ELSEVIER

Theoretical Computer Science 270 (2002) 895–911

Theoretical Computer Science

www.elsevier.com/locate/tcs

Mathematical Games

# *Rush Hour* is PSPACE-complete, or "Why you should generously tip parking lot attendants"

Gary William Flake *, Eric B. Baum

*NEC Research Institute, 4 Independence Way, Princeton, NJ 08540, USA*

Received June 1999; revised February 2001; accepted February 2001
Communicated by A. Fraenkel

**Abstract**

*Rush Hour* is a children's game that consists of a grid board, several cars that are restricted to move either vertically or horizontally (but not both), a special target car, and a single exit on the perimeter of the grid. The goal of the game is to find a sequence of legal moves that allows the target car to exit the grid. We consider a slightly generalized version of the game that uses an $n \times n$ grid and assume that we can place the single exit and target car at any location we choose on initialization of the game.

In this work, we show that deciding if the target car can legally exit the grid is PSPACE-complete. Our constructive proof uses a lazy form of dual-rail reversible logic such that movement of "output" cars can only occur if logical combinations of "input" cars can also move. Emulating this logic only requires three types of devices (two switches and one crossover); thus, our proof technique can be easily generalized to other games and planning problems in which the same three primitive devices can be constructed. 

*Keywords:* Games; PSPACE-completeness; Reversible logic; Motion planning; Dual-rail logic

## 1. Introduction

*Rush Hour* [1] is a children's game (for ages 8 and above) that is played on a square grid (see Fig. 1). The object of the game is to move cars on the grid in such a way that a special car, the *target car*, is allowed to leave the grid through a single exit located on the perimeter of the grid. Cars may occupy either 2 or 3 grid cells (depending on the car size) and are restricted to move either vertically or horizontally (depending on the initial orientation) but not both. Initial configurations of the game

* Corresponding author.
*E-mail addresses:* flake@research.nj.nec.com (G.W. Flake), eric@research.nj.nec.com (E.B. Baum).

[1] The name "Rush Hour" is a trademark of Binary Arts, Inc.

0304-3975/02/$ - see front matter 
PII: S0304-3975(01)00173-6

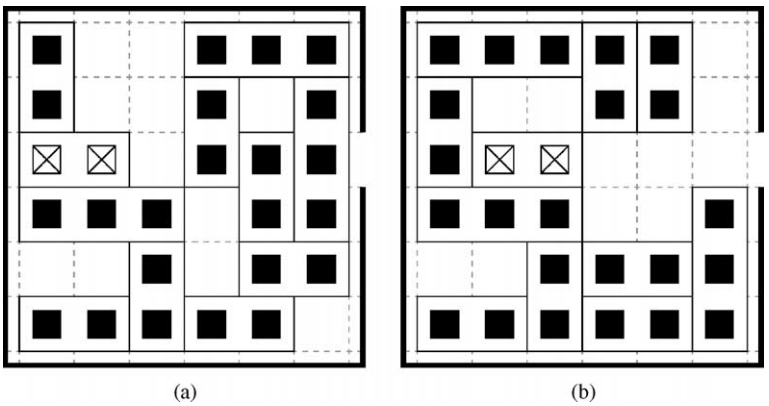


Fig. 1. An instance of the original Rush Hour game: (a) an initial configuration, (b) one possible solution configuration. The target car is the car drawn with crosses.

usually contain a traffic jam that prohibits the target car from exiting the grid. Even when played on a $6 \times 6$ grid, solutions may take over 40 moves in order to free the target car; hence, Rush Hour, while simple to play, can reflect subtle and complicated dependencies among the cars.

We show that a slightly generalized version of Rush Hour (GRH) is PSPACE-complete. While this basic result is not too surprising, given the status of other motion planning problems [19, 9, 17, 5], our proof technique may offer a simpler method for showing that related problems are also PSPACE-complete. Our proof works in two steps. First, we show that GRH can emulate a lazy form of reversible dual-rail logic via three primitive devices. Second, we show that a lazy reversible random access machine can be built from the three primitive devices. Our units are "lazy" in the sense that no particular car movement is ever required at any time step; however, the only possible car movements are those that maintain logical consistency between input and output cars. As a result, we can construct GRH circuits in such a way that the only way to free the target car is to emulate a reversible dual-rail random access machine.

Since the three primitive devices that we construct (two types of switches and one crossover unit) are extremely simple, one can construct similar devices within other game and motion planning frameworks. Because of the generality of our constructive proof, any game or motion planning problem that supports our three primitive devices will also be PSPACE-complete.

This work is divided into five sections. Section 2 describes Rush Hour in greater detail, contains a more rigorous definition of GRH, and briefly describes related work. In Section 3, we construct our three primitive devices and build more complicated gadgets from the primitives that are sufficient to emulate combinatorial Boolean logic. Section 4 focuses on the problem of emulating recurrent circuitry. We show that with a memory buffer and a simple control unit (both built from our primitive devices), GRH is capable of emulating a general-purpose computing device and is, therefore, PSPACE-complete. Finally, Section 5 summarizes our results and gives our conclusions.

## 2. Background

Previous works on the computational complexity of motion planning have often used elaborate proof techniques that are very specific to the underlying problem formulations. Our goal in this work is not to merely prove the complexity status of another motion planning problem; instead, we emphasize our proof technique, which may be generalized into other motion planning domains. To give this work the proper context, we first define our problem domain and then briefly sample earlier related works.

