PSPACE-Completeness of Sliding-Block Puzzles and Other Problems through the Nondeterministic Constraint Logic Model of Computation

Robert A. Hearn* Erik D. Demaine*

Abstract

We present a nondeterministic model of computation based on reversing edge directions in weighted directed graphs with minimum in-flow constraints on vertices. Deciding whether this simple graph model can be manipulated in order to reverse the direction of a particular edge is shown to be PSPACE-complete by a reduction from Quantified Boolean Formulas. We prove this result in a variety of special cases including planar graphs and highly restricted vertex configurations, some of which correspond to a kind of passive constraint logic. Our framework is inspired by (and indeed a generalization of) the "Generalized Rush Hour Logic" developed by Flake and Baum [4].

We illustrate the importance of our model of computation by giving simple reductions to show that several motion-planning problems are PSPACE-hard. Our main result along these lines is that classic unrestricted sliding-block puzzles are PSPACE-hard, even if the pieces are restricted to be all dominoes (1×2 blocks) and the goal is simply to move a particular piece. No prior complexity results were known about these puzzles. This result can be seen as a strengthening of the existing result that the restricted Rush Hour™ puzzles are PSPACE-complete [4], of which we also give a simpler proof. We also greatly strengthen the conditions for the PSPACE-hardness of the Warehouseman's Problem [6], a classic motion-planning problem. Finally, we strengthen the existing result that the pushing-blocks puzzle Sokoban is PSPACE-complete [2], by showing that it is PSPACE-complete even if no barriers are allowed.

1 Introduction

Motivating Application: Sliding Blocks. Motion planning of rigid objects is concerned with whether a collection of objects can be moved (translated and rotated), without intersection among the objects, to reach a goal configuration with certain properties. Typically, one object is distinguished, the remaining objects serving as obstacles, and the goal is for that object to reach a particular position. This general problem arises in a variety of applied contexts such as robotics and graphics. In addition, this problem arises in the recreational context of *sliding-block puzzles* [7], where the pieces are typically integral rectangles, L shapes, etc., and the goal is simply to move a particular piece to a specified target position. See Figure 1 for an example.

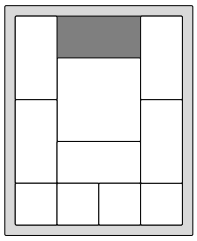


Figure 1: The Donkey Puzzle: move the large square to the bottom center.

The *Warehouseman's Problem* [6] is a particular formulation of this problem in which the objects are rectangles of arbitrary side lengths, packed inside a rectangular box. In 1984, Hopcroft, Schwartz, and Sharir [6] proved that deciding whether the rectangular objects can be moved so that each object is at its specified final position is PSPACE-hard. Their construction critically requires that some rectangular objects have dimensions that are proportional to the box dimensions.

*MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139, U.S.A., {rah,edemaine}@csail.mit.edu

Although not mentioned in [6], the Warehouseman's Problem captures a particular form of sliding-block puzzles in which all pieces are rectangles. However, two differences between the two problems are that sliding-block puzzles typically require only a particular piece to reach a position, instead of the entire configuration, and that sliding-block puzzles involve blocks of only constant size. More generally, it is natural to ask for the complexity of the problem as determined by the goal specification and the set of allowed block types.

In this paper, we prove that the Warehouseman's Problem and sliding-block puzzles are PSPACE-hard even for 1×2 rectangles (dominoes) packed in a rectangle. In contrast, there is a simple polynomial-time algorithm for 1×1 rectangles packed in a rectangle. Thus our results are in some sense tight.

Hardness Framework. To prove that sliding blocks and other problems are PSPACE-hard, this paper builds a general framework for proving PSPACE-hardness which simply requires the construction of a couple of gadgets that can be connected together in a planar graph. Our framework is inspired by the one developed by Flake and Baum [4], but is simpler and more powerful. We prove that several different models of increasing simplicity are equivalent, permitting simple constructions of PSPACE-hardness. In particular, we derive simple constructions for sliding blocks, Rush Hour [4], and a restricted form of Sokoban [2].

Nondeterministic Constraint Logic Model of Computation. Our framework can also be viewed as a model of computation in its own right. We show that a Nondeterministic Constraint Logic (NCL) machine has the same computational power as a space-bounded Turing machine. Yet, it has a more concise formal description, and has a natural interpretation as a kind of logic network. Thus, it is reasonable to view NCL as a simple computational model that corresponds to the class PSPACE, just as, for example, deterministic finite automata correspond to regular languages.

Roadmap. Section 2 describes our model of computation in more detail. Section 3 proves increasingly simple formulations of NCL to be PSPACE-complete. Section 4 proves various puzzles and motion-planning problems to be PSPACE-hard using the restricted forms of our model of computation. Section 5 presents an alternative formulation of the NCL problem.

2 Nondeterministic Constraint Logic

In this section we formally define the nondeterministic constraint logic (NCL) model of computation, and give a family of related decision problems whose complexity we are interested in. We then describe in detail a special case of NCL graphs, called *AND/OR constraint graphs*. In the following section we will show the decision problems to be PSPACE-complete, even for the restricted AND/OR constraint graphs.

2.1 Graph Formulation

An NCL "machine" is specified by a *constraint graph*: an undirected graph together with an assignment of nonnegative integers (*weights*) to edges and integers (*minimum in-flow constraints*) to vertices. A configuration of this machine is an orientation (direction) of the edges such that the sum of incoming edge weights at each vertex is at least the minimum in-flow constraint of that vertex. A move from one configuration to another configuration is simply the reversal of a single edge such that the minimum in-flow constraints remain satisfied. The standard decision question from a particular NCL machine and configuration is whether a specified edge can be eventually reversed by a sequence of moves. We can view such a sequence as a nondeterministic computation.

3 种形式的 NCL

Two related decision problems are also of interest:

1. Given two configurations A and B of an NCL machine, is there a sequence of moves from A to B?
2. Given two edges E_A and E_B of an NCL machine, and orientations for each, are there configurations A and B such that E_A has its desired orientation in A, E_B has its desired orientation in B, and there is a sequence of moves from A to B?

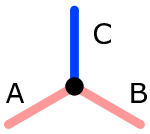
We refer to these three decision problems as *configuration-to-edge*, *configuration-to-configuration*, and *edge-to-edge*. We will show that all of these problems are PSPACE-complete. (The fourth possibility, *edge-to-configuration*, is the same as *configuration-to-edge*, by reversibility.)

