The Nondeterministic Constraint Logic Model of Computation: Reductions and Applications

Robert A. Hearn[1] and Erik D. Demaine[2]

[1] Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 200 Technology Square, Cambridge, MA 02139, USA, rah@ai.mit.edu
[2] Laboratory for Computer Science, Massachusetts Institute of Technology, 200 Technology Square, Cambridge, MA 02139, USA, edemaine@mit.edu

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 [2].
We illustrate the importance of our model of computation by giving simple reductions to show that multiple 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$^{\mathrm{TM}}$ puzzles are PSPACE-complete [2], of which we also give a simpler proof. Finally, we strengthen the existing result that the pushing-blocks puzzle Sokoban is PSPACE-complete [1], by showing that it is PSPACE-complete even if no barriers are allowed.

1 Introduction

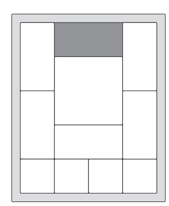


Fig. 1. Move the large square to the bottom center.

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* [5], 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 Fig. 1 for an example.

The *Warehouseman's Problem* [4] 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 [4] 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 [4], 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.

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 [2], 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 [2], and a restricted form of Sokoban [1].

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 the regular expressions.

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 NCL.

2 Nondeterministic Constraint Logic

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. Note that any loop edge can trivially be reversed; see, e.g., Fig. 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.

Fig. 2 shows one translation to normal form that we will use frequently. In fact, every constraint graph has an equivalent normal form which can be computed in polynomial time. We omit the proof in this abstract; it is not needed for our main results.

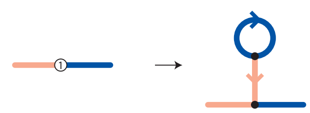


Fig. 2. Normalizing red-to-blue conversion.

2.2 Circuit Formulation

In this Section and the following we show that a normal-form 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.

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

Fig. 3. Gates. Inputs are at bottom; outputs are at top.

AND and OR Gates. An And gate (Fig. 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. Similarly, an Or gate (Fig. 3(b)) has the constraint that its output may be active only if at least one input is active.

SPLIT Gate. A Split (Fig. 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 (Fig. 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.

WEAK OR Gate. A Weak Or gate (Fig. 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 (Fig. 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. The idea of an inverter does not map onto the passive nature of NCL: ports are *permitted*, but not required, to change state when their constraints are satisfied.

The approach in [2] 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 normal-form 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.

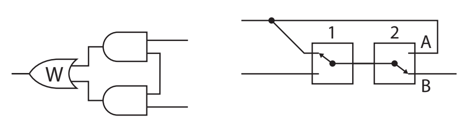
(a) red-red-red (b) red-red-blue (c) red-blue-blue (d) blue-blue-blue

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

Lemma 1. *Normal-form constraint graphs and* AND *and* OR *circuits are polynomial-time equivalent.*

Proof sketch. Apply the conversions in Fig. 4. □

Lemma 1 shows 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 [2], 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

Fig. 5. Gate emulations. Inputs are on the left; outputs are on the right.

Lemma 2. LATCH *may be emulated with* AND *and* WEAK OR*;* WEAK OR *may be emulated with* OR*;* OR *may be emulated with* SPLIT *and* LATCH*.*

Proof sketch. Fig. 5 shows the LATCH and the OR constructions; OR may substitute directly for WEAK OR. □

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

Theorem 1. *The following are polynomial-time equivalent: normal-form constraint graphs,* AND/OR *circuits,* AND/LATCH *circuits, and* AND/WEAK OR *circuits.*

Proof. Lemmas 1 and 2. □

Corollary 1. *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 [3], even when the formula is in conjunctive normal form. A simple argument then shows that NCL is in PSPACE, and therefore PSPACE-complete.

