1-1-2003

# The Computational Complexity of Motion Planning

Jeff R.K. Hartline '01
*Harvey Mudd College*

Ran Libeskind-Hadas
*Harvey Mudd College*

Recommended Citation
J. R. Hartline and R. Libeskind-Hadas, "The Computational Complexity of Motion Planning," SIAM Review, Vol. 45, No. 3, October 2003, pp. 543-557. DOI: 10.1137/S003614450139517

SIAM Review
Vol. 45, No. 3, pp. 543–557

# The Computational Complexity of Motion Planning*

Jeffrey R. Hartline†
Ran Libeskind-Hadas‡

**Abstract.** In this paper we show that a generalization of a popular motion planning puzzle called Lunar Lockout is computationally intractable. In particular, we show that the problem is PSPACE-complete. We begin with a review of NP-completeness and polynomial-time reductions, introduce the class PSPACE, and motivate the significance of PSPACE-complete problems. Afterwards, we prove that determining whether a given instance of a generalized Lunar Lockout puzzle is solvable is PSPACE-complete.

**Key words.** motion planning, NP-completeness, PSPACE-completeness

**AMS subject classifications.** 03D15, 68Q17, 68Q25

**DOI.** 10.1137/S0036144501395174

## 1. Introduction.

Motion planning is an important area in robotics which addresses computationally planning the motions of one or more robots to achieve a specific goal. For example, the goal might be moving a robotic arm to a specified location or planning the path of a robot through an environment with obstacles. Many of these problems are known to be computationally intractable or "hard." For example, a number of motion planning problems have been shown to be NP-complete, implying that they belong to a group of equally hard problems including the famous traveling salesman problem, among others.

Some motion planning problems appear to be even harder than NP-complete. The PSPACE-complete problems are a class of problems that are at least as hard as the NP-complete problems and are believed to be even harder. Establishing that a problem is PSPACE-complete suggests that it is probably not fruitful to search for an efficient algorithm that solves the problem optimally. Instead, it is desirable to develop heuristics for the problem or use approximation techniques.

In this paper we show that a generalization of a puzzle called *Lunar Lockout*[1] is PSPACE-complete. While our discovery of this result is interesting in its own right, our proof is particularly illustrative of a general technique that has been used, by us and by others, to prove the PSPACE-completeness of motion planning puzzles. Our proof is the simplest example of this technique that we have seen. At the end of the

*Received by the editors September 12, 2001; accepted for publication (in revised form) August 19, 2002; published electronically August 11, 2003.
http://www.siam.org/journals/sirev/45-3/39517.html

†Department of Computer Science, Cornell University, Ithaca, NY 14853 (jhartlin@cs.cornell.edu). This work was performed when the first author was at Harvey Mudd College.

‡Department of Computer Science, Harvey Mudd College, Claremont, CA 91711 (hadas@cs.hmc.edu).

[1]Lunar Lockout is a trademark of Binary Arts, Inc.

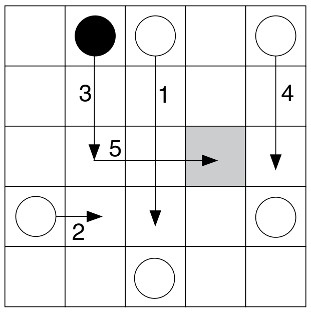

**Fig. 1** *A solution to a Lunar Lockout puzzle. The target robot is indicated in black and the target cell is indicated in gray.*

paper we revisit the proof technique to show how it could potentially be applied to a wide variety of other motion planning problems.

Lunar Lockout is played on a $5 \times 5$ grid. Several robots are placed on the grid, with each robot occupying a single grid cell. At each move, any robot may be slid up, down, right, or left, but only if there is another robot in that direction. The robot is slid until it touches the first robot it encounters in that direction. One of the robots is designated the *target robot* and one grid cell is designated the *target cell*. The objective is to move the target robot to the target cell. An example is shown in Figure 1, where the target robot is indicated in black and the target cell is indicated in gray. One solution is indicated by arrows labeled with a possible ordering of the moves.

We define *Generalized Lunar Lockout (GLL)* to be the version of this puzzle played on an $m \times m$ board. We also define a variant of this puzzle, called *Generalized Lunar Lockout Variant (GLLV)*, to be the version of GLL in which some robots may be designated as being stationary. It should be noted that GLL and GLLV are not exactly the same puzzle. In this paper we will show that GLLV is PSPACE-complete. It is not known if GLL is also PSPACE-complete. We discuss this issue in more detail at the end of the paper.

In the next section we provide a brief review of computational complexity and discuss PSPACE-completeness and its significance. In the two subsequent sections we prove that GLLV is PSPACE-complete. We conclude with a discussion of how this technique can be used to show that other motion planning puzzles are PSPACE-complete and offer suggestions for further related reading.

## 2. A Brief Review of Computational Complexity.

In this section we review some of the fundamental ideas of computational complexity. We assume that the reader has some familiarity with the concepts of Turing machines, NP-completeness, and polynomial-time reductions, equivalent to what is typically found in an undergraduate course in the theory of computation. After reviewing these concepts we introduce the notions of PSPACE and PSPACE-completeness.

### 2.1. Measuring Complexity with Turing Machines.

The computational complexity of a problem is a measure of how much of a particular resource, such as time or space (memory), is required to solve the problem on a computer. Since the complexity is measured as a function of the size of the problem, it is not very interesting to restrict our attention to a problem of fixed size. For this reason, we generalized the

Lunar Lockout puzzle to be played on a board of arbitrary size rather than examining puzzles played only on $5 \times 5$ boards.

In order to standardize the notion of a "computer," it is conventional to use a *Turing machine* as the canonical model of computation. It is not difficult to show that the amount of time or space used by a Turing machine is "essentially" the same as the amount of that resource used on any other "reasonable" model of computation. Most introductory texts on the theory of computation quantify this notion more precisely (see, for example, [9]).

