1×1 Rush Hour with Fixed Blocks Is PSPACE-Complete

Josh Brunner
Massachusetts Institute of Technology, Cambridge, MA, USA
brunnerj@mit.edu

Lily Chung
Massachusetts Institute of Technology, Cambridge, MA, USA
ikdc@mit.edu

Erik D. Demaine
Massachusetts Institute of Technology, Cambridge, MA, USA
edemaine@mit.edu

Dylan Hendrickson
Massachusetts Institute of Technology, Cambridge, MA, USA
dylanhen@mit.edu

Adam Hesterberg
Massachusetts Institute of Technology, Cambridge, MA, USA
achester@mit.edu

Adam Suhl
Algorand, Boston, MA, USA

Avi Zeff
Massachusetts Institute of Technology, Cambridge, MA, USA
avizeff@mit.edu

Abstract

Consider $n^2 - 1$ unit-square blocks in an $n \times n$ square board, where each block is labeled as movable horizontally (only), movable vertically (only), or immovable – a variation of Rush Hour with only 1×1 cars and fixed blocks. We prove that it is PSPACE-complete to decide whether a given block can reach the left edge of the board, by reduction from Nondeterministic Constraint Logic via 2-color oriented Subway Shuffle. By contrast, polynomial-time algorithms are known for deciding whether a given block can be moved by one space, or when each block either is immovable or can move both horizontally and vertically. Our result answers a 15-year-old open problem by Tromp and Cilibrasi, and strengthens previous PSPACE-completeness results for Rush Hour with vertical 1×2 and horizontal 2×1 movable blocks and 4-color Subway Shuffle.

2012 ACM Subject Classification Theory of computation → Problems, reductions and completeness

Keywords and phrases puzzles, sliding blocks, PSPACE-hardness

Digital Object Identifier 10.4230/LIPIcs.FUN.2021.7

Related Version This paper is also available on arXiv at `https://arXiv.org/abs/2003.09914`.

Acknowledgements We thank the many colleagues over the years for their early collaborations in trying to resolve the 1×1 Rush Hour problem (when E. Demaine mentioned it to various groups over the years): Timothy Abbott, Kunal Agrawal, Reid Barton, Punyashloka Biswal, Cy Chen, Martin Demaine, Jeremy Fineman, Seth Gilbert, David Glasser, Flena Guisoresac, MohammadTaghi Hajiaghayi, Nick Harvey, Takehiro Ito, Tali Kaufman, Charles Leiserson, Petar Maymounkov, Joseph Mitchell, Edya Ladan Mozes, Krzysztof Onak, Mihai Pătraşcu, Guy Rothblum, Diane Souvaine, Grant Wang, Oren Weimann, Zhong You (MIT, November 2005); Jeffrey Bosboom, Sarah Eisenstat, Jayson Lynch, and Mikhail Rudoy (MIT 6.890, Fall 2014); and Joshua Ani, Erick Friis, Jonathan Gabor, Josh Gruenstein, Linus Hamilton, Lior Hirschfeld, Jayson Lynch, John Strang, Julian Wellman (MIT 6.892, Spring 2019, together with the present authors).

10th International Conference on Fun with Algorithms (FUN 2021).
Editors: Martin Farach-Colton, Giuseppe Prencipe, and Ryuhei Uehara; Article No. 7; pp. 7:1–7:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

In a ***sliding block puzzle***, the player moves blocks (typically rectangles) within a box (often a rectangle) to achieve a desired configuration. Such puzzles date back to the 15 Puzzle, invented by Noyes Chapman in 1874 and popularized by Sam Loyd in 1891 [13], where the blocks are unit squares. One of the first puzzles to use rectangular pieces is the Pennant Puzzle by L. W. Hardy in 1909, popularized under the name Dad's Puzzle from 1926, whose 10 pieces require a whopping 59 moves to solve [5]. In general, such puzzles are PSPACE-complete to solve, even for 1×2 blocks in a square box [7, 8], which was the original application for the hardness framework Nondeterministic Constraint Logic (NCL). For unit-square pieces (as in the 15 Puzzle), such puzzles can be solved in polynomial time, though finding a shortest solution is NP-complete [10, 3].