### 2.1. Rush hour

Rush Hour, in its original form, is played on a $6 \times 6$ grid. An initial configuration consists of several cars, each of which has a size that is either 2 or 3, a position that neither overlaps with any other car nor allows any portion of the car to be outside the grid, and an orientation that is either vertical or horizontal. Additionally, there is an exit located near the center of the right edge of the grid from which only the target car may move through. Cars can be moved one at a time and only into empty spaces. Moreover, cars can never change orientation. The goal of the game is to move the cars so that the target car may exit the grid.

Figs. 1(a) and (b) show a single instance of Rush Hour in the initial state and in the solution state, respectively. To achieve this particular solution, seven car movements had to be performed. Other, more complicated configurations may require dozens of car movements, with some cars being forced to move back and forth multiple times in order to free the target car.

Generalized Rush Hour (GRH) is a variant of the original game with two simple modifications. First, we allow the grid size to be a rectangle of arbitrary width and height. Second, we allow the exit for the target car to be at any location on the perimeter of the grid.

**Definition 1.** A *GRH instance* is a tuple $\langle w, h, x, y, n, \mathscr{C} \rangle$ such that:

- $(w, h) \in \mathbb{N}^2$ are the grid dimensions;
- $(x, y) \in \mathbb{N}^2$ are the coordinates of the exit which must lie on the perimeter of the grid;
- $n \in \mathbb{N}$ is the number of non-target cars;
- $\mathscr{C} = \{c_0, \ldots, c_n\}$ is a set of $n+1$ car tuples $c_i = \langle x_i, y_i, o_i, s_i \rangle$, $(x_i, y_i) \in \mathbb{N}^2$ are the car coordinates, $o_i \in \{N, S, E, W\}$ is the car orientation, $s_i \in \{2, 3\}$ is the car size, and $c_0$ is the target car.

Note that $\mathscr{C}$ must be consistent in the sense that all cars are properly contained within the grid perimeter and that no two cars overlap.

**Definition 2.** A *GRH solution* consists of a sequence of $m$ moves, where each move consists of a car index, $i$, a direction that is consistent with the initial orientation of

$c_i$, and a distance. Each move, in sequence, must be consistent with itself and with the previous configuration prior to the move; moreover, in order to move a distance, $d$, the configuration must be consistent for all $d', 0 \leqslant d' \leqslant d$ (thus, "teleportation" over obstacles is not allowed).

We consider two variants of GRH. The path version of GRH explicitly requires a solution path and the decision problem version of GRH asks only if such a solution exists. As will be shown later, the path version of the game may have a solution description that is exponential in the input description.

### 2.2. Related work

The earliest PSPACE-hardness result in motion planning is due to Reif [19] who constructively proved the result for the generalized mover's problem. The task in this case consists of moving a collection of linked polyhedral bodies (the object), through a finite polyhedral path, and ending at a specific target location. Reif's proof uses a three-dimensional tunnel and a three-dimensional multi-armed object. The tunnel encodes a Turing machine (TM) state transition rule set, the position of the arms encode the content of the TM's tape, and the position of the object relative to the tunnel corresponds to the TM's current state.

The Warehouseman's Problem is very similar to Rush Hour in that both deal with motion planning of rectangular objects. However, in the Warehouseman's Problem, one is allowed to rotate objects, unlike the fixed object orientation in Rush Hour. Like Rush Hour, the Warehouseman's Problem, is PSPACE-complete [9]. The constructive proof contains a reduction from the symbol transposition problem, and uses objects of many different sizes; in fact, the proof fails if all objects have the same size [17].

The generalized 15-puzzle game is also structurally similar to Rush Hour in that game pieces are restricted to movements in a rectangular lattice (with the goal of moving all pieces to target locations simultaneously). However, generalized 15-puzzle game pieces are square in shape and are allowed to move both vertically and horizontally. Deciding if a solution exists is in P, but finding the shortest solution is NP-complete [18].

Finally, Sokoban is a game in which a porter moves around a rectilinear maze while pushing barrels onto target locations. Sokoban was proved PSPACE-complete in [5], which used an ingenious construction of a TM. Interestingly, Sokoban is inherently irreversible because movements of the porter can render a solvable instance insolvable.

## 3. GRH is NP-hard

We now consider how to construct GRH configurations that correspond to combinatorial Boolean circuits (i.e., Boolean circuits without recurrent connections). In the first subsection, we describe the basic idea behind all of our devices, namely, that each device requires car configurations that are surrounded by a frozen core of deadlocked

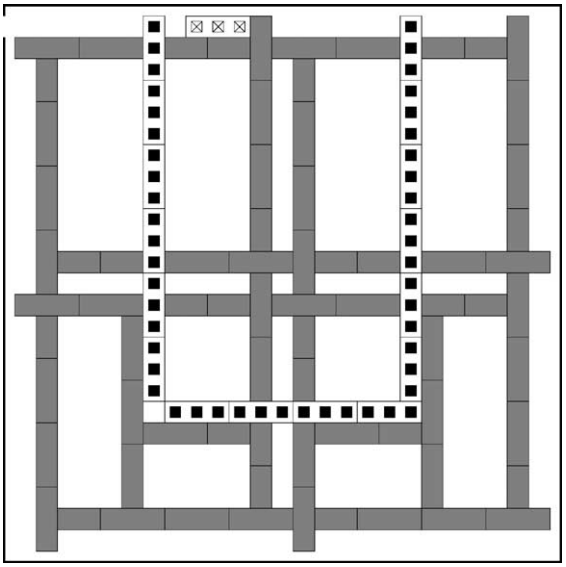

Fig. 2. A GRH block that demonstrates block tiling, simple plumbing, output gating, and input constraints: the target car (shown with the crosses) can exit the grid only when the left trigger line opens. The two vertical trigger lines are constrained so that only one can open at any given time; thus, the two trigger lines can be used to encode a ternary value.

cars, and illustrate how this frozen core can be constructed. In the Section 3.2, we define three primitive GRH devices that can be combined to form more complicated devices. In Sections 3.3 and 3.4, we show how the devices can be constructed to emulate dual-rail logic, thus proving that GRH is NP-hard.

