Suppose there is a solution, and consider the last time a solution of minimal length visits a previously visited location, assuming there is any such time. Let v be the vertex of this last self-intersection. After leaving v for the last time, every transition the robot makes is through a tunnel that it had not previously traversed. Since the gadget does not have interacting tunnels, these tunnels have the same traversability when the robot goes through them as they do originally. We modify the solution by 'shortcutting': remove the portion of the solution between the first visit to v and the last visit to v, so the robot only visits v once, and skips the loop that begins and ends at v. The new path is still a solution: the segment before v is identical to the unmodified solution, and the segment after v consists of tunnels whose traversability is not changed before the robot goes through them. The shortcut path is shorter than the original solution, which was assumed to be minimal. Thus a solution of minimal length has no self-intersections.

We'll want to treat the system of gadgets as though it were a directed graph by replacing each tunnel with an edge in the appropriate direction, or a pair edges if it is traversable in either direction. We can locally walk through all the available transitions in a gadget, assess which locations they lead to, and non-deterministically pick one to try, allowing this to be executed in NL. A path from the start location to the end location in this graph is exactly a solution for the system of gadgets with no self-intersections; the traversability of each tunnel used in such a solution does not change before the tunnel is used.

Since reachability in directed graphs is in NL, the motion planning problem is also in NL. Moreover, if the gadget has any state in which a tunnel can be traversed in one direction but not the other, the motion planning problem is NL-complete, and otherwise it is in L. ◀

2.1 Reducing to Locking 2-Toggles

In this section, we introduce the locking 2-toggle shown in Figure 1a, and we show that all interacting-k-tunnel reversible deterministic gadgets can simulate it. The proof first examines what constraints on a gadget are implied by being interacting-k-tunnel, reversible, and deterministic, and goes on to identify that all such gadgets have a pair of special states with some useful common properties. From this pair of states we construct a 1-toggle, and then combine that with our special states to build a locking 2-toggle. One of the major insights is identifying this special pair of states which belongs to all gadgets in the class, and after that the primary challenge is in preventing undesired transitions, which are plentiful when allowing such a wide class of gadgets.

▶ **Theorem 3.** *Every interacting-k-tunnel reversible deterministic gadget simulates a locking 2-toggle.*

Proof. We begin by examining an arbitrary interacting-k-tunnel reversible deterministic gadget, as shown in Figure 2. Because the gadget has interacting tunnels, we can find a pair

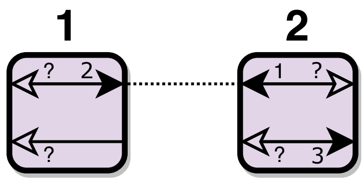
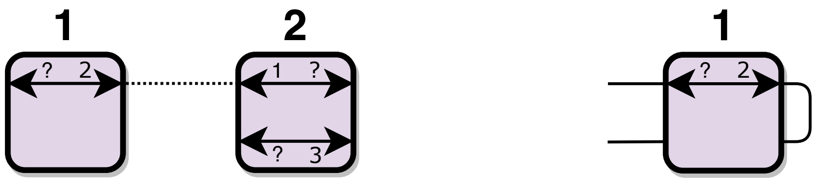


Figure 2 An arbitrary interacting-k-tunnel reversible deterministic gadget. Hollow arrows indicate traversals that may or may not be possible. Solid or absent arrows indicate traversals that are or are not possible, respectively.



(a) State graph, refining Figure 2. **(b)** Simulating a one-directional edge.

Figure 3 An arbitrary interacting-k-tunnel reversible deterministic gadget which has no one-directional edge.

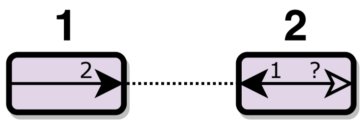
of states in which traversing the top line can change the traversability of the bottom line to the right. Since it is also reversible, the inverse transition is also possible, so traversing the top line can change in either direction the left-to-right traversability of the bottom line. Then without loss of generality, the gadget has the form shown in Figure 2: in state 1, traversing the top line to the right switches to state 2, and the bottom line is not traversable to the right. In state 2, traversing the top line to the left switches to state 1, and the bottom line is traversable to the right, say to state 3 (which may be the same as state 1). All other traversals may or may not be possible in either state, indicated by the question marks.

▶ **Lemma 4.** *Every interacting-k-tunnel reversible deterministic gadget simulates a **one-directional edge**, that is, a tunnel which (in some state) can be traversed in one direction but not the other.*

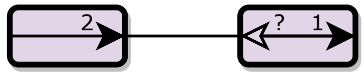
Proof. If in some state, some edge in the gadget can be traversed in one direction but not the other, then it is a one-directional edge. Otherwise, the gadget has the form shown in Figure 3a. Then the construction in Figure 3b is equivalent to a one-directional edge: currently the gadget is in state 1, so the path from the bottom to the top is blocked by the bottom edge, but from the top, you can go across the top edge, switching the gadget to state 2, and then back across the bottom edge. ◀

▶ **Lemma 5.** *Every interacting-k-tunnel reversible deterministic gadget simulates a 1-toggle (Figure 1b).*

Proof. By the previous lemma, we can build a one-directional edge, which has the structure shown in Figure 4a: in state 1, we can traverse the edge to the right and switch to state 2, but not to the left. In state 2, we can undo this transition, and possibly also traverse the edge to the right. The construction in Figure 4b is then a 1-toggle. In the current state, it can be traversed to the right but not to the left because of the gadget on the left. After making this traversal, it becomes the rotation of the current state, and it cannot be traversed to the right again because of the gadget on the right. ◀



(a) Form of state graph.



(b) Simulating a 1-toggle.

Figure 4 A one-directional edge gadget.

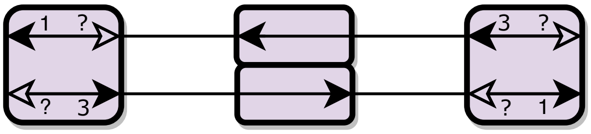


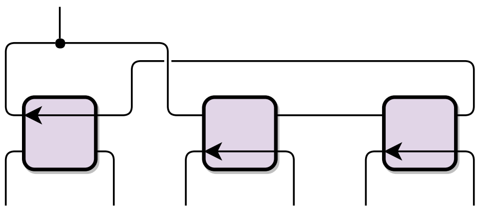
Figure 5 An arbitrary interacting-k-tunnel reversible deterministic gadget and a 1-toggle simulate a locking 2-toggle.

To build a locking 2-toggle, we put the arbitrary gadget (in state 2), an antiparallel pair of 1-toggles, and the rotation of the arbitrary gadget (also in state 2) in series, as in Figure 5. Currently, the top edge is traversable to the left and the bottom edge is traversable to the right, but not in the other direction. After traversing the top edge to the left, the 1-toggles prevents us from traversing either edge to the left, and the leftmost gadget (in state 1) prevents us from traversing the bottom edge to the right, so the only legal traversal is going back across the top edge to the right. Similarly after traversing the bottom edge, the only legal traversal is across the bottom edge in the opposite direction. Thus this construction is equivalent to a (antiparallel) locking 2-toggle.

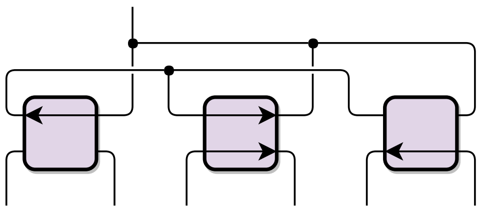
Traversing the simulated locking 2-toggle takes either 4 or 6 transitions of the raw gadget, depending on whether it contains a one-directional edge (from Lemma 4). For simplicity, we can include additional gadgets (e.g. another pair of 1-toggles) to ensure it always takes exactly 6 transitions; this will be relevant to timing considerations in multiplayer games. ◀

2.2 PSPACE-hardness

In this section, we show that 1-player motion planning with the locking 2-toggle is PSPACE-complete by a reduction from Nondeterministic Constraint Logic (NCL). See Appendix A.1 for a definition of NCL. We represent edges by pairs of locking 2-toggles. The construction requires ***edge gadgets*** which are directed and can be flipped, as well as AND and OR

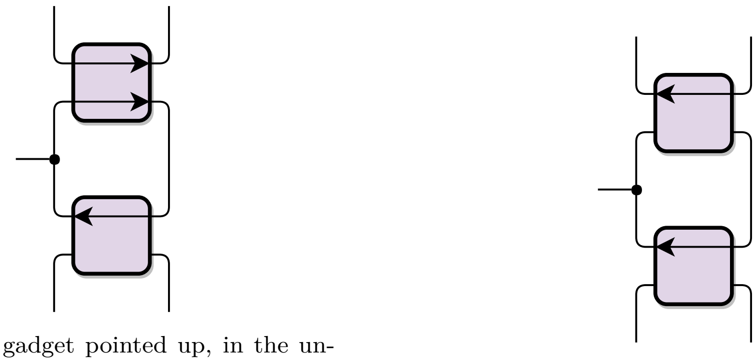


(a) An AND vertex gadget. The leftmost edge has weight two and is pointing in (up). The other edges have weight one and are pointing away (down).



(b) An OR vertex gadget. All edges are weight 2. The leftmost edge is pointing in (up), the middle edge is free, and the rightmost edge is pointing away (down).

Figure 6 Vertex gadgets in the NCL reduction.



(a) An edge gadget pointed up, in the unlocked state. The gadget is accessed by the loose end on the left.

(b) The same edge gadget in the locked state.

Figure 7 Edge gadget in the NCL reduction.

vertex gadgets which apply constraints on how many edges must be directed towards them at any given point in time.

▶ **Theorem 6.** *1-player motion planning with the locking 2-toggle is PSPACE-complete.*

Proof. Motion planning with the gadget is in PSPACE by Lemma 1. We use a reduction from Nondeterministic Constraint Logic (NCL) to show PSPACE-hardness. See Appendix A.1 for a definition of NCL.

The **edge gadget**, shown in Figure 7, contains two locking 2-toggles, each of which is also attached to a vertex gadget. It is oriented towards one of the vertices, can be either ***locked*** or ***unlocked***. Specifically, the edge gadget is unlocked (Figure 7a) if either locking 2-toggle is in the middle state (with both lines traversable), and locked (Figure 7b) otherwise. It is oriented towards the vertex attached to the locking 2-toggle whose edge not accessible from the edge gadget is traversable. The robot can access the free line on the left. If the edge gadget is unlocked, the robot can traverse a loop through one edge of each locking 2-toggle to change the orientation of the edge gadget. The edge gadget switches between being locked and unlocked when the robot moves through a vertex gadget to traverse one of the edges not accessible from the edge gadget.

The **vertex gadgets** are shown in Figure 6. The robot can access the free line on the top, and traverse loops to lock and unlock edge gadgets, enforcing the constraints of vertices. Specifically, if all three edges are pointing towards an AND vertex, the robot can traverse a loop to lock both weight-1 edges and unlock the weight-2 edge, or vice versa. If multiple edges are pointing towards an OR vertex, the robot can traverse a loop to unlock the currently locked edge and lock another edge. Observe that for both vertex gadgets, the sum of the weights of locked edges does not change.

Given an NCL graph, we construct a maze of locking 2-toggles. Each edge in the graph corresponds to an edge gadget (Figure 7). Each locking 2-toggle in the edge gadget corresponds to a vertex incident to the edge. When three edges meet at a vertex, we put a vertex gadget on the locking 2-toggles corresponding to that vertex. We use an AND vertex gadget (Figure 6a) or an OR vertex gadget (Figure 6b) depending on the type of vertex. The vertical ‘entrance’ line on each vertex gadget and horizontal ‘entrance’ line on each edge gadget is connected to the starting location. Each edge is oriented as in the NCL graph. For each vertex, we pick a set of edges initially pointing at the vertex with total weight 2. The edge gadgets corresponding to the chosen edges are locked, and other edge gadgets are unlocked. The goal location is placed inside the edge gadget corresponding to the target edge so that it is reachable if and only if the target edge is unlocked.

If the original NCL graph is solvable, the robot can perform the same sequence of edge flips, visiting vertex gadgets to lock and unlock edges as necessary, and reach the goal location. If the robot can reach the goal location, the same sequence of edge flips solves the NCL graph. So the maze is solvable if and only if the NCL graph was. ◀

This reduction is also possible without edge gadgets, and leads to a system with only one L2T for each constraint logic edge. We use edge gadgets because the reduction is easier to understand, and adaptations of this construction in Sections 2.3, 3, and 4 will need them.

▶ **Corollary 7.** *1-player motion planning with any interacting-k-tunnel reversible deterministic gadget is PSPACE-complete.*

Proof. Hardness follows from Theorems 3, and 6. For any such gadget, we have a reduction from mazes of locking 2-toggles to mazes of that gadget by replacing each locking 2-toggle with a simulation of one built from the arbitrary gadget. Motion planning with the gadget is in PSPACE by Lemma 1. ◀

2.3 Planarity

In this section, we show that interacting-k-tunnel reversible deterministic gadgets are PSPACE-complete even for the planar 1-player motion planning problem. We once again work with the locking 2-toggle, showing that each of its planar versions can simulate each other. From there we use the crossing locking 2-toggle to build an A / BA crossover, which is less powerful than a full crossover but will suffice to make our reduction in Section 2.2 planar. An interesting question is whether the locking 2-toggle is powerful enough to build a full crossover, which can be done with any of the 2 state gadgets. Although not needed here, it would allow the multiplayer game results later in this paper to carry over to the planar case.