In the 1970s, two famous puzzle designers – Don Rubin in the USA and Nobuyuki "Nob" Yoshigahara (1936–2004) in Japan – independently invented [9] a new type of sliding block puzzle, where each block can move only horizontally or only vertically. The motivation is to imagine each block as a car that can drive forward and reverse, but cannot turn; the goal is to get one car (yours) to "escape" by reaching a particular edge of the board. The original forms – Rubin's "Parking Lot" [11] and Nob's "Tokyo Parking" [15] – imagined a poor parking-lot attendant trying to extract a car. Binary Arts (now ThinkFun) commercialized Nob's 6×6 puzzles as ***Rush Hour*** in 1996, where a driver named Joe is "figuring things out on their way to the American Dream" [16]. The physical game design led to a design patent [18] and many variations by ThinkFun since [16].[1] Computer implementations of the game at one point led to a lawsuit against Apple and an app developer [9].

The complexity of Rush Hour was first analyzed by Flake and Baum in 2002 [4]. They proved that the game is PSPACE-complete with the original piece types – 1×2 and 1×3 cars, which can move only in their long direction – when the goal is to move one car to the edge of a square board. In 2005, Tromp and Cilibrasi [17] strengthened this result to use just 1×2 cars (which again can move only in their long direction), using NCL. Hearn and Demaine [8, 6] simplified this proof, and proved analogous results for triangular Rush Hour, again using NCL. In 2016, Solovey and Halperin [14] proved that Rush Hour is also PSPACE-complete with 2×2 cars and immovable 0×0 (point) obstacles.[2] Unlike 1×2 cars, which have an obvious direction of travel (the long direction), 2×2 cars need to have a specified direction, horizontal or vertical.

Back in 2002, Hearn, Demaine, and Tromp [7, 17][3] raised a curious open problem: might 1×1 cars suffice for PSPACE-completeness of Rush Hour? Like 2×2 cars, each 1×1 car has a specified direction, horizontal or vertical. This **1×1 *Rush Hour*** problem behaves fundamentally differently: deciding whether a specified car can move at all is polynomial time [7, 8, 17], whereas the analogous questions for 1×2 or 2×2 Rush Hour (or for 1×2 sliding blocks) are PSPACE-complete [7, 8]. Tromp and Cilibrasi [17] exhaustively searched all 1×1 Rush Hour puzzles of a constant size, and found that the length of solutions grew rapidly, suggesting exponential-length solutions; for example, the hardest 6×6 puzzle requires 732

[1] Sadly, to our knowledge, Rush Hour the puzzle was not an inspiration for Rush Hour the 1998 buddy cop film starring Jackie Chan and Chris Tucker.

[2] Solovey and Halperin [14] state their result in terms of unit-square cars amidst polygonal obstacles, but crucially allow the cars to be shifted by half of the square unit. Phrased as cars aligned on a unit grid, these cars are effectively 2×2.

[3] The open problem was first stated in the ICALP 2002 version of [7], based on discussions with John Tromp, as mentioned in [17], which is cited in the journal version of [7].

moves. They also suggested a variant where some cars cannot move at all (perhaps they ran out of gas?), which we call ***fixed blocks*** by analogy with pushing block puzzles [2],[4] as potentially easier to prove hard.

In this paper, we settle the latter open problem by Tromp and Cilibrasi [17] by proving that 1×1 Rush Hour with fixed blocks is PSPACE-complete. This result is the culmination of many efforts to try to resolve this problem since it was posed in 2005; see the Acknowledgments.

Our reduction starts from NCL, and reduces through another related puzzle game, Subway Shuffle. In his 2006 thesis, Hearn [6, 8] introduced this type of puzzle as a generalization of 1×1 Rush Hour, again to help prove it hard. Subway Shuffle involves motion planning of colored tokens on a graph with colored edges, where the player can repeatedly move a token from one vertex along an incident edge of the same color to an empty vertex, and the goal is to move a specified token to a specified vertex. Despite the generalization to graphs and colored tracks, the complexity remained open until 2015, when De Biasi and Ophelders [1] proved it PSPACE-complete by a reduction from NCL. Their proof works even when the graph is planar and uses just four colors.

We use a variant on Subway Shuffle where the graph is directed, and tokens can travel only along forward edges. In Section 3, we prove that directed Subway Shuffle is PSPACE-complete even with planar graphs and just two colors, by a proof similar to that of De Biasi and Ophelders [1]. In Section 4, we then show that this construction uses a limited enough set of vertices that it can actually be embedded in the grid and simulated by 1×1 Rush Hour, proving PSPACE-completeness of the latter with fixed blocks. We conclude with open problems in Section 5.

2 Basics

First we precisely define the problems introduced above.