2.2 AND/OR Constraint Graphs

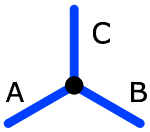
Certain vertex configurations in NCL graphs are of particular interest. A vertex with minimum in-flow constraint 2 and incident edge weights of 1, 1, and 2 behaves as a logical AND, in the following sense: the weight-2 edge may be directed outward if and only if both weight-1 edges are directed inward.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

Similarly, a vertex with minimum in-flow constraint 2 and incident edge weights of 2, 2, and 2 behaves as a logical OR: a given edge may be directed outward if and only if at least one of the other two edges is directed inward. We will call such a vertex an *OR vertex*.

In our reductions we will be concerned primarily with graphs containing only AND and OR vertices; we call such graphs *AND/OR constraint graphs*. [1] When drawing graphs, we will follow the convention that all vertices have in-flow constraint 2, red (light gray) edges have weight 1, and blue (dark gray) edges have weight 2. Figure 2 shows AND and OR vertices.



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



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

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

Circuit Interpretation. With these AND and OR vertex interpretations, it is natural to view an AND/OR graph as a kind of digital logic network, or circuit. One can imagine signals flowing through the graph, as outputs activate when their input conditions are satisfied. This is the picture that motivates our description of NCL as a model of computation, rather than simply as a set of decision problems. Indeed, it is natural to expect (even before our PSPACE-completeness results) that a finite assemblage of such logic gadgets could be used to build a polynomial-space bounded computer.

However, several differences between AND/OR constraint graphs and ordinary digital logic circuits are noteworthy. First, NCL machines are inherently nondeterministic; digital logic circuits are deterministic. Second, with the above AND and OR vertex interpretations, there is nothing to prohibit "wiring" a vertex's "output" (e.g. the weight-2 edge of an AND vertex) to another "output", or an "input" to an "input"; in digital logic circuitry, such connections would be illegal, and meaningless. Finally, although we have AND- and OR-like devices, there is nothing like an inverter (or NOT gate) in NCL; inverters are essential in ordinary digital logic.

This last point deserves some elaboration. The logic that is manifested in NCL graphs is a passive constraint logic; nothing forces an AND vertex, say, to direct its weight-2 edge outward when its other two edges are directed inward. A signal is thus *permitted*, but not required, to flow. For there to be an inverter vertex in NCL would require that an edge be permitted to be directed outward if and only if another edge is not permitted to be directed inward. But there is no way for the vertex to know, so to speak, whether an edge *can* be directed inward; the constraints are in terms of what *is* directed inward.

Flake and Baum require the use of inverters in a similar computational context [4]. They define gadgets ("both" and "either") that are essentially the same as our AND and OR vertices, but rather than use them

[1] We mention without proof that every NCL graph is logspace-reducible to an equivalent AND/OR constraint graph (equivalent with respect to the decision problems). The reduction is rather elaborate, and the result is not needed for any of our main results, so we omit it.

as primitive logical elements, they use their gadgets to construct a kind of dual-rail logic. With this dual-rail logic, they can represent inverters. We do not need inverters for our reductions, so we may omit this step.

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

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

Red-Blue Conversion. Viewing AND/OR graphs as circuits, we might want to connect the output of an OR, say, to an input of an AND. We can't do this directly by joining the loose ends of the two edges, because one edge is blue and the other is red. But we can insert a subgraph that has the desired effect, allowing the AND input edge to activate (point inward) just when the OR output edge is activated (pointing outward). More generally, the subgraph on the left side of Figure 3 effectively copies a signal between a red edge and a blue edge. [2] Edge C permanently satisfies the incident OR vertex's constraint, allowing D to point away from it. This lets E point down, providing an extra in-flow of 1 to the vertex between A and B. Either A or B can now satisfy this vertex's constraint by pointing inward; the other edge is free to point away. In other words, a signal can propagate out from A precisely if it was passed in via B, and vice versa.

This subgraph has an extra red edge (F), whose direction is not constrained; we must somehow deal with its loose end. A little reflection shows that such extra edges must always occur in pairs: red edges only exist on red-red-blue vertices, therefore the sequence F, E, A, must continue on in an unbranching chain, ending at another unattached red edge. We can then identify F with that edge, forming a cycle.

We use the shorthand notation on the right side of Figure 3 to denote this subgraph; this will simplify the figures.

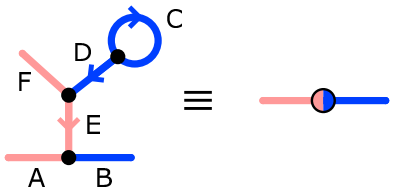


Figure 3: Red-to-blue conversion subgraph, with shorthand notation.

3 PSPACE-completeness

In this section, we show that all three NCL decision problems are PSPACE-complete, and extend these results to apply to simplified forms of NCL.

3.1 Nondeterministic Constraint Logic

We show that configuration-to-edge is PSPACE-hard by giving a reduction from Quantified Boolean Formulas (QBF), which is known to be PSPACE-complete [5], even when the formula is in conjunctive normal form (CNF). A simple argument then shows that configuration-to-edge is in PSPACE, and therefore PSPACE-complete. The PSPACE-completeness of the other two decision problems also follows simply.

[2] Our notion of graph allows loop edges and multiple edges between a single pair of vertices (sometimes called a *multigraph* or *pseudograph*). In Section 3.4 we show how to reduce these graphs to simple graphs (without loops or multiple edges); however, this step is not strictly necessary for our applications.