Recall that a deterministic Turing machine is a device which comprises a *tape* of discrete tape cells and a movable *tape head* which is used to read and write symbols on the tape. The tape has a left endpoint but is infinite to the right. The machine can be in any one of a finite number of states at each step of computation. An input string is encoded beginning at the left end of the tape and the remaining tape cells contain blank symbols. The machine begins computation with its tape head at the leftmost tape cell in a special "starting" state. At each step, it reads the symbol under the tape head and consults its *transition function* which specifies a symbol to write at that tape cell, the new state of the machine, and a direction (either right or left) in which the tape head should move one cell. If the machine enters specially designated "accepting" or "rejecting" states, it halts and "accepts" or "rejects" its input, respectively.

More formally, a Turing machine, $M$, is specified by a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}})$, where

1. $Q$ is the finite set of states of the machine.
2. $\Sigma$ is the finite set of symbols which are used in the input string. For simplicity and concreteness, we assume that $\Sigma = \{0, 1\}$. In other words, the input string is encoded in binary.
3. $\Gamma$ is the finite set of symbols in $\Sigma$, along with the blank symbol and any additional symbols which are used to facilitate the computation. Again, for simplicity, we assume that $\Gamma = \{0, 1, B\}$, where $B$ is the blank symbol.
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\text{Left, Right}\}$ is the transition function. The transition $\delta(q, s) = (q', s', D)$ means that when the Turing machine is in state $q \in Q$ with the symbol $s \in \Gamma$ at the position of the tape head, it should transition to state $q' \in Q$, write the symbol $s' \in \Gamma$ at this tape cell, and move the tape head one step in direction $D \in \{\text{Left, Right}\}$.
5. $q_{\text{start}}$, $q_{\text{accept}}$, $q_{\text{reject}}$ are the starting, accepting, and rejecting states in $Q$, respectively.

The time used by a Turing machine is the number of moves it makes before halting in either the accepting or rejecting state. Similarly, the space used by a Turing machine is the number of tape cells visited before halting. We say that a Turing machine uses *polynomial time* or *space* to mean that the amount of time or space, respectively, is upper bounded by some polynomial in the size of the input.

**2.2. Decision Problems and Reductions.** Complexity theory generally assumes that the problem under consideration is a *decision problem*: a problem whose answer is either "yes" or "no." A Turing machine solves the problem if, given an encoding of any instance of the problem, it always enters the accepting state in finite time if the answer is "yes" and enters the rejecting state in finite time if the answer is "no." In the case of GLLV, for example, we would like to encode an initial configuration of the puzzle on the tape of a Turing machine. The machine would accept the input if that instance of the problem were solvable; otherwise it would reject the input. Although it may seem unnatural to restrict our attention to determining merely whether an instance of GLLV can or cannot be solved, note that if the decision problem is hard, then finding an actual solution to the puzzle is at least as hard.

A decision problem $\Pi'$ is said to be *reducible* to a decision problem $\Pi$ if there exists a Turing machine which, given any instance of $\Pi'$, produces an instance of $\Pi$ such that if the instance of $\Pi'$ has answer "yes," then the constructed instance of $\Pi$ also has answer "yes." In addition, if the answer to the instance of $\Pi'$ is "no," then the answer to the constructed instance of $\Pi$ is also "no." We say that $\Pi'$ is *polynomial-time reducible* to $\Pi$ if the reduction can be performed in time polynomial in the size of the instance of $\Pi'$. Notice that if $\Pi'$ is polynomial-time reducible to $\Pi$ and, in addition, a polynomial-time Turing machine is found for $\Pi$, then $\Pi'$ can be solved in polynomial time: First the instance of $\Pi'$ is reduced to a corresponding instance of $\Pi$ in polynomial time. Then, the hypothetical polynomial-time Turing machine for $\Pi$ is used to determine if the answer to the $\Pi$ instance is "yes" or "no." By the definition of a reduction, a "yes" answer to the instance of $\Pi$ implies a "yes" answer to the instance of $\Pi'$ and a "no" answer to the instance of $\Pi$ implies a "no" answer to the instance of $\Pi'$.

## 2.3. NP-Completeness.

A problem is said to be in the class *NP* if a solution to the problem can be verified in polynomial time or, equivalently, the problem can be solved by a nondeterministic Turing machine in polynomial time.[2] A problem $\Pi$ is said to be *NP-complete* if it is in NP, and for every problem $\Pi' \in$ NP, $\Pi'$ is polynomial-time reducible to $\Pi$.

The NP-completeness of a problem $\Pi$ is strong evidence that the problem is computationally difficult. In particular, if a polynomial-time algorithm were found for $\Pi$, then we could solve all problems in NP in polynomial time as suggested by our above discussion of polynomial-time reductions.

One apparent difficulty is that to show that a problem $\Pi$ is NP-complete, we need to demonstrate polynomial-time reductions from every NP problem to $\Pi$. This would seem impossible given that there are a huge number of known problems in NP and potentially many problems that are not yet known. One of the most important and surprising results in the theory of NP-completeness is a theorem due to Cook [1]. Cook's theorem states that a particular logic problem known as "satisfiability" (SAT) is NP-complete. Using a clever technique employing Turing machines, Cook showed that *every* problem in NP can be reduced in polynomial time to SAT. We refer the reader unfamiliar with this result to excellent discussions in books by Garey and Johnson [6] and Sipser [9].

The fact that SAT is NP-complete can be used to demonstrate that other problems are NP-complete. For example, to show that some problem $\Pi$ is NP-complete, it suffices to show that $\Pi$ is in the class NP and that SAT is polynomial-time reducible to $\Pi$. This is substantially less work than trying to show that every problem in NP is polynomial-time reducible to $\Pi$. The reason this is valid is that the NP-completeness of SAT implies that every problem in NP can be reduced to SAT in polynomial time.

[2]A *nondeterministic Turing machine* is similar to a deterministic one except the transition function may specify multiple outputs for a given input. In other words, when the Turing machine is in a particular state and reading a particular symbol, there are possibly many next state/symbol/move triplets from which the Turing machine can choose. We say that a nondeterministic machine accepts its input if there exists some sequence of choices (sometimes called "guesses") that lead the machine to the accepting state. A nondeterministic machine accepts its input in polynomial time or space if, in addition, all sequences of choices cause the machine to halt in polynomial time or space, respectively.

If SAT can be reduced to our problem $\Pi$ in polynomial time, then we can construct a polynomial-time reduction from any problem $\Pi'$ in NP to SAT in polynomial time and then reduce SAT to $\Pi$ in polynomial time. The composition of these two polynomial-time reductions results in a polynomial time reduction from any problem $\Pi'$ in NP to $\Pi$. In fact, by the same reasoning, to show that a problem $\Pi$ is NP-complete it suffices to demonstrate a reduction from any *one* known NP-complete problem to $\Pi$.