▶ **Definition 2.1.** *In **Rush Hour**, we are given a square grid containing nonoverlapping **cars**, which are rectangles with a specified orientation, either horizontal or vertical. A legal move is to move a car one square in either direction along its orientation, provided that it remains within the square and does not intersect another car. The goal is for a designated special car to reach the left edge of the board. We also allow **fixed blocks**, which are spaces cars cannot occupy.*

▶ **Definition 2.2.** **1×1 *Rush Hour*** *is the special case of Rush Hour where each car is 1×1.*

▶ **Definition 2.3.** *In **Subway Shuffle**, we are given a planar undirected graph where each edge is colored and some vertices contain a colored token. A legal move is to move a token across an edge of the same color to an empty vertex. The goal is for a designated special token to reach a designated target vertex.*

▶ **Definition 2.4.** *In **oriented Subway Shuffle**, we are given a planar directed graph where each edge is colored and some vertices contain a colored token. A legal move is to move a token across an edge of the same color, in the direction of the edge, to an empty vertex, and then flip the direction of the edge. The goal is for a designated special token to reach a designated target vertex.*

[4] Tromp and Cilibrasi [17] refer to 1×1 Rush Hour as "Unit (Size) Rush Hour" and the fixed-block variant as "Walled Unit Rush Hour".

Figure 1 The valid Subway Shuffle vertices with degree 3. Every vertex with degree 1 or 2 is valid.

▶ **Lemma 2.5.** *Subway Shuffle, oriented Subway Shuffle, and Rush Hour are in PSPACE.*

Proof. We can solve these problems in nondeterministic polynomial space by guessing each move, and accepting when the special car or token reaches its goal. So all three problems are contained in NPSPACE, and by Savitch's theorem [12] they are in PSPACE. ◀

3 2-color Oriented Subway Shuffle is PSPACE-complete

In this section, we show that 2-color oriented Subway Shuffle is PSPACE-complete. To do so, we reduce from nondeterministic constraint logic, which is PSPACE-complete [8]. Our reduction is an adaption of the proof in [1] in which the gadgets use only two colors (instead of four) and work in the oriented case.

We actually prove a slightly stronger result in Theorem 3.1: that 2-color oriented Subway Shuffle is PSPACE-complete even with a restricted vertex set, and with a single unoccupied vertex. A vertex is ***valid*** if it has degree at most 3, and has at most 2 edges of a single color attached to it; these vertices are shown in Figure 1. Our proof of PSPACE-hardness will only use valid vertices.

▶ **Theorem 3.1.** *2-color oriented Subway Shuffle with only valid vertices and exactly one unoccupied vertex is PSPACE-complete.*

Proof. Containment in PSPACE is given by Lemma 2.5. To show hardness, we reduce from planar NCL with AND and protected OR vertices.

In constraint logic, a ***protected OR vertex*** is an OR vertex (one with three blue edges) such that two edges, due to global constraints, cannot simultaneously point towards the vertex. NCL is still PSPACE-complete when every OR vertex is protected [8]. Because of this global constraint, there are only five possible states that a protected OR vertex can be in. In particular, there are only four possible transitions between the states of a protected OR vertex. Our OR gadget only allows these four transitions; in particular it does not allow the transition between only the leftmost edge pointing inward and only the rightmost edge pointing inward, which is the defining transition that a protected OR vertex does not have compared to a normal OR vertex.

The Subway Shuffle instance we construct will have only a single empty vertex (other than the target vertex), called the ***bubble***, which moves around the graph opposite the motion of tokens. Our vertex and edge gadgets work by having the bubble enter them, move around a cycle, and then exit at the same vertex. The effect is that each edge in the cycle flips and each token in the cycle moves across one edge, except that one token by the entrance moves twice.

The general structure of the reduction is as follows. First, we choose any rooted spanning tree on the dual graph of the constraint logic graph. This rooted spanning tree will determine the path the bubble takes to get from one vertex or edge to another. For each edge and

vertex in the constraint logic graph, we will replace it with a subway shuffle gadget. The constraint logic edges which are part of our spanning tree will have a path for the bubble to cross through them. Each face of the CL graph has paths connecting vertex gadgets and edge gadgets as necessary to allow the bubble to visit each gadget.

When playing the constructed Subway Shuffle instance, the bubble begins at the root of the spanning tree. The bubble can move down the tree by crossing edge gadgets until reaching a desired face. It then enters a vertex or edge gadget, goes around a cycle, and exits. A sequence of moves of this form corresponds to flipping a constraint logic edge or reconfiguring a vertex (that is, changing which constraint logic edge(s) are used to satisfy that vertex and therefore are locked from being flipped away from it). The bubble can always travel back up the spanning tree to the root, and from there visit any face and then any CL vertex or edge.