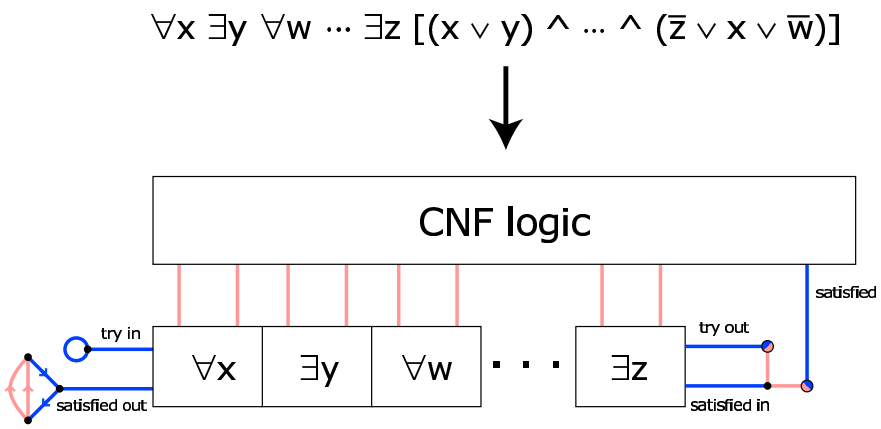


Figure 4: Schematic of the reduction from Quantified Boolean Formulas to NCL.

Reduction. First we give an overview of the reduction and the gadgets we need; then we analyze the gadgets' properties.

The reduction is illustrated schematically in Figure 4. We translate a given quantified Boolean formula ϕ into an AND/OR constraint graph, so that a particular edge in the graph may be reversed if and only if ϕ is true.

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

This is essentially the strategy our reduction shall employ. We define *quantifier gadgets*, which are connected together into a string, one per quantifier in the formula, as in Figure 5(a). Each quantifier gadget outputs a pair of edges corresponding to a variable assignment. These edges feed into the *CNF network*, which corresponds to the unquantified formula. The output from the CNF network connects to the rightmost quantifier gadget; the output of our overall graph is the satisfied out edge from the leftmost quantifier gadget. (We use the attached subgraph for the other decisions problems.)

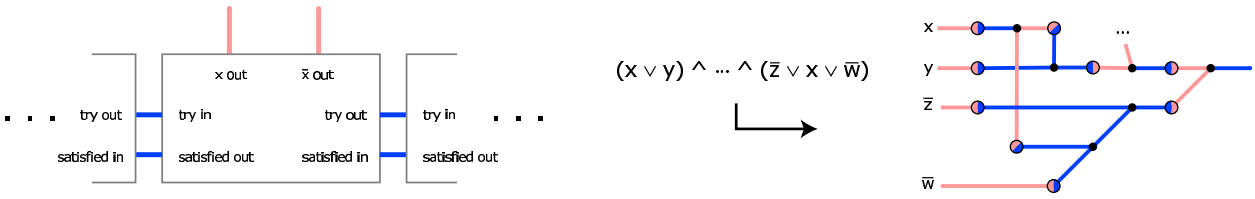
Quantifier Gadgets. When a quantifier gadget is *activated*, all quantifier gadgets to its left have fixed particular variable assignments, and only this quantifier gadget and those to the right are free to change their variable assignments. The activated quantifier gadget can declare itself *satisfied*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

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

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

Lemma 1 *A quantifier gadget's satisfied in edge may not be directed inward unless its try out edge is directed outward.*

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



(a) Quantifier gadget connections

(b) Part of a CNF formula graph

Figure 5: QBF wiring.

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

Because the formula is in conjunctive normal form, and we have edges representing both literal forms of each variable (true and false), we don't need an inverter for this construction. We use the signal-splitting and red-blue conversion techniques described in Section 2.2 to construct the graph. Part of such a graph is shown in Figure 5(b).

Lemma 2 *The satisfied out edge of a CNF subgraph may be directed outward if and only if its corresponding formula is satisfied by the variable assignments on its input edge orientations.*

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

(a) Locked, A active

(b) Unlocked, A active

(c) Unlocked, B active

(d) Locked, B active

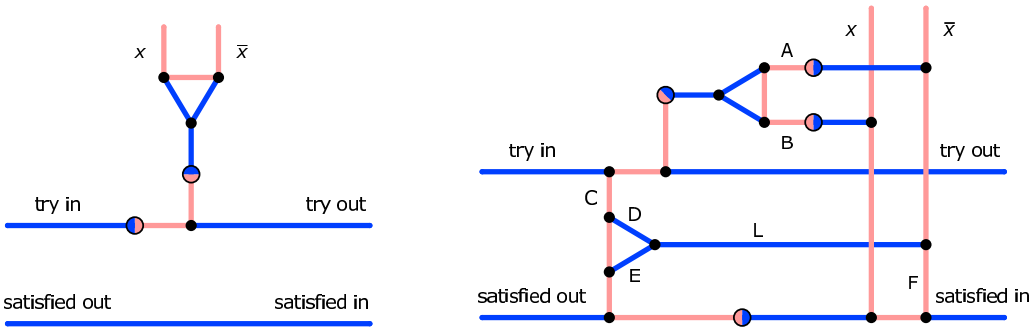
Figure 6: Latch gadget, transitioning from state A to state B.

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

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

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

Universal Quantifier. A universal quantifier gadget is more complicated (Figure 7(b)). It may only direct satisfied out leftward if the formula is true under both variable assignments. Again we use a latch for the variable state; this time we split the variable outputs, so they can be used internally. In addition, we use



(a) Existential quantifier

(b) Universal quantifier

Figure 7: Quantifier gadgets.

a latch internally, as a memory bit to record that one variable assignment has been successfully tried. With this bit set, if the other assignment is then successfully tried, satisfied out is allowed to point out.

Lemma 3 *A universal quantifier gadget may direct its satisfied out edge outward if and only if at one time its satisfied in edge is directed inward while its variable state is locked in the false ($\overline{x}$) assignment, and at a later time the satisfied in edge is directed inward while its variable state is locked in the true (x) assignment, with try in directed inward throughout.*

Proof: First we argue that, with try in directed outward, edge E must point right. The try out edge must be directed inward in this case, so by Lemma 1, satisfied in must be directed outward. As a consequence, F must point down, and thus L must point right. On the other hand, C must point up and thus D must point left. Therefore, E is forced to point right in order to satisfy its Or vertex.