### *3.1. Inductive constraints for cells*

The key to performing logic in GRH is to only allow very specific types of car movements. Since GRH has no notion of a fixed object (other than the perimeter of the grid) we create small areas of fixed cars by essentially building traffic jams that are anchored to the perimeter of the grid. Let the term *packed line* refer to a column or row of cars that has no spacing between cars and with all cars oriented either vertically or horizontally, respectively. A packed line is *anchored* when it is impossible for any car in the packed line to move. A *constrained line* is a line of cars that can shift by at most two cells.

Anchoring a packed line can be done in several ways. Each end of the packed line must touch two blocking obstacles that can be any combination of a perimeter wall, another anchored line, or a constrained line. In the last case, the packed line must touch the constrained line at a location where no possible shifts of the constrained line can permit an end car of the packed line to escape.

Fig. 2 illustrates a simple GRH configuration that has many properties that are in our constructions. The configuration consists of four *anchored rooms*. The gray blocks

are all anchored because both end points touch other anchored points. The two lines of cars in the middle are *trigger lines* that can be used to pass information through the grid. In most cases, a trigger line will only need to shift by one cell in order to pass information; in no case does a trigger line need to shift more than two cells.

Constrained rooms can be easily packed to one another in a tiled manner. The packed rooms are guaranteed to maintain their anchoring points as long as the external anchoring points are anchored to the perimeter wall. Hence, we can inductively conclude that an entire grid configuration made of constrained rooms maintains its structural integrity as long as the assumptions concerning the external anchor points and the external trigger lines all are fulfilled.

### 3.2. Primitive GRH devices

All information in our GRH constructions will ultimately be transmitted in the form of a car having (or not having) the ability to back up one cell. Emulating non-trivial functions in this manner requires that trigger lines be allowed to intersect (to allow for non-planar circuits) and combined via simple switches.

#### 3.2.1. CROSSOVER *block*

Figs. 3(a)–(d) show a CROSSOVER block in four states. In the figures, the "input" ends of the triggers are found on the left and bottom sides of the constrained room and the "outputs" reside on the top and right sides. The dark gray cars are anchored while the light gray cars are only constrained. In this construction, the anchored and constrained cars are sufficient to guarantee that the "outputs" of the trigger lines can only be open (i.e., move in) if, and only if, the corresponding input line is also open (i.e., pulled out). Moreover, the construction is sound in the sense that the constrained and anchored portions cannot be disassembled through any movement of the trigger lines.

Thus, in the CROSSOVER block, the two intersecting trigger lines behave exactly like two independent trigger lines: an output can open only if its input is open, and an input can close only if its output is closed. In no case does the state of one trigger line in the CROSSOVER block interfere with the other.

#### 3.2.2. BOTH *and* EITHER *blocks*

The two remaining building blocks perform simple switching. While the two blocks resemble AND and OR gates, neither is sufficient to perform Boolean logic in the strictest sense. As such, we refer to these two building blocks as a BOTH block and an EITHER block, respectively. Figs. 4(a) and (b) show the two block types in the closed state. The BOTH block requires that both inputs be open for the output to open. If either input is closed, then the output must be closed. The EITHER block can open if either or both of its inputs are open.

The validity of the BOTH block can be easily verified because it is constructed entirely with anchor and trigger lines. The validity of the EITHER block is a little more difficult to