Now we will describe the various gadgets that implement constraint logic in Subway Shuffle. Many places in the gadget figures have an empty vertex attached to them; this represents where the gadget is connected to the spanning tree. Entering through these vertices is the only way the bubble can interact with a gadget.

The edge gadget is shown in Figure 2. The two vertices and edge at the bottom and top of the edge gadget (in a gray box in the figure) are shared with the connecting vertex gadget. The edge gadget consists of five interlocking cycles. The edge can be flipped by rotating each of the five cycles in order, as shown in Figure 3. The bubble rotates a cycle by entering at the appropriate white vertex, and then moving around the cycle, and finally exiting where it entered.

If the edge is in the spanning tree, we include the rightmost vertex called the ***exit***, which allows the bubble to visit the edge gadget and pass through to face on the other side of the edge. We place the edge gadget in the orientation so that the entrance is on the face closer to the root of the spanning tree of the dual graph.

There are two kinds of edges in constraint logic: red and blue edges. The only difference is that they have different weights for the constraint logic constraints. Blue edges are as shown in Figure 2 (and can be rotated); red edges are the same gadget, but reflected.

Edge gadgets connect to vertex gadgets by sharing the two vertices and edge marked in a gray box. In the edge gadget, when the vertex colors and edge direction are as shown in the edge gadget figure, the edge is ***unlocked***, which means that the bubble is free to flip the direction of that edge. The vertex gadget's colors take precedence for the shared edges and vertices. When they do not match those shown in the edge gadget figure, we say the edge is ***locked***. When this happens, it becomes impossible for the bubble to rotate first cycle, and thus prevents the bubble from flipping the edge. This mechanism is what allows the gadgets to enforce the constraints of the vertices in the constraint logic graph. Edges are only ever locked while pointing into a vertex because all of the constraints in constraint logic only give lower bounds on the number of inward pointing edges. When an edge is pointing away from a vertex, some of the cycles in the vertex will be impossible to rotate, preventing the bubble from unlocking other edges.

The AND vertex gadget is shown in Figure 4. Whenever the bubble is not visiting the vertex gadget, either the blue (weight 2) edge or both red (weight one) edges are locked to point towards the vertex. If all three edges are pointing towards the vertex, the bubble can visit the vertex gadget (at the top entrance) and go around the cycle to switch which edges are locked. This implements the constraints on a NCL AND vertex.

Our protected OR vertex gadget is shown in Figure 5. The two protected edges are the leftmost and rightmost edges, so we can assume that they never both point towards the vertex. The gadget has three entrances. The gadget can be in five possible states corresponding