**2.4. NP-Completeness Proofs and Gadgets.** Before examining PSPACE-completeness and the Lunar Lockout puzzle, we briefly review the notion of "gadgets" and how they are used in NP-completeness proofs. Gadgets are an important concept and a key ingredient in our proof of the PSPACE-completeness of GLLV.

Recall that the 3-satisfiability problem (3SAT) is the following logic problem: We are given a collection of boolean variables $x_1, \ldots, x_v$ and a collection of clauses $C_1, \ldots,$ $C_k$, where each clause contains the disjunction (logical OR) of exactly three literals (a literal is just a variable or its negation). The decision problem is to determine whether or not there exists an assignment of **true** or **false** to each variable such that every clause evaluates to true. Such an assignment is called a *satisfying assignment* and the 3SAT instance is said to be *satisfiable* if such an assignment exists. For example, using variables $x_1, x_2, x_3$ and the clauses $(x_1 \vee x_2 \vee \overline{x_3}), (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$, $(\overline{x_1} \vee x_2 \vee x_3)$, $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3})$, the answer to this instance is "yes" since one possible satisfying assignment is $x_1 = \textbf{true}$, $x_2 = \textbf{false}$, and $x_3 = \textbf{true}$.

The 3SAT problem is easily shown to be NP-complete by a polynomial-time reduction from the SAT problem. The SAT and 3SAT problems are in fact virtually identical, with the exception that SAT permits an arbitrary number of literals in each clause, whereas 3SAT requires exactly three literals per clause. We now examine a famous problem from graph theory, known as the vertex cover problem (VC), and discuss the polynomial-time reduction from 3SAT.

Given a graph $G = (V, E)$, where $V$ is the set of vertices and $E$ is the set of edges, a vertex cover is a subset $S$ of the vertices such that every edge has at least one of its two endpoints in the set $S$. If an edge has endpoints $u$ and $v$ and vertex $u$ is in the vertex cover, then we say that "$u$ covers the edge" or that "the edge is covered by $u$." Of course the edge could be covered by $v$ instead or by both $u$ and $v$. In VC, we are given a graph and a positive integer $\ell$ and we wish to answer the question, "Does there exist a vertex cover of size $\ell$ for this graph?"

It is easily verified that VC is in the class NP. We now demonstrate that 3SAT can be reduced to VC in polynomial time, thereby proving that VC is NP-complete. For any instance of 3SAT we must construct an instance of VC such that the answer to the 3SAT instance is "yes" ("yes, there exists a satisfying assignment") if and only if the answer to the corresponding VC instance is "yes" ("yes, there exists a vertex cover of size $\ell$ for this graph").

Consider an arbitrary instance of 3SAT. For each of the $v$ boolean variables, $x_i$, we construct a pair of vertices labeled $x_i$ and $\overline{x_i}$ and connect this pair of vertices by an edge. This pair of vertices connected by an edge is called a "gadget." More specifically, we will call this a "variable gadget" since, as we shall see, the gadget represents the two possible values of the corresponding boolean variable.

Next, for each of the $k$ clauses, $C_j$, in the 3SAT instance, we construct three vertices and completely connect these vertices using three edges. This group of vertices and edges is labeled $C_j$ and is called a "clause gadget." Notice that each clause has exactly three literals and each corresponding clause gadget has exactly three vertices.

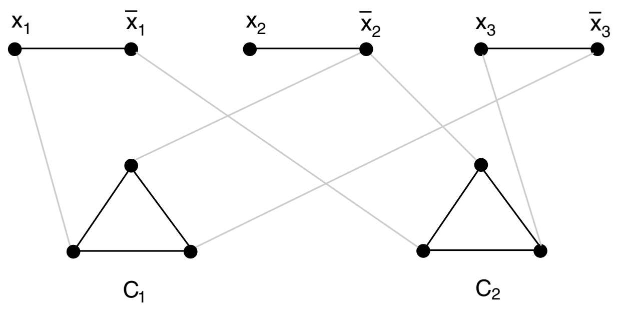

**Fig. 2** *The graph constructed from the* $3SAT$ *instance with boolean variables* $x_1, x_2, x_3$ *and clauses* $C_1 = (x_1 \vee \overline{x_2} \vee \overline{x_3})$, $C_2 = (\overline{x_1} \vee \overline{x_2} \vee x_3)$. *The three variable gadgets are at the top and the two clause gadgets are at the bottom. The gray edges indicate connections between the variable gadgets and the clause gadgets.*

Now, we connect the variable and clause gadgets as follows. For each literal in clause $C_j$, connect one distinct vertex in the corresponding clause gadget to the vertex in the variable gadget labeled with this literal. These edges will henceforth be called *gray edges*. In this way, each of the three vertices in a clause gadget will have exactly one gray edge to a vertex in a variable gadget. For example, for the small 3SAT instance with boolean variables $x_1, x_2, x_3$ and two clauses $C_1 = (x_1 \vee \overline{x_2} \vee \overline{x_3})$, $C_2 = (\overline{x_1} \vee \overline{x_2} \vee x_3)$, we have the corresponding graph shown in Figure 2, where the variable gadgets are on top and the clause gadgets are below. The gray edges are indicated as well.

The VC problem consists of both a graph and a positive integer $\ell$ (recall that the question being asked is, "Does there exist a vertex cover in the graph of size $\ell$?"). We have just described the construction of the graph. The value of $\ell$ in this reduction is $v + 2k$ for reasons that will become apparent very shortly. (Recall that $v$ is the number of variables and $k$ is the number of clauses in the 3SAT instance.)

It is not difficult to verify that the reduction that we have just described can be performed in time polynomial in the size of the 3SAT instance. What remains to be shown is that the answer to the given 3SAT instance is "yes" if and only if the answer to the constructed VC instance is "yes." That is, the 3SAT instance has a satisfying boolean assignment if and only if the constructed graph has a vertex cover of size $\ell = v + 2k$.