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

We summarize the behavior of both types of quantifiers with the following property:

Lemma 4 *A quantifier gadget may direct its satisfied out edge out if and only if its try in edge is directed in, and the formula read from the corresponding quantifier to the right is true given the variable assignments that are fixed by the quantifier gadgets to the left.*

Proof: By induction. By Lemmas 1 and 3, if a quantifier gadget's satisfied in edge is directed inward and the above condition is inductively assumed, then its satisfied out edge may be directed outward only if the condition is true for this quantifier gadget as well. For the rightmost quantifier gadget, the precondition is explicitly satisfied by Lemma 2 and the construction in Figure 4. □

Theorem 5 *Configuration-to-edge is PSPACE-complete, even when the constraint graph is restricted to an* And/Or *graph.*

Proof: The graph is easily seen to have a legal configuration with the quantifier try in edges all directed leftward. We start with the graph in some such configuration. Because of the blue loop edge attached to the leftmost quantifier's try in edge, we may direct that edge rightward and activate its quantifier. By Lemma 4, the satisfied out edge of the leftmost quantifier gadget may be directed leftward if and only if ϕ is true.

Therefore, deciding whether that edge may reverse also decides the QBF problem, so configuration-to-edge is PSPACE-hard.

Configuration-to-edge is in PSPACE because the state of the constraint graph can be described in a linear number of bits, specifying the direction of each edge, and because the list of possible moves from any state can be computed in polynomial time. Thus we can nondeterministically traverse the state space, at each step nondeterministically choosing a move to make, and maintaining the current state but not the previously visited states. Savitch's Theorem [8] says that this NPSPACE algorithm can be converted into a PSPACE algorithm. □

Corollary 6 *Edge-to-edge and configuration-to-configuration are PSPACE-complete, even when the constraint graph is restricted to an* AND/OR *graph.*

Proof: For edge-to-edge, we use the leftmost try in edge as the input edge; then, as described above, there is a legal initial configuration. Again, we use the leftmost satisfied out edge as the target edge.

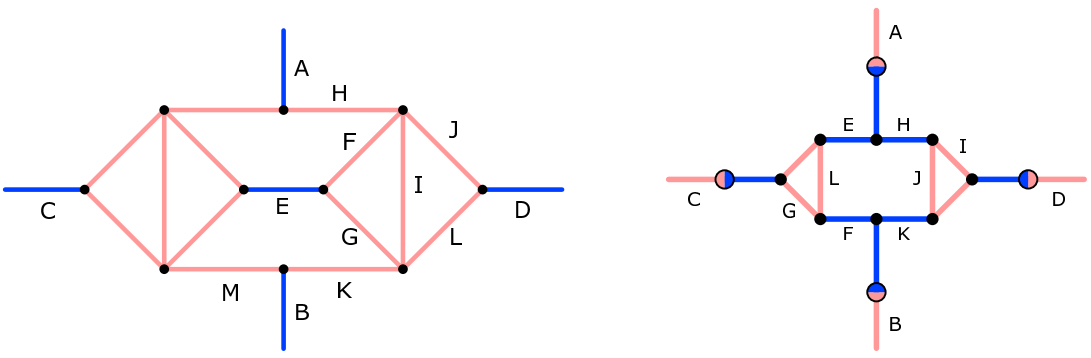
For configuration-to-configuration, we start in some configuration with the leftmost try in edge directed left, and with the four edges attached to the output subgraph in Figure 4 directed as indicated. Then this same configuration, but with those four edges reversed, is reachable if and only if ϕ is true. This subgraph is actually a latch: directing satisfied out left allows those edges to reverse. Then satisfied out can be directed right again, and the entire move sequence can be reversed.

The same algorithm as above serves to show that these tasks are also in PSPACE. □

3.2 Planar Nondeterministic Constraint Logic

The result obtained in the previous section used particular constraint graphs, which turn out to be nonplanar. Thus, reductions from NCL to other problems must provide a way to encode arbitrary graph connections into their particular structure. For 2D motion-planning kinds of problems, such a reduction would typically require some kind of crossover gadget. Crossover gadgets are a common requirement in complexity results for these kinds of problems, and can be among the most difficult gadgets to design. For example, the crossover gadget used in the proof that Sokoban is PSPACE-complete [2] is quite intricate. A crossover gadget is also among those used in the Rush Hour proof [4].

In this section we show that any AND/OR constraint graph can be translated into an equivalent planar AND/OR constraint graph (with respect to the three decision problems), obviating the need for crossover gadgets in reductions from NCL.



(a) Crossover (b) Half-crossover

Figure 8: Planar crossover gadgets.

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

Lemma 7 *In a crossover subgraph, each of the edges A and B may face outward if and only if the other faces inward, and each of the edges C and D may face outward if and only if the other faces inward.*

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

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

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

Next, we must show how to represent the degree-4 vertices in Figure 8(a) with equivalent AND/OR subgraphs. The necessary subgraph is shown in Figure 8(b). Note that this subgraph implicitly contains red-blue conversions subgraphs (Figure 3); we must be careful to keep the graph planar when joining their free red edges. We join these edges in pairs: A's to D's, and B's to C's.

Lemma 8 *In a half-crossover gadget, at least two of the edges A, B, C, and D must face inward; any two may face outward.*

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

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

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

Theorem 9 *Theorem 5 and Corollary 6 remain valid even when the input* AND/OR *constraint graphs are planar.*

Proof: Lemmas 7 and 8. Any crossing edge pairs may be replaced by the above constructions; a crossing edge may be reversed if and only if a corresponding crossover edge (e.g., A or C) may be reversed. For two of the decision problems, we must also specify configurations in the replacement graph corresponding to source or target configurations, but this is easy: pick any legal configuration of the crossover subgraphs with matching crossover edges. □

3.3 Protected OR Graphs

So far we have shown that the decision problems for planar AND/OR constraint graphs are PSPACE-complete. We can make the conditions required for PSPACE-completeness still weaker; this will make our following puzzle reductions simpler.

We call an OR vertex *protected* if there are two of its edges that, due to global constraints, cannot simultaneously be directed inward. Intuitively, graphs with only protected ORs are easier to reduce to

another problem domain, since the corresponding OR gadgets need not function correctly in all the cases that a true OR must. We can simulate an OR vertex with a subgraph all of whose OR vertices are protected, as shown in Figure 9.

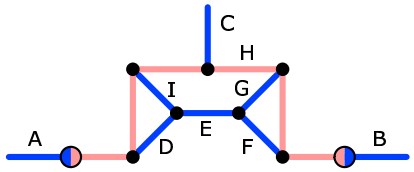


Figure 9: OR vertex made with protected OR vertices.