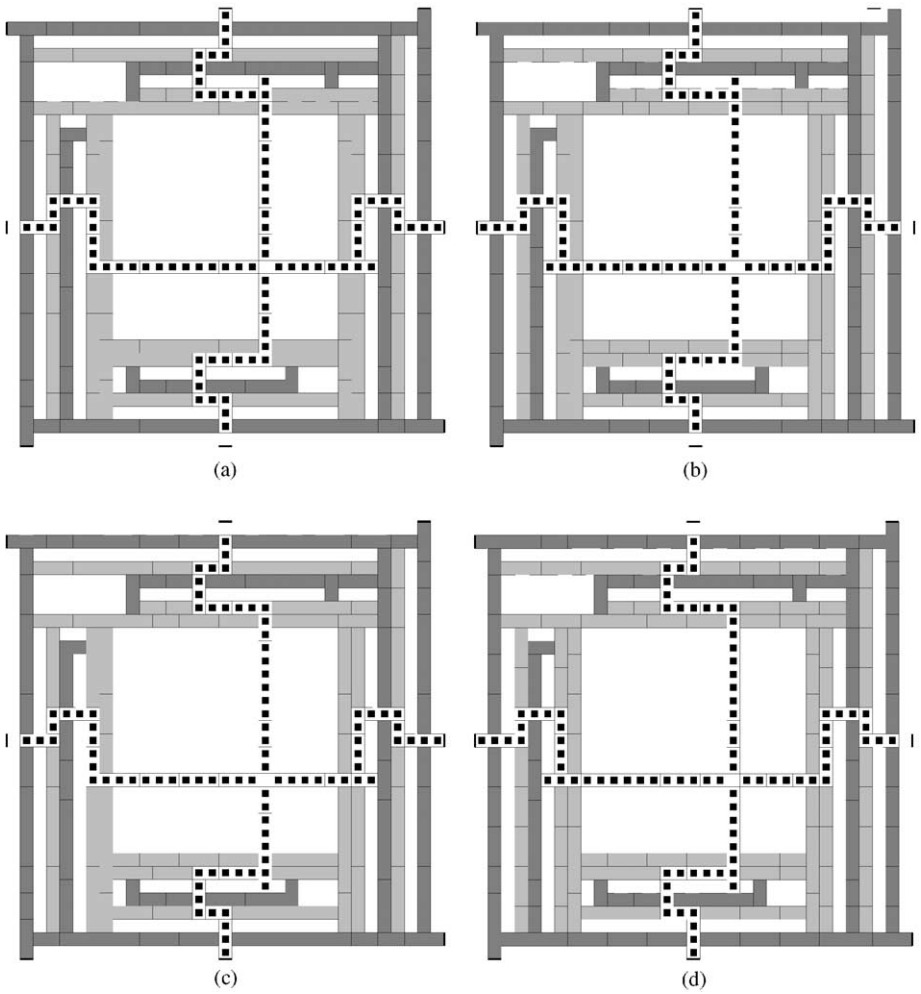


Fig. 3. A CROSSOVER block in four different states: (a) both closed (b) left open, bottom closed, (c) left closed, bottom open, (d) both open. Dark gray cars are anchored and light gray cars are constrained.

verify. Note that the main section of the block (where the bulk of the trigger lines are located) is surrounded by a core of constraining lines. These, in turn, are surrounded by anchor lines; thus, it is clearly impossible for any of the trigger or constrained lines to escape from the anchor lines. If it were possible for the output to open with both inputs closed, the output trigger line would have to shift down by two cells into the main chamber. However, due to the construction, there is simply no way for this to happen while both inputs are closed.

Since the dimensions of the constraining rooms of all the primitive blocks are identical, and since the inputs and output triggers are located in the same position relative

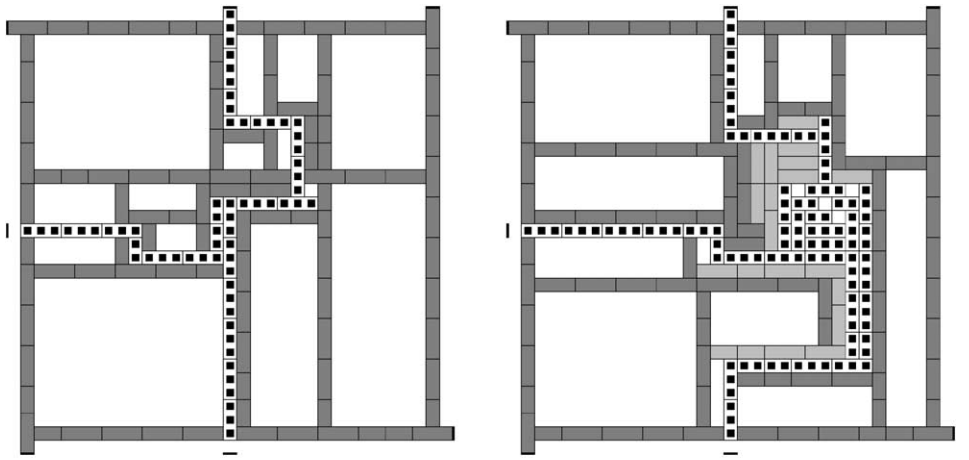


Fig. 4. (a) BOTH block and (b) EITHER block: inputs are on the left and bottom sides, outputs are on the top. Dark gray cars are anchored and light gray cars are constrained. A SPLIT block (which duplicates a trigger line) is equivalent to an upside-down BOTH block so that the roles of the inputs and outputs are reversed.

to the perimeter of the constraining rooms, the primitives can be tiled together in a meaningful manner. [2]

### 3.2.3. Block representations and reuse

Because all of our blocks have well-defined inputs and outputs, and since each I/O trigger line must assume one of the two states, each block type can be unambiguously described by a finite state machine (FSA) where each FSA state contains all block trigger states at a single instance and the FSA transitions represent legal movements among the cars. In this FSA representation, transitions can only occur between states that differ by one trigger movement. Later, we use an FSA representation to show how blocks can be used in atypical ways by mixing the roles of input and output trigger lines.

Having our blocks be instances of a more abstract FSA representation is a key to simplifying our proof and generalizing our result. Thus, showing that another system supports our three basic devices is equivalent to showing that the underlying FSA is identical.

In this spirit, we also reuse the BOTH block for the purpose of duplicating trigger signals by reversing the roles of the inputs and output. While we refer to this device as a SPLIT block, one can easily verify that it is equivalent to an inverted (upside-down)

[2] Tromp [22] has improved our constructions by showing that basic blocks identical in function to our own can be constructed using cars of only size two.

BOTH block by constructing the equivalent FSA and inverting the trigger states in the FSA composite states.

### 3.3. GRH Logic

While it is tempting to think of the GRH states *open* and *closed* as mapping to Boolean **true** and **false**, respectively, the act of blocking the motion of a car can never actually trigger an event—only a non-event. With such a mapping, it would be impossible to build an inverter gate, that is, a GRH block that only opens when its input is closed. We get around this problem by mapping Boolean values to a pair of trigger lines, similar to the dual-rail logic used in self-timed circuits [6]. In this framework, one line represents **true** and the other **false**; thus, both states can potentially trigger events in other blocks, and inversion becomes a simple matter of crossing over the two trigger lines. However, care must be taken to properly handle non sense values such as both lines being open or both lines being closed.