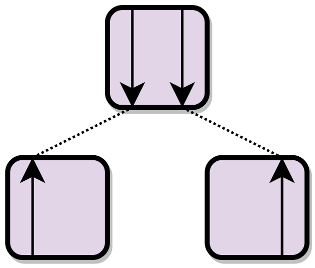
Recall for the planar problem we allow rotations and reflections of gadgets. This leaves three distinct embeddings of the locking 2-toggle into a plane: parallel, antiparallel, and crossing, shown in Figure 8, and which we abbreviate PL2T, APL2T, and CL2T. (Up to only rotation, there are four, the other being the antiparallel locking 2-toggle with the other handedness). We will allow reflections of gadgets, so these are the three kinds of locking 2-toggles we will consider.

▶ **Lemma 8** ([6]). *Parallel, antiparallel, and crossing locking 2-toggles all simulate each other in planar graphs.*

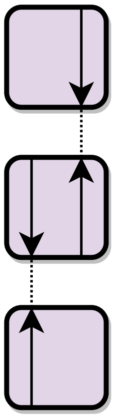
Proof. Figure 9 shows APL2T simulating CL2T, Figure 10 shows CL2T simulating PL2T, and Figure 11 shows PL2T simulating APL2T. Note that we use both APL2Ts of both handednesses, so we need to be able to reflect gadgets. ◀

▶ **Theorem 9.** *Every interacting-k-tunnel reversible deterministic gadget simulates each type of locking 2-toggle in planar graphs.*

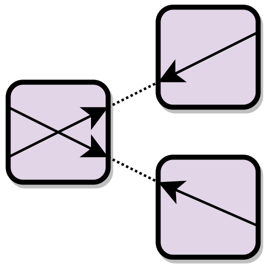
Proof. We follow the proof of Theorem 3. As before, we assume that traversing a line to switch from state 1 to state 2 makes a traversal on another line legal. This new traversal can be parallel to, antiparallel to, or cross the first traversal; we consider each case. If the new traversal is parallel, the construction in the proof of Theorem 3 works to simulate an APL2T in a planar graph.



(a) A parallel locking 2-toggle (PL2T).



(b) An antiparallel locking 2-toggle (APL2T).



(c) A crossing locking 2-toggle (CL2T).

Figure 8 Types of locking 2-toggles in planar mazes.

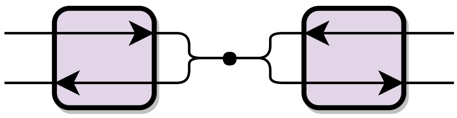


Figure 9 APL2T simulating CL2T. (Based on [6, Figure 4].)

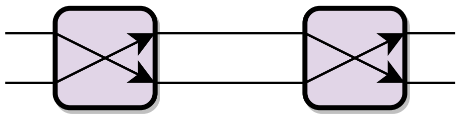


Figure 10 CL2T simulating PL2T. (Based on [6, Figure 5].)

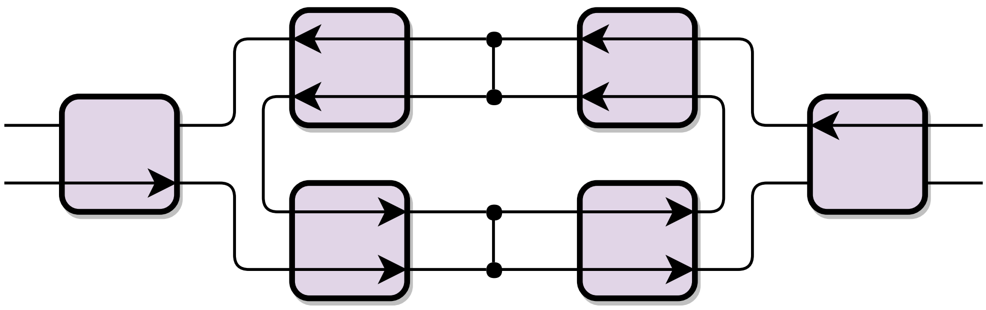


Figure 11 PL2T simulating APL2T. (Based on [6, Figure 13].)

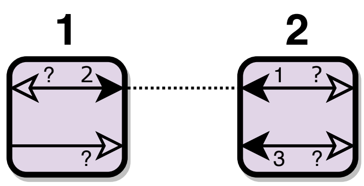


Figure 12 The antiparallel case of an arbitrary interacting-k-tunnel reversible deterministic gadget.

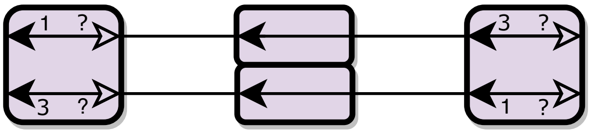


Figure 13 An arbitrary antiparallel interacting-k-tunnel reversible deterministic gadget and a 1-toggle simulate a PL2T.

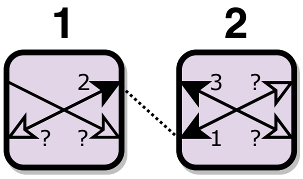


Figure 14 The crossing case of an arbitrary interacting-k-tunnel reversible deterministic gadget.

If it is antiparallel, the gadget has the form shown in Figure 12. Either the gadget has a one-directional edge, or it has the form in Figure 3a, and simulates a one-directional edge by the construction in Figure 3b. Thus it simulates a 1-toggle by the construction in Figure 4b. Then the construction in Figure 13 simulates a PL2T: currently either edge can be traversed to the left, if the top edge is traversed, the left gadget blocks the bottom edge, and if the bottom edge is traversed, the right gadget blocks the top edge.

Finally, if the new traversal crosses the first traversal, the gadget has the form shown in Figure 14. Either it has a one-directional edge, or the construction in Figure 15 simulates a one-directional edge, similarly to Lemma 4. So the gadget simulates a 1-toggle by the construction in Figure 4b. Then the construction in Figure 16 simulates a PL2T, similarly to the previous case.

Once the gadget simulates some locking 2-toggle, we can use Lemma 8 to simulate all three types. ◀

▶ **Theorem 10.** *1-player planar motion planning with any interacting-k-tunnel reversible deterministic gadget is PSPACE-complete.*

Proof. We begin by constructing some weak crossover gadgets. The crossover locking 2-toggle is itself a very weak crossover. We use it to construct an ***A/BA crossover***, shown in Figure 17a. Calling the traversal from top to bottom A and that from left to right B, we can perform either of the sequences A and BA. Since everything is reversible and deterministic, we can also undo those sequences. The A/BA crossover is sufficient for the rest of the proof; we abbreviate it as shown in Figure 17b.

We modify the proof of Theorem 6, giving a reduction from planar NCL to planar mazes with locking 2-toggles. By Theorem 9, this is sufficient to show PSPACE-hardness. Our gadgets use PL2Ts, CL2Ts, and A/BA crossovers; they do not use APL2Ts.

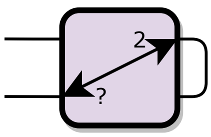


Figure 15 A crossing interacting-k-tunnel reversible deterministic gadget simulates a one-way edge.

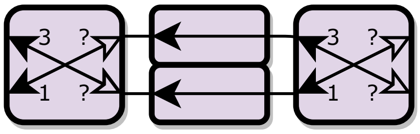


Figure 16 An arbitrary crossing interacting-k-tunnel reversible deterministic gadget and a one-toggle simulate a PL2T.

The edge gadget is shown in Figure 18, and vertex gadgets are shown in Figure 19. Given a planar NCL graph, we construct a mazes as follows.

Pick a rooted spanning tree of the dual of the NCL graph, directed away from the root; the robot will use this tree to navigate the graph. The system of gadgets will contain a vertex for each face f of the NCL graph, which is a vertex of the spanning tree.

For each edge of the graph, we place an edge gadget. When an edge is in the spanning tree, we orient it so that the A/BA crossover points, from entrance to exit, in the same direction as the edge points in the spanning tree (left to right in Figure 18, and away from the root). If an edge is in the spanning tree and has target f, we connect its exit to f. For each edge e, we connect its entrance to the vertex f corresponding to the face containing its entrance, i.e. the face adjacent to e to which we can connect its entrance without crossings. If e is in the spanning tree, this connects the entrance of e to the source f of e.

Now we place a vertex gadget of the appropriate type for each vertex of the NCL graph, so that the gadget shares a PL2T with each incident edge gadget. AND vertex gadgets must be oriented so the weight-2 edge has the appropriate PL2T (the bottom one in Figure 19a). The entrance of each vertex gadget is connected to the vertex f corresponding to the face containing the entrance.

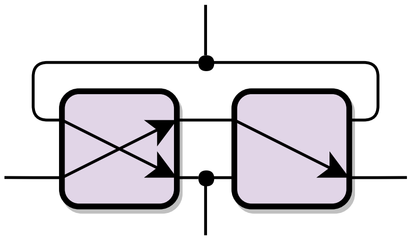
We set each edge gadget to the orientation of its corresponding edge. For each vertex, we select edges directed towards it with total weight 2, and set the selected edges to locked and other edges to unlocked. The goal location is placed inside the target edge so that reaching it requires flipping the target edge. The starting location is the vertex corresponding to the root of the spanning tree.

Play on this maze proceeds as follows: the robot travels down the spanning tree, crossing edges until it reaches some face. It goes into an edge or vertex attached to that face, and manipulates it. Then the robot travels back up the spanning tree and down a different branch, manipulating another edge or vertex, and so on. The edge and vertex gadgets enforce the NCL constraints. If the target edge can be flipped, the robot can reach the goal location. Thus the maze is solvable if and only if the NCL graph was. The maze is planar by its construction, using the planarity of the NCL graph.

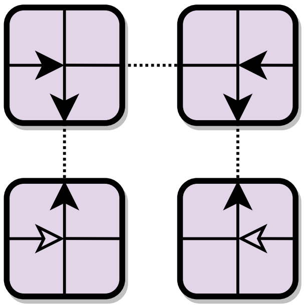
This completes the proof of PSPACE-hardness. Containment in PSPACE is by Lemma 1, so the problem is PSPACE-complete. ◀

3 2-Player Unbounded Motion Planning

In this section, we analyze 2-player motion planning games with k-tunnel reversible deterministic gadgets. We show that any such game which includes an interacting-tunnels gadget is EXPTIME-complete. We do so by a reduction from 2-player unbounded constraint logic, allowing us to reuse some of the work in the prior section. In addition to building the single player AND and OR vertices, we show how to adapt the gadgets to allow different players to



(a) Simulating an A/BA crossover using CL2Ts.



(b) A state diagram and notation for the A/BA crossover.

Figure 17 An A/BA crossover gadget: the robot can traverse top to bottom (A), or traverse left to right (B) and then top to bottom. Thinking of the gadget as a crossing pair of 1-toggles, the vertical 1-toggle is always traversable, and the horizontal 1-toggle is traversable when the vertical one is pointing down.

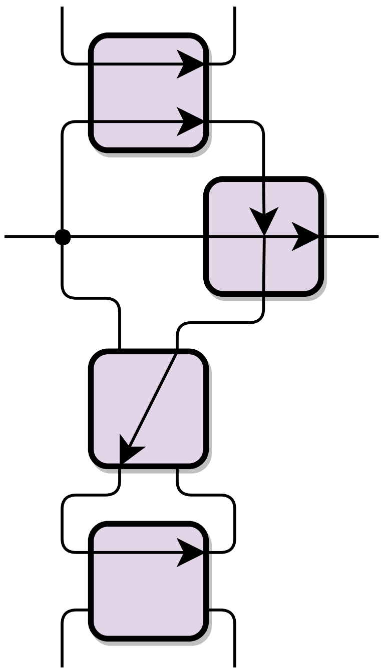
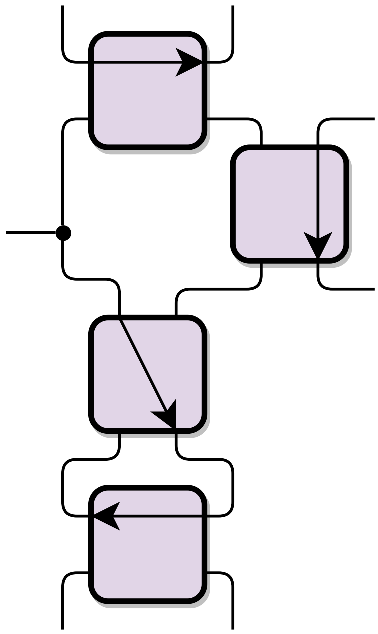
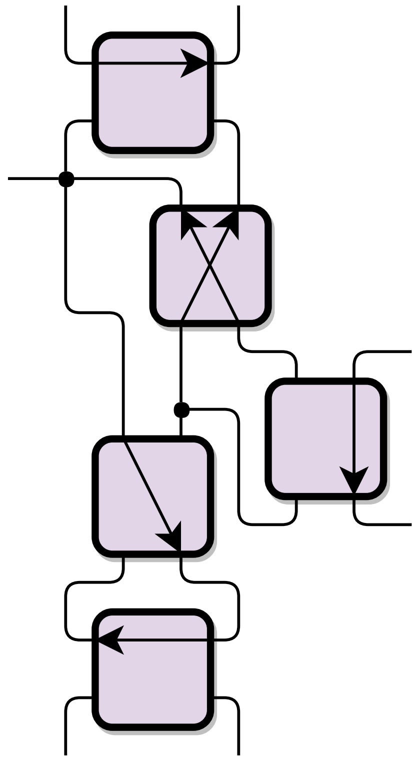


Figure 18 An edge gadget for planar graphs, currently unlocked and directed up. This is analogous to Figure 7, with two changes. First, the bottom PL2T is 'twisted' to have the same handedness as the top PL2T for connecting to vertex gadgets; the CL2T is sufficient for the crossing caused by this. Second, the A/BA crossover allows the robot to cross the edge from left to right, regardless of the state of the edge. We call the line on the left the ***entrance*** and the line on the right, on the other side of the A/BA crossover, the ***exit***.



(a) An AND vertex for planar graphs. Currently the weight-2 edge, connected at the bottom PL2T, is directed towards the vertex and locked, and both weight-1 edges are directed away. If the weight-1 edges become directed towards the vertex, the robot can visit the vertex gadget and traverse a loop through all three PL2Ts, locking the weight-1 edges and unlocking the weight-2 edge. The CL2T is a sufficient crossover.