Lemma 10 *Edges A, B, and C in Figure 9 satisfy the same constraints as an OR vertex; all ORs in this subgraph are protected.*

Proof: Suppose that edges A and B are directed outward. Then D and F must be directed away from E. Assume without loss of generality that E points left. Then so must G; this forces H right and C down, as required. Then, if B points left, the following move sequence is possible (moves on unlabeled edges omitted): I right, E right, G right, H left, F left, E left, I left, C up. Similarly, we can direct A in, and B and C out.

The two OR vertices shown in the subgraph are protected: edges I and D cannot both be directed inward, due to the red edge they both touch; similarly, G and F cannot both be directed inward. The red-blue conversion subgraphs also contain OR vertices, but these are also protected. (We connect the free red edges from the conversion subgraphs together, preserving planarity.) □

Theorem 11 *Theorem 9 remains valid even when all of the* OR *vertices in the input graph are protected.*

Proof: Lemma 10. Any OR vertex may be replaced by the above construction; an OR edge may be reversed if and only if a corresponding subgraph edge (A, B, or C) may be reversed. For two of the decision problems, we must also specify configurations in the replacement graph corresponding to source or target configurations: pick any legal configuration of the subgraphs with matching edges. □

3.4 Nondeterministic Constraint Logic on a Polyhedron

In this section we give a reduction from NCL to a particularly simple geometric form. (This result is presented for its own sake, and is not needed for our further reductions.) We show that any AND/OR constraint graph can be translated into an equivalent simple planar 3-connected graph. Steinitz's Theorem [10, 13] says that a simple planar 3-connected graph is isomorphic to the edges of a convex polyhedron in 3D. Therefore, any NCL problem can be thought of as an edge redirection problem on a convex polyhedron.

(a) Adding a spur to a red edge

(b) Adding a spur to a blue edge

Figure 10: 3-connectivity method.

We use the constructions in Figure 10 to perform the conversion. We may replace any red edge with a subgraph yielding an extra unconstrained blue edge, as shown in Figure 10(a): the original red edge may be reversed in the original graph if and only if the top (equivalently bottom) red edge may be reversed in the new graph. (This is like performing two consecutive red-blue conversions, but with an extra blue edge.) Likewise, we may replace any blue edge with a similar subgraph, as shown in Figure 10(b).

Theorem 12 *Every* AND/OR *constraint graph has an equivalent simple planar 3-connected* AND/OR *graph which can be computed in polynomial time.*

Proof: First, we make the graph planar, by Theorem 9.

Next, we make the graph simple: if any two vertices are joined by more than one edge, we may replace one with a subgraph from Figure 10. We also choose an arbitrary edge on a common face with the replaced edge, and replace this edge as well with such a subgraph, and join the two free blue edges. We eliminate loop edges similarly.

Suppose the resulting graph is not 3-connected. Then there exist zero, one, or two vertices which, if removed, would separate the graph into multiple pieces. From each of two such pieces, choose an edge that lies on a common face, replace these edges with subgraphs from Figure 10, and join the two free blue edges. (This step preserves planarity and simplicity.) Now these pieces will not be separated by the vertex removal. By repeating this process, we may make the graph simple, planar, and 3-connected.

As in Theorem 9, we must also specify a mapping from original to modified graph configurations; again, we simply map orientations of the replaced edges to consistent subgraph configurations. □

4 Applications

In this section, we apply our results from the previous section to various puzzles and motion-planning problems. One result (sliding blocks) is completely new, and provides a tight bound; one result (Rush Hour) reproduces an existing result, with a simpler construction; the last result (Sokoban) strengthens an existing result.

4.1 Sliding Blocks

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

We are interested in the difficulty of this problem, for various allowed integral block sizes. We give a reduction from configuration-to-edge for protected OR graphs showing that Sliding Blocks is PSPACE-hard even when all the blocks are 1×2 rectangles (dominoes). (Somewhat simpler constructions are possible if larger blocks are allowed.) In contrast, there is a simple polynomial-time algorithm for 1×1 blocks; thus, our results are tight.

The *Warehouseman's Problem* [6] is a related problem in which there are no restrictions on block size, and the goal is to achieve a particular total configuration. Its PSPACE-hardness also follows from our result.

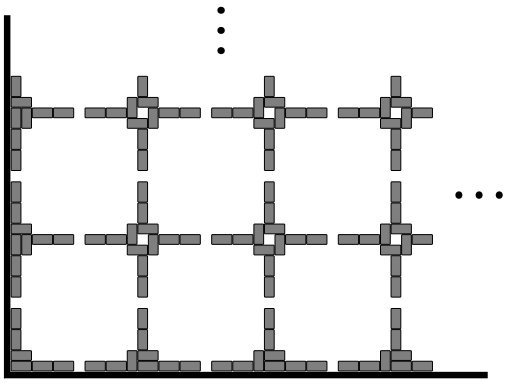
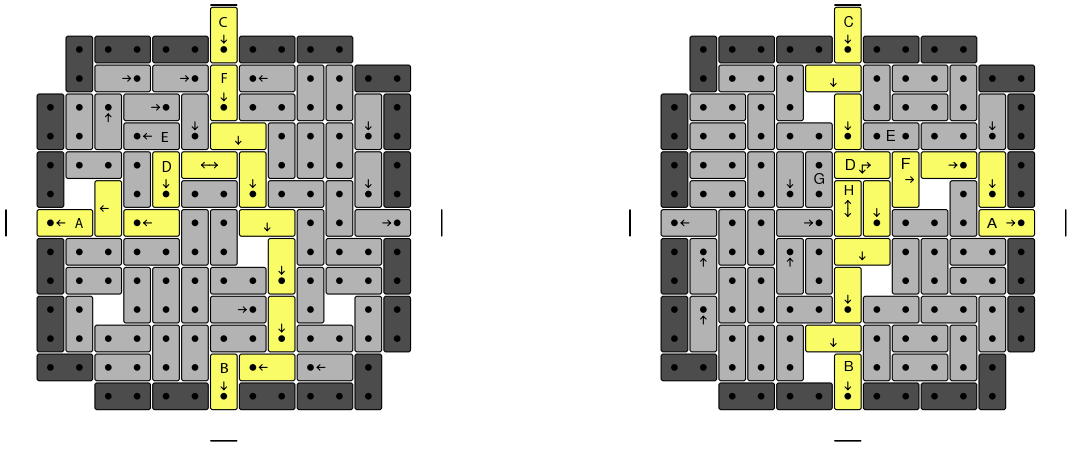


Figure 11: Sliding Blocks layout.