**Definition 3.** A *trigger state*, $X$, is denoted $X^+$ or $X^-$ to indicate that $X$ is opened or closed, respectively. Thus, the superscripts $+$ and $-$ can be considered functions that map trigger states to Boolean values, such that each superscript is the inverted function of the other.

When thinking about trigger line states, it may be useful to read $X^+$ to mean "$X$ is currently open or may be backed up by one cell length into the open position". Similarly, $X^-$ can be read as "$X$ is currently forced into the closed state". Thus, being in the open state indicates the potential for movement, while being in the closed state indicates that movement is impossible without a change in the preconditions of the trigger line.

**Definition 4.** Let $X \dot{\wedge} Y \to Z$ be an operator that represents whether the output of a BOTH block can open with the two input trigger lines $X$ and $Y$. Logically, the $\dot{\wedge}$ operator is described by

$$Z = X \dot{\wedge} Y \equiv \begin{cases} Z^+ & \text{if } X^+ \wedge Y^+, \\ Z^- & \text{if } X^- \vee Y^-. \end{cases}$$

We refer to $\dot{\wedge}$ as the BOTH operator.

**Definition 5.** Let $X \dot{\vee} Y \to Z$ be an operator that represents whether the output of an EITHER block can open with the two input trigger lines $X$ and $Y$. Logically, the $\dot{\vee}$ operator is described by

$$Z = X \dot{\vee} Y \equiv \begin{cases} Z^+ & \text{if } X^+ \vee Y^+, \\ Z^- & \text{if } X^- \wedge Y^-. \end{cases}$$

We refer to $\dot{\vee}$ as the EITHER operator.

**Definition 6.** A dual-rail *state* is a tuple $\bar{X} = \langle X_T, X_F \rangle$, where $X_T$ and $X_F$ represent the **true** and **false** trigger lines of $\bar{X}$, respectively. The four possible dual-rail states are **true** $\equiv \langle X_T^+, X_F^- \rangle$, **false** $\equiv \langle X_T^-, X_F^+ \rangle$, **null** $\equiv \langle X_T^-, X_F^- \rangle$, and **tautology** $\equiv \langle X_T^+, X_F^+ \rangle$.

Later, we show how **tautology** is always prohibited from our constructions; however, we allow **null** to appear in our GRH constructions for transitions from **true** to **false**.

**Definition 7.** The GRH logic operators, $\bar{\wedge}, \bar{\vee}$, and $\bar{\neg}$ (respectively, AND, OR, and NOT), are defined as follows:

$$\bar{X} \bar{\wedge} \bar{Y} \equiv \langle X_T \dot{\wedge} Y_T, X_F \dot{\vee} Y_F \rangle,$$

$$\bar{X} \bar{\vee} \bar{Y} \equiv \langle X_T \dot{\vee} Y_T, X_F \dot{\wedge} Y_F \rangle,$$

$$\bar{\neg} \bar{X} \equiv \langle X_F, X_T \rangle.$$

Note that when $\bar{X}$ and $\bar{Y}$ are restricted to values in {**true**, **false**, **null**}, then the GRH logic operators implement ternary logic, a proper superset of Boolean logic.

Fig. 2 illustrates the only remaining parts needed to show that GRH is NP-hard. In the figure, the two vertical trigger lines are constrained so that only one of the two can be open (down) at any time. As a result, the trigger lines can only assume a composite state that is consistent with ternary logic. Also shown in the figure is a target car that can exit only when the left trigger line is open. In this way, one can constrain a target car so that it can exit only if some ternary expression is satisfied.

**Theorem 1.** *GRH is NP-hard.*

**Proof.** To reduce SAT to GRH, we build a GRH configuration that implements a SAT expression, $E$, with the GRH logic operators from Definition 7. For every variable in the SAT expression, a corresponding GRH input pair is used, as shown in Fig. 2, which restricts the input values to legal ternary states. The output **true** trigger line of the GRH configuration output restricts the single target car.

In order for the target car to escape the GRH grid, GRH configuration must be able to open the **true** trigger line of the output. Since this can only happen when the inputs of the GRH configuration assume states that correspond to satisfying values for $E$, the target car can be freed if and only if a satisfying set of variable assignments exists for $E$. □

## 4. GRH is PSPACE-complete

We now turn our attention to the more interesting question of how to emulate arbitrary recurrent circuits within GRH. We introduce a lazy reversible dual-rail machine (LRD) that is capable of emulating a finite memory random access machine (RAM)

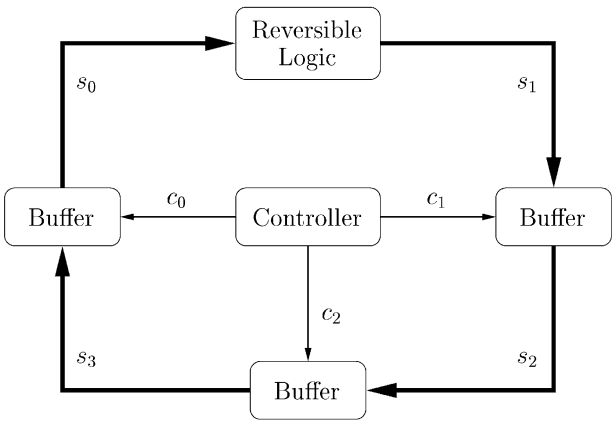


Fig. 5. A LRD machine emulates a reversible random access machine: the memory contents of the machine are contained within the trigger lines that pass through the buffers. The controller can produce signals that force the computation forwards or backwards.

and, hence, an irreversible linear bounded TM. While we build the LRD machine from GRH parts, we emphasize that our results apply to any problem domain that can emulate the CROSSOVER, BOTH and EITHER blocks. As a result, our construction of an LRD machine is actually a meta-construction, despite the fact that we illustrate specific components with GRH parts.

