The Nondeterministic Constraint Logic Model of Computation: Reductions and Applications

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 [3].

We illustrate the importance of this model of computation by giving simple reductions to show that various 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 [3], of which we also give a simpler proof. 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* [6], 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.

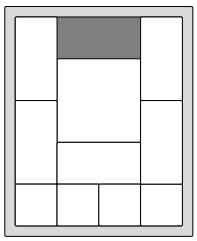


Figure 1: The Donkey Puzzle: move the large square to the bottom center.

The *Warehouseman's Problem* [5] 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 [5] 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.

Although not mentioned in [5], 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,

*Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 200 Technology Square, Cambridge, MA 02139, U.S.A., rah@ai.mit.edu

†MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, U.S.A., edemaine@mit.edu

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 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 [3], 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 [3], 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, and that is the focus of this paper. 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, formulated in terms of both graphs and circuits. Section 3 proves increasingly simple formulations of NCL to be PSPACE-complete. Section 4 proves various motion-planning problems to be PSPACE-hard using the restricted forms of our model of computation. Section 5 shows that some related optimization and puzzle-design variations are still in PSPACE. Appendix A gives an additional simple formulation of NCL.

2 Nondeterministic Constraint Logic

In this section we formally define the nondeterministic constraint logic (NCL) model of computation, and give two main equivalent formulations: one in terms of directed graphs, and the other in terms of gates and circuits.

2.1 Graph Formulation

The simplest description of NCL arises as reversal of edges in a directed graph. A "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.

Equivalent Forms. A constraint graph G_2 is an *equivalent form* of constraint graph G_1 if every configuration of G_1 can be reached if and only if a corresponding configuration of G_2 can be reached, and the configuration map preserves identity of non-loop edges: that is, every non-loop edge in G_1 may be assigned an edge in G_2 such that reversing one always corresponds to reversing the other.

This definition captures all the properties we will need for our reductions. Note that any loop edge can trivially be reversed; see, e.g., Figure 2.

Normal Form. We say that a constraint graph is in *normal form* if all edge weights are 1 or 2, all minimum in-flow constraints are 2, and all vertices have degree 3. This is the form of NCL that we shall be primarily concerned with. In all graph diagrams, we adopt the convention that red (light gray) edges have weight 1, blue (dark gray) edges have weight 2, and unlabeled vertices have a minimum in-flow constraint of 2.

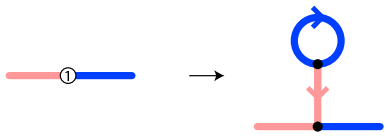


Figure 2: Red to blue conversion, in unrestricted and normal forms.

Figure 2 shows one translation to normal form that we will use frequently. A degree-2 vertex with a red edge (weight 1) and a blue edge (weight 2) and a minimum in-flow constraint of 1 serves to convert a "red signal" into a "blue signal", and vice-versa; one edge may be directed out if and only if the other is directed in. By attaching a spur with a free red edge (the blue edge satisfies the upper vertex's constraint), we convert the construction to normal form.

2.2 Circuit Formulation

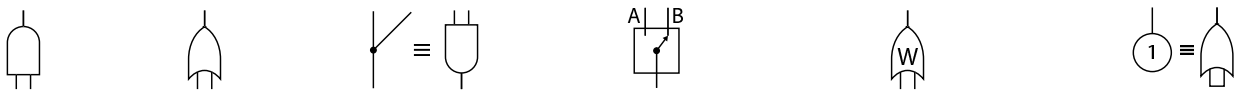
In this section and the next, we show that a constraint graph may be viewed as a kind of circuit made up of various kinds of logic gates wired together. The circuit model is useful for visualizing how some graphs work, and is also useful for reductions to various other problems. We give three different sets of primitive gates, any one of which is sufficient to represent any constraint graph.

Gates. A *gate* is an object with a set of *ports* (each of which is either an *input* or an *output*), possibly an internal state, and a set of constraints relating the port states and the internal state. A port may be either *active* or *inactive*.

Circuits. A *circuit* is a collection of gates togther with a one-to-one pairing of all of their ports. The pairs are called *wires*. We do not require that wires connect inputs to outputs; in fact, much of the special character of NCL circuits results from constraints induced by wiring inputs to inputs or outputs to outputs. Actually, the input/output labeling is not necessary, and merely serves to place the gates in a familiar digital logic context.

We require consistency of ports connected by wires, as follows: (1) an inactive output may not be connected to an active input, (2) two active inputs may not be connected, and (3) two inactive outputs may not be connected. That is, the input/output distinction has the effect of reversing the notions of active and inactive; indeed, this is the only effect of the input/output labeling.

We give circuits a "kinematics" by allowing any sequence of individual changes to port or internal gate states consistent with the constraints. We do not, however, give circuits a "dynamics"; a circuit's state evolution is nondeterministic.



(a) AND (b) OR (c) SPLIT (d) LATCH (e) WEAK OR (f) 1

Figure 3: Gates. Inputs are at bottom; outputs are at top.

AND Gate. An AND gate (Figure 3(a)) is a gate with two inputs and one output, and the constraint that the output may be active only if the inputs are both active.

Or Gate. An Or gate (Figure 3(b)) is a gate with two inputs and one output, and the constraint that the output may be active only if at least one input is active.

Split Gate. A Split (Figure 3(c)) is a gate with one input and two outputs, and the constraint that the outputs may be active only if the input is active. Because Split is not a symmetric gate, we are careful to distinguish the input side by drawing the outputs at a 45° angle.

In fact, Split is equivalent to And with the inputs and outputs reversed. That is, if we replace an And in a circuit by a Split, using the Split outputs in place of the And inputs and vice-versa, then the Split imposes the same constraints on the surrounding circuit behavior as the And gate did. However, we will often use Splits in circuit diagrams, to indicate the normal direction of information flow.