Assume that the given instance of 3SAT is satisfiable. Then we construct a vertex cover $S$ of size $v+2k$ in the corresponding graph as follows: For each variable $x_i$, if $x_i$ is assigned the value **true** in the satisfying assignment, then we include the vertex with label $x_i$ in $S$. Similarly, if $x_i$ is assigned the value **false**, then we include the vertex with label $\overline{x_i}$ in $S$. Next, we examine each clause gadget, $C_j$. The clause gadget has three incident gray edges. Denote these edges by $e_1 = (u_1, v_1)$, $e_2 = (u_2, v_2)$, and $e_3 = (u_3, v_3)$, where $u_1, u_2, u_3$ are vertices in the clause gadget and $v_1, v_2, v_3$ are vertices in variable gadgets. Since we started with a satisfying assignment, our construction guarantees that clause $C_j$ has at least one literal which evaluates to **true**, and consequently at least one of the three vertices $v_1, v_2, v_3$ has already been included in $S$. Without loss of generality assume that $v_1$ was included in $S$. Then we include vertices $u_2$ and $u_3$ from the clause gadget in $S$ as well. These two vertices cover all three of the edges in the clause gadget as well as the other two gray edges incident on the clause gadget. In this way, all edges incident on vertices in the $C_j$ gadget are covered. By repeating this process for each clause gadget, we see that $S$ is a vertex cover. Since exactly one vertex is selected from each variable gadget and exactly two vertices are selected from each clause gadget, this vertex cover has size $v + 2k$.

Conversely, assume that there exists a vertex cover of size $v + 2k$ in the graph. Each edge in a variable gadget has at least one of its two endpoints included in the vertex cover. Moreover, each clause gadget must have at least two of its vertices included in the vertex cover in order to ensure that the three edges in the clause gadget are covered. Since the vertex cover has size $v + 2k$, exactly one vertex in each variable gadget and exactly two vertices in each clause gadget must be included in the vertex cover. We now construct an assignment of the boolean variables as follows: If the vertex labeled $x_i$ is included in the vertex cover, then we let $x_i$ be **true** and otherwise we let $x_i$ be **false**. This assignment necessarily satisfies every clause in the 3SAT instance. To see this, consider any clause $C_j$ in the 3SAT instance. In the corresponding clause gadget in the VC instance, at least one of the three gray edges must have an endpoint in a variable gadget that was included in the vertex cover. By construction, this means that clause $C_j$ was satisfied in our assignment.

Notice that the gadgetry here is used to enforce a correspondence between 3SAT and VC. Another way of saying this is that the gadgetry allows us to emulate any instance of the 3SAT problem by a carefully constructed instance of the VC problem. In general, when reducing one problem $\Pi'$ to another problem $\Pi$, gadgetry is used to construct an instance of $\Pi$ that corresponds to the given instance of $\Pi'$.

**2.5. PSPACE-Completeness.** Although NP-completeness is often regarded as the paragon of computational intractability, an evidently harder class of problems are the *PSPACE-complete* problems. A problem is said to be in the class PSPACE if it can be solved in space polynomial in the size of its input. Clearly, a Turing machine which uses polynomial time also uses at most polynomial space. However, a Turing machine which uses polynomial space may use an exceedingly large amount of time before halting. A problem $\Pi$ is said to be PSPACE-complete if it is in PSPACE and if every problem $\Pi' \in$ PSPACE is polynomial-time reducible to $\Pi$.

Observe that if a PSPACE-complete problem, $\Pi$, can be solved in polynomial time, then every problem $\Pi'$ in PSPACE can be solved in polynomial time as well; we use the method described above to reduce the instance of $\Pi'$ to an instance of $\Pi$ with the same answer. Moreover, it is easily verified that NP is a subset of PSPACE. Consequently, showing that a PSPACE-complete problem can be solved in polynomial time immediately implies that all NP-complete problems can be solved in polynomial time as well. Thus, proving that a problem is PSPACE-complete means that it is at least as hard as an NP-complete problem. It is widely conjectured, although not proved, that there exist problems in PSPACE that are not in NP. Thus, the PSPACE-complete problems are believed to be strictly harder than the NP-complete problems.

In the remainder of this paper we show that GLLV, the version of Generalized Lunar Lockout in which some robots may be designated as stationary, is PSPACE-complete. We use a powerful and general technique in our proof. We use this particular puzzle to demonstrate the technique because the mechanisms used in the reduction, that is, the gadgets, are relatively simple. Afterwards, we revisit the technique and discuss how it can be applied in general.

## 3. The Lunar Lockout Gadgetry.

In this section we describe several gadgets which can be built in the GLLV model. These gadgets will be used in the next section to prove that GLLV is PSPACE-complete. In the following figures we use dark circles

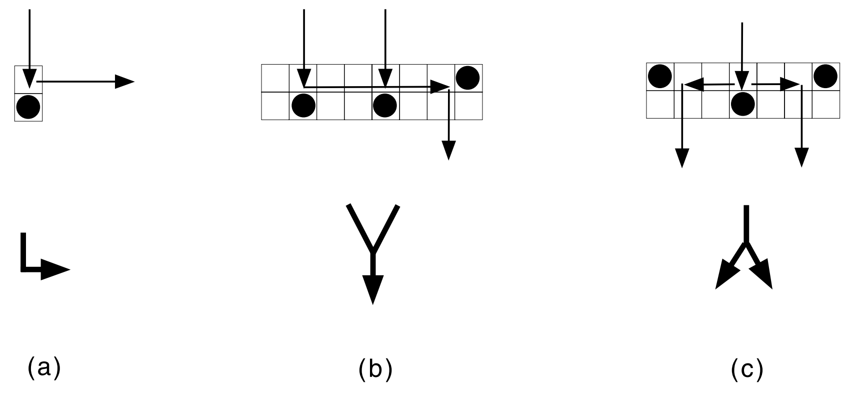

**Fig. 3** *The three basic gadgets with their iconic representations below.* (a) *One-way turn gadget. To ensure that only a right turn is permitted, no other stationary robots may be placed in the same row to the left of this gadget. Three other turn gadgets can be obtained by rotating this gadget* 90, 180, *and* 270 *degrees.* (b) *Merge gadget. To ensure that the paths are merged as intended, no other stationary robots may be placed in the same rows as this gadget to its left.* (c) *Splitter gadget. To ensure that the path is split as intended, no other stationary robots may be place above this gadget's second or sixth columns.*

to indicate fixed robots and directed lines to indicate the path of the target robot. Next to each gadget we give the iconic representation of the gadget which we use in the next section. The basic gadgets are as follows.