Sliding Blocks Layout. We fill the box with a regular grid of gate gadgets, within a "cell wall" construction as shown in Figure 11. The internal construction of the gates is such that none of the cell-wall blocks may move, thus providing overall integrity to the configuration.



(a) AND (b) Protected OR

Figure 12: Sliding Blocks vertex gadgets.

AND and OR Vertices. We construct NCL AND and protected OR vertex gadgets out of dominoes, in Figures 12(a) and 12(b). Each figure provides the bulk of an inductive proof of its own correctness, in the form of annotations. A dot indicates a square that is always occupied; the arrows indicate the possible positions a block can be in. For example, in Figure 12(b), block D may occupy its initial position, the position one unit to the right, or the position one unit down (but not, as we will see, the position one unit down and one unit right). Because we allow continuous motions, all intermediate block positions are also possible, but this is irrelevant to the vertex properties. We also note that the constructions are such that no block is ever free to rotate.

For each vertex gadget, we show that the annotations are correct, by inductively assuming for each block that its surrounding annotations are correct; its correctness will then follow. The few exceptions are noted below. The annotations were generated by a computer search of all reachable configurations, but are easy to verify by inspection.

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

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

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

Lemma 13 *The construction in Figure 12(a) satisfies the same constraints as an NCL* AND *vertex, with* A *and* B *corresponding to the* AND *red edges, and* C *to the blue edge.*

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

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

The annotations in this case serve as a complete proof of their own correctness, with one exception. Block D appears as though it might be able to slide upward, because block E may slide left, yet D has no upward arrow. However, for E to slide left, F must first slide down, but this requires that D be first be slid down. So when E slides left, D is not in a position to fill the space it vacates.

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

Lemma 14 *The construction in Figure 12(b) satisfies the same constraints as an NCL protected* Or *vertex, with A and B corresponding to the protected edges.*

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

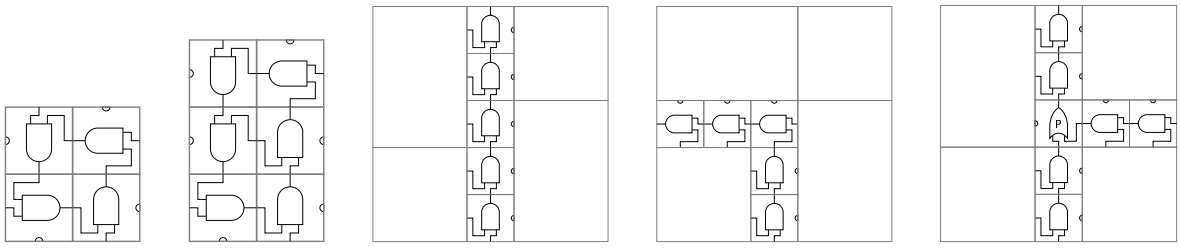
First, observe that these motions are possible. If A moves right, D may move right, releasing the blocks above it. If B moves down, the entire central column may also move down.

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

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

Graphs. Having shown how to make And and protected Or gates out of sliding-blocks configurations, we must now show how to connect them together into arbitrary planar graphs. First, note that the box wall constrains the facing port blocks of the vertices adjacent to it to be retracted (see Figure 11). This does not present a problem, however, as we will show. The unused ports of both the And and protected Or vertices are unconstrained; they may be slid in or out with no effect on the vertices. Figures 13(a) and 13(b) show how to make (2×2)-vertex and (2×3)-vertex "filler" blocks out of Ands. (We use conventional "and" and "or" icons to denote the vertex gadgets.) Because none of the Ands need ever activate, all the exterior ports of these blocks are unconstrained. (The unused ports are drawn as semicircles.)

块拼接
实现转弯和
无交叉与门
（保护）或门



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

Figure 13: Sliding Blocks wiring.

We may use these filler blocks to build (5×5)-vertex blocks corresponding to "straight" and "turn" wiring elements (Figures 13(c) and 13(d)). Because the filler blocks may supply the missing inputs to the Ands, the "output" of one of these blocks may activate (slide in) if and only if the "input" is active (slid out). Also, we may "wrap" the And and protected Or vertices in 5×5 "shells", as shown for protected Or in Figure 13(e). (Note that "left turn" is the same as "right turn"; switching the roles of input and output results in the same constraints.)

We use these 5×5 blocks to fill the layout; we may line the edges of the layout with unconstrained ports. The straight and turn blocks provide the necessary flexibility to construct any planar graph, by letting us extend the vertex edges around the layout as needed.

Theorem 15 *Sliding Blocks is PSPACE-hard, even for* 1×2 *blocks.*

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

Corollary 16 *The Warehouseman's Problem is PSPACE-hard, even for* 1×2 *blocks.*

Proof: As above, but using configuration-to-configuration instead of configuration-to-edge. The NCL graph initial and desired configurations correspond to two block configurations; the second is reachable from the first if and only if the NCL problem has a solution. □

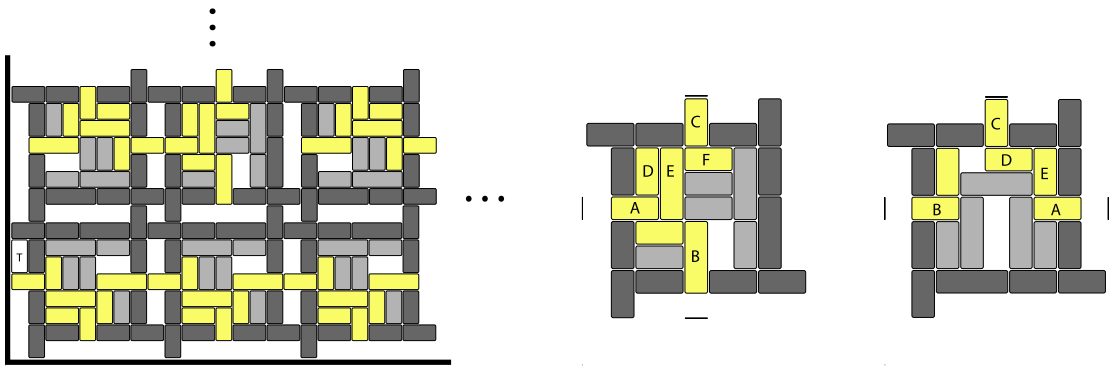
If we restrict the block motions to unit translations, then these problems are also in PSPACE, as in Theorem 5.