Latch Gate. A Latch (Figure 3(d)) is a gate with one input, two outputs, and one Boolean internal state variable. The internal state can change only while the input is active. One output (A) may be active only if the input is active or the internal state is false; and the other output (B) may be active only if the input is active or the internal state is true.

As it turns out, a Latch is often easier to construct than an Or gate as a gadget used in a reduction from NCL, and we will show that it is just as useful.

A latch may be viewed as a limited kind of Or, as follows. If we reverse the input and output labels, then activating either input is necessary and sufficient to activate the output. However, the behavior is not quite the same as that of a true Or gate, because the following sequence of operations is not permitted: A activates, internal state becomes true, output activates, B activates, A deactivates. The internal state would have to change to allow this, but it is locked as long as the output is active.

Weak Or Gate. A Weak Or gate (Figure 3(e)) is identical to an Or gate, except that we require that any circuit containing a Weak Or must make it impossible for both inputs to be active at once.

Like Latch, we will show that Weak Or is just as useful as Or; it is often easier to construct for reductions, because a Weak Or gadget built out of something else (such as sliding blocks) need not function correctly in all the cases an Or must.

1 Gate. A 1 gate (Figure 3(f)) has a single output, which is unconstrained (and thus may serve to supply an active input to another gate). This is merely a shorthand for an Or with the inputs wired together.

Inverter Gate. Although it is not needed for our construction, we point out for comparison that it is impossible to make an inverter, that is, a gate whose output is active exactly when its input is inactive. In the first place, if we had such a gate, our circuit kinematics would not permit it to operate, because state changes must happen one at a time, and we would need both ports to change at once. More importantly, the idea of an inverter does not map onto the passive nature of NCL. In all the other gates, outputs are *permitted*, but not required, to change state when their constraints are satisfied.

The approach in [3] requires inverters in a similar computational context, and Flake and Baum show how to construct inverters by using a kind of dual-rail logic. However, our reductions have no need of inverters, so we may omit this step, and view individual wires as representing our logic values.

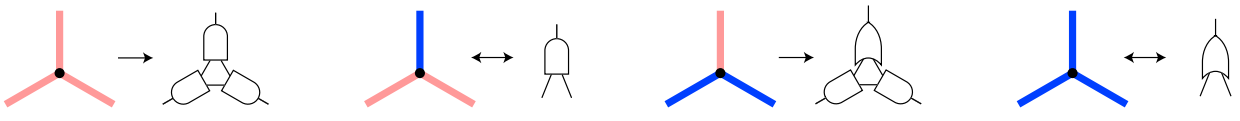
2.3 Universal Gate Sets

Here we show that circuits made with And and Or gates are equivalent to constraint graphs: graphs and And/Or circuits are merely two different languages for describing the same computational processes.

We define equivalence as for constraint graphs, substituting "port" for "edge" where appropriate.

Lemma 1 *Every normal-form constraint graph has an equivalent* And *and* Or *circuit which can be computed in polynomial time.*

Proof: In a normal-form graph, there can be only four kinds of vertices: red-red-red, red-red-blue, red-blue-blue, and blue-blue-blue. Each of these has an equivalent subcircuit, shown in Figure 4. We map



(a) red-red-red (b) red-red-blue (c) red-blue-blue (d) blue-blue-blue

Figure 4: Converting normal-form vertices into AND/OR subcircuits.

inward-directed edges to active inputs or inactive outputs, and outward-directed edges to inactive inputs or active outputs. The subcircuits are equivalent in that they obey the same constraints as the graph vertices.

For 4(a), note that any one AND may be active, but that forces one input of each other AND, and thus their outputs, to be inactive. This is the same as the constraint for red-red-red vertices: at most one edge may be outward-directed, because the minimum in-flow constraint is 2.

For 4(b), either both red edges must be directed inward (inputs active) or the blue one must (output inactive); these are the same constraints as an AND gate (or a SPLIT).

For 4(c), observe that the red edge is unconstrained, and one blue edge must be inward-directed. Likewise, both AND gates cannot be active at once; the inactive one may provide the necessary input to activate the OR.

For 4(d), a particular blue edge may be outward directed if and only if one of the other two is inward-directed; these are the same constraints as an OR gate. OR is thus revealed to be an inherently symmetric gate; any two ports may be used as inputs.

If we wire all the vertices' corresponding subcircuits together as the edges connect the vertices, then the wire constraints ensure that an edge may be redirected if and only if a corresponding gate port may reverse its state. (An edge actually maps to a pair of ports; either may be chosen as the corresponding port.) □

Lemma 2 *Every* AND *and* OR *circuit has an equivalent normal-form constraint graph which can be computed in polynomial time.*

Proof: The conversions shown in Figures 4(b) and 4(d) also serve to turn AND and OR gates into equivalent subgraphs. We connect these subgraphs together by joining edges as in Figure 2; this same red edge-blue loop spur technique can also be used to connect two blue edges. (We do this, rather than identify corresponding subgraph edges, to ensure that each port maps to a unique edge.)

A given port may then change state if and only if its corresponding edge may reverse. □

Lemmas 1 and 2 show that AND and OR are universal gates. As we will show in Section 3.1, this means that we may show a problem to be PSPACE-hard by showing how to construct an AND and OR circuit as an instance of the problem. In comparison, our AND and OR gates have essentially the same properties as the "both" and "either" gates in [3], but their Generalized Rush Hour Logic requires additional machinery to build Boolean "and" and "or" operations because of their use of dual-rail logic. Furthermore, here we show that two other sets of gates, which are often easier to construct, work just as well. In Section 4, we show three different problems PSPACE-hard; in each one, a different set of gates proves most convenient.

(a) LATCH built from AND and WEAK OR (b) OR built from SPLIT and LATCH

Figure 5: Gate emulations. Inputs are at bottom; outputs are at top.