(b) An OR vertex for planar graphs. Currently the edge containing the bottom PL2T is directed towards the vertex, and the other edges are directed away. If multiple edges are ever directed towards the vertex, the robot can visit the vertex gadget, unlock the locked edge, and lock another edge.

Figure 19 NCL vertex gadgets for planar graphs, analogous to the gadgets in Figure 6. In each gadget, each of the three PL2Ts is also part of an edge gadget. The robot enters at the line on the left, called the ***entrance***, traverses loops that enforce the NCL constraints, and then leaves at the entrance.

have control of different edges. We also build up the needed infrastructure to enforce turn taking in the simulated game.

The construction of crossovers using interacting-k-tunnel reversible deterministic gadgets with two states should allow one to show hardness for the planar version of this problem with those gadgets and any others that simulate them. Care must be taken with the layout, timing, and interaction between crossovers so we do not go on to prove such a result in this paper. Unfortunately, the crossover created by the locking 2-toggle in Section 2.3 does not suffice and thus leaves the question partially open. In addition, the question of noninteracting-k-tunnels reversible deterministic gadgets has not been resolved. We are not able to show problems with such gadgets are easy, and Section 6 suggests they should be at least PSPACE-hard.

▶ **Lemma 11.** *2-player motion planning with any set of gadgets is in EXPTIME.*

Proof. A configuration of the maze consists of the state of each gadget and the location of the robot, and has polynomial length. There is a polynomial-space alternating Turing

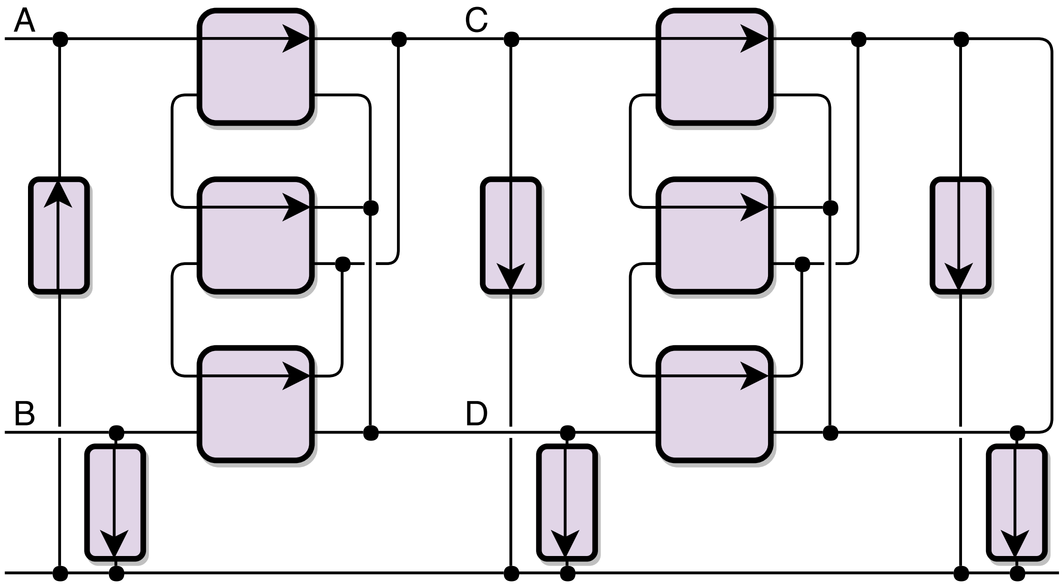


Figure 20 The timer gadget used in the 2CL reduction, made of PL2Ts and 1-toggles. In order to travel between A and B, a player must travel between C and D three times. The timer can be extended to the right; two iterations are shown.

machine which nondeterministically guesses moves for each player and keeps track of the configuration, using existential quantifiers for player 1 and universal quantifiers for player 2. This Turing machine accepts exactly when player 1 has a forced win. Thus the problem is in APSPACE = EXPTIME. ◀

▶ **Theorem 12.** *2-player motion planning with the locking 2-toggle gadget is EXPTIME-complete.*

Proof. This game is in EXPTIME by Lemma 11. We use a reduction from 2-player Constraint Logic (2CL) to show EXPTIME-completeness. See Appendix A.1 for a definition of 2CL.

We begin by describing a timer gadget, shown in Figure 20. Suppose one player has access to the bottom line. They can enter the gadget at A, and begin going through the timer, eventually reaching a victory gadget at B. The timer has two key properties:

1. Reaching B takes a number of transitions exponential in the size of the timer. In order to get from A to B, the player goes though the top PL2T to C, recursively travels from C to D, goes around the loop through the top two PL2Ts, goes back from D to C, traverses the bottom loop, once again goes from C to D, and finally proceeds to B. If traveling between C and D takes m transitions, then traveling between A and B takes $3m + 6$ transitions. If the timer gadget is repeated k times, it takes at least 3^k transition to get from A to B.
2. A player in the timer has an opportunity to exit the timer at least every 2 turns, and exiting takes 1 turn; in particular, they can always exit within 3 turns while progressing the timer. The player uses a 1-toggle to exit to the bottom line. They can then later reenter using the same 1-toggle, resuming their work on the timer where they left off. If the player is in the timer, the next step in progressing the timer is either traversing a loop between to PL2Ts, which takes 2 transitions, or moving horizontally between timer segments, which takes 1 transition. Thus in 3 transition, the player can complete the current or next step and exit to the bottom line.

The constraint logic gadgets are similar to those used in Theorem 6 for the 1-player game, with the modification shown in Figure 21. We have added 1-toggles allowing a player at an edge to visit and configure the incident vertices, without allowing the player to travel to other edges. Each player's goal location is inside the gadget corresponding to their target edge, so that they can reach it if they can flip the edge.

Unlike the 1-player version, we need gadgets to enforce the turn order. The overall construction is shown in Figure 22. The maze consists of three main regions: the White area, the Black area, and the constraint logic. Each player will spend most of their time in their own area, occasionally entering the constraint logic to flip an edge. The players' areas are designed to enforce turn order and progression of the game. A player can never enter the other player's area.

There is a single L2T separating the constraint logic area from each player's area. This prevents both players from being in the constraint logic at the same time.

Each player's area contains an edge selection gadget, which consist of a locking 2-toggle for each edge they can control. The other line in the L2T is accessible by entering the constraint logic area and passing through a delay line four 1-toggles, and is connected to the corresponding edge gadget. In order to access an edge gadget, the player must activate the appropriate L2T, which requires deactivating the previously activated L2T. This ensures that only one edge gadget is accessible by each player at any time. There is a 1-toggle separating the edge selection gadget from the rest of the player's area, so that switching the selected edge requires at least 4 turns (we use one tunnel of a L2T for a 1-toggle).

Each player's area has a timer, of length t_w for White and t_b for black. If a player finishes their timer, they win.

Each player begins inside their edge selection gadget, and White goes first. The game begins with White picking an edge and going to the constraint logic area, while Black goes to their timer.

A round of normal play proceeds as follows:

- White moves from edge selection to the constraint logic area. Black is currently in their timer.
- White enters the constraint logic, walks to their selected edge, and flips it. Black continues working on their timer.
- White returns through their constraint logic delay line. Once they pass the first 1-toggle, Black finishes their current step in the timer and exits, moving towards edge selection.
- White begins working on their timer. Black selects an edge, enters the constraint logic, and flips the edge.
- Black returns through their constraint logic delay line. Once they pass the first 1-toggle, White exits their timer and moves to edge selection.
- White selects an edge as Black enters their timer.

Suppose Black has just flipped an edge gadget; they have nothing to do but return through the delay line of length 4. When Black is past the first 1-toggle, White will leave their timer to flip an edge. Black might try turning around to go back to the constraint logic area. It takes Black at least 6 turns to flip the edge back, during which White has enough time to select an edge and reenter their timer. The game is now in the same situation as before, except that White has progressed their timer; thus Black does not want to do this.

Black might instead try waiting at the central L2T after White has selected an edge. White will then go to their timer, forcing Black to exit eventually. When Black is not next to the central L2T, White exits their timer and moves to constraint logic. Because of the 1-toggle separating edge selection from the central L2T, for Black to change their selected

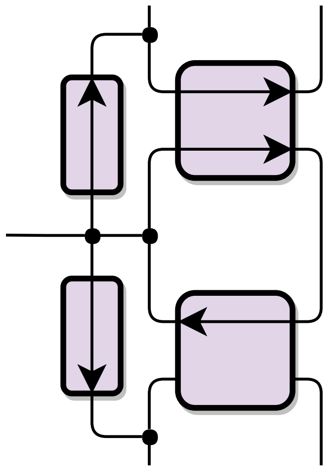


Figure 21 A modified edge gadget for the 2CL reduction. A player can visit the vertex gadgets attached to the edge gadget, and then return to the edge gadget.

edge, they must spend multiple turns away from the L2T, allowing White to enter constraint logic; similarly if Black works on their timer, White can enter constraint logic. So Black has no choice but to pass the turn to White.

Since White can always exit their timer within 3 turns, and Black has three more 1-toggles to get through when White begins looking to exit, White will reach edge selection before Black can reach edge selection, so White will be the first player ready to enter constraint logic again. Nothing Black can do will prevent White from taking the next turn in the 2CL game. Similarly after White flips an edge, Black will be able to take a turn next. So either player can force the alternation of constraint logic turns.

The sizes of the timers are chosen to satisfy the following. First, if White cannot win the constraint logic, Black should win, so Black's timer is shorter: $t_b < t_w$. Second, if White can win the constraint logic game, White should win first, even if Black ignores the constraint logic game and just works on their timer. If the constraint logic graph has n edges, it takes at most 2^n constraint logic turns for White to win. Each constraint logic turn for White takes 6 turns to select an edge and return to the constraint logic, 8 turns to cross the constraint logic delay line twice, 4 turns to access and flip an edge, and up to 5 turns to access and configure an incident vertex, so 25 turns in total during which Black can work on their timer. Both players might be in their timers simultaneously at most 4 times each cycle, and each time for at most 4 turns, so Black spends at most 41 turns in their timer for each constraint logic turn. Thus, since it takes Black at least 3^{t_b} turns to win through the timer, we need $41 \cdot 2^n < 3^{t_b}$; $t_b = n + 6$ suffices, and we can set $t_w = 2n + 12$.

Using these timer sizes, it is clear that the constraint logic game will resolve before either timer if the players follow normal play. We need the timers so that Black cannot force a draw by sitting in the constraint logic forever, preventing White from winning; White will progress on their timer if Black attempts this.

Hence White has a forced win in the motion planning game if and only if they have a forced win in the constraint logic game. Since 2CL is EXPTIME-complete, the 2-player game on systems of locking 2-toggles is EXPTIME-hard. The maze used in the reduction has only $O(n)$ L2Ts. ◀

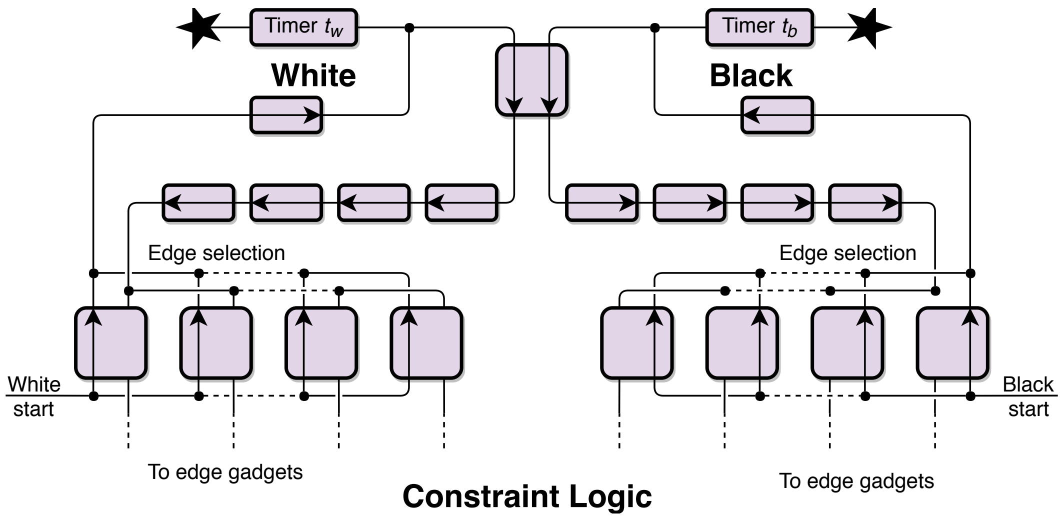


Figure 22 The overall structure and turn enforcement gadget. Each player's edge selection area has a L2T for each edge that player can flip; four are shown for each player. The bottom line from each such L2T connects to the corresponding edge gadget. The timers are as shown in Figure 20, with t_w and t_b repetitions. The inside connection to each timer is connected to its access line, and the outside connection (to a win gadget) is at B in Figure 20. The goal location past each timer is for the player whose side it is on.

▶ **Theorem 13.** *2-player motion planning with any interacting-k-tunnel reversible deterministic gadget is EXPTIME-complete.*

Proof. This game is in EXPTIME by Lemma 11. We adapt the 2CL reduction in the proof of Theorem 12. Replace each locking 2-toggle in that 2CL reduction with the simulation of a locking 2-toggle from the arbitrary gadget in Theorem 3. In the new maze, each tunnel in a simulated L2T takes 6 transitions to traverse, so the game goes 6 times slower.

The simulation still works with two players, as long as both players do not have access to the gadget at the same time. Each L2T in the turn enforcement area is accessible only by one player, and only one player can be in the constraint logic area at any time. The only L2T both players have simultaneous access to is the central gadget which gives access to the constraint logic area, so we look more carefully at that gadget.