4.2 Rush Hour

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

Flake and Baum [4] showed that the generalized problem is PSPACE-complete, by showing how to build a kind of reversible computer from Rush Hour gadgets that work like our And and Or vertices, as well as a crossover gadget. Tromp [11] strengthened their result by showing that Rush Hour is PSPACE-complete even if the blocks are all 1×2.

Here we give a simpler construction showing that Rush Hour is PSPACE-complete, again using the traditional 1×2 and 1×3 blocks which must slide lengthwise. We only need an And and a protected Or, which turns out to be easier to build than Or; because of our generic crossover construction (Section 3.2), we don't need a crossover gadget. (We also don't need the miscellaneous wiring gadgets used in [4].)



(a) Layout (b) And (c) Protected Or

Figure 14: Rush Hour layout and vertex gadgets.

Rush Hour Layout. We tile the grid with our vertex gadgets, as shown in Figure 14(a). One block (T) is the target, which must be moved to the bottom left corner; it is released when a particular port block slides into a vertex.

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

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

Lemma 17 *The construction in Figure 14(b) satisfies the same constraints as an NCL* AND *vertex, with A and B corresponding to the* AND *red edges, and C to the blue edge.*

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

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

Lemma 18 *The construction in Figure 14(c) satisfies the same constraints as an NCL protected* OR *vertex, with A and B corresponding to the protected edges.*

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

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

Graphs. We may use the same constructions here we used for sliding-blocks layouts: 5×5 blocks of Rush Hour vertex gadgets serve to build all the wiring necessary to construct arbitrary planar graphs (Figure 13).

In the special case of arranging for the target block to reach its destination, this will not quite suffice; however, we may direct the relevant signal to the bottom left of the grid, and then remove the bottom two rows of vertices from the bottommost 5×5 blocks; these can have no effect on the graph. The resulting configuration, shown in Figure 14(a), allows the target block to be released properly.

Theorem 19 *Rush Hour is PSPACE-complete.*

Proof: Reduction from configuration-to-edge for planar protected OR graphs, by the construction described. The output port block of a particular vertex may move if and only if the corresponding NCL graph edge may be reversed. We direct this signal to the lower left of the grid, where it may release the target block.

Rush Hour is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 5. □

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

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

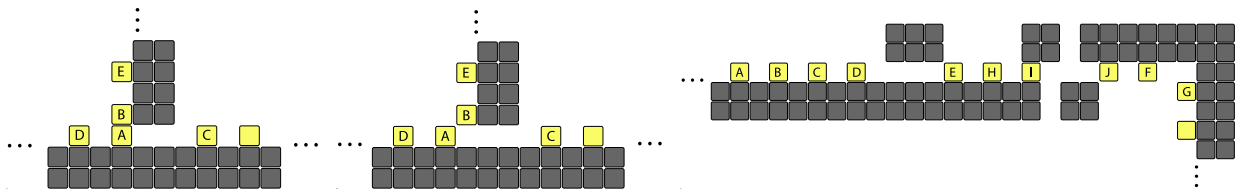
Therefore, a straightforward application of our technique cannot show this problem hard; however, the complexity of moving a given block to a given position is not obvious. Tromp and Cilibrasi [12] provide some empirical indications that minimum-length solutions for 1×1 Rush Hour may grow exponentially with puzzle size.

4.3 Sokoban

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

Culberson [2] proved that Sokoban is PSPACE-complete, by showing how to construct a Sokoban position corresponding to a space-bounded Turing machine. Using NCL, we give an alternate proof. Our result applies even if there are no barriers allowed in the Sokoban position, thus strengthening Culberson's result.

Unrecoverable Configurations. The idea of an *unrecoverable configuration* is central to Culberson's proof, and it will be central to our proof as well. We construct our Sokoban instance so that if the puzzle is solvable, then the original configuration may be restored from any solved state by reversing all the pushes. Then any push which may not be reversed leads to an unrecoverable configuration. For example, in the partial configuration in Figure 15(a), if block A is pushed left, it will be irretrievably stuck next to block D; there is no way to position the pusher so as to move it again. We may speak of such a move as being prohibited, or impossible, in the sense that no solution to the puzzle can include such a move, even though it is technically legal.



(a) AND (b) OR (c) Utility gadgets

Figure 15: Sokoban gadgets.

AND and OR Vertices. We construct NCL AND and OR vertex gadgets out of partial Sokoban positions, in Figure 15. (The pusher is not shown.) The dark-colored blocks in the figures, though unmovable, are not barriers; they are simply blocks that cannot be moved by the pusher because of their configuration. The light-colored ("trigger") blocks are the ones whose motion serves to satisfy the vertex constraints. In each vertex, blocks A and B represent outward-directed edges; block C represents an inward-directed edge. A and C switch state by moving left one unit; B switches state by moving up one unit. We assume that the pusher may freely move to any empty space surrounding a vertex. We also assume that block D in Figure 15(a) may not reversibly move left more than one unit. Later, we show how to arrange both of these conditions.

Lemma 20 *The construction in Figure 15(a) satisfies the same constraints as an NCL* AND *vertex, with A and B corresponding to the* AND *red edges, and C to the blue edge.*

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

Lemma 21 *The construction in Figure 15(b) satisfies the same constraints as an NCL* OR *vertex.*

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

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

However, if we push C left without first pushing A left or B up, then, as in Lemma 20, we will be left in an unrecoverable configuration. □

Graphs. We have shown how to make AND and OR vertices, but we must still show how to connect them up into arbitrary planar graphs. The remaining gadgets we shall need are illustrated in Figure 15(c).

The basic idea is to connect the vertices together with alternating sequences of blocks placed against a double-thick wall, as in the left of Figure 15(c). Observe that for block A to move right, first D must move right, then C, then B, then finally A, otherwise two blocks will wind up stuck together. Then, to move block D left again, the reverse sequence must occur. Such movement sequences serve to propagate activation from one vertex to the next.