Lemma 3 LATCH *may be emulated with* AND *and* WEAK OR.

Proof: We show the construction in Figure 5(a). First, note that the WEAK OR circuit constraint is satisfied: for both inputs to be active, both AND outputs would have to be active; because of the wired-together inputs, however, one of them must be inactive. The ANDs thus provide the required input protection to the WEAK OR.

Suppose the WEAK OR output is active. Then one of the ANDs must be active, and thus one of the subcircuit outputs must be inactive. If the WEAK OR output is inactive, both ANDs may be inactive, thus both subcircuit outputs may be active; either AND may activate before the WEAK OR output activates. These are the same constraints LATCH has, treating the WEAK OR output as the LATCH input, and the free AND inputs as the LATCH outputs, and remembering to appropriately reverse the sense of activation. □

Lemma 4 WEAK OR *may be emulated with* OR.

Proof: Definition of WEAK OR and OR. □

Lemma 5 OR *may be emulated with* SPLIT *and* LATCH.

Proof: The construction is shown in Figure 5(b). Observe that if the inputs are inactive, the output must also be inactive. The SPLIT must be inactive, so both of LATCH 1's outputs must be active; its input must therefore also be active, so LATCH 2's input must be inactive, and its state must be locked. Port A is active (because the SPLIT is inactive), so port B must be inactive. If either input activates, LATCH 2's input may activate, unlocking it, and allowing B to activate. Suppose the second input activates. Then both LATCHes are free to change state. LATCH 2 must lock its state in order to deactivate its input and release LATCH 1's state, but it may leave port B active while doing do, and then change state arbitrarily after unlocking again. Thus, after one input activates, and B activates, and the other input activates, the device may enter the state that would have resulted from the inputs activating in the reverse order. Therefore, any one input may deactivate with B remaining active. These are the same constraints OR has. □

We summarize all of these results with the following theorem, recalling that AND is equivalent to SPLIT:

Theorem 6 *The following are polynomial-time equivalent: normal-form constraint graphs,* AND/OR *circuits,* AND/LATCH *circuits, and* AND/WEAK OR *circuits.*

Proof: Lemmas 1, 2, 3, 4, and 5. □

Corollary 7 *The following are polynomial-time equivalent: planar normal-form constraint graphs, planar* AND/OR *circuits, planar* AND/LATCH *circuits, and planar* AND/WEAK OR *circuits.*

Proof: All of the relevant reductions use planar subcircuits and subgraphs. □

3 PSPACE-completeness

In this section, we show that NCL is PSPACE-complete, and provide reductions showing that some simplified forms of NCL are also PSPACE-complete.

3.1 Nondeterministic Constraint Logic

We show that NCL is PSPACE-hard by giving a reduction from Quantified Boolean Formulas (QBF), which is known to be PSPACE-complete [4], even when the formula is in conjunctive normal form. A simple argument then shows that NCL is in PSPACE, and therefore PSPACE-complete.

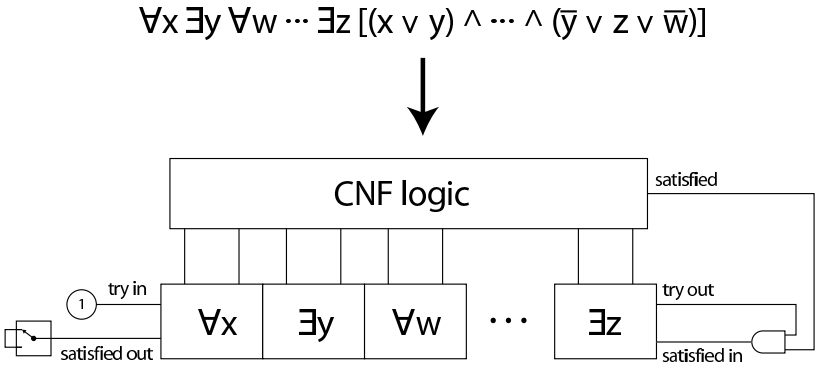


Figure 6: Schematic of the reduction from Quantified Boolean Formulas to NCL.

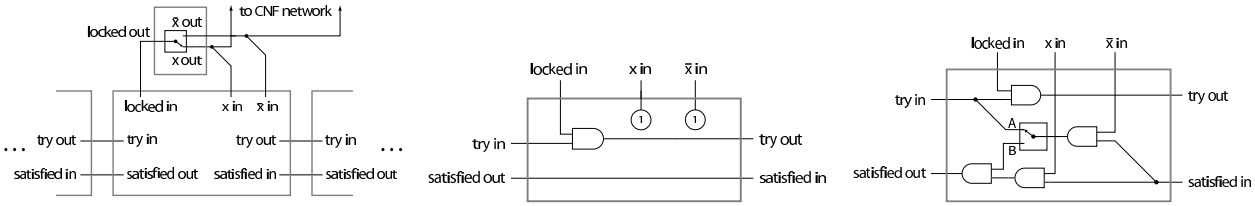
Reduction. First we give an overview of the reduction and the gadgets we need; then we analyze the gadgets' properties.

We use the circuit form of NCL for the reduction. The reduction is illustrated schematically in Figure 6. We translate a given quantified Boolean formula ϕ into an instance of NCL, so that a particular gate in the resulting circuit may be activated 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 *variable gadgets* and *quantifier gadgets* (Figure 7). The quantifier gadgets are connected together into a string, one per quantifier in the formula. Each quantifier gadget is connected to its own variable gadget. The variable gadgets 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 circuit is the satisfied out port from the leftmost quantifier gadget. (We use the attached LATCH to show a related result.)

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) Variable and connections (b) Existential quantifier (c) Universal quantifier

Figure 7: QBF reduction gadgets.