The state with both edges traversable is shown in Figure 5 (the 1-toggle simulation still works). Note that the simulation is of an APL2T, but the gadget in the 2CL reduction is a PL2T; this is not a problem because we are not concerned with planarity. Suppose both players approach the gadget, one from the right on the top line and one from the left on the bottom line. Whoever reaches the gadget first should 'win the race,' and lock out the other player. The simulation implements this correctly, provided that the player who arrives first is a full turn ahead in the L2T maze, or 6 turns ahead in the new maze. The only time the players might be within 6 turns of each other is at the very beginning of the game, so we put a delay of 6 turns for Black to get from their start location to edge selection to ensure White wins the race by 6 turns. If a player would arrive less than 6 turn before the other player, they should go to their timer instead; since this is a zero-sum game and the players would have to collaborate to break the simulation, one player will choose not to.

The other way players can interact at this gadget is when one player is exiting the constraint logic area, and the other player is waiting just outside and enters as soon as they can. The state of the simulation is shown in Figure 23 (the other possible state is symmetric).

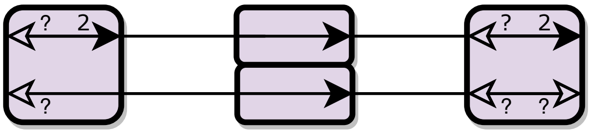


Figure 23 Another state of the construction shown in Figure 5. The leftmost gadget is in state 1, and the rightmost gadget is in state 3.

One player, say White, has traversed the top edge to enter the constraint logic area, and is about to exit by traversing the top line to the right. Black is waiting at the left end of the bottom line, ready to enter the constraint logic area. The leftmost gadget prevents Black from making any transitions until White begins exiting. Once White begins exiting, the leftmost gadget switches to state 2, so Black can follow parallel to White and one turn behind. As long as White continues through the construction at full speed, Black interacts with the construction as though White has already finished their traversal, so it correctly simulates a L2T. Again breaking the simulation would require the players to cooperate, and the game is zero-sum, so at least one player will ensure the simulation works. ◀

4 Team Unbounded

In this section, we show that team imperfect information games with interacting-k-tunnel reversible deterministic gadgets is RE-complete, implying the problem is Undecidable. The reduction is from Team Private Constraint Logic (TPCL); see Appendix A.1 for a definition. We use many of the ideas and constructions from Section 3, but various modifications are needed to deal with the additional player and the model of player knowledge. Recall in this model we have three players on two different teams, each controlling a single robot. All players start knowing the configuration of the entire game; however, after that point players can only observe the states of the gadgets that their robots can reach via the connection graph. Adaptations for the planar version and the complexity of such games with noninteracting-tunnel gadgets remains open as in Section 3.

▶ **Lemma 14.** *Team motion planning with any set of gadgets is in RE (recursively enumerable).*

Proof. Suppose the White team has a forced win on some system of gadgets, and consider the tree of possible positions when White follows their winning strategy. The branches in the tree correspond to choices the Black team might make. Since White forces a win, every branch of the tree is finite. Since Black has finitely many choices at each turn, the tree is finitely branching, so by Kőnig's infinity lemma [14], the tree is finite. In particular, there is a finite bound on the number of turns it takes for White to win, so the winning strategy can be described in a finite amount of space. So there are countably many potential winning strategies, and we can sort them lexicographically.

Given a potential winning strategy, the problem of determining whether it is actually a winning strategy is decidable: an algorithm can explore every choice Black might make, and see whether White always wins. There are only finitely many choices to check because the strategy only describes a finite number of turns.

We use the following algorithm to determine whether White has a forced win. For each potential winning strategy in lexicographic order, check whether it is a winning strategy.

If it is, accept. This algorithm accepts whenever White has a forced win, and runs forever otherwise, so it recognizes the games in which White has a forced win. ◀

Although [7] only mentions undecidability and not RE-completeness, it follows that TPCL is RE-complete. Containment in RE is given by an argument nearly identical to the proof of Lemma 14. The proof of undecidability is ultimately by a reduction from acceptance of a Turing machine on an empty input, which is RE-complete, implying that TPCL is RE-hard.

▶ **Theorem 15.** *Team motion planning with the locking 2-toggle gadget is RE-complete (and thus undecidable).*

Proof. Containment in RE is given by Lemma 14. For RE-hardness, we use a reduction from TPCL, with a similar construction as in the proof of Theorem 12. The overall construction is shown in Figure 24. Capital letters label L2Ts, and lowercase letters label lengths of delay lines. The two tunnels in the same L2T are labelled the same, instead of being positioned next to each other. The three players B, W_1, and W_2 each have their own region. Each region contains an edge selection area with k edges initially active, access to the constraint logic, and some additional gadgets. We need to ensure the following:

1. Turn order is enforced. That is, the players take turns in the order B, W_1, W_2, and neither team can gain anything by deviating from this. We use L_1 and L_2 to prevent B from being in the constraint logic area at the same time as W_1 or W_2, and appropriate delays to ensure each player is ready for their turn. The timer in W_2's region forces B to eventually pass the turn to W_1.
2. Each player can flip up to k edges each turn. If k edges are initially accessible for each player, the edge selection area allows them to select any k of their edges, and a player must end their turn in order to change their selection.
3. The White players have the correct information about the state of the game. Each of them has a visibility area, which allows them to see the orientation of the appropriate constraint logic edges. We must not allow W_1 and W_2 to both access the same L2T, as they could then use it to communicate. So we need a more complicated mechanism to prevent both White players from being to the constraint logic area at the same time.

For visibility, we modify the edge gadget as shown in Figure 25. The appropriate line is connected to each White player's visibility area if they should be able to see that edge.

A round of normal play proceeds as follows:

- B begins their turn by passing down through L_1 and L_2. W_1 waits next to V, and W_2 walks through their timer.
- B flips some edges, and returns, passing V. When W_1 sees this happen, they go to their visibility area, and then select k edges. W_2 continues in the timer.
- B finishes exiting through the delay b. Once B has passed L_1, W_1 enters the constraint logic area. W_2 reaches the end of the timer, finds S to be closed, and comes back.
- B is stuck on the side of L_1 away from the constraint logic area, and can select edges. W_1 flips edges and returns to just below L_1. W_2 goes to their visibility area, and then selects edges.
- After a number of turns large enough that both White players are definitely ready, W_1 exits L_1. The same round, W_2 enters L_2, passing the turns from W_1 to W_2.
- W_2 takes their turn. B waits just to the right of L_2, and W_1 waits above X.
- W_2 exits L_2 and goes to the timer. B passes through L_2 to take their turn, and W_1 waits.

We place each player's starting location to be at the end of a chain of 1-toggles leading to their region, so they arrive after an appropriate delay. We can set B to have no delay and

W_1 and W_2 to have $2k$ delay, so B has time to select edges before the White players arrive. The first turn has slightly strange timing since W_2 starts the timer later than normal, but this is not important.

We consider ways in which player might deviate from normal play, and see that in each case they do not gain anything by deviating.

B enters the constraint logic through L_2 as soon as W_2 passes L_2 on their way out, at which point W_2 enters the timer. B need to be able to take a full turn and go back through S before W_2 reaches the end of the timer; this takes up to $2(b + c + 2) + 2 + 11k$ turns, since flipping each edge now takes up to 11 turns. So we need $t > 2(b + c + 2) + 2 + 11k$. The timer forces B to return through S within $t + 2$ turns, since otherwise W_2 wins.

The gadget V lets W_1 know when B is done, since W_1 can see whether B is past V while waiting at L_1. Specifically, W_1 waits until they see B stay past V for $2c$ turns, and then return. For B to be unable to flip edges after this, we need $4c > t$. Then W_1 goes to visibility and sees the current configuration, selects k edges for their next turn, and waits at L_1 again. For W_1 to have time to do this before B gets out, we need $b > 2k + 2$.

Once B exits L_1, W_1 goes in and flips edges. The delay d ensures that if W_1 (or W_2) flips any edges, then B will be ready for their next turn; we need $2d > 2k + 4$. W_2 returns through the timer, checks visibility, and selects edges. If W_2 enters constraint logic before W_1 leaves, B can win through X and Y, so W_2 must wait until W_1 leaves. The White players coordinate using the fact that the length of an entire round is bounded, so they can wait long enough to ensure that they are both ready, and then W_1 exits X immediately before W_2 enters Y. Since W_1 was past L_1, B is locked outside of L_1, so W_2 can get past L_2; the W_1 can safely pass the turn to W_2.

While W_1 is past X, B might try going through Z and X, trapping W_1. In this case, W_2 can win through Z, so B will only go through Z if both X and Y are traversable.

During W_2's turn in the constraint logic, W_1 must not be past X to prevent B from winning through X and Y. So B can go through L_1, and go through L_2 as soon as W_2 exits. That is, W_2 cannot pass the turn back to W_1.

W_2 might try to stay in the timer, forcing B to stay out of the constraint logic to prevent W_2 from winning through S. Then W_1 might be able to take extra turns in the constraint logic. If the White team attempts this, B will win through P and Q. If B goes through R and P when Q is not traversable in order to trap W_1, W_2 will win through R; these three L2Ts are analogous to X, Y, and Z.

Assuming the constraints mentioned are satisfied, no player or team can usefully deviate from normal play, and normal play simulates the TPCL game. Thus White has a forced win in the team motion planning game if and only if they have a forced win in the TPCL game.

We can satisfy all the constraints, e.g by $b = 2k+3$, $c = 8k+7$, $d = k+3$, and $t = 31k+27$ (the constraints are not tight, but they suffice). The number of L2Ts in the resulting system of gadgets is only linear in the number of edges in the constraint logic graph. ◀

▶ **Theorem 16.** *Team motion planning with any interacting-k-tunnel reversible deterministic gadget is RE-complete.*

Proof. Containment in RE is given by Lemma 14. For RE-hardness, we adapt the TPCL reduction in Theorem 15 to work for the arbitrary gadget. As in the 2-player case of Theorem 3, it is almost sufficient to replace each L2T with the simulation in Theorem 3. We examine the L2Ts that are shared between two players.

First, L_1 and L_2 are analogous to the central L2T in Theorem 3: if two player are racing to enter, the player who should win is at least 6 turns ahead, and if one player exits and

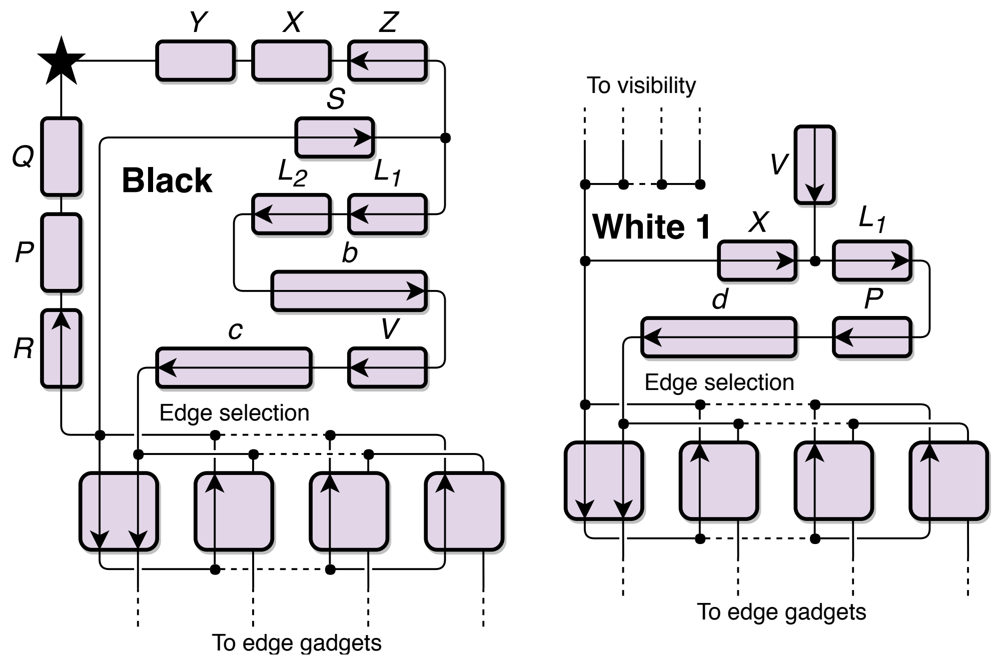


Figure 24 The turn enforcement gadget for the team game. Each player has their own region which contains an edge selection area, a path to the edge gadgets they can control, and some other constructions. Each White player has a visibility area which allows them to see the state of some edge gadgets in constant time. There is no good layout for the whole gadget, so we use pairs of 1-toggles that share a (capital) label to represent L2T. Long boxes with lowercase labels represent chains of 1-toggles with length given by the label. The win gadgets are for the obvious players, and the tunnels currently not traversable (P, Q, R, S, X, Y, and Z) will directed toward the win gadget when they become traversable.

another enters, is works correctly.

For S, P, Q, R, X, Y, and Z, we use a single copy of the arbitrary gadget with 5 extra gadgets for delay, instead of the simulation. Considering the gadget as in Figure 2, we use state 1, and put the bottom edge in the position next to a win gadget. For S, Q, Y, R, and Z, if the bottom edge is traversed from state 2, the game is over, so the gadget is never in a state other than 1 or 2 while the game is going. For P and X, we know that B cannot safely wait past those gadgets, so the game must be about to end in Black victory if they ever reach state 3.

For V and the visibility gadgets on edges, we use the construction in Figure 26. B has three paths to choose from in the process of crossing the bottommost 1-toggle, and always two of them are align with that 1-toggle, so B has two options. The White player, say W_1 can see the state of a gadget in all three paths, and thus determine the orientation. If W_1 goes through one of these gadgets, B will use the other path. If there were only one path, W_1 could go through the gadget, forcing B to either not flip that edge or get a gadget into an unknown state (for L2Ts, we used the fact that W_1 could never traverse that tunnel in one direction). This visibility gadget allows W_1 to see the orientation of a constraint logic edge or V without being able to interfere.