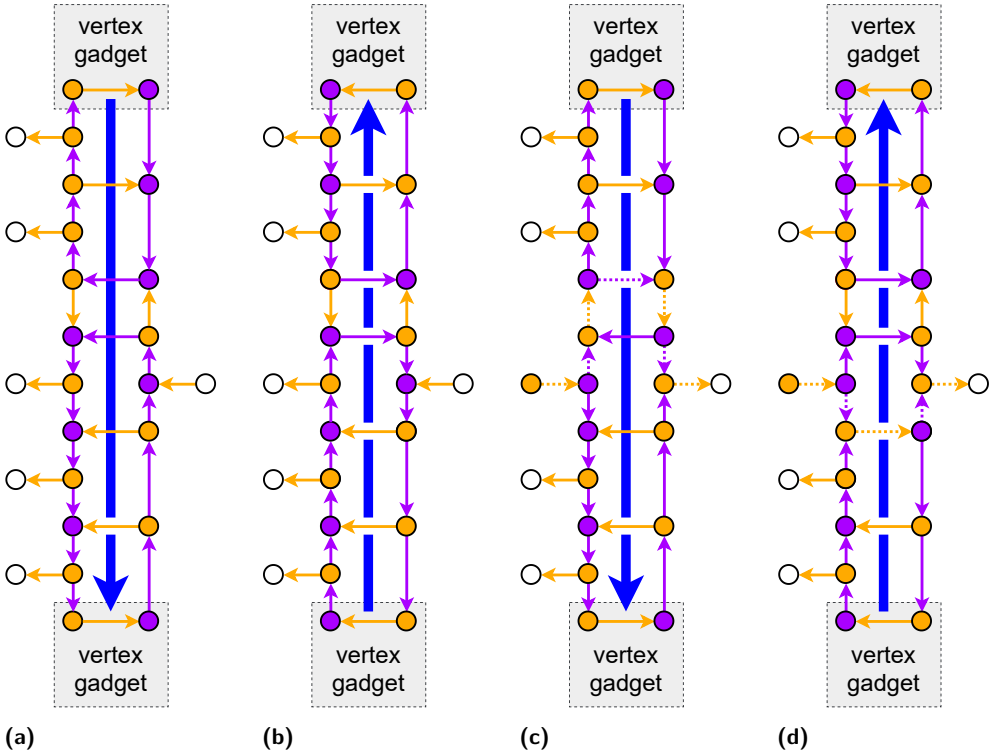


Figure 2 The edge gadget for 2-color oriented Subway Shuffle, shown (a) directed down and unlocked, (b) directed up and unlocked, (c) directed down after the bubble has passed through, and (d) directed up after the bubble has passed through. This gadget is based on the edge gadget in [1].

to the five possible states of a CL protected OR. In each state, the edges which are locked in correspond to the set of edges that are pointing inward in the corresponding state of a CL protected OR. In the first state, the left edge is locked, and the other two are free. In the second state, the middle edge is also locked. In the third state, only the middle edge is locked. In the fourth state, both the middle and right edges are locked. Finally, in the fifth state, only the right edge is locked. To get from one state to the next, the bubble rotates a single cycle. The fives states and the transitions between them are shown in Figure 5. The only transitions between states are to the next and previous states. To transition from one state to the next, the bubble goes around the cycle indicated by the dotted edges.

Our last gadget is the win gadget, shown in Figure 6. It is placed attached to the edge gadget corresponding to the target edge in the constraint logic instance, and allows the player to win the Subway Shuffle instance when that edge can be flipped.

In the first state shown, the target edge is pointing away. If the bubble arrives at the win gadget, it cannot accomplish anything. If the target edge is flipped so it now points toward the win gadget, we will be in the second state. Then the bubble can enter the win gadget at the top entrance and go around the indicated cycle, moving the special token one to the left. Finally, the bubble can enter at the bottom entrance to move the special token across to the target vertex.

To allow the bubble to reach every gadget, we connect the entrances and exits of gadgets which are on the same face of the CL graph. This simply requires a tree connecting these vertices for each face. Each face other than the root of the spanning tree has exactly one edge exit on it; we orient the edges on that face to point towards this exit. The color of these

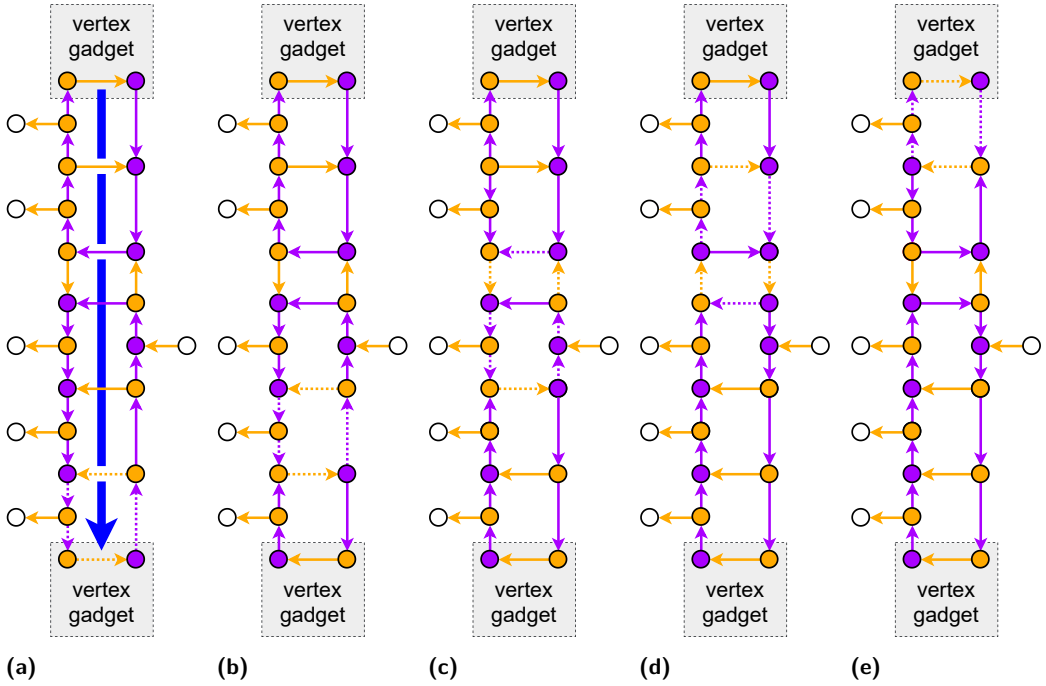
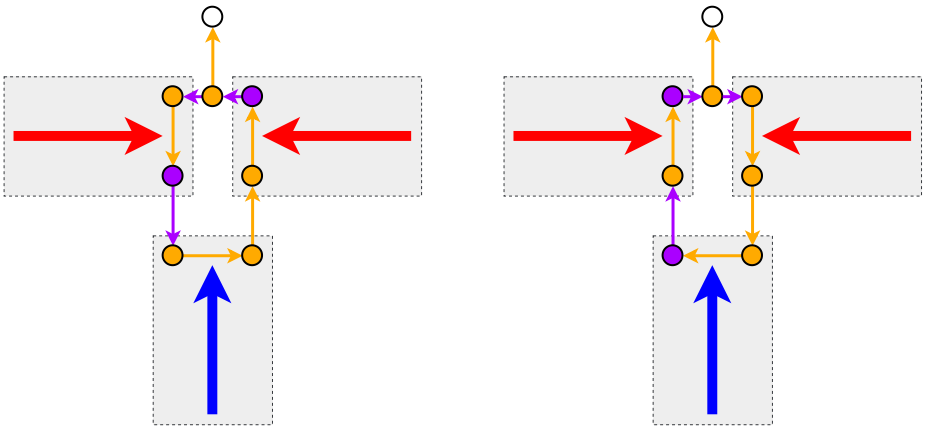