First, we give a general overview for how the LRD machine works. Afterwards, we examine the individual components of the LRD machine, which is followed by a brief example of a counter device implemented as an LRD machine. We conclude this section by confirming that our construction satisfies all of the requirements for PSPACE-completeness.

### *4.1. LRD machine*

Fig. 5 shows a rough sketch of the LRD machine, which basically emulates a finite memory RAM in a reversible manner. The flow of information is shown by the bold trigger path on the perimeter of Fig. 5. This path represents a fixed but very large number of trigger line pairs. In forward operation, the LRD machine calculates a next state (via the reversible logic) and propagates the next state clockwise, through the buffers, and ultimately back to the input side of the reversible logic.

At any given instant, the flow of information can be reversed so that the computation reverses, proceeding backwards (i.e., counter-clockwise). The direction of the computation is determined by the controller, which is explained in greater detail in the next subsection.

The buffers simply pass information. When the control signal into a buffer is asserted, the buffer's behavior is indistinguishable from a plain trigger line, that is, the buffer can only output values that are consistent with the inputs. By "consistent" we

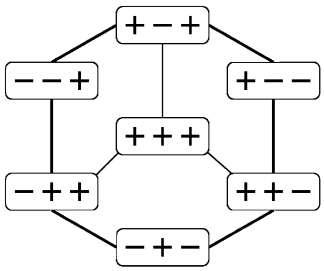


Fig. 6. FSA representation of the controller: the symbols "+" and "−" correspond to a control signal being asserted or negated, respectively. By design of the controller, at least one control signal must be asserted at all times, and bold lines represent possible transitions. By design of the reversible logic, the middle state "+++" is explicitly prohibited. Thus, the controller can only move (within the finite state diagram) clockwise or counter-clockwise along the perimeter, which corresponds to driving the reversible logic forwards or backwards.

mean that the output is either identical to the input or it is **null**, and never equal to **tautology**.

When the control signal is not asserted in a buffer, then a buffer's input and output may differ; however, both inputs and outputs are prohibited from assuming the **tautology** state. When a buffer's control signal is reasserted while a buffer's input and output differ, either the input or the output must change state (becoming identical) in order for the control signal assertion to complete. Otherwise, the control signal cannot be asserted and the computation in the LRD machine must stop.

In physical systems like Rush Hour, unbuffered recurrent circuits can give rise to circular dependencies that would not be present in the equivalent electrical circuit. This phenomenon happens because inputs may not be able to close until dependent outputs close first. For this reason, we need the buffer in our construction to allow for motion in the recurrent circuit that is not jammed due to circular dependencies. In this way, the combined buffer and controller allow for state transitions in a precise way.

### *4.2. Special devices*

#### *4.2.1. Controller*

The controller in the LRD machine is actually just an EITHER block used so that all three triggers are treated as the control input signals to the buffers. Referring back to Fig. 4b, note that at all times the EITHER block must have at least one trigger line pulled away from the block; in other words, it is impossible for all three triggers to be pulled into the block (inputs closed, with output open). As a result, the legal states of the controller are constrained to obey the finite state diagram shown in Fig. 6.

A deterministic reversible machine that is capable of emulating a linear bounded TM has the property that all configurations (the machine state and content of the tape) have at most one predecessor state and one successor state. Moreover, the initial and halting states can be made to have no legal predecessor and successor states, respectively. As a result, the finite state description of such a machine resembles a linear

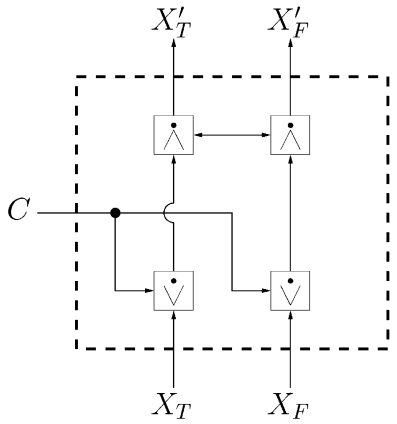


Fig. 7. A GRH buffer unit. The two BOTH blocks in the top-half share an input trigger line such that only one of $X'_T$ or $X'_F$ can be open at a given time. When the control signal, $C$, is closed (asserted, or moved inside of the block), then the inputs and outputs can only assume the same non-**null** values. When $C$ is open, the inputs and outputs may differ. The SPLIT device (shown as a black circle) is actually an upside-both BOTH block.

chain of states, with each non-terminal state having exactly one predecessor and one successor.

Looking back at the control unit, if all control lines were to be simultaneously asserted (i.e., the controller is in the "+++" state), then the LRD machine would have all of its memory trigger lines (in positions $s_0, s_1, s_2$, and $s_3$) in identical states. However, assuming that the reversible logic is constructed so that the initial state has no legal predecessor state and that the final state has no legal successor state, it is impossible for the controller to be in the "+++" state. For the controller to be in this state, the logic would have to be irreversible. We assume, by construction, that the logic in the LRD machine is reversible.