1. A *one-way turn gadget* illustrated in Figure 3(a). To ensure that only a right turn is permitted, no other stationary robots may be placed in the same rows as this gadget to its left. Three other turn gadgets can be obtained by rotating this gadget 90, 180, and 270 degrees.
2. A *merge gadget* illustrated in Figure 3(b) for the case in which two incoming paths are merged to a single outgoing path. To ensure that the paths are merged as intended, no other stationary robots may be placed in the same rows as this gadget to its left. The gadget can be trivially extended to merge any arbitrary number of incoming paths.
3. A *splitter gadget* illustrated in Figure 3(c). To ensure that the path is split as intended, no other stationary robots may be placed above this gadget's second or sixth columns, so that a stationary robot passing through this gadget does not attempt to go "up" rather than "down." A splitter gadget with an arbitrary number of outputs can be constructed as a tree of two-way splitter gadgets.

Note that these gadgets are constructed such that they are inherently "one-way" devices: A robot passing through one of these gadgets can never attempt to reverse its path. For example, immediately after the mobile robot has turned right in the one-way gadget illustrated in Figure 3(a), it cannot reverse its path since, by design, there is no stationary robot to its left in this row. The cases for merge and splitter gadgets are analogous.

Other larger gadgets can be built using these three basic gadgets as "building blocks." In particular, the merge and splitter gadgets are used to build larger gadgets, and the one-way turn gadgets are used to "glue" these merge and splitter gadgets together. In assembling these basic gadgets into a larger gadget, it is important that no unintended interactions can occur. For example, if we intend for a mobile

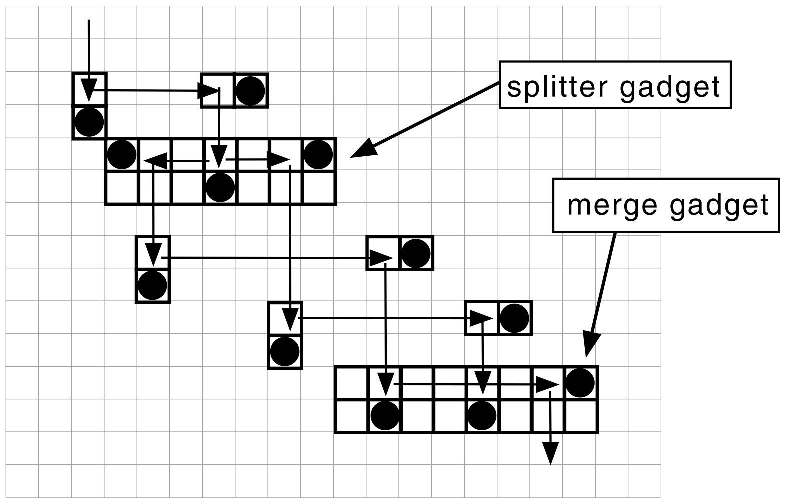

**Fig. 4** *A splitter gadget and a merge gadget occupying different rows and columns connected by one-way turn gadgets.*

robot to move down through a splitter gadget, we want to make sure that there is no opportunity for the robot to move up instead! To avoid such interactions, a larger gadget is constructed by laying out several basic gadgets along a diagonal of the puzzle board so that the merge and splitter gadgets do not share any rows or columns. Then, the output of one merge or splitter gadget is connected to the input of the next merge or splitter gadget using one-way turn gadgets.

A small example is illustrated in Figure 4. In this example, the mobile robot enters at the upper left. It then passes through a one-way turn gadget. Notice that the robot may only turn right at this point since there are no other robots to the left of this gadget. Now, the mobile robot enters another one-way turn gadget where it is forced to turn downwards. Next, the mobile robot enters a splitter gadget. Notice that the mobile robot can only move left or right inside this splitter gadget since we have ensured that there are no stationary robots placed above the two columns of the one-way turn gadget through which the stationary robot just passed. Assume, for example, that the mobile robot now moves to the right in the splitter gadget, stopping when it collides with the rightmost stationary robot in that gadget. Now the mobile robot may only move down since no stationary robots were allowed to be placed above this gadget in this column. Next, the mobile robot is forced to make two turns via one-way turn gadgets to arrive at a merge gadget. This merge gadget is placed so that it does not interact with any other merge or splitter gadgets. This big gadget that we've constructed isn't very useful, but it illustrates how basic gadgets can be placed on the board so as to interact only as intended.

Next, we describe the *lockable door gadget*. Conceptually, this gadget is a door that can be in either a locked or an unlocked configuration. The ability to set and test the configuration of this gadget allows us to store and retrieve a single bit of information. More specifically, the lockable door gadget has the following properties: The gadget has a one-way entrance and a one-way exit denoted the door entrance and exit, respectively. In addition, the gadget has a one-way entrance and a one-way exit denoted the lock entrance and exit, respectively. The gadget may be in one of two

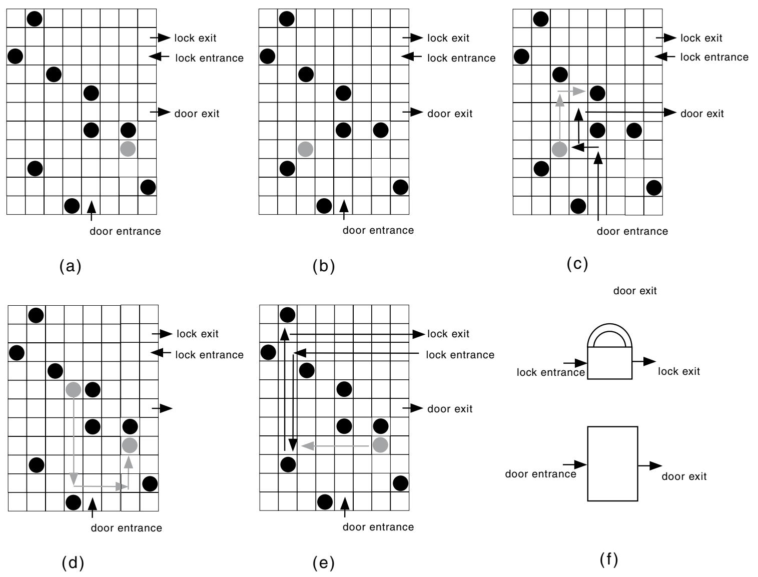

**Fig. 5** (a) *The lockable door in the locked configuration.* (b) *The lockable door in the unlocked configuration.* (c) *Target robot's path through the unlocked door.* (d) *Relocking sequence after passage through unlocked door.* (e) *Unlocking sequence for a locked door.* (f) *Iconic representations of the lock and door.*