Figure 3 The five cycles that the bubble rotates to flip the orientation of an edge gadget. For each cycle, the bubble enters at the white vertex, goes around the dotted cycle, and leaves where it entered. Note that the colors and orientation of the edge and vertices that connect to the vertex gadget will not always match what is shown in this figure. When they do not, we say the edge is ***locked*** by the corresponding vertex gadget, and it is not possible to rotate the dotted cycle.

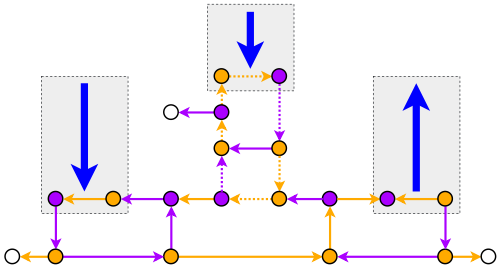


(a) All edges oriented in, with the blue edge locked.

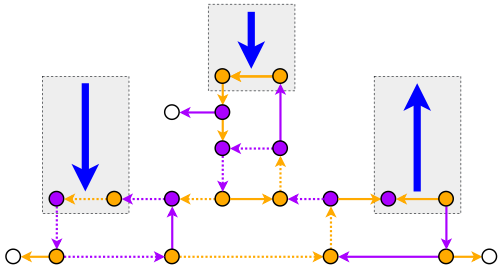
(b) All edges oriented in, with both red edges locked.

Figure 4 The AND vertex gadget for 2-color oriented Subway Shuffle. This gadget is based on the AND vertex gadget in [1].

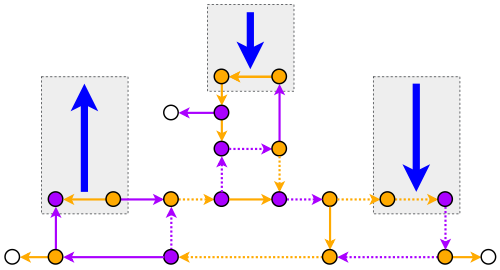
edges does not matter, provided all vertices are valid and the token at the tail of an edge is the same color. For the face which is the root of the spanning tree, the tree connecting entrances has one vertex without a token, and the edges point towards it; this is where the bubble starts.



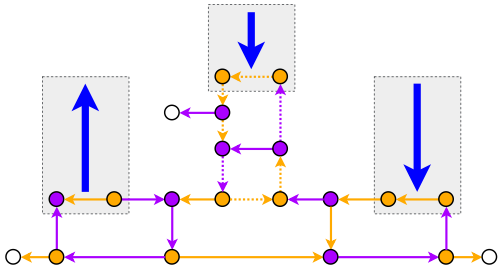
(a) State 1: The left edge is locked. The middle edge is unlocked and pointing in. The right edge is pointing out.