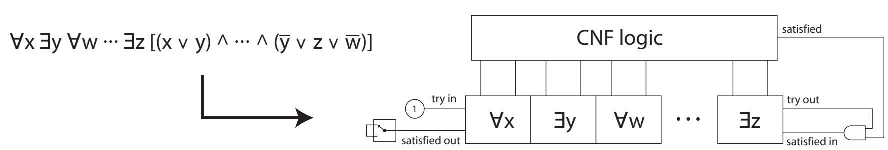


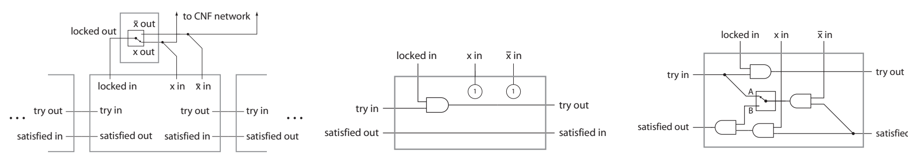
Fig. 6. Schematic of the reduction from Quantified Boolean Formulas to NCL.

Reduction. First we will give an overview of the reduction and the gadgets we will need; then we will analyze the gadgets' properties. We use the circuit form of NCL. The reduction is illustrated schematically in Fig. 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 quantifer, 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* (Fig. 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 satisified 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; connections (b) Existential quantifier (c) Universal quantifier

Fig. 7. QBF reduction gadgets.

Variable Gadget. A variable gadget (shown in Fig. 7(a)) 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. 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 evaulate 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 Fig. 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.

Lemma 3. *A quantifier gadget's satisfied in port may not activate unless its try out port is active.*

Proof sketch. Induct from right to left, using properties of quantifier gadgets. □

Existential Quantifier. For an existential quantifier gadget (Fig. 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 4. *An existential quantifier gadget may activate its satisfied out port if and only if its satisfied in port is active with its variable locked in some state.*

Proof. By Lemma 3 and the definition of the existential quantifer gadget. □

Universal Quantifier. A universal quantifer gadget (Fig. 7(c)) 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 assignmnent has been successfully tried, and then enable satisfied out only if the bit so indicates, and the other assignment is currently satisfied. To ensure that the bit resets appropriately, the other LATCH state is constrained to be active when try in is inactive.

Lemma 5. *A universal quantifier gadget may activate its satisfied out port if and only if its satisfied in port is at one time active with its variable locked in the false ($\overline{x}$) state, and at a later time is again active with its variable locked in the true (x) state, with try in remaining active throughout.*

Proof sketch. The constraints on the gates essentially force this route. □

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

Lemma 6. *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 sketch. By induction from right to left using Lemmas 4 and 5. □

Theorem 2. *NCL is PSPACE-complete.*

Proof. Lemma 6 establishes PSPACE-hardness. A simple nondeterministic algorithm traverses the state space, maintaining only the current state, so NCL is in NPSPACE, and Savitch's Theorem [6] says that NPSPACE = PSPACE. □

Corollary 2. *Deciding whether a specified configuration of an NCL graph is reachable is PSPACE-complete.*

Proof sketch. The configuration which is identical to the initial configuration, but with the attached LATCH state switched, is reachable just when ϕ is true. □

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 [1] is quite intricate. A crossover gadget is also among those used in the Rush Hour proof [2].

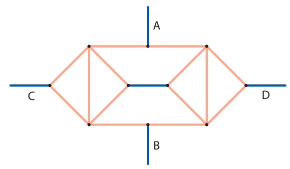


Fig. 8. Planar crossover.

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. Fig. 8 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.

Lemma 7. *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 sketch. The constraints on the vertices force this behavior. □

The crossover subgraph is not in normal form, and Corollary 1 only applies to graphs in normal form. To solve this problem, we replace each degree-4 vertex in Fig. 8 with the equivalent subgraph in Fig. 9.

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 sketch. A similar but simpler constraint analysis as in Lemma 7. □

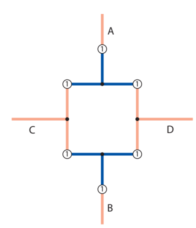


Fig. 9. Half-crossover.

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

Proof. Lemmas 7 and 8. □

3.3 Nondeterministic Constraint Logic on a Polyhedron

Nondeterministic Constraint Logic has a particularly simple geometric form. Any NCL graph can be translated into an equivalent simple planar 3-connected graph, which 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. We omit the construction and the proof in this abstract.