configurations: the *locked configuration* or the *unlocked configuration*. If the gadget is in the locked configuration, the target robot cannot pass from the door entrance to the door exit. If the gadget is in the unlocked configuration, the target robot may pass from the door entrance to the door exit. Afterwards, the gadget automatically returns to the locked configuration. Finally, if the gadget is in the locked configuration, the target robot may enter the lock entrance, set the gadget to the unlocked configuration, and exit through the lock exit.

The gadget is implemented using nine stationary robots and one movable robot, the *lock robot*, which is distinct from the target robot. The gadget is shown in Figure 5. The stationary robots are indicated as dark circles and the movable lock robot is indicated as a gray circle. If the lockable door is in the locked configuration, shown in Figure 5(a), the target robot is unable to pass from the door entrance to the door exit. If the lockable door is in the unlocked configuration, shown in Figure 5(b), the target robot is able to pass from the door entrance to the door exit using the sequence of moves shown in Figure 5(c). After the target robot has passed through the door, the gadget can be returned to the locked configuration via the relocking sequence in Figure 5(d). Finally, a gadget in the locked configuration can be put into the unlocked configuration when the target robot enters the lock entrance, performs the sequence of moves shown in Figure 5(e), and leaves through the lock exit.

We have described above the intended use of the lockable door gadget. We now show that this gadget can only be used in this way. First, if the gadget is in the locked configuration in Figure 5(a), the target robot may enter via the door entrance but there is no possible next move. If the gadget is in the unlocked configuration in Figure 5(b), an unintended move is for the target robot to slide north into the door and for the lock robot to then slide east into the target robot. In this case, the lock robot may make up to three more moves (south, east, and north), but the target robot becomes stuck inside the gadget in all cases. Thus, we may assume that upon entering the unlocked gadget, the target robot moves north and then slides west into the lock robot. If the lock robot does not now perform the moves in Figure 5(c), then the target robot may repeatedly slide south, east, north, and then west, but again remains stuck within the gadget.

Next, we observe that after the target robot has passed through the unlocked gadget, the lock robot may remain at its current location or make up to three moves (south, east, north) as shown in Figure 5(d), culminating in the locked configuration. If the robot makes fewer than these three moves, the target robot cannot subsequently pass through the door and thus the gadget is effectively locked.

If the target robot enters via the lock entrance, it may either immediately leave through the lock exit (west, north, east) or may move west and south. In the latter case, if the gadget is in the unlocked configuration, the only possible unintended move is for the target robot to slide east to the lock robot. In this case there is no possible next move and the target robot becomes stuck in the gadget. If the gadget is in the unlocked configuration, the only unintended moves are for the lock robot to perform some part of the north, east, south, west, north sequence which effectively locks the gadget.

Observe that by ensuring that a lockable door gadget is placed such that there are no other gadgets in its rows or columns (except for the one-way turn gadgets which are used to connect it to other gadgets in different rows or columns), it is easily seen that no unintended interactions with other gadgets can occur.

The iconic representations of the lock and door are shown in Figure 5(f). We note that although the lockable door is a single gadget, it will be convenient in the next section to draw the lock separately from the door.

## 4. PSPACE-Completeness of GLLV.

We are now ready to show that GLLV is PSPACE-complete. We must first show that GLLV is in PSPACE. In other words, we must show that a polynomial-space Turing machine can be constructed which takes as input the encoding of a GLLV puzzle and accepts the input if and only if the puzzle is solvable. A general technique for doing this is to appeal to a powerful result due to Savitch [8].

Savitch's theorem states that the set of problems solvable in polynomial space with deterministic Turing machines (PSPACE) is exactly equivalent to the set of problems solvable in polynomial space with nondeterministic Turing machines (known as *NPSPACE*). This result is perhaps surprising since the corresponding statement for *time* would be "P is the same as NP." This is not known to be true, and in fact the relationship between P and NP is one of the most famous open problems in computer science. Intuitively, the result that PSPACE = NPSPACE is plausible since space, unlike time, may be reused. Thus, a deterministic polynomial-space Turing machine can simulate a nondeterministic polynomial-space Turing machine by erasing and reusing portions of the tape during the simulation. The details of this simulation are clever and beautiful, and the reader is referred to [9], or any other good introductory text on the theory of computation, for the details.

Given an instance of GLLV on an $m \times m$ board, there are at most $4^{m^2}$ distinct configurations of the board that can arise during play since each of the $m^2$ cells can either be empty, contain the target robot, contain a mobile robot other than the target robot, or contain a stationary robot. Therefore, if the instance of GLLV is solvable, there exists a sequence of at most $4^{m^2}$ moves from the initial configuration to a solution. Therefore, we build a nondeterministic Turing machine which repeatedly "guesses" the next move of the puzzle. This move is then made within the encoding of the puzzle on the tape, using just the space allocated for the input. In order to ensure that the nondeterministic computation does not run forever, we use another part of the tape as a "timer" so that the computation ends if more than $4^{m^2}$ moves have been made. Note that $4^{m^2}$ can be stored in binary using $\log_2 4^{m^2} = O(m^2)$ tape cells. Thus, the total amount of space used of an input of size $n = m^2$ (for the $m \times m$ board) is $O(m^2)$, which is not just polynomial, but in fact linear, in the size of the problem. Therefore, we have shown the following.

Theorem 1. *GLLV is in PSPACE.*

Next we prove the second part of PSPACE-completeness; every problem in PSPACE is polynomial-time reducible to GLLV. As discussed in section 2, we can either attempt to reduce every problem in PSPACE to GLLV in polynomial time or reduce a single known PSPACE-complete problem to GLLV in polynomial time. The vast majority of completeness proofs take the latter approach since it is generally much easier and more convenient. Surprisingly, in this case it appears to be more convenient to show explicitly that every problem in PSPACE can be polynomial-time reduced to GLLV. Our approach is analogous to Cook's theorem, which shows explicitly that every problem in NP can be polynomial-time reduced to SAT.