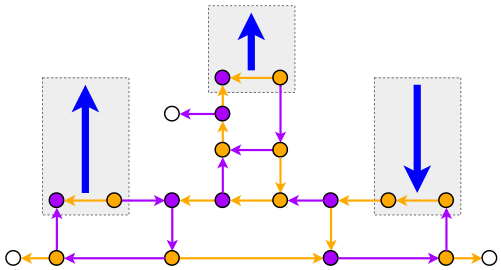
(b) State 2: The left and middle edges are locked. The right edge is pointing out.



(c) State 3: The left edge is pointing out. The middle edge is locked. The right edge is unlocked and pointing in.



(d) State 4: The left edge is pointing out. The middle and right edges are locked.

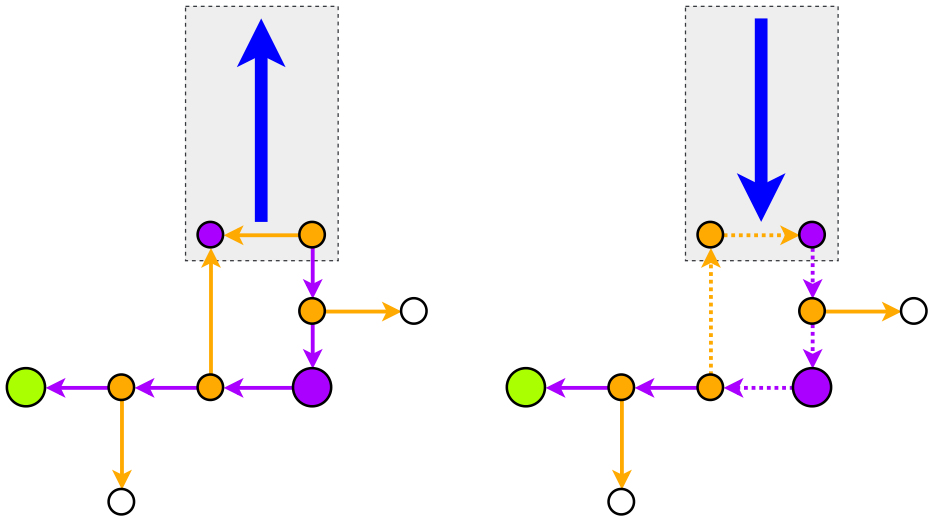


(e) State 5: The left and middle edges are pointing out. The right edge is locked.

Figure 5 The five states of the protected OR vertex. The dotted edges show the cycle that is rotated to transition to the next state. Note that each state is defined only by which edged are locked in; the other unlocked edges can be either pointing in or out in each state.

Now we show how the gadgets prevent any moves other than the moves outlined above that simulate the NCL instance. First we consider the edge gadget. It is easy to check that while rotating any of the dotted cycles in an edge gadget, there are only two legal moves other than continuing the cycle. The first one is leaving through the exit vertex during the third cycle. This is equivalent to the bubble just using the throughway in the edge gadget to reach the rest of the spanning tree after turning only the first two cycles. By Lemma 3.3, this is never useful. The other legal move is while turning the first, fourth, or fifth cycle, it is possible for the bubble to move into the connecting vertex gadget through the shared vertices. We will show that nothing useful can be accomplished here when we consider the vertex gadgets. Similarly, it will also be possible for the bubble to come from a vertex and enter the edge gadget through the shared vertices. We show this is not useful in Lemma 3.2.

▶ **Lemma 3.2.** *It is never useful for the bubble to enter an edge gadget directly from a vertex gadget through the shared vertices.*



(a) The locked win gadget. The bubble cannot do anything here.

(b) The unlocked win gadget. Now that the blue edge is pointing into the gadget, the indicated cycle can be rotated.

Figure 6 The win gadget for 2-color oriented Subway Shuffle. The target edge starts pointing away. If it is flipped, the bubble can enter the win gadget once at each entrance to move the (bottom right) purple special token to the (bottom left, green) target vertex. This gadget is based on the FINAL gadget in [1].

Proof. We need to check the up, down, up traversed, and down traversed configurations.