Because of these properties, the EITHER block works as a reversible timer since it always forces one of the buffers to be asserted (with its inputs and output identical). Moreover, referring back to Fig. 6, since the center state "+++" cannot be realized, the controller itself exists in a simple state loop such that movement in a clockwise direction drives the LRD machine into forward logic, and movement in a counter-clockwise direction drives the LRD machine into reverse logic.

### *4.2.2. Buffer*

A buffer unit from Fig. 5 is schematically shown in Fig. 7. Each of the three main buffers in Fig. 5 contains as many buffer units as there are bits in the memory of the LRD machine, and within one of the main buffers each individual buffer unit receives the exact same control signal via multiple SPLIT blocks. Thus, the control signal to a main buffer cannot be asserted until all sub-control signals are asserted in the individual buffer units.

Table 1
Information and control flow in a simple counter circuit

| Step | $s_0$ | $s_1$ | $s_2$ | $s_3$ | $c_0$ | $c_1$ | $c_2$ |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | + | + | − |
| 2 | 0 | 1 | 1 | 0 | − | + | − |
| 3 | 0 | 1 | 1 | 1 | − | + | − |
| 4 | 0 | 1 | 1 | 1 | − | + | + |
| 5 | 0 | 1 | 1 | 1 | − | − | + |
| 6 | 1 | 2 | 1 | 1 | − | − | + |
| 7 | 1 | 2 | 1 | 1 | + | − | + |
| 8 | 1 | 2 | 1 | 1 | + | − | − |
| 9 | 1 | 2 | 2 | 1 | + | − | − |
| 10 | 1 | 2 | 2 | 1 | + | + | − |

The buffers allow for discrepancies to exist between its inputs and outputs. Temporarily allowing such discrepancies is the only way for information to be propagated around the LRD machine. The buffer unit has a hidden state contained in the interior trigger lines between the EITHER and BOTH blocks. When the control line is negated, the hidden state may assume any value, including **tautology**. However, the two BOTH blocks only allow ternary values to escape through the buffer unit outputs. Since the two BOTH blocks share an input trigger, only one of the two output triggers can be opened at any time; thus, only legal values can escape from the buffer unit's output.

When the control signal is asserted, the hidden states of the buffer unit must be identical to the inputs. Therefore, the outputs of the buffer must be equal to the inputs or equal to **null**.

#### *4.2.3. Reversible logic*

The reversible logic core in Fig. 5 is similar to the combinatorial logic explored in Section 3. Since the logical core of the LRD machine effectively emulates an irreversible device, it must be built from reversible gates [20, 21] and use internal bookkeeping [11, 1] to insure that the entire computation can be reversed at any instant. While we omit these details to simplify the presentation, we note that this form of emulation has been well-established in the literature [12, 4, 14, 13, 8], and that the time and space bounds for emulation of an irreversible machine by a reversible machine can be done in time and space that is polynomial in the original requirements [2], or in linear space at the expense of having an exponential slowdown [10].

#### *4.2.4. Counter circuit example*

To better explain how the logical core interacts with the rest of the LRD machine components, consider Table 1 which illustrates how a simple counter circuit would increment its internal state. In the table, the states of the four sets of trigger lines are shown along with the transitions that occur in the control signals.

At step 2, $c_0$ is negated. Note that with $c_1$ still asserted, $s_1$ and $s_2$ must be equal. Furthermore, $s_0$ is also locked in place because it cannot assume a value which conflicts with $s_1$. (Since the logic implements a counter, $s_1 = s_0 + 1$ will always be true.) As a result, the only memory state that can change is $s_3$. Conceivably, $s_3$ could assume any set of ternary values; however, only a value of 1 will allow $c_2$ to be asserted at step 4.

At step 5, $c_1$ is negated, which allows $s_0$ and $s_1$ to assume the values of 1 and 2, respectively. The information continues to flow clockwise, until at step 10 the LRD machine is identical to its initial state except that the memory contents have been incremented.

Clearly, the control signals could have been reversed at any instance, forcing the LRD machine to also reverse. However, the only way for a halting state to be realized is for the entire computation to be emulated by the LRD machine. Interestingly, the fact that a GRH configuration can only be solved by emulating a counter-circuit implies that the output description of a GRH solution can be exponential in the problem instance size.

**Theorem 2.** *GRH is PSPACE-complete.*

**Proof.** By construction of the LRD machine, deciding GRH is equivalent to emulating a general purpose computing device with finite memory. Due to the space results of emulating irreversible TM with reversible TM [2, 10], an LRD machine can be constructed so that the space requirements are quadratic in the input description (due to the planar construction). Moreover, since GRH and emulating an LRD machine are clearly in PSPACE, GRH is PSPACE-complete. □

## 5. Conclusions

We have shown that Generalized Rush Hour can express logical expressions in such a way that empty spaces in the GRH configuration can only propagate through the grid space if cars emulate the exact form of information flow that must occur in Boolean logic. Mapping arbitrary Boolean expressions into GRH configurations is a relatively simple task, and from this result we can conclude that GRH is NP-hard.

We have extended this result to show that GRH can emulate recurrent logic as well. Due to the explicit reversibility of all actions in GRH, we were required to examine the properties of reversible logic in order to show that GRH can emulate a general purpose computing device. Nevertheless, the emulation can be performed in time and space that is polynomial in the size of the device being emulated.