Variable Gadget. A variable gadget (shown in Figure 7(a) connected to a quantifier) is simply a LATCH, with the input port used to lock or release the variable state, and the output ports used to indicate that the variable is either true or false. While the input port is inactive, the internal state is locked, and one of the output ports is forced inactive; the other is unconstrained, and represents the current variable assignment.

With the input port active, the internal state is free to change, and the output ports are unconstrained. Note that the LATCH input port serves as the variable locked out port. This input/output switch reverses the sense of activation: for locked out to activate, the LATCH input must be inactive, locking it.

Quantifier Gadgets. A quantifier gadget is activated by activating its try in port. Its try out port is enabled to activate only if try in is active, and its variable gadget is locked. Thus, 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. For satisfied out to activate, indicating that the formula from this quantifier on is currently satisfied, we require (at least) that satisfied in is active.

We need both existential and universal quantifier gadgets, described below.

CNF Formula. In order to evaluate the formula for a particular variable assignment, we construct an AND and OR network corresponding to the unquantified part of the formula, fed inputs from the variable gadgets, and feeding into the satisfied in port of the rightmost quantifier gadget, as in Figure 6. The satisfied in port of the rightmost quantifier gadget is further protected by an AND gate, so it may activate only if try out is active and the formula is currently satisfied.

Because the formula is in conjunctive normal form, and we have wires representing both literal forms of each variable (true and false), we don't need an INVERTER gate for this construction.

Lemma 8 *A quantifier gadget's satisfied in port may not activate unless its try out port is active.*

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 active before try out activates, and requires satisfied in to be active before satisfied out activates. □

Existential Quantifier. For an existential quantifier gadget (Figure 7(b)) we use the basic circuitry required to meet the definition of a quantifier gadget; we leave the variable ports unconstrained by connecting them to 1 gates. If the formula is true under some assignment of an existentially quantified variable, then its quantifier gadget may lock the variable gadget in the corresponding state, and recursively receive the satisfied in signal, releasing its satisfied out port. Here we exploit the nondeterminism in the model to choose between true and false.

Lemma 9 *An existential quantifier gadget may activate its satisfied out port if and only if its satisfied in port is active while its variable gadget is locked in some assignment.*

Proof: By Lemma 8 and the definition of the existential quantifier gadget. □

Universal Quantifier. A universal quantifier gadget is more complicated (Figure 7(c)). It may only enable satisfied out if the formula is true under both variable assignments. We use a LATCH as a memory bit to record that one variable assignment has been successfully tried, and then enable satisfied out only if the memory bit so indicates, and the other variable assignment is currently satisfied. To ensure that the memory bit resets before the quantifier gadget deactivates, the other LATCH state is constrained to be active when try in is inactive. To enforce this constraint, a SPLIT output is connected to a LATCH output (A); recall that connecting two outputs has the effect of forcing one of them to be active. When try in is inactive, the SPLIT output cannot be active, so A must be.

Lemma 10 *A universal quantifier gadget may activate its satisfied out port if and only if its satisfied in port is at one time active while its variable gadget is locked in the false ($\overline{x}$) assignment, and at a later time is again active while its variable gadget is locked in the true (x) assignment, with try in remaining active throughout.*

Proof: Suppose that the variable is locked in the false state (the $\overline{x}$ in port is active), and that satisfied in activates. These two conditions allow the LATCH input to activate. The LATCH may then change internal state and deactivate, leaving output A inactive and output B active. Now, all the activations that occurred after activating try in and before activating the LATCH may be reversed, because none of them are constrained by the LATCH state. 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 (the x in port is active). This condition, along with the LATCH B output, is both necessary and sufficient to activate satisfied out. Because of the constraint between the try in port and A, the LATCH must have been set to the B state since try in was last enabled. □

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

Lemma 11 *A quantifier gadget may activate its satisfied out port if and only if its try in port is active, 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 9 and 10, if a quantifier gadget's satisfied in port activates and the above condition is inductively assumed, then its satisfied out port may activate only if the condition is true for this quantifier gadget as well. For the rightmost quantifier gadget, the precondition is explicitly satisfied by the construction. □

Theorem 12 *NCL is PSPACE-complete.*

Proof: By Lemma 11, the satisfied out port of the leftmost quantifier gadget may activate if and only if ϕ is true. This port corresponds to an edge in a constraint graph; therefore, deciding whether that edge may reverse also decides the QBF problem, so NCL is PSPACE-hard.

NCL is in PSPACE because the state of the constraint graph can be described in a linear number of bits, specifying the direction of each edge, and 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 [7] says that this NPSPACE algorithm can be converted into a PSPACE algorithm. □

Corollary 13 *Deciding whether a specified configuration of an NCL graph is reachable is PSPACE-complete.*

Proof: To show this we use the LATCH connected to the leftmost satisfied out port. We may activate the LATCH if and only if the satisfied out port may be activated. After doing so, we may change the LATCH state, and undo all the other moves that were made. Thus, the configuration which is the same as the initial configuration, but with the LATCH state switched, is reachable if and only if ϕ is true.

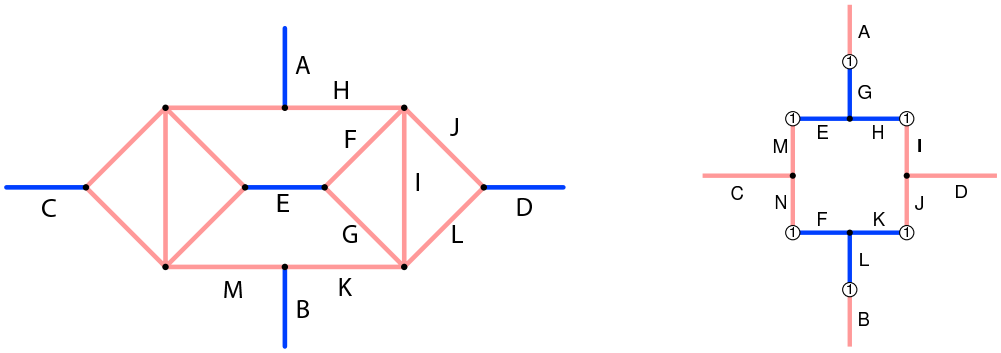
The same algorithm as above serves to show that this task is also in PSPACE. □

3.2 Planar Nondeterministic Constraint Logic

The result obtained in the previous section used particular constraint graphs (represented as circuits), 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 [3].