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

We satisfy the requirement that block D in Figure 15(a) may not reversibly move left more than one unit by protecting the corresponding edge of every AND with a turn; observe that in Figure 15(c), block F may not reversibly move right more than one unit. The flip gadget solves our one remaining problem: how to position the pusher freely wherever it is needed. Observe that it is always possible for the pusher to cross a string through a flip gadget. (After moving J right, we may actually move I *three* spaces right.) If we simply place at least one flip along each wire, then the pusher can get to any side of any vertex.

Theorem 22 *Sokoban is PSPACE-complete, even if no barriers are allowed.*

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

Sokoban is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 5. □

5 Alternative Formulation

In this section we give an alternative formulation of the NCL problem, *Sliding Tokens*, in terms of tokens sliding along graph edges. This formulation is even simpler than the edge-reversal formulation in Section 2. However, it lacks the inherent computational flavor of AND/OR constraint graphs; furthermore, it seems to be less suitable for reductions. Therefore, we have chosen to use AND/OR constraint graphs as our primary formulation; this section may be viewed as an additional application.

Our "machine" in this case is a graph. A configuration of a machine is a subset of its vertices containing tokens, such that no two tokens are adjacent along an edge. A move from one configuration to another is made by moving a token from one vertex to an adjacent one, resulting in a valid configuration. The decision question is whether a given token can eventually be moved by a sequence of moves.

We give a reduction from configuration-to-edge for AND/OR graphs showing that this problem is PSPACE-complete.

(a) AND (b) OR

Figure 16: Sliding Tokens vertex gadgets.

AND/OR Graphs. We construct NCL AND and OR vertex gadgets out of sliding-token subgraphs, in Figures 16(a) and 16(b). The edges that cross the dotted-line gadget borders are "port" edges. A token on an outer port-edge vertex represents an inward-directed NCL edge, and vice-versa. Given an AND/OR graph and configuration, we construct a corresponding sliding-token graph, by joining together AND and OR vertex gadgets at their shared port edges, placing the port tokens appropriately.

Theorem 23 *Sliding Tokens is PSPACE-complete.*

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

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

Sliding Tokens is in PSPACE: a simple nondeterministic algorithm traverses the state space, as in Theorem 5. □

Discussion. This problem formulation is interesting for several reasons. It is a dynamic version of the Independent Set problem, which is NP-complete [5]. Similarly, the natural two-player-game version of Independent Set, called Kayles, is also PSPACE-complete [5]. Just as many NP-complete problems become PSPACE-complete when turned into two-player games [9], it is also natural to expect that they become PSPACE-complete when turned into dynamic puzzles.

From a recreational standpoint, sliding-token graphs are similar both to sliding-coin puzzles and to 1×1 sliding-block puzzles, many forms of which are in P [3]. Typically one needs some structure of the pieces beyond an atomic token or 1×1 block to add complexity to a motion-planning puzzle. In this case, however, the nonadjacency requirement suffices.

Computationally, sliding-token graphs also superficially resemble Petri nets.

6 Conclusion

We proved that one of the simplest possible forms of motion planning, sliding 1×2 blocks (dominoes) around in a box, is PSPACE-hard. This result is a major strengthening of previous results. The problem has no artificial constraints, such as the movement restrictions of Rush Hour; it has object size constraints which are tightly bounded, unlike the unbounded object sizes in the Warehouseman's Problem. Also compared to the Warehouseman's Problem, the task is simply to move a block at all, rather than to reach a total configuration.

Along the way, we presented a model of computation of interest in its own right, and which can be used to prove several motion-planning problems to be PSPACE-hard. Our hope is to apply this approach to several other motion-planning problems whose complexity remain open, for example:

1. 1×1 **Rush Hour.** While 1×1 sliding blocks can be solved in polynomial time, if we enforce horizontal or vertical motion constraints as in Rush Hour, does the problem become PSPACE-complete [12]? Deciding whether a block may move at all is in P, so a straightforward application of our technique will not work, but what is the complexity of moving a given block to a given position?

2. **Retrograde Chess.** Given two configurations of chess pieces in a generalized $n \times n$ board, is it possible to play from one configuration to the other if the players cooperate? This problem is known to be NP-hard [1]; is it PSPACE-complete?

Acknowledgments

We thank John Tromp, Shafira Goldwasser, and Albert Meyer for several useful comments and suggestions. We also thank Timothy Chow for introducing us to the Retrograde Chess problem mentioned in the conclusion, and for pointing out the reference [1].

References

[1] Hans Bodlaender. Re: Is retrograde chess NP-hard? Usenet posting to `rec.games.abstract`, March 16 2001.

[2] Joseph Culberson. Sokoban is PSPACE-complete. In *Proceedings of the International Conference on Fun with Algorithms*, pages 65–76, Elba, Italy, June 1998.

[3] Erik D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In Jiří Sgall, Aleš Pultr, and Petr Kolman, editors, *Proceedings of the 26th Symposium on Mathematical Foundations in Computer Science (MFCS 2001)*, volume 2136 of *Lecture Notes in Computer Science*, pages 18–32, Marianske Lazne, Czech Republic, August 27–31 2001.

[4] Gary William Flake and Eric B. Baum. *Rush Hour* is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1–2):895–911, January 2002.

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

[6] J. E. Hopcroft, J. T. Schwartz, and M. Sharir. On the complexity of motion planning for multiple independent objects: PSPACE-hardness of the 'Warehouseman's Problem'. *International Journal of Robotics Research*, 3(4):76–88, 1984.

[7] Edward Hordern. *Sliding Piece Puzzles*. Oxford University Press, 1986.

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

[9] Thomas J. Schaefer. Complexity of decision problems based on finite two-person perfect-information games. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 41–49, Hershey, PA, 1976. ACM Press.

[10] Ernst Steinitz and Hans Rademacher. *Vorlesungen über die Theorie der Polyeder*. Springer-Verlag, Berlin, 1934. Reprinted 1976.

[11] John Tromp. On size 2 Rush Hour Logic. Manuscript, December 2000. `http://turing.wins.uva.nl/~peter/teaching/tromprh.ps`.

[12] John Tromp and Rudy Cilibrasi. Limits of Rush Hour Logic complexity. Manuscript, June 2004. `http://www.cwi.nl/~tromp/rh.ps`.

[13] Günter M. Ziegler. *Lectures on Polytopes*, lecture 4, pages 103–126. Graduate Texts in Mathematics. Springer-Verlag, New York, 1995.