Once we make these replacements, the new maze with the arbitrary gadget has a forced win by White if and only if the maze with L2Ts did. ◀

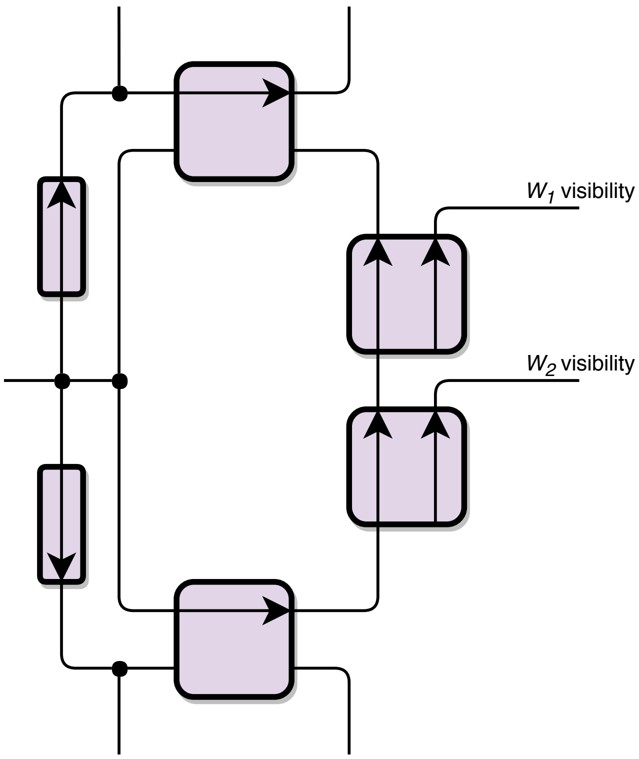


Figure 25 An edge gadget for the TPCL reduction. This is the same as a 2CL edge gadget, except two L2Ts have been added that allow W_1 or W_2 to see the state of the edge if it is connected to their visibility area, but they cannot make any transitions.

5 1-Player Bounded Motion Planning

In this section, we consider a broad class of gadgets which are naturally in NP and give a dichotomy classifying them as NP-complete or in NL. We examine all gadgets in tunnels whose state-transition graph forms a DAG. We will call these DAG gadgets for short. Our proof of hardness further applies to a larger class of gadgets, however a full classification of more general, simple to describe classes of gadgets will require more insight or much more case-work. Also, our constructions require the use of a crossover gadget.

The results in this section can be seen as similar to Viglietta's Metatheorem 1 about location traversal (being implemented by the interacting tunnels in gadgets) and single-use paths [19]. It also bears resemblance to Metatheorem 4 about pressure plates which only affect one door [19]. However, our proof goes through 3SAT rather than Hamiltonian Path, uses a different underlying model which makes different features salient, and gives generalizations in a different direction. Structurally the proof follows that used to show Mario as well as many other games are NP-hard [1].

▶ **Lemma 17.** *All DAG gadgets contain a single-use transition unless they are a transitionless gadget.*

Proof. First, find a node which only has transitions to ***terminal states***, ones with no possible further transitions. To find one, begin by removing all terminal states from the graph. Of the remaining nodes, all of them which are now terminal states must have pointed to at least one terminal state in the original graph or it would have been removed, and it

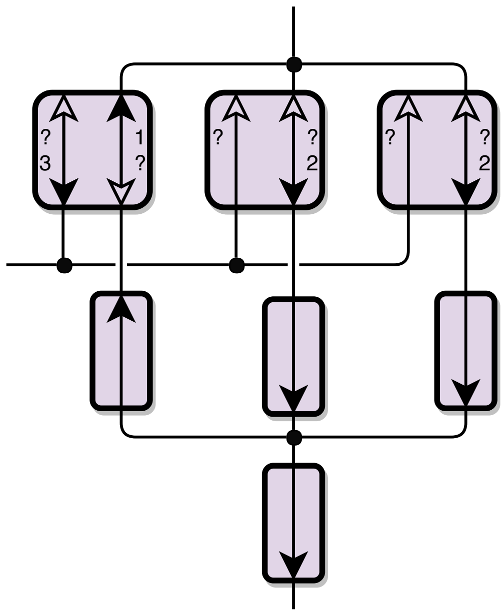


Figure 26 A visibility gadget for the TPCL reduction. The Black player can travel between the top and bottom, and a White player can enter the side to see which direction was traversed most recently.

must have only pointed to terminal states or it would not be a terminal node. Terminal nodes have no transitions. Thus the node we discovered has an available transition which closes all tunnels in the new state. The gadget starting from that state is a single-use gadget. ◀

In a system of gadgets, each DAG gadget can only be traversed polynomially many times. This is the core reason that motion planning involving these gadgets is always in NP.

▶ **Lemma 18.** *1-player motion planning with any set of DAG gadgets is in NP.*

Proof. If a gadget and its state is a sink in the state-transition graph, then no transitions are available from that state. Each time a gadget is traversed the state of the gadget is moved down the graph. All paths from any vertex to a leaf are of polynomial length and thus each gadget can only be traversed polynomially many times before it no longer has any open tunnels. Thus we can give a polynomial size witness consisting of the order in which gadgets are visited, as well as the transition made at each gadget. To verify this certificate we check that each specified transition is legal, and that the location after each transition is connected to the location before the next transition in the witness. ◀

Recall from Theorem 2 that all gadgets without interacting tunnels are in NL. Thus one might hope to show that all interacting-k-tunnel DAG gadgets are NP-complete. This is true for deterministic gadgets but false in general; nondeterministic gadgets require a more careful categorization. We will define two behaviors a DAG gadget might have, 'distant opening' and 'forced distant closing,' and show that either behavior guarantees NP-hardness, while having neither one puts the gadget in NL.

A ***distant opening*** in a DAG gadget is a transition in some state across a tunnel which opens a different tunnel.

▶ **Lemma 19.** *1-player motion planning with any k-tunnel DAG gadget with a distant opening is NP-hard.*

Proof. We show this problem is hard by a standard reduction from 3SAT. See Appendix A.2 for a definition of 3SAT.

We construct our reduction as follows. We use the tunnel which is traversed in the distant opening and one of the tunnels it opens. Each literal in a 3-CNF formula will be represented by those two tunnels in a single gadget, in the state of the distance opening. Each variable x_i is represented by a connection to two different paths, one which goes through the opening transitions for the x_i literals, and one for the $\neg x_i$ literals. We place a single-use gadget at the start and end of each branch of each variable to ensure only one side of the variable is traversed. The single-use gadget prevents the agent from returning on the same branch, and if the agent returns via the other branch, they will not be able to proceed to the next variable.

Each clause contains connections between the openable tunnels for each of its literals. All variable gadgets are laid out in series followed by the clause gadgets, with the goal location at the end of the clause gadgets. Each clause gadget can only be traversed if at least one of its corresponding variable gadgets has been traversed, allowing at least one passage to be open. The agent can reach the goal location exactly when it has a path through the variable gadgets which makes each clause gadget traversable, which corresponds to a satisfying assignment of the 3-CNF formula. ◀

When a transition across a tunnel closes another tunnel, the situation is more complicated, since the agent may be able to cross the same tunnel through a different transition, choosing not to close the other tunnel. For distant openings, the agent always chooses to open the other tunnel. We will now consider only ***monotonically closing*** DAG gadgets, which are DAG gadgets with no distant openings. We clarify some terminology regarding k-tunnel DAG gadgets. A ***transition*** is an edge in the transition graph, which is a legal move between locations which changes the state of the gadget. A ***traversal*** in a state is an orientation of a tunnel which is open in that state. A traversal may correspond to multiple transitions; a gadget being deterministic is equivalent to each traversal having only one transition. An ***orientation*** of a set of tunnels in a state contains, for each tunnel in the set, a single traversal of the tunnel the state.

For NP-completeness one might suggest there exist a traversal such that all of its transitions close some other traversal. However, this fails in a simple two tunnel case where one transition closes one direction of the other tunnel and the other transition closes the other direction. This leads us to a more complex definition. A ***forced distant closing*** in a state of a DAG gadget is a traversal across a tunnel in that state and an orientation of some other tunnels in the state such that, for each transition corresponding to the traversal, the transition closes some traversal in the orientation. The ***size*** of a forced distant closing is the number of traversals in the orientation.

▶ **Lemma 20.** *1-player motion planning with any monotonic k-tunnel DAG gadget with a forced distant closing is NP-hard.*

Proof. Consider all states which have forced distant closings, and let s be such a state that is minimal in the state-transition DAG, so that after making a transition from state s there are no forced distant closings. We will use a forced distant closing in s with smallest size; say this forced distant closing traverses tunnel t and has size i. We chain the i tunnels in the orientation for the forced distant closing, in the directions specified by the orientation, to make what is effectively a single long tunnel r. We will use the tunnels t and r in a reduction from 3SAT, and they have two important properties:

- If the agent traverses t, it cannot later traverse r: since we are using a forced distant closing, after traversing t at least one (oriented) tunnel in r is not traversable. Since there are no distant openings, this tunnel cannot become traversable again.
- The agent can traverse r from state s: in state s, each tunnel in r is open. The agent begins by traversing the first tunnel in r. This cannot be a forced distant closing for the remaining $i-1$ tunnels, since we assume the smallest forced distant closing has size i. So the agent can choose a transition which leaves the remaining tunnels in r open. After this first traversal, there are no more forced distant closings, so the robot can always choose a transition which leaves the remaining tunnels in r open.

We can now describe the reduction, which is very similar to the reduction in the proof of Lemma 19. Each literal in a 3-CNF formula is represented by a gadget in state s, with the tunnels r chained together. Each variable x_i is represented by a connection to two different paths, one which goes through t for the x_i literals, and one for the $\neg x_i$ literals. We place a single-use gadget at the start and end of each branch of each variable to ensure only one side of the variable is traversed. The single-use gadget prevents the agent from returning on the same branch, and if the agent returns via the other branch, they will not be able to proceed to the next variable.

When the agent goes through the x_i (resp $\neg x_i$) path of a variable, it closes r in the gadget for each literal x_i ($\neg x_i$), which corresponds to assigning x_i to false (true). This is reversed from the reduction for gadgets with distant openings.

Each clause contains connections between the r for each of its literals. All variable gadgets are laid out in series followed by the clause gadgets, with the goal location at the end of the clause gadgets. Each clause gadget can only be traversed if at least one of its corresponding variable gadgets has *not* been traversed, leaving at least one passage r open. The agent can reach the goal location exactly when it has a path through the variable gadgets which leaves each clause gadget traversable, which corresponds to a satisfying assignment of the 3-CNF formula. ◀

▶ **Lemma 21.** *1-player motion planning with any monotonic k-tunnel DAG gadget with no forced distant closing is in NL.*

Proof. The proof follows that of Theorem 2, though we must be more careful to account for optional distant closings. As in Theorem 2, if a system of gadgets has a solution, then a solution of minimal length does not intersect itself. This only requires that the gadget has no distant openings, since then making transitions can never increase traversability, and the shortcutting argument applies.

We locally convert the system of gadgets into a directed graph, and show a path in the graph from the start location to the goal location corresponds to a solution to the system of gadgets which does not intersect itself. Given a (not self-intersecting) path in the graph, we follow the corresponding path through the system of gadgets. When we make a traversal, we must pick a transition to avoid closing tunnels we will need later. This is always possible because there are no forced distant closings; we can always choose a transition which does not close any traversal in the orientation consisting of the traversals the path will later take. By doing this, we ensure that every traversal we need is available when we get to it, so the system of gadgets is solvable.

Suppose there is a solution to the system of gadgets that does not intersect itself. Since it uses each tunnel at most once, and the gadget has no distant openings, the traversability of each tunnel does not change before the solution uses it. Thus the solution is also a path in the directed graph.

So the system of gadgets has a solution iff there is a path from the start location to the end location in the directed graph. Since we can locally convert the system of gadgets to the graph in logarithmic space and solve reachability in NL, the motion planning problem is in NL. ◀

Combining Lemmas 17, 18, 19, 20, and 21, we have our dichotomy:

▶ **Theorem 22.** *1-player motion planning with a k-tunnel DAG gadget is NP-complete if the gadget has a distant opening or forced distant closing, and otherwise is in NL.*

It is natural to wonder whether this condition for hardness can be checked in polynomial time. That is, is there a polynomial-time algorithm which determines whether 1-player motion planning with a given DAG gadget is NP-complete? For all of our other dichotomies, the question of whether a gadget of the appropriate type satisfies the condition for hardness is clearly in P; in fact, in L. But a forced distant closing involves an orientation of the tunnels in the gadget, so there may be exponentially many potential forced distant closings to check. We will show that whenever it is necessary to search through each potential forced distant closing, the number of states of the gadget is exponential in the number of tunnels, so the search takes time polynomial in the number of states.

First, it is easy to determine whether a DAG gadget has a distant opening in polynomial time, since we can iterate through the transitions and see whether each one opens another tunnel. So we consider gadgets with no distant openings, and wish to determine whether they have a forced distant closing.

▶ **Lemma 23.** *Suppose a monotonic DAG gadget has a state s with k open tunnels, and there are no forced distant closings from states reachable from s. Then the gadget has at least 2^k states reachable from s.*

Proof. For each subset of the open tunnels in s, we will find a state that has exactly those tunnels open. Since there are 2^k such subsets, this implies there are at least 2^k states. Assume without loss of generality that each tunnel is traversable from left to right in state s.

Given a subset X of the open tunnels, we perform transitions starting from s as follows. For each tunnel not in X, traverse the tunnel repeatedly until it is closed in both directions; this must happen eventually because the gadget is a DAG. At each traversal, choose a transition which does not close any other tunnel from left to right. If there were no such choice of transition, that traversal with all other tunnels oriented from left to right would be a forced distant closing, which does not exist by assumption.