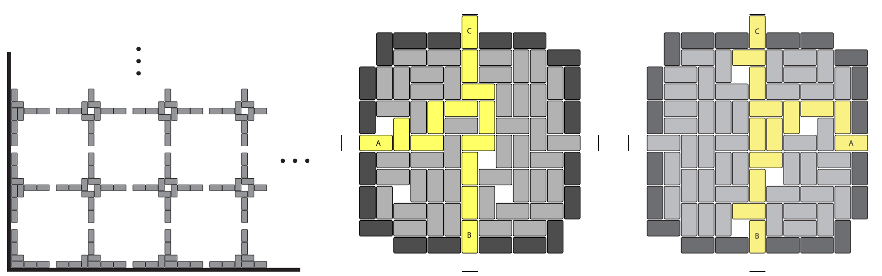
4 Applications

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 (Rush Hour) reproduces an existing result, with a simpler construction; the last (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 give a reduction from PNCL showing that Sliding Blocks is PSPACE-hard even when all the blocks are 1×2 rectangles (dominoes). In contrast, there is a simple polynomial-time algorithm for 1×1 blocks; thus, our results are tight. The *Warehouseman's Problem* [4] 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.

Fig. 10 shows a schematic of our reduction and the two required gate gadgets. A and B are (inactive) inputs; C is the (inactive) output. Activation proceeds by moving "holes" forward: A activates by moving out, C by moving in.



(a) Layout (b) AND (c) WEAK OR

Fig. 10. Sliding Blocks layout and gates.

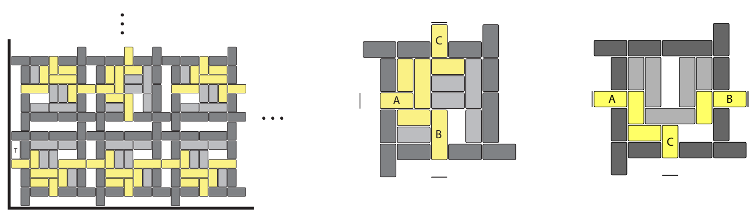
To build arbitrary planar circuits, we also need "straight" and "turn" blocks. These may be formed from (5×5)-gate combinations of AND gates. The construction is omitted in this abstract.

4.2 Rush Hour

In the puzzle *Rush Hour*, one is given a sliding-block configuration with the additional constraint 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 [2] 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 [7] 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, as shown in Fig. 11.



(a) Layout (b) And (c) Latch

Fig. 11. Rush Hour layout and gadgets.

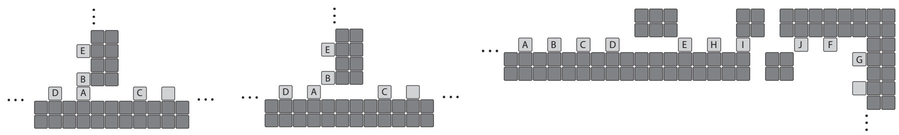
4.3 Sokoban

In the 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 [1] 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 construction. Our result applies even if there are no barriers allowed, thus strengthening Culberson's result.

Figs. 12(a) and 12(b) illustrate the And and Or gates. In each, block C may be reversibly pushed left only when A and/or B have been pushed left/up, as appropriate. Fig. 12(c) shows gadgets for basic wiring (A can move right only if D has), parity switching (D to E), wire flipping and pusher pass-through (H to J), and turning corners (F to G). These gadgets permit arbitary planar circuits.

The target position corresponds to a desired PNCL configuration; this ensures that moves that violate the above conditions do not permit solution.



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

Fig. 12. Sokoban gadgets.

5 Conclusion

We proved that one of the simplest possible forms of motion planning, involving sliding 1×2 blocks (dominoes) around in a rectangle, 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 remains 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, but how hard is moving a given block to a given position?
2. **Lunar Lockout.** In this 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?

Acknowledgment. We thank John Tromp for several useful suggestions.

References

1. Joseph Culberson. Sokoban is PSPACE-complete. In *Proceedings of the International Conference on Fun with Algorithms*, pages 65–76, Elba, Italy, June 1998.
2. 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.
3. Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.
4. 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.
5. Edward Hordern. *Sliding Piece Puzzles*. Oxford University Press, 1986.
6. Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.
7. John Tromp. On size 2 Rush Hour logic. Manuscript, December 2000.