While this complexity result is not especially surprising given other well-known complexity results, we believe our proof technique to be of interest in its own right. In particular, by abstracting the emulation to the lazy reversible dual-rail machine, our proof should be applicable to any game or motion planning problem in which the three

basic devices (the CROSSOVER, EITHER, and BOTH blocks) can be emulated. For example, the PSPACE-completeness of the Warehouseman's Problem [9] is easily retrieved from our results.

While several other games are known to be PSPACE-complete, GRH is the simplest game that we know of that has this property. In particular, we note that GRH has no special purpose hardware built into it (such as switches, fixed objects, gates, timers, etc.). Moreover, the rules of the game are exceptionally simple and are clearly realizable in the classical world of physics (i.e., there is no "teleportation" of objects, or "transmutation" of pieces from one color to another, etc.).

We also note that GRH closely resembles other physically realizable reversible models of computation [7, 16, 3], especially the rod logic of [15]. GRH, or some variation of it, may actually be realizable as a physical system because none of our cars need move by more than four cells; thus, our results may provide additional motivation for studying reversible systems that can be built with nanotechnology.

## Acknowledgements

We thank Igor Durdanovic, Stephen Judd, Cris Moore, Harold Stone, John Tromp, and the anonymous reviewers for helpful comments, suggestions, and insights.

## References

[1] C.H. Bennett, Logical reversibility of computation, IBM, J. Res. Development 17 (6) (1973) 525–532.
[2] C.H. Bennett, Time/space trade-offs for reversible computation, SIAM J. Comput. 18 (4) (766–776).
[3] C.S. Calude, J. Casti, M.J. Dinneen (Eds.), Unconventional Models of Computation, Springer, Berlin, 1998.
[4] P. Crescenzi, C.H. Papadimitriou, Reversible simulation of space-bounded computation, Theoret. Comput. Sci. 143 (1995) 159–165.
[5] J. Culberson, Sokoban is pspace-complete, in: E. Lodi, L. Pagli, N. Santoro (Eds.), Proc. Int. Conf. on Fun with Algorithms, Elba, June, 1998, Proc. Informatics, Vol. 4, Carleton Scientific, Waterloo, Canada, 1999, pp. 65–76.
[6] M. Dean, T. Williams, D. Dill, Efficient self-timing with level-encoded 2-phase dual-rail (LEDR), in: C.H. Séquin (Ed.), Advanced Research in VLSI, MIT Press, Cambrdige, MA, 1991, pp. 55–70.
[7] M.P. Frank, Physically-motivated models of computation for complexity theory, MIT Reversible Computing Project Memo #M5, March 1997.
[8] M.P. Frank, M.J. Ammer, Separations of reversible and irreversible space–time complexity classes, extended abstract, CCC-98, 1997, submitted for publication.
[9] J.E. Hopcroft, J.T. Schwartz, M. Sharir, On the complexity of motion planning for multiple independent objects; pspace-hardness of the "warehouseman's problem", Internat. J. Robotics Res. 3 (4) (1984) 76–88.
[10] K.-J. Lange, P. McKenzie, A. Tapp, Reversible space equals deterministic space, Proc. 12th Annu. IEEE Conf. on Computational Complexity (CCC '97), June 1997, pp. 45–50.
[11] Y. Lecerf, Machines de Turing réversibles. Insolubilité récursive en $n \in N$ de l'équation $u = \theta^n$, où $\theta$ est un $\ll$ isomorphisme de codes $\gg$ (Reversible Turing machines. Recursive insolubility in $n \in N$ of the equation $u = \theta^n$, where $\theta$ is an "isomorphism of codes"). C. R. Acad. Sci. (Weekly Proc. Acad. Sci.) 257 (1963) 2597–2600.
[12] R.Y. Levine, A.T. Sherman, A note on Bennett's time–space tradeoff for reversible computation, SIAM J. Comput. 19 (4) (1990) 673–677.

[13] Ming Li, J. Tromp, P. Vitányi, Reversible simulation of irreversible computation by pebble games, Physica D 120 (1998) 168–176.
[14] Ming Li, P.M.B. Vitányi, Reversibility and adiabatic computation trading time and space for energy, Proc. Roy. Soc. London. A 452 (1996) 1–21.
[15] R.C. Merkle, Two types of mechanical reversible logic, Nanotechnology 4 (1993) 114–131.
[16] R.C. Merkle, K.E. Drexler, Helical logic, Nanotechnology 7 (4) (1996) 325–339.
[17] C. Papadimitriou, P. Raghavan, M. Sudan, H. Tamaki, Motion planning on a graph, IEEE Symp. on Foundations of Computer Science (1994) 511–520.
[18] D. Ratner, M. Warmuth, Finding a shortest solution for the $(n\times n)$-extension of the 15-puzzle is intractable, J. Symbolic Comput. 10 (1990) 111–137.
[19] J. Reif, Complexity of the mover's problem and generalizations, IEEE Symp. on Foundations of Computer Science (1979) 421–427.
[20] T. Toffoli, Reversible computing, Technical Memo MIT/LCS/TM-151, MIT Lab for Computer Science, February 1980.
[21] T. Toffoli, Reversible computing, in: J.W. de Bakker, J. van Leeuwen (Eds.), Automata, Languages and Programming (7th Colloquium, Noordwijkerhout, the Netherlands, Jul. 14–18, 1980), Lecture Notes in Computer Science, Vol. 85, Springer, Berlin, 1980, pp. 632–644.
[22] J. Tromp, private communication, July 1999.