In this section we show that any normal-form NCL graph can be translated into an equivalent normal-form planar NCL (PNCL) graph, obviating the need for crossover gadgets in reductions from NCL.

Figure 8(a) illustrates the reduction. All vertices have minimum in-flow constraints of 2, so the blue-red-red vertices need either the blue edge or both red edges to be directed inward. The degree-4 vertices need two edges to be directed inward.



(a) Crossover (b) Half-crossover

Figure 8: Planar crossover gadgets.

Lemma 14 *In a crossover gadget, 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. □

This does not quite complete the desired construction, because the crossover subgraph is not in normal form, and the planar circuit equivalence shown by Corollary 7 only applies to graphs in normal form. To solve this problem, we replace each degree-4 vertex in Figure 8(a) with the equivalent subgraph in Figure 8(b). (This is a shorthand for a planar normal-form subgraph; see Figure 2.)

Lemma 15 *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, and G must face up. This forces H to face left and I to face up; then D must face left and J must face up. But then K must face right, and L and 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 M and N up, E right, I and J down, and K 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. □

Theorem 16 *Every normal-form constraint graph has an equivalent planar normal-form constraint graph which can be computed in polynomial time.*

Proof: Lemmas 14 and 15. 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. □

3.3 Nondeterministic Constraint Logic on a Polyhedron

In this section we give a reduction from NCL to a particularly simple geometric form. We show that any normal-form NCL graph can be translated into an equivalent simple planar 3-connected graph. Steinitz's Theorem [8, 10] 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 9: 3-connectivity method.

We use the constructions in Figure 9 to perform the conversion. We may replace any red edge with a subgraph yielding an extra unconstrained blue edge, as shown in Figure 9(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. (Here we are using the same trick as in Figure 2 again.) Likewise, we may replace any blue edge with a similar subgraph, as shown in Figure 9(b).

Theorem 17 *Every normal-form constraint graph has an equivalent simple planar 3-connected graph which can be computed in polynomial time.*

Proof: First, 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 9. We replace any other edge in the graph with such a subgraph as well, and join the two free blue edges.

Next, we make the graph planar, by Theorem 16. (This construction keeps the graph simple.)

Suppose the resulting graph is not 3-connected. Then there exists a cut of zero, one, or two edges separating the graph into two pieces. Choose an edge on the exterior of each piece, replace them with subgraphs from Figure 9, and join the two free blue edges. Now an extra edge must be cut to separate the two pieces.

By repeating this process, we may make the graph simple, planar, and 3-connected. □

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 PNCL 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* [5] 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.

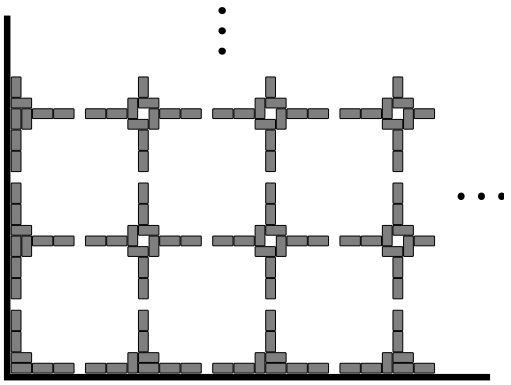
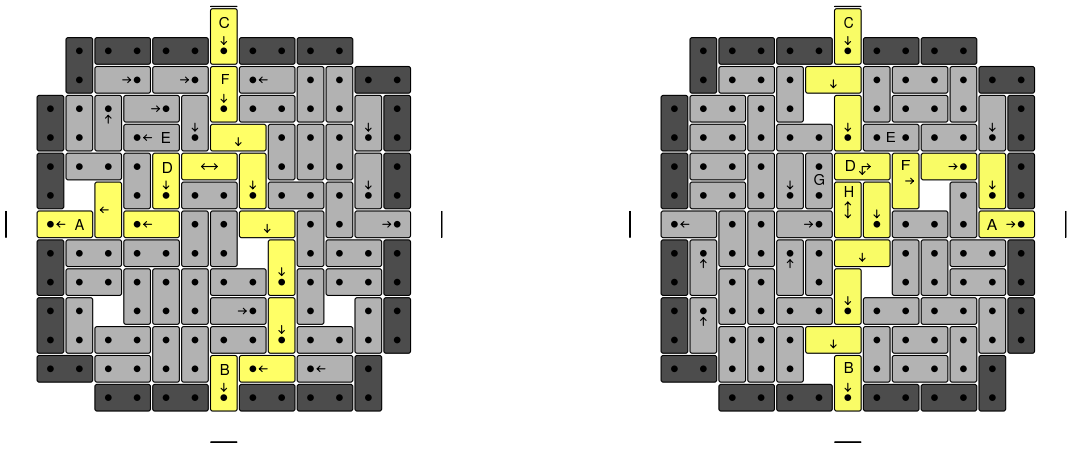


Figure 10: Sliding Blocks layout.

Sliding Blocks Layout. We fill the box with a regular grid of gate gadgets, within a "cell wall" construction as shown in Figure 10. 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) Weak Or

Figure 11: Sliding Blocks gates. Inputs are on the bottom and sides; outputs are on the top.