After making these transitions, we have closed each tunnel not in X without closing any tunnels in X. Since the gadget is monotonic, we have not reopened any tunnel. So the final state has exactly the tunnels in X open. ◀

▶ **Theorem 24.** *Deciding whether a 1-player motion planning with a k-tunnel DAG gadget is NP-complete can be done in polynomial time.*

Proof. The following algorithm checks in polynomial time whether 1-player motion planning with a given a DAG gadget is NP-complete.

- For each transition, see whether it is a distant opening. If it is, accept.
- Iterate through the states of the gadget in reverse order; i.e. check each state reachable from s before checking s. For each state, and for each traversal from that state:

- Suppose the state has k open tunnels other than the tunnel of the traversal. If every transition corresponding to the traversal leaves fewer than k of these tunnels open, accept.
- Enumerate the 2^k orientations of these k open tunnels, and check for each orientation whether it is a forced distant closing with the traversal. If it is, accept.

- Reject.

If the gadget has a distant opening, the algorithm notices it in the first step. Otherwise, we check for each state and traversal whether it has a forced distant closing. If every transition for a traversal reduces the number of other open tunnels, than any orientation of the other tunnels gives a forced distant closing. Otherwise, we check for each orientation whether it gives a forced distant closing. So the algorithm accepts exactly when the gadget has a distant opening or a forced distant closing, which is when 1-player motion planning with the gadget is NP-complete by Theorem 22.

The only step of the algorithm which does not obviously take polynomial time is running through all 2^k orientations of tunnels. Suppose the algorithm reaches this step for some state and traversal. Then there are no forced distant closings after making a transition from this state, since we would have accept already if there were. Also, there is some transition corresponding to the traversal which leaves all k other open tunnels open. By Lemma 23, there are at least 2^k states reachable after making this transition. In particular, the gadget has more than 2^k states, so enumerating the 2^k orientations takes time polynomial in the number of states. Thus the algorithm runs in polynomial time. ◀

6 2-Player Bounded Motion Planning

In this section, we show that it is PSPACE-complete to decide who wins in a 2-player race with any nontrivial DAG gadget (having at least one transition). To do so we give a construction that shows hardness for single-use paths and single-use one-way gadgets by a reduction from QBF. A simpler construction is possible, but this construction is more easily adapted to the team game in Section 7. This gives us a nice example of the 2-player local motion planning problem fitting into the canonical complexity class for two-player bounded games. It is also of interest because of how incredibly simple this gadget is. Two-location gadgets trivially do not have interacting tunnels (there is no other tunnel to interact with) and thus the 1-player version of these problems are contained in NL by Theorem 2.

▶ **Lemma 25.** *2-player motion planning with any set of DAG gadgets is in PSPACE.*

Proof. Since each gadget can undergo only a polynomial number of transitions, the length of the game is polynomially bounded. An alternating Turing machine which uses ∀ states to pick Black's moves and ∃ state to pick White's moves can simulate the game in polynomial time, so the motion planning problem is in AP = PSPACE. ◀

▶ **Lemma 26.** *2-player motion planning with the single-use bidirectional gadget is PSPACE-complete.*

Proof. Containment in PSPACE follows from Lemma 25. For PSPACE-hardness, we reduce from quantified boolean formulas (QBF). See Appendix A.2 for a definition of QBF.

We begin by describing the gadgets used in the reduction. The variable gadget is shown in Figure 27. Most of the gadget is two branches, corresponding to a variable and its negation. Each branch has a series of forks separated by single-use paths. There will be a number of forks depending on the number of occurrences of a literal in the formula; two forks are shown. Each side of each fork has two single-use paths in series. The game will be constructed so that White always prefers the top side of a fork to be traversable, and Black prefers them to be not traversable; the top of a fork will be used later in evaluating the formula.

During the game, both players will pass through each variable gadget, with one player taking each of the two branches. White will take the bottom side of each fork on their branch, and Black will take the top side. Afterwards, only the branch which White took will have forks whose top sides are traversable. Thus we consider the assignment of the variable to be the literal corresponding to the branch White takes.

Suppose both players are at the left end of a variable gadget, and it is Player 1's (who may be White or Black) turn. Player 1 picks a branch, and Player 2 must walk down the other branch. Player 1 arrives at the right end of the branches immediately before Player 2. If Player 1 proceeds along the bottom path, Player 2 wins, so Player 1 must take the top path, which takes one turn longer. After traversing the variable gadget, both players are at the right end, and it is Player 2's turn, so the other player gets to choose a branch in the next variable gadget.

The clause gadget is shown in Figure 28. There are three paths from the left end to the right end, corresponding to the literals in a clause. Each path goes through a fork in a variable gadget. After variables are assigned, the single-use paths on each end of the fork are used, as are either those on the top or those on the bottom of each fork. If the top single-use paths are used, that path through the clause gadget is blocked, and if the bottom paths are used, that path is open. White will ultimately win by traversing each clause gadget, so White prefers to use the bottom side of a fork, and Black prefers to use the top side.

Each path has a large amount of delay (gadgets in series) before and after the fork, so that trying to use the clause gadget during variable assignment results in losing before reaching the end of the delay.

The race gadget is shown in Figure 29. It ensures both players proceed though variable gadgets as fast as possible. Let Player 1 be the player who reaches the race gadget first in this situation, immediately before Player 2; they are also the player who did not pick the assignment of the last variable. If Player 1 takes the bottom path, Player 2 will win, so Player 1 takes the top path. Then Player 2 takes the bottom path, and now the two players have been separated.

If Player 1 arrives more than a turn ahead of Player 2, they can take the bottom path. The next turn, before Player two can do anything at the race gadget, Player 1 wins. If Player 2 reaches the race gadget first, they can take the top path and win.

Given a quantified boolean formula with V variables and C clauses, we construct a system of gadgets as follows. We assume the QBF has alternating quantifiers beginning with $\exists$. There is a series of variable gadgets connected end-to-end corresponding to the variables of the formula, in the order of quantification. The goal location inside each variable gadget is a win for alternating players, beginning with Black. The branches of the variable gadget corresponding to x correspond to the literals x and $\neg x$. Each branch of that variable gadget has enough forks that each instance of a x or $\neg x$ in the formula corresponds to a fork, and the two branches have the same number of forks.

There is a clause gadget for each clause in the formula, connected in series. The three branches of a clause gadget correspond to the three literals in the clause. Each branch goes through the fork in the appropriate variable gadget corresponding to that instance of the literal. The delay before and after each fork consists of $9C + 3V$ single-use paths. The right end of the last clause is connected to a White goal location.

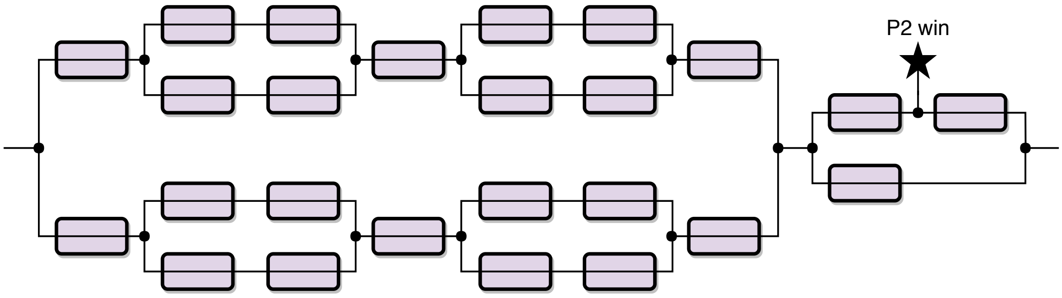


Figure 27 A variable gadget. The players arrive at the left, each take one path across, and exit at the right.

A race gadget is connected to the right end of the last variable gadget, with the goal locations such that Player 1 is the player with a win gadget inside the last variable gadget. The path with a White win gadget, which Black will walk down, is followed by $C(18C+6V+1)+2$ of single-use paths in series leading to a Black win gadget. The other path, which White will walk down, is connected to the first clause gadget.

Both players begin at the left end of the first variable gadget, and White goes first.

The game begins with White choosing a branch of the first variable gadget, corresponding to a choice of variable, and Black taking the other branch. Then Black chooses a branch of the second variable gadget, choosing the assignment of the variable based on the path White is forced to take. The players continue to take turns assigning variables. If either player deviates from this, such as by going into the delay in a clause gadget or by going backwards along another path, the other player will reach the race gadget first and win; the delay in clause gadgets is long enough to ensure that they do not have time to get through the clause gadget before losing. Otherwise both players arrive at the race gadget, and are sent down different branches.

White then proceed through each clause in series. Each branch of a clause is traversable if and only if the corresponding literal is true (since White took the bottom side and Black took the top side of each clause). The single-use paths between forks ensure that White cannot do anything other than progress through each clause gadget. If the formula is satisfied, White has a path through the clauses, and wins after $C(18C + 6V + 1)$ turns. If the formula is not satisfied, Black, who is walking down their long path, wins after slightly longer. Thus White has a forced win if and only if the quantified formula is true. ◀

▶ **Lemma 27.** *2-player motion planning with the single-use one-way gadget is PSPACE-complete.*

Proof. We again reduce from QBF. In the reduction in Lemma 26, neither player ever has to move through a single-use gadget to the left. Thus we can replace each bidirectional single-use gadget with a one-way single-use gadget pointing to the right, and the reduction still works. ◀

▶ **Corollary 28.** *2-player motion planning with any nontrivial DAG gadget is PSPACE-complete.*

Proof. As noted in Section 5 all DAG gadgets contain a single-use transition. This can be bidirectional or one-way, which are both shown to be PSPACE-hard in Lemmas 26 and 27. Containment in PSPACE is given by Lemma 25. ◀

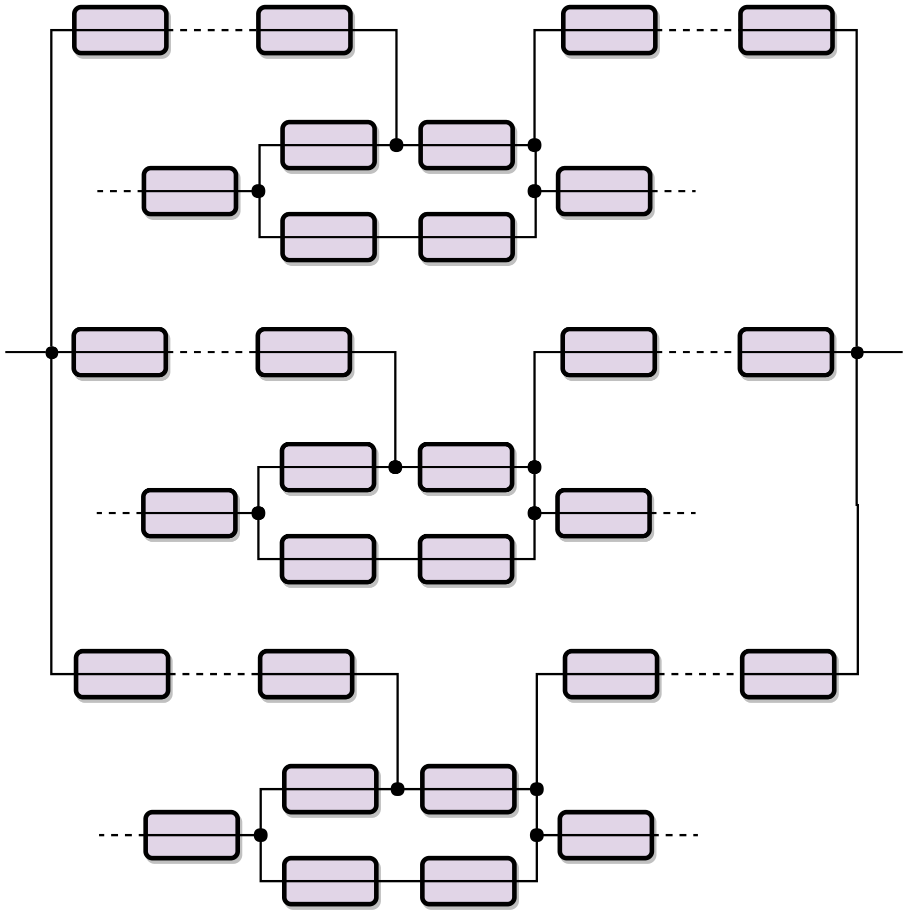


Figure 28 A clause gadget. Each literal is also part of a variable gadget. Each branch has a long series of gadgets so that it takes a large amount of time to traverse.

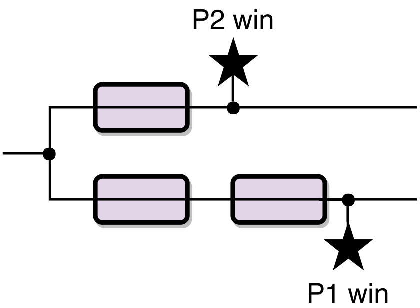


Figure 29 A race gadget. If Player 1 arrives at the left immediately before Player 2, each player ends up on one of the right exits. Otherwise, the player who arrives first wins.

7 Team Bounded Motion Planning

In this section we characterize the complexity of team imperfect information motion planning games with DAG gadgets. Since DAG gadgets are inherently bounded, the problem is in NEXPTIME, shown in Lemma 29. We go on to show in Lemma 30 that any nontrivial DAG gadget is NEXPTIME-complete by first giving a reduction from dependency quantified boolean formula (DQBF) for the single-use gadget. We then show that this proof adapts for single-use one-way gadgets. Since all DAG gadgets with at least one transition contain at least one of these, we achieve hardness for all such DAG gadgets.