Let $\Pi'$ be any problem in PSPACE. Let $M' = (Q, \Sigma, \Gamma, \delta, q_{\text{start}}, q_{\text{accept}}, q_{\text{reject}})$ be a polynomial-space Turing machine for problem $\Pi'$ and let $p(n)$ denote the polynomial amount of space used by $M'$ on an input of length $n$. Thus, given an input string $w$ of length $n$, $M'$ accepts or rejects $w$ using space $p(n)$. Without loss of generality, we assume that $\Sigma = \{0, 1\}$ and $\Gamma = \{0, 1, B\}$, where $B$ is the blank symbol. The construction can be performed identically for any finite alphabets $\Sigma$ and $\Gamma$.

We transform $M$ and $w$ in polynomial time into an instance of GLLV such that $M$ accepts $w$ if and only if the target robot can reach the target cell. Specifically, we construct an instance of GLLV which simulates the behavior of $M$ on $w$. To this end, we construct a *tape cell gadget* which simulates a tape cell in $M$. This gadget has $|Q| - 2$ entrances, one for each possible state (except for the special accepting and rejecting states) in which the Turing machine could be upon entering the corresponding tape cell. The gadget has $|Q|$ exits to each of its adjacent tape cell gadgets, one for each possible state in which the Turing machine could be after evaluating the transition function. The exit corresponding to the accepting state is routed directly to the target cell, corresponding to successful completion of the GLLV puzzle. The exit corresponding to the rejecting state is routed to a row or column in which there are no other robots, corresponding to unsuccessful completion of the GLLV puzzle. Each gadget internally stores the symbol at the corresponding tape cell as well as the entire transition function of the Turing machine.

The constructed GLLV instance comprises $p(n)$ tape cell gadgets, each occupying distinct rows and column in the grid to avoid unintended interactions between tape cells. Adjacent tape cells are connected using one-way turn gadgets. The first $n$ tape cell gadgets store the input string, $w$, and the remaining tape cell gadgets store the blank symbol $B$. The target robot enters the leftmost tape cell gadget at the entrance corresponding to the start state. A representation of this construction is shown in Figure 6, where each tape cell gadget is represented by a large square, the target robot is represented by a circle at upper left, and the target cell is represented by a small square at lower right. The upwards facing lines indicate the exits corresponding to the rejecting state.

Finally, we describe the construction of the tape cell gadget. The gadget comprises three groups of lockable door gadgets. The first group are called *guess* lockable doors.

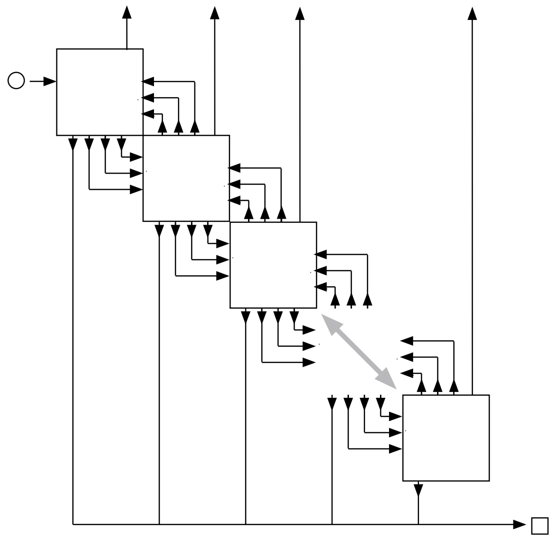

**Fig. 6** *Embedding of the Turing machine tape onto the GLLV grid. The circle at upper left represents the initial location of the target robot. The square at lower right represents the target cell. Upwards facing lines represent exits corresponding to the rejecting state.*

Conceptually, these lockable doors are used to store the state of the Turing machine and a "guess" of the symbol stored in the tape cell. The second group are the *symbol* lockable doors. These lockable doors are used to store and verify the symbol in the tape cell. There is one such lockable door for each of the three tape symbols. Before the target robot enters this tape cell gadget, exactly one of the symbol lockable doors is unlocked, storing the symbol written there. The third group are the *temporary storage* lockable doors. These lockable doors are used to temporarily store the current state and symbol.

The target robot enters the gadget in one of the $|Q| - 2$ entrances, representing the current state of the Turing machine. In Figure 7 only two entrances, $q$ and $r$, are shown for simplicity. Each entrance $i$ is connected via a splitter gadget to the lock components of guess lockable doors $G_i^0$, $G_i^1$, and $G_i^B$, all of which are initially locked. The target robot may unlock only one of these lockable doors, corresponding to storing the state $i$ of the Turing machine and a "guess" of the symbol at this cell.

Next, the outputs of all of the locks are merged using a merge gadget. The resulting line is then split, using a split gadget, to the door components of the symbol lockable doors $S^0$, $S^1$, and $S^B$. Prior to entering the tape cell gadget, exactly one of these lockable doors is unlocked, representing the symbol stored at this tape cell. Thus, the target robot can only pass through the door corresponding to the symbol at this cell. Recall that once the target robot passes through a door, the door automatically returns to the locked configuration.

From the output of each $S^\sigma$, $\sigma \in \{0, 1, B\}$, a splitter is used to connect to the door components of $G_i^\sigma$ for all states $i$. Only one of these doors is unlocked by merit of the target robot's initial guess. Thus, the target robot may pass through the door $G_i^\sigma$ if and only if it entered the tape cell at entrance $i$ (corresponding to state $i$) and correctly guessed that the tape cell contains symbol $\sigma$. The exit of this door is directly connected to the lock entrance of the temporary storage lockable door gadget $M_i^\sigma$.