NCL Gates. We construct NCL And and Weak Or gates out of dominoes, in Figures 11(a) and 11(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 11(b), block D may occupy its initial position, the position one unit to the right, or the position one unit down (but not 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 gate properties. We also note that the constructions are such that no block is ever free to rotate.

For each gate, 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 gray) may not move outward; we then need to show they may not move inward. The light gray ("trigger") blocks are the ones whose motion serves to satisfy the gate constraints; the medium gray blocks are fillers. Some of them may move, but none may move in such a way as to disrupt the gates' correct operation.

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

In the figures, all the ports are shown inactive. Activation proceeds by moving "holes" forward. If the bottom port block of the AND gate is moved down (extended from the gate), then we say the input has activated. If the top port block is moved down (retracted into the gate), then we say the output has activated. (Note that sliding a block *out* of a gate thus corresponds to directing an edge *in* to a vertex.)

Lemma 18 *The construction in Figure 11(a) satisfies the same constraints as an NCL* AND *gate.*

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 19 *The construction in Figure 11(b) satisfies the same constraints as an NCL* WEAK OR *gate.*

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.

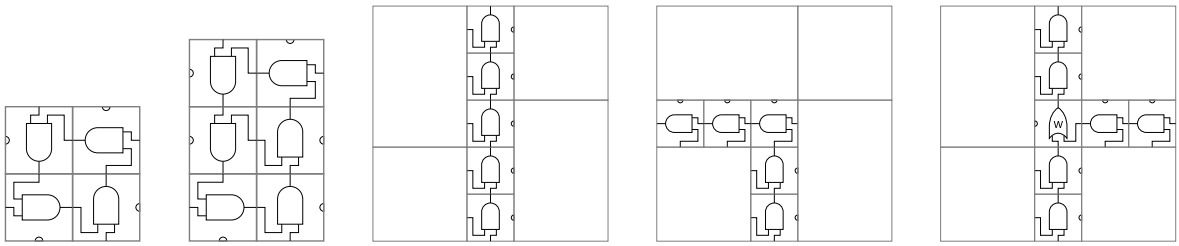
The annotations again provide the bulk of the proof of their own correctness. In this case there are three exceptions. Block E looks as if it might be able to move down, because D may move down and F may move right. However, D may only move down if B moves down, and F may only move right if A moves right. Because this is a WEAK OR, we are guaranteed that this cannot happen: the gate will be used only in circuits such that at most one input can ever be active. 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 both inputs to be active.

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. □

Circuits. Having shown how to make AND and OR gates out of sliding-blocks configurations, we must now show how to wire them together into arbitrary planar circuits. First, note that the box wall constrains the facing port blocks of the gates adjacent to it to be retracted (see Figure 10). This does not present a problem, however, as we will show. The unused ports of both the AND and OR gates are unconstrained; they may be slid in or out with no effect on the gates. Figures 12(a) and 12(b) show how to make (2×2)-gate and (2×3)-gate "filler" blocks out of ANDs. Because none of the ANDs need ever activate, all the exterior ports of these blocks are unconstrained. (The unused ports are drawn as semicircles.)

We may use these filler blocks to build (5×5)-gate blocks corresponding to "straight" and "turn" wiring elements (Figures 12(c) and 12(d)). Because the filler blocks may supply the missing inputs to the ANDs, the output of one of these blocks may activate if and only if the input is active. Also, we may "wrap" the AND and WEAK OR gates in 5×5 "shells", as shown for WEAK OR in Figure 12(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 circuit.



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

Figure 12: Sliding Blocks wiring.

Theorem 20 *Sliding Blocks is PSPACE-hard, even for* 1×2 *blocks.*

Proof: Reduction from PNCL, by the construction described. The output port block of a particular gate may move if and only if the corresponding NCL graph edge may be reversed. □

Corollary 21 *The Warehouseman's Problem is PSPACE-hard, even for* 1×2 *blocks.*

Proof: By Corollary 13, deciding whether a given NCL configuration is reachable is PSPACE-hard. 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 12.

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 [3] 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 gates, as well as a crossover gadget. Tromp [9] strengthened their result by showing that Rush Hour is PSPACE-complete even if the blocks are all 1×2.

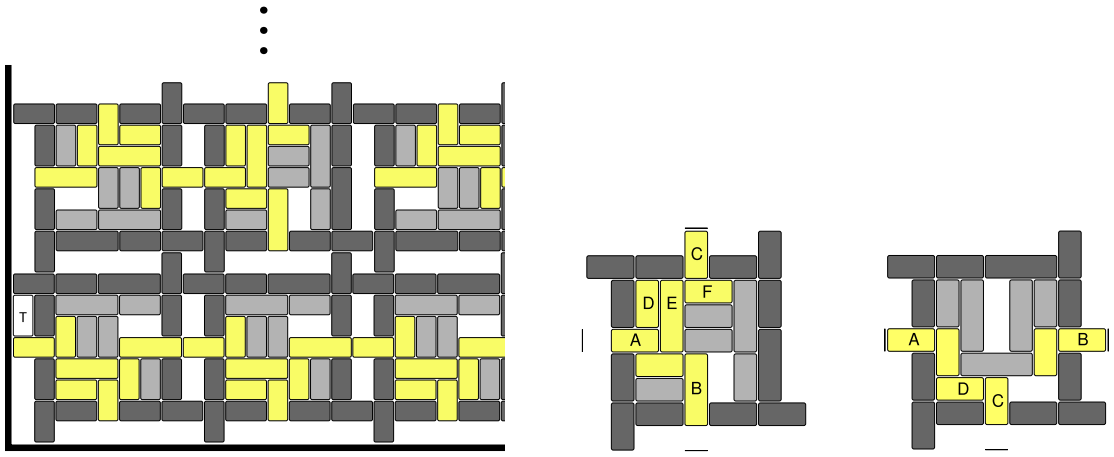
Here we give a simpler construction showing that Rush Hour is PSPACE-complete, again using the traditional 1×2 and 1×3 blocks which must slide lengthwise. We only need an And and a Latch, which here turns out to be easier to build than Or; because of our generic crossover construction (Section 3.2),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

Rush Hour Layout. We tile the grid with our gate gadgets, as shown in Figure 13(a). One block (T) is the target, which must be moved to the bottom left corner; it is released when a particular gate is activated.