▶ **Lemma 29.** *Team motion planning with any set of DAG gadgets is in NEXPTIME.*

Proof. A ***partial history*** for a player is the sequence of visible gadget states and moves made by that player, up to some point in the game. A ***strategy*** is a family of functions, one for each White player, that assign to each possible partial history a legal move from the position at the end of the partial history.

Since the gadget is a DAG, the game lasts a polynomial number of turns. Each player has polynomially many choices for each move, so there are only exponentially many possible sequences of moves, and only exponentially many possible partial histories for each player. Thus a strategy can be written in an exponential amount of space.

To determine whether White has a forced win in the team game, first nondeterministically pick a strategy. Then, for each possible sequence of moves the Black players could make, simulate the game with the White players following the strategy. If Black ever wins, reject; if White always wins, accept. This nondeterministic algorithm accepts if and only if there is some strategy White can use to force a win. The algorithm runs in exponential time because there are exponentially many sequences of moves the Black players might make, and the game for each such sequence takes a polynomial amount of time to simulate. Thus the algorithm decides the team game on systems of the gadget in NEXPTIME. ◀

▶ **Lemma 30.** *Team motion planning with the single-use bidirectional gadget is NEXPTIME-complete.*

Proof. Containment in NEXPTIME follows from Lemma 29. For NEXPTIME-completeness, we reduce from dependency quantified boolean formulas (DQBF). See Appendix A.2 for a definition of DQBF. In this reduction White represents the existential variables and Black represents the universal variables.

The reduction uses the same gadgets as that in Lemma 26, except that the clause gadget is modified as in Figure 30. This allows the White player checking the formula to try each literal, and return to the start of the clause gadget if the literal is false. This is necessary because the White player cannot see the state of the literals until arriving at them. For variable gadgets, we do not include the portion with a win gadget for Player 2 (the rightmost quarter or so in Figure 27), since we no longer want players to alternate choosing variables.

We construct the system of gadgets as follows. The overall structure is shown in Figure 31. For each set of variables $\vec{x}_1$, $\vec{x}_1$, $\vec{y}_1$, and $\vec{y}_2$, there is a corresponding set of variable gadgets (without the win gadget component) connected in series, followed by a race gadget. For simplicity, we will put C forks in each branch of each variable, where the formula has C clauses, though usually we need much fewer. Then each variable gadget takes $k = 3C + 1$ turns to traverse. We call the top path of a race gadget the ***fast exit*** and the bottom path the ***slow exit***, since (in normal play) the first (second) player to arrive leaves through the fast (slow) exit. It will become clear which player each win gadget in a race gadget is for.

The turn order will be B, then W_1, then W_2. Both B and W_1 start at the beginning of the variable gadgets for $\vec{x}_1$. W_2 starts next to a delay line of length d_1. The fast exit of the race gadget for $\vec{x}_1$ and the end of this delay line both connect to the beginning of the $\vec{x}_2$ variable gadgets. The slow exit connects to a delay line of length d_2. The end of this delay line and the fast exit of the $\vec{x}_2$ race gadget connect to the beginning of the $\vec{y}_1$ variable gadgets, and the slow exit connects to a delay line of length d_3. The end of this delay line is connected to the *slow* exit of the $\vec{y}_1$ race gadget and the beginning of the $\vec{y}_2$ variable gadgets. The fast exit of the $\vec{y}_1$ race gadget is connected to yet another delay line of length d_4. The slow exit of the $\vec{y}_2$ race gadget is connected to a long delay line of length d_5 followed by a win gadget for B, and the fast exit is connected to a longer delay line of length $d_5 + 3$.

This all serves to accomplish the following. First, B chooses the assignment for $\vec{x}_1$ accompanied by W_1, so W_1 learns the assignment. Then B and W_1 are separated, and B assigns $\vec{x}_2$ accompanied by W_2. Next, W_1 chooses $\vec{y}_1$ accompanied by B, and finally W_2 chooses $\vec{y}_2$ accompanied by B. The delays d_1 through d_4 are chosen so that the White players arrive at exactly the right time; we have $d_1 = |\vec{x}_1|k + 1$, $d_2 = |\vec{x}_2|k - 1$, $d_3 = |\vec{y}_1|k$, and $d_4 = |\vec{y}_2|$. If a player deviates during variable assignment, they will arrive at their next race gadget too late, and lose.

The end of the final delay line for W_1, of length d_4, is connected to the first clause gadget, and the clause gadgets are connected in series corresponding to the clauses of the formula. The delay lines in each branch of each clause gadget have length Vk, where V is the number of variables; this ensures that if a player enters one of the delay lines during variable selection, an opponent will reach a race gadget and win before they accomplish anything. The end of the last clause gadget is connected to a win gadget for W_1. When W_1 reaches each clause gadget, they try the literals one at a time. When they cross the delay line to the fork, if the fork is traversable, they move on to the next clause. Otherwise they return through the other delay line and try the next literal. Each clause takes up to $6Vk + 1$ turns to cross.

If the formula is satisfied, W_1 eventually gets through all the clauses and wins. Otherwise, B wins after walking through their delay line of length d_5, which we can set to $C(6Vk+1)+1$.

We have seen that no player or team can benefit by deviating from normal play, and normal play is equivalent to the game corresponding to the DQBF. Thus White has a forced win if and only if the DQBF is true. ◀

▶ **Lemma 31.** *Team motion planning with the single-use one-way gadget is NEXPTIME-complete.*

Proof. The reduction in Lemma 30 still works when we replace each single-use bidirectional gadget with a one-way bidirectional gadget. We have to be a bit more careful than in Lemma 27: of the two paths in a clause gadget from the beginning to a fork, we need one path to point to the right and the other to point to the left, allowing W_1 to return from that fork. All other gadgets point to the right. ◀

▶ **Corollary 32.** *Team motion planning with any nontrivial DAG gadget is NEXPTIME-complete.*

Proof. Every DAG gadget has a single-use transition, which may be either bidirectional or one-way. Both cases are shown to be NEXPTIME-hard in Lemmas 30 and 31. Containment in NEXPTIME is Lemma 29. ◀

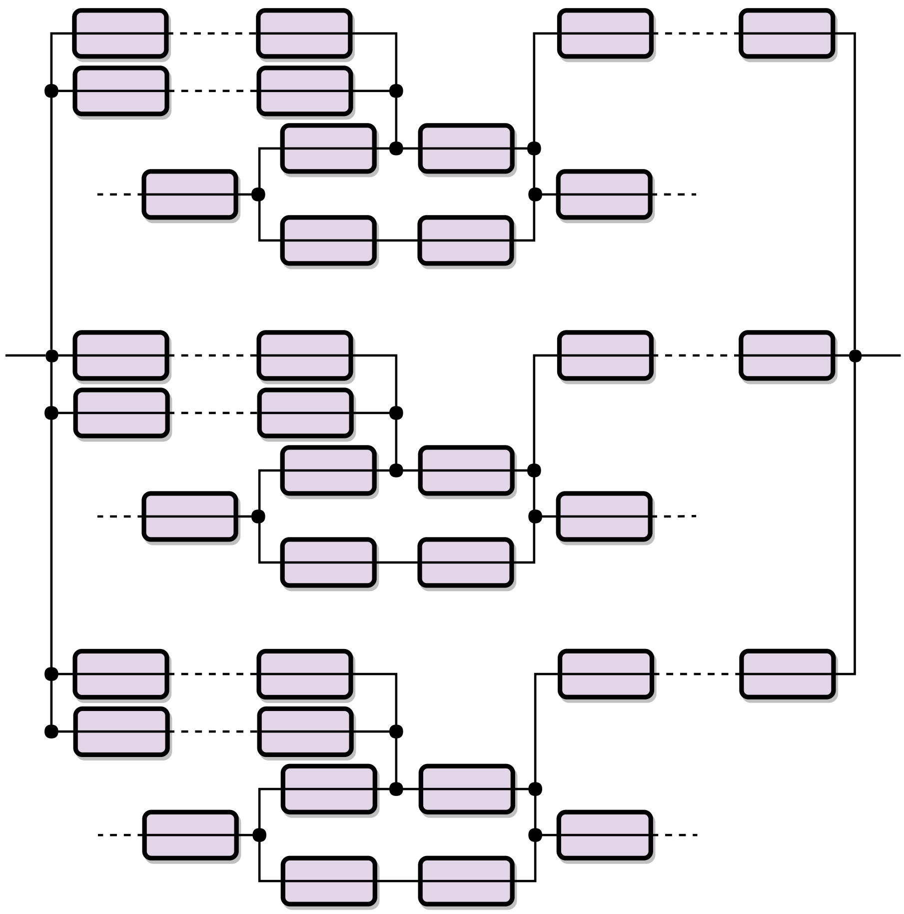


Figure 30 A clause gadget for team games. There are now two paths from the entrance of the clause to each fork, so the White player traversing the clause can return if they discover the fork is not traversable.

8 Applications

In this section we give examples of some known hard problems whose proofs can be simplified by using this motion planning framework.

8.1 PushPull-1F

In this section, we use the results of this paper to provide a simple proof that a Sokoban variant called PushPull-1F is PSPACE-hard, by reducing from motion planning in planar systems of locking 2-toggles (Section 2.3). This problem, and many related problems, were considered in [5] and were shown to be PSPACE-complete in [15] by a reduction from nondeterministic constraint logic; our reduction is much more straightforward using the infrastructure of the gadget framework.

▶ **Definition 33.** *In **PushPull-1F**, there is a square grid containing movable blocks, fixed blocks, an agent, and a goal location. The agent can freely move through empty squares, but can't move through blocks. The agent can push or pull one movable block at a time. The agent wins by reaching the goal location. The corresponding decision problem is whether a*

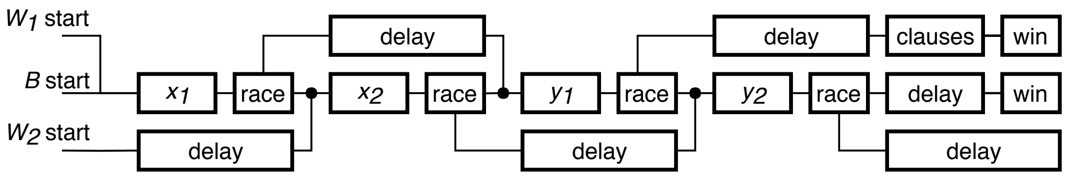


Figure 31 The high-level structure of the DQBF reduction.

*given instance of PushPull-*1*F is winnable.*

In the notation 'PushPull-1F,' 'PushPull' indicates that the agent can both push and pull, '1' indicates the number of blocks which can be moved at a time, and 'F' indicates the existence of fixed blocks [5].

▶ **Theorem 34** ([15])**.** *PushPull-kF is PSPACE-hard for $k \geq 1$.*

Proof. We reduce from 1-player planar motion planning with locking 2-toggles, shown PSPACE-complete in Theorem 10. The (planar) connection graph is implemented using tunnels built with fixed blocks, and the agent and target location are placed appropriately. It suffices to build a gadget which behaves as a locking 2-toggle.

Such a gadget is shown in Figure 32. The two tunnels, currently both traversable, go from top to left and right to bottom. They interact in the center, where traversing either tunnel requires pushing a block into the middle square, which blocks the other tunnel. This is surrounded by four 1-toggles, which prevent additional traversals which aren't possible in a locking 2-toggle. Each 1-toggle is a room with 3 blocks, which can only be entered on one side. Upon entry, the agent can move the blocks to reveal the other exit, but doing so requires blocking the entrance taken, which flips the 1-toggle.

◀

8.2 Mario Kart

Mario Kart is a popular Nintendo racing game whose computational complexity was considered in [3] which showed NP-completeness for 1 player races and PSPACE-completeness for 2 player races with reductions from 3SAT and QSAT respectively. Using results from this paper, the 2 player proof now only needs a single, simple gadget, reducing a several page proof to a paragraph.

▶ **Theorem 35.** *Deciding if a player can force a win in two player Mario Kart is PSPACE-hard.*

Proof. A single-use one-way gadget can be constructed from a ramp and Dash Mushroom in Mario Kart. We place a ramp before a gap in the track long enough that a racer going at the normal maximum speed will not be able to make the jump and will fall onto another track that will take a long time to reach the finish line, ensuring they lose. However, this gap is small enough that, if the player uses a Dash Mushroom before, the increase in speed will allow them to make the jump. We put a single Dash Mushroom power-up before each ramp, ensuring the first racer to arrive can pick up the item and use it to cross the gap. To ensure a racer does not pick up the item and then keep it for later use, we precede the mushroom and ramp with a one-way gadget implemented by a long-fall. Along with the trivial existence of crossovers and the finish line as a location based win condition, Mario Kart is PSPACE-hard by Theorem 27. ◀

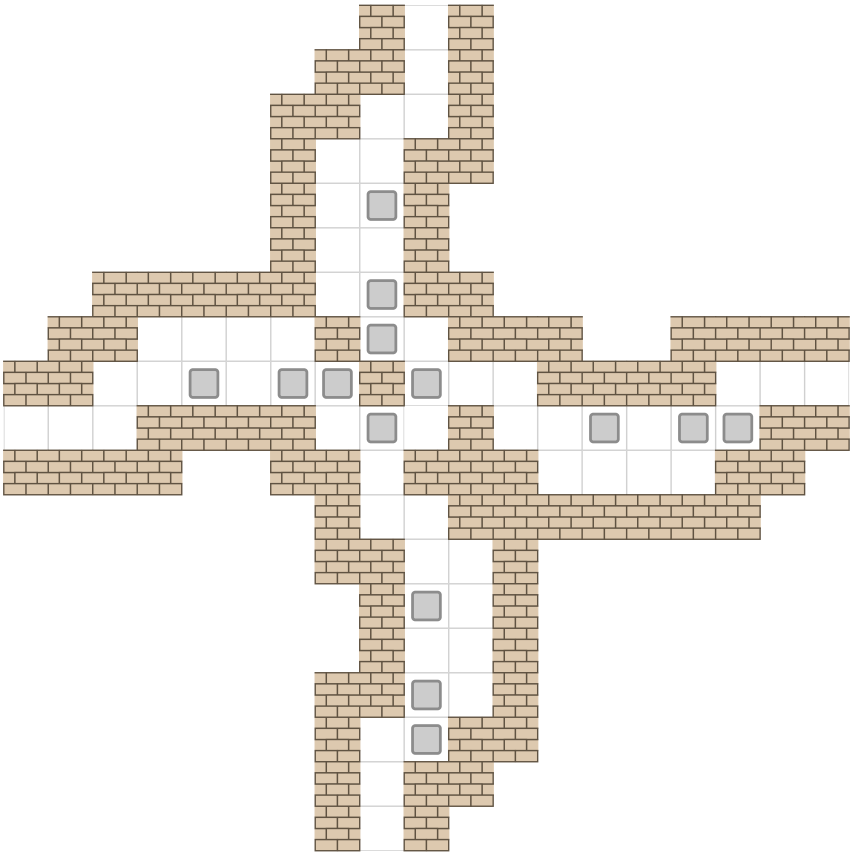


Figure 32 A locking 2-toggle in PushPull-1F.