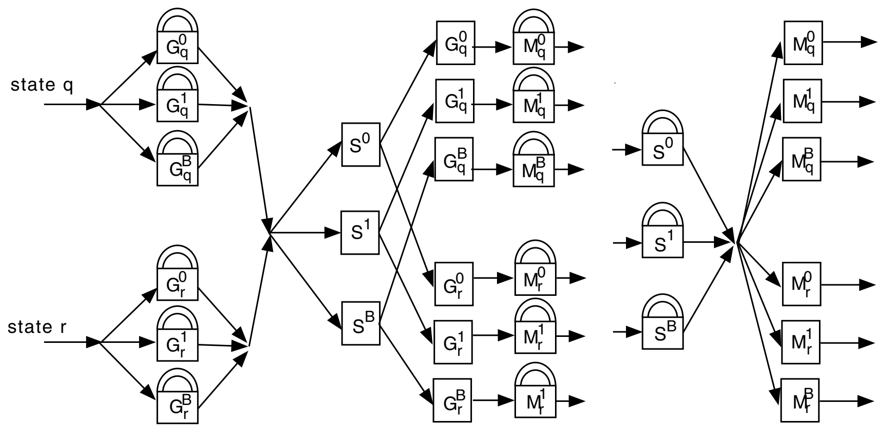

**Fig. 7** *The tape cell gadget. For simplicity, only two input lines, $q$ and $r$, are shown.*

Passing through this lock unlocks this door, thereby storing the fact that the machine is in state $i$ with symbol $\sigma$ under the tape head.

Now that the state and symbol are known, the transition function is applied. A line representing symbol $\sigma$ and state $i$ is connected to the entrance of the lock component of the lockable door $S^\tau$, where $\tau$ is the new symbol for this tape cell as specified by the transition function. This unlocks this symbol lockable door, thereby storing this symbol in the tape cell.

The outputs from the symbol lockable doors are now merged and then immediately split to all of the temporary storage door entrances. The target robot can only pass through the door $M_i^\sigma$ if it was in state $i$ with $\sigma$ at the tape cell before applying the transition function. Thus, the temporary storage doors allow us to retain the state and original tape symbol, even after the new symbol has been stored at the tape cell. Now the transition function is applied again to the original state and symbol to determine the new state and tape head direction. Thus, from the door exit of $M_i^\sigma$, there is a connection to the appropriate one of $2|Q|$ exits from the tape cell gadget.

Notice that the tape cell gadget is constructed using merge, splitter, and lockable door gadgets. As discussed earlier, these gadgets can be placed in separate rows and columns of the puzzle board and connected by one-way turn gadgets so as to avoid any unintended interactions.

Finally, we note that this reduction is easily verified to take time polynomial in the length of the input. To see this, note that each tape cell gadget has some constant size which depends only on the number of symbols in $\Gamma$ (in this case three) and the number of states in the Turing machine, but not on the length $n$ of the input string. Thus, each tape cell gadget can be constructed in constant time. There are $p(n)$ such tape cell gadgets which must be constructed. Thus, the entire construction requires only polynomial time. Therefore, we have shown the following.

THEOREM 2. *Every problem in PSPACE is polynomial-time reducible to GLLV.*

Together, Theorems 1 and 2 imply that GLLV is PSPACE-complete.

## 5. Conclusion.

In this paper we have described a new result showing that a generalization of the Lunar Lockout game, called GLLV, is PSPACE-complete. The fundamental mechanisms used in our GLLV PSPACE-completeness proof are the basic connector gadgets (one-way turn, merge, and split) and the lockable door gadget.

From these gadgets we can assemble the tape cell gadget. Using tape cell gadgets and the basic connector gadgets, we can then build an emulator for a polynomial-space Turing machine.

We note that in laying out the tape cell gadget on the grid, some paths may intersect (see Figure 7). In GLLV, a robot slides either horizontally or vertically until it hits another robot. Therefore, there is no worry that a robot traveling horizontally, for example, will turn vertically (midway through its slide) onto an intersecting perpendicular path. On the other hand, in some motion planning puzzles the robot takes only one step at a time and may move in any of four directions. In such puzzles, a *crossover* gadget may be required to ensure that the robot travels in the intended direction. We see, therefore, that if the basic connector gadgets, a crossover gadget, and a lockable door gadget can be constructed under the rules of a given motion planning problem, the problem can be shown to be PSPACE-complete.

It is not known how to build all of these gadgets for GLL if stationary robots are not permitted. Stationary robots were heavily used in our gadgetry for the PSPACE-completeness proof of GLLV. In particular, if all robots are mobile, then the gadgets that we have described may interact with one another in a number of unintended ways. The problem of whether GLL is also PSPACE-complete remains an interesting open problem.

In addition to the Lunar Lockout problem discussed here, this general technique has been used by Dor and Zwick [4] and by Culberson [2] to show that motion planning puzzles related to the Sokoban puzzle are PSPACE-complete. Flake and Baum have recently shown that the generalization of another popular puzzle, Rush Hour, is also PSPACE-complete [5]. A number of other motion planning puzzles, as well as a variety of other puzzles and games, have been studied in recent years. The interested reader is referred to the excellent survey by Demaine [3]. For more general background reading on NP- and PSPACE-completeness, we refer the reader to the books by Garey and Johnson [6], Sipser [9], and Papadimitriou [7].

**Acknowledgments.** The authors wish to thank the anonymous reviewers for their valuable suggestions, which substantially improved the exposition of this paper. In addition, numerous students at Harvey Mudd College provided suggestions that also improved this paper.

## REFERENCES

[1] S. Cook, *The complexity of theorem-proving games*, in Proceedings of the Third ACM Symposium on Theory of Computing, ACM, New York, 1971, pp. 151–158.

[2] J. Culberson, *Sokoban is PSPACE-complete*, in Proceedings of the International Conference on Fun with Algorithms, 1998, pp. 65–76.

[3] E. Demaine, *Playing games with algorithms: Algorithmic combinatorial game theory*, in Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science, 2001.

[4] D. Dor and U. Zwick, *Sokoban and other motion planning problems*, Comput. Geom., 13 (1999), pp. 215–228.

[5] G. Flake and E. Baum, *Rush hour is PSPACE-complete or "why you should generously tip parking attendants,"* Theoret. Comput. Sci., 270 (2002), pp. 895–911.

[6] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, CA, 1979.

[7] C. Papadimitriou, *Computational Complexity*, Addison-Wesley, Reading, MA, 1994.

[8] W. Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. System Sci., 4 (1970), pp. 177–192.

[9] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing, Boston, MA, 1997.