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

As in the sliding-blocks construction, input blocks are activated by sliding them out of the gate; output blocks are activated by sliding them into the gate. The layout ensures that no port block may ever slide out into an adjacent gate; this helps keep the cell walls fixed.

Lemma 22 *The construction in Figure 13(b) satisfies the same constraints as an NCL* And *gate. Blocks* A *and* B *are inputs; block* C *is the output.*



(a) Layout (b) AND (c) LATCH

Figure 13: Rush Hour layout and gadgets.

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 23 *The construction in Figure 13(c) satisfies the same constraints as an NCL* LATCH *gate. Block C is the input; blocks A and B are outputs.*

Proof: Block D represents the internal state. When the input is inactive (C is not extended), the internal state may not change, and only one output may activate (retract). When C activates, D may move so as to release both outputs; it may also switch sides, so that after C deactivates only the other output may be active. These are the same constraints an NCL LATCH has. □

Circuits. We may use the same constructions here we used for sliding-blocks layouts: 5×5 blocks of Rush Hour gates serve to build all the wiring necessary to construct arbitrary planar circuits (Figure 12).

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 gates from the bottommost 5×5 blocks; these can have no effect on the circuit. The resulting configuration, shown in Figure 13(a), allows the target block to be released properly.

Theorem 24 *Rush Hour is PSPACE-complete.*

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

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

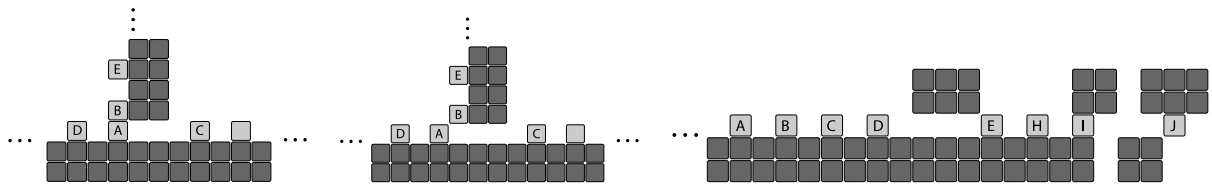
Generalized Problem Bounds. We may consider the more general *Constrained Sliding Block* problem, where blocks need not be 1×2 or 1×3, and may have a constraint direction independent of their dimension. In this context, the existing Rush Hour results do not yet provide a tight bound; the complexity of the problem for 1×1 blocks has not been addressed. We note that deciding whether a block may move at all is in P, so a straightforward application of our technique will not work; however, the complexity of moving a given block to a given position is not obvious.

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 Sokoban is PSPACE-complete, by showing how to construct a Sokoban position corresponding to a space-bounded Turing machine. Using PNCL, 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 14(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 14: Sokoban gadgets.

NCL Gates. We construct NCL And and Or gates out of partial Sokoban positions, in Figure 14. The dark gray blocks in the figures, though unmovable, are not barriers; they are simply blocks that because of their configuration it is not possible to move. The light gray ("trigger") blocks are the ones whose motion serves to satisfy the gate constraints. In each gate, blocks A and B represent inactive inputs; block C represents an inactive output. A and C activate by moving left one unit; B activates by moving up one unit. We assume that the pusher may freely move to any empty space surrounding a gate. We also assume that block D in Figure 14(a) may not reversibly move left more than one unit. Later, we show how to arrange both of these conditions.

Lemma 25 *The construction in Figure 14(a) satisfies the same constraints as an NCL* And *gate.*

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 26 *The construction in Figure 14(b) satisfies the same constraints as an NCL* Or *gate.*

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 25, we will be left in an unrecoverable configuration. □

Circuits. We have shown how to make AND and OR gates, but we must still show how to connect them up into arbitrary planar circuits. The remaining gadgets we shall need are illustrated in Figure 14(c).

The basic idea is to wire the gates together with alternating sequences of blocks placed against a double-thick wall, as in the left of Figure 14(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 gate port 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 gate, 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 14(a) may not reversibly move left more than one unit by protecting this input of every AND with a turn; observe that in Figure 14(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 gate.

Theorem 27 *Sokoban is PSPACE-complete, even if no barriers are allowed.*

Proof: Reduction from PNCL. Given a planar NCL circuit, we build a Sokoban configuration as described above. We use the result that to achieve a specified NCL configuration is PSPACE-hard (Corollary 13). Our circuit has two relevant states, those corresponding to the initial and desired NCL states. Our initial configuration is the former; we place a target at every position that would be occupied by a block in the latter. Since NCL is inherently reversible, and our construction emulates NCL, then the solution configuration must also be reversible, as required for the unrecoverable configuration constraints.

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

5 Related Problems

Two variations of the motion-planning family of problems suggest themselves for consideration. First, suppose we want the *shortest* sequence of moves that solves a given puzzle. This problem is certainly PSPACE-hard for e.g. Sliding Blocks; is it harder than PSPACE?

Second, *puzzle design* asks, for example, to find the most difficult puzzle with a particular box size and set of pieces. For simplicity, we define the *difficulty* of a puzzle to be the smallest number of moves required to solve it. This problem is intuitively at least as hard as the preceding one; is it harder?

In fact, both of these problems are also in PSPACE, at least for discrete versions of these puzzles. (It is not obvious that the Sliding Blocks problem itself is in PSPACE when allowing continuous motions.)