9 Open Problems

This paper characterizes the complexity of two large classes of gadgets (DAG gadgets and reversible deterministic gadgets). Ideally, we could fully characterize the complexity of motion planning for every gadget type (and set of gadgets) as being easy or hard. There are many specific steps we might take towards this grand goal:

1. Is 2-player motion planning with 1-toggles EXPTIME-complete? This would complete our characterization for 2-player games with k-tunnel reversible deterministic gadgets. As an easier target, we could prove PSPACE-hardness, perhaps by adapting the 2-player proof for one-way closing gadgets.
2. Can we extend our characterizations of k-tunnel reversible deterministic gadgets to remove one of these restrictions? Specifically, non-tunnel gadgets, non-reversible gadgets, and nondeterministic gadgets are all interesting (and challenging) goals.
3. Which motion planning problems remain hard on planar systems of gadgets, like we proved for 1-player reversible deterministic? Are there any examples of gadgets where the planar version of the motion planning problem has a different complexity?

While we focused in this paper on general theory building, we can also explore the application of this motion planning framework to analyze the complexity of specific problems of interest. We conjecture that the results of this paper simplify many past hardness proofs, which can now be reduced to one or two figures showing how to build any hard gadget according to our characterization, and how to connect gadgets together. See the hardness surveys [8, 11, 12, 4] for a large family of candidate problems. Of course, we also hope that this framework will enable the solution of many open problems in this space.

References

1 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (NP-)hard. In *Proceedings of the 7th International Conference on Fun with Algorithms (FUN 2014)*, Lipari Island, Italy, July 2014.

2 Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015. Originally at FUN 2014.

3 Jeffrey Bosboom, Erik D Demaine, Adam Hesterberg, Jayson Lynch, and Erik Waingarten. Mario kart is hard. In *Japanese Conference on Discrete and Computational Geometry and Graphs*, pages 49–59. Springer, 2015.

4 Erik D. Demaine. 6.890: Algorithmic lower bounds: Fun with hardness proofs. MIT lecture notes and videos, 2014. http://courses.csail.mit.edu/6.890/fall14/.

5 Erik D. Demaine, Isaac Grosof, and Jayson Lynch. Push-pull block puzzles are hard. In *Proceedings of the 10th International Conference on Algorithms and Complexity*, pages 177–195, Athens, Greece, May 2017. URL: https://doi.org/10.1007/978-3-319-57586-5_16, `doi:10.1007/978-3-319-57586-5_16`.

6 Erik D. Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, pages 18:1–18:21, La Maddalena, Italy, June 13–15 2018.

7 Erik D. Demaine and Robert A. Hearn. Constraint Logic: A uniform framework for modeling computation as games. In *Proceedings of the 23rd Annual IEEE Conference on Computational Complexity*, pages 149–162, June 2008.

8 Erik D. Demaine and Robert A. Hearn. Playing games with algorithms: Algorithmic combinatorial game theory. In Michael H. Albert and Richard J. Nowakowski, editors, *Games of No Chance 3*, volume 56 of *Mathematical Sciences Research Institute Publications*, pages 3–56. Cambridge University Press, 2009.

9 Michal Forisek. Computational complexity of two-dimensional platform games. In *Proceedings International Conference on Fun with Algorithms (FUN 2010)*, pages 214–227, 2010.

10 Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

11 Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A. K. Peters, Ltd., Natick, MA, USA, 2009.

12 Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008. URL: http://www.bibsonomy.org/bibtex/25cc2e9e313951aad1b2127959e6b2269/dblp.

13 Sanjeev Khanna, Madhu Sudan, Luca Trevisan, and David P. Williamson. The approximability of constraint satisfaction problems. *SIAM Journal on Computing*, 30(6):1863–1920, 2000. URL: https://doi.org/10.1137/S0097539799349948, `doi:10.1137/S0097539799349948`.

14 Dénes Kőnig. Über eine Schlussweise aus dem Endlichen ins Unendliche. *Acta Sci. Math. (Szeged)*, 3(2–3):121–130, 1927.

15 André G. Pereira, Marcus Ritt, and Luciana S. Buriol. Pull and PushPull are PSPACE-complete. *Theoretical Computer Science*, 628:50–61, 2016. `doi:https://doi.org/10.1016/j.tcs.2016.03.012`.

16 Gary L. Peterson and John H. Reif. Multiple-person alternation. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, SFCS '79, pages 348–363, Washington,

DC, USA, 1979. IEEE Computer Society. URL: https://doi.org/10.1109/SFCS.1979.25, doi:10.1109/SFCS.1979.25.
17 Thomas J. Schaefer. The complexity of satisfiability problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, pages 216–226, San Diego, California, May 1978.
18 Tom C Van Der Zanden and Hans L Bodlaender. PSPACE-completeness of Bloxorz and of games with 2-buttons. In *International Conference on Algorithms and Complexity*, pages 403–415. Springer, 2015.
19 Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.
20 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

A Problem Definitions

In this appendix, we give formal definitions for the known hard problems used in this paper. In the paper we use single player, 2-player, and team imperfect information versions of Constraint Logic and Boolean Formula Games. The exact problems are specified in the following sections.

A.1 Constraint Logic

Constraint Logic [7, 11] is a uniform family of games — one-player, two-player, or team, with both bounded and unbounded variants — with the appropriate complexity in each case (as in Table 1). We will only describe the unbounded variants of Constraint Logic, as we use formula games for our bounded reductions. We also do not describe zero-player Constraint Logic, as we do not need it here.

In general, a ***constraint graph*** is an undirected maximum-degree-3 graph, where each edge has a weight of 1 (called a red edge) or 2 (called a blue edge). A ***legal configuration*** of a constraint graph is an orientation of the edges such that, at every vertex, the total incoming weight is at least 2. A ***legal move*** in a legal configuration of a constraint graph is a reversal of a single edge that results in another legal configuration.

In ***1-player*** Constraint Logic (also called ***Nondeterministic Constraint Logic or NCL***), we are given a legal configuration of a constraint graph and a target edge e, and we want to know whether there is a sequence of legal moves ending with the reversal of target edge e. In this game, two types of vertices suffice for PSPACE-completeness: an OR vertex has exactly three incident blue edges, and an AND vertex has exactly one incident blue edge and exactly two incident red edges. We can also assume that each OR vertex can be assigned two "input" edges, and the overall construction is designed to guarantee that at most one input edge is incoming at any time; thus, we only need a "Protected OR" gadget which does not handle the case of two incoming inputs. Furthermore, the problem remains PSPACE-complete for planar constraint graphs.

In ***2-player Constraint Logic (2CL)***, each edge of a constraint graph is also colored either black or white, and two players named Black and White alternate making valid moves where each player can only reverse an edge of their color. Given a legal configuration of a constraint graph, a target white edge for White, and a target black edge for Black, the goal is to determine whether White has a forced win, i.e., a strategy for reversing their target edge before Black can possibly reverse their target edge. In this game, six types of vertices suffice for EXPTIME-completeness: AND and OR vertices where all edges are white, AND vertices where all edges are black, AND vertices where the blue edge is white and one or both of the red edges are black, and degree-2 vertices where exactly one edge is black.

In ***Team Private Constraint Logic (TPCL)***, there are two players on the White team and one player on the Black team, who play in round-robin fashion. In each move, the player can reverse up to a constant number k of edges of their color. Each player has a target edge to reverse, and can see the orientation of a specified set of edges, including edges of their own color and edges incident to those edges. Given a legal configuration of a constraint graph, the goal is to determine whether the White team has a forced win; i.e., whether one of the White players can reverse their target edge before Black can. In this game, all possible black/white colorings of AND and OR vertices suffice for RE-completeness. (Only undecidability has been claimed before, but RE-completeness follows by the same arguments.)

A.2 Formula Games

A ***3-CNF formula*** is a boolean formula φ of the form $C_1 \wedge \cdots \wedge C_k$, where each ***clause*** C_i is the disjunction of up to three ***literals***, which are variables or their negations. An ***assignment*** for such a formula specifies a truth value for each variable, and is ***satisfying*** if the formula is true under the assignment.

In ***3SAT***, we are given a 3-CNF formula, and we want to know whether it has a satisfying assignment. 3SAT is NP-complete [10].

A ***partially quantified boolean formula*** is a formula of the form $Q_1 x_1 : \cdots : Q_n x_n : \varphi$, where Q_i is one of the quantifiers $\forall$ or $\exists$, x_i is a (distinct) variable, and φ is a 3-CNF formula. An ***assignment*** for a partially quantified boolean formula specifies a truth value for each variable in φ that is not any x_i, called ***free*** variables. For a partially quantified boolean formula $\psi = Q_1 x_1 : \cdots : Q_n x_n : \varphi$ with $n > 0$, let $\psi' = Q_2 x_2 : \cdots : Q_n x_n : \varphi$. Given an assignment S for ψ, define assignments $S + x_1$ and $S + \neg x_1$ for ψ' which assign the same truth value as S to each free variable of φ and assign 'true' and 'false' to x_1, respectively. The truth value of ψ under S is defined recursively as follows:

- If $n = 0$ (so $\psi = \varphi$), ψ is true under S if and only if φ is true under S.
- If $n > 0$ and $Q_1 = \forall$, ψ is true under S if and only if ψ' is true under both $S + x_1$ and $S + \neg x_1$.
- If $n > 0$ and $Q_1 = \exists$, ψ is true under S if and only if ψ' is true under at least one of $S + x_1$ and $S + \neg x_1$.

A ***quantified boolean formula*** is a partially quantified boolean formula with no free variables. A quantified boolean formula has only one assignment (which is empty), so we say it is true if it is true under this unique assignment.

The truth value of a quantified boolean formula $\psi = Q_1 x_1 : \cdots : Q_n x_n : \varphi$ is equivalent to whether $\exists$ has a forced win in the following game: two players $\exists$ and $\forall$ choose an assignment for φ by assigning variables in the order they are quantified, with player Q_i choosing the truth value of x_i. $\exists$ wins if the assignment satisfies φ.

In ***QBF***, we are given a (fully) quantified boolean formula, and we want to know whether it is true. QBF is PSPACE-complete, even if we restrict to formulas with alternating quantifiers beginning with $\exists$. This restriction is equivalent to that $\exists$ and $\forall$ take alternating turns, with $\exists$ going first [10].

A ***dependency quantified boolean formula*** is a formula of the form $\forall x_1 : \cdots : \forall x_m : \exists y_1(s_1) : \cdots : \exists y_n(s_n) : \varphi$, where x_i and y_j are (distinct) variables, φ is a 3-CNF formula, and s_j is a subset of $\{x_i \mid i \leq m\}$. We also require that every variable in φ is some x_i or

y_j (φ has no free variables). A ***strategy*** for a dependency quantified boolean formula is a collection of functions $f_j : \{\text{true}, \text{false}\}^{s_j} \to \{\text{true}, \text{false}\}$ for $j = 1, \ldots, n$. A strategy ***solves*** a dependency quantified boolean formula if for every map $S : \{x_i \mid i \leq m\} \to \{\text{true}, \text{false}\}$, the assignment given by $x_i \mapsto S(x_i)$ and $y_j \mapsto f_j(S|_{s_j})$ satisfies φ. Intuitively, y_j is only allowed to depend on the variables in s_j. A quantified boolean formula is a special case of a dependency quantified boolean formula, where each $s_j = \{x_i \mid i < k\}$ for some k. A dependency quantified boolean formula is ***true*** if there is a strategy that solves it.

The truth value of a dependency quantified boolean formula $\forall x_1 : \cdots : \forall x_m : \exists y_1(s_1) : \cdots : \exists y_n(s_n) : \varphi$ is equivalent to whether the $\exists$ team has a forced win in the following game, which puts a team of one player $\forall$ against a team of players $\exists_j$ for $j = 1, \ldots, n$: $\forall$ picks a truth value for each x_i. $\exists_j$ sees the truth value for each element of s_j (and nothing else) and picks a truth value for y_j. The $\exists$ team wins if the resulting assignment satisfies φ.

In the ***DQBF*** problem, we are given a dependency quantified boolean formula, and we want to know whether it is true. DQBF is NEXPTIME-complete even if we restrict to formulas of the form $\forall \vec{x}_1 : \forall \vec{x}_2 : \exists \vec{y}_1(\vec{x}_1) : \exists \vec{y}_2(\vec{x}_2) : \varphi$, where $\vec{x}_i$ and $\vec{y}_i$ may contain multiple variables, and each variable in $\vec{y}_i$ can depend on all the variables in $\vec{x}_i$. This restriction is equivalent to requiring that the $\exists$ team has two players who each choose multiple variables, and they see disjoint exhaustive subsets of the variables $\forall$ picks [16].