Theorem 28 *Finding the shortest sequence of moves that solves a given discrete motion-planning problem is in PSPACE.*

Proof: To find the shortest sequence of moves, first look for length-1 solutions, then length-2, etc. Each search invokes a subprogram which is the deterministic version of a nondeterministic search algorithm, as described in Theorem 12. We need a counter for the current maximum solution length, and the current move

count. These are both bounded by the possible number of configurations: if there's a solution, there must be a solution in at most this many moves. If n bits are required to represent a configuration, then at worst, the number of configurations is 2^n, so the counter space is linear. □

Theorem 29 *Finding the most difficult discrete motion-planning problem with a particular box size and set of pieces is in PSPACE.*

Proof: We can use the same trick here as above, except this time we need a way to enumerate the puzzles. To do this, just step through all 2^n bit patterns in the configuration space, and for each valid configuration, find the shortest solution by using the algorithm above. We need to remember the current most difficult puzzle; again, this takes a linear number of bits. □

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? 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. **Lunar Lockout.** In this commercial puzzle, robots are placed on a grid, and each move slides a robot in one direction until it hits another robot; robots are not allowed to fly off to infinity. The goal is to bring a particular robot to a specified position. Is this problem PSPACE-complete?

3. **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 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] 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.

[4] 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.

[5] 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.

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

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

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

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

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

A AND/OR Constraint Graphs

In Section 2.3, we showed that every AND/OR circuit has an equivalent normal-form constraint graph. This reduction is useful, because it is often most convenient to reduce from the circuit formulation of NCL to show problems PSPACE-hard. However, the graph formulation is more concise, and is the more natural version of the problem to state.

Here we show that NCL is PSPACE-complete even when the constraint graphs are restricted to planar graphs containing only red-red-blue (AND) and blue-blue-blue (OR) vertices. This formulation of the problem is both concise and convenient to reduce from.

This result does not follow automatically from the results in Sections 2.3 and 3.2. An AND/OR circuit's equivalent constraint graph contains red-blue-blue vertices, which convert from red edges to blue edges. (See Figure 2.) To find a graph with only AND and OR vertices, we must find another way to perform this conversion. We use instead the subgraph shown in Figure 15.

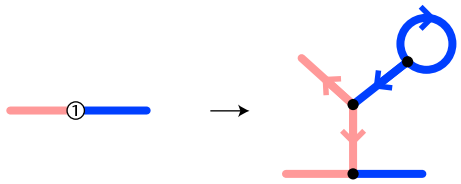


Figure 15: Alternate red-to-blue conversion.

This subgraph also supplies the necessary extra input to convert red edges to blue edges, but it has an unattached red edge as well. Using only AND and OR vertices, red edges only come in pairs; therefore, there is no self-contained subgraph that can absorb a single extra red edge. However, there are an even number of red-to-blue conversions required: because there are no red-red-red vertices, the red edges in the graph form a set of unbranching open chains or closed loops. At each end of each chain is a red-blue vertex requiring conversion. Therefore, we may join the extra red edges in Figure 15 in pairs.

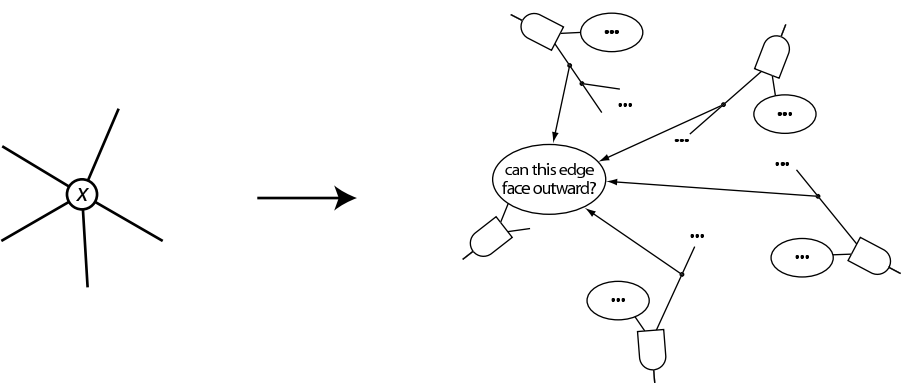
The resulting graph may not be planar, but it can be made planar by using the crossover gadgets in Section 3.2. There are three complications. First, the half-crossover (Figure 8(b)) contains two red-red-red vertices. We replace each with three AND vertices, as in Figure 4(a); this change eliminates four of the red-to-blue conversions required (becuase the AND outputs are blue edges), and introduces two new ones (to keep edges C and D red).

The second complication is the remaining four red-to-blue conversions in the half-crossover. These conversions can use the new subgraph in Figure 15, joining the extra red edges in pairs, preserving planarity.

The final crossover complication is that the crossover subgraph has blue edge terminals. We can cross red edges over red edges, by converting each terminal blue edge in the crossover to red using Figure 15, joining the free red edges in two pairs. We can avoid having to cross red edges over blue edges, by inserting extra red edges where necessary. Any blue edge may be replaced by a blue-red-blue edge sequence, using two copies of Figure 15 with their free edges joined. (This is similar to the construction in Figure 9(b), but does not result in any extra edges.) Then the original blue edge may be effectively crossed by crossing two red edges instead.

Theorem 30 *Every normal-form constraint graph has an equivalent planar* AND/OR *constraint graph which can be computed in polynomial time.*

Proof: We first convert the graph to an AND/OR circuit as in Lemma 1. Then we convert the circuit back to a normal-form graph using Lemma 2. The resulting graph contains no red-red-red vertices, and we replace the red-blue-blue vertices with copies of Figure 15 as described above. Finally, we make the graph planar, using Theorem 3.2, modified suitably as described above. □


X
can this edge
face outward?