In most configurations, there are no legal moves to enter the edge gadget from the shared vertices. The only configuration where this is possible is from the orange token on the top left of the upward pointing edge gadget. From here, it can move through a path of three tokens before it gets stuck. At that point, the only legal move is to undo the last three moves and exit the same way it entered. ◀

▶ **Lemma 3.3.** *It is never useful to turn some of the cycles in an edge gadget without turning all of them.*

Proof. If you turn some of the cycles, but not all of them, then both ends of the edge gadget will be in the pointing outward configuration. For all of the vertex gadgets, there are no transitions that require the outward pointing configuration, so the edge gadget being in this configuration never lets you make a move that you could not make if you finished turning all of the cycles in an edge gadget.

We also need to make sure that turning only some cycles, and then entering an edge gadget from a vertex gadget (as in Lemma 3.2), does not allow you to do anything. If we look at all of the partial edge configurations as shown in Figure 3, there is no way to access anything from any of these configurations. We also need to check the configurations that arise from partially rotating an edge and then traversing it. Since it is not possible to reach the traverse paths from entering from a vertex gadget, these configurations also do not let the bubble do anything else useful. ◀

Now we consider the AND gadget. Since the entire gadget is a single cycle, there is nothing the bubble can do within the gadget while turning the cycle. While turning the cycle, the bubble can try to enter an edge gadget through one of the shared vertices; however, we have already shown that the this is never useful in Lemma 3.2.

We also need to consider if the bubble enters the vertex gadget from an edge on one of the shared vertices. It will never be able to move around the entire cycle because the orange vertex at the top will not be accessible. The only other thing the bubble can do is try to enter a different edge gadget, but we already showed this is not useful in Lemma 3.2.

Now we consider the OR gadget. First we look at each of the four cycles. While turning the first cycle, the only legal move that is not continuing the cycle is moving the purple token just to the right of the cycle. However, from here, the only moves lead to dead ends so there is not anything useful for the bubble to do besides immediately return to the cycle. There are no other legal moves while turning the second cycle. While rotating the third cycle, it is possible for the bubble to reach the shared vertices of the leftmost edge gadget, but by Lemma 3.2 this does not help. While rotating the fourth cycle, it is possible for the bubble to reach the shared vertices of the rightmost edge gadget, but again this does not help.

Now we consider when the bubble enters the OR vertex gadget from an edge gadget through one of the shared vertices. In the first state, there are no legal moves after entering from the top or right edges. In the second state, from either the top or left edges it can enter and traverse most of the gadget but cannot complete any loop and thus cannot make progress by Lemma 3.4. In the third state, the bubble has no legal moves after entering from the left or top edge. From the right edge it can traverse most of the gadget but cannot complete any loops. From the fourth state, again, while the bubble can traverse most of the gadget after entering from the top edge, it does not complete any loops so it has no effect. In the fifth state, there are no legal moves after entering the vertex gadget.

▶ **Lemma 3.4.** *If the bubble takes any path from any vertex to the same vertex which does not complete a nontrivial loop, then the state of the Subway Shuffle instance must not have changed.*

Proof. If the bubble never completed a loop, then the only way for it to get back to where it started is to take the same path in reverse. By the definition of Subway Shuffle moves, this exactly undoes these moves returning the instance back to its original state. ◀

Finally, we check the win gadget. While using the win gadget, there are no legal moves other than completing the one loop. There is only one edge connected to the win gadget. If the bubble tries to enter the win gadget here, it cannot leave anywhere else or complete any loops, so by Lemma 3.4 it must return with no effect.

Since the constraint logic graph is planar, the reduction yields a planar graph for 2-color oriented Subway Shuffle. Since the constructed instance Subway Shuffle is winnable exactly when the constraint logic instance is, and the reduction can clearly be done in polynomial time, this shows 2-color oriented Subway Shuffle is PSPACE-hard. All of the gadgets used, including the trees connected gadget entrances, use only valid vertices, so it is still PSPACE-hard with only valid vertices. ◀

4 1×1 Rush Hour is PSPACE-complete

In this section, we show that 1×1 Rush Hour is PSPACE-complete by a reduction from 2-color oriented Subway Shuffle with only valid vertices and only a single empty vertex, which was shown to be PSPACE-complete in the previous section. 1×1 Rush Hour is played on a large square grid. We allow for fixed blocks, which are spaces marked impassable in the grid.

We will simulate Subway Shuffle vertices with individual cars at intersections, and edges as paths of cars. In general, purple edges and vertices will be horizontal cars, and orange edges and vertices will be vertical cars. Like in the Subway Shuffle, we will have a single ***bubble*** which is a single empty space that moves around as cars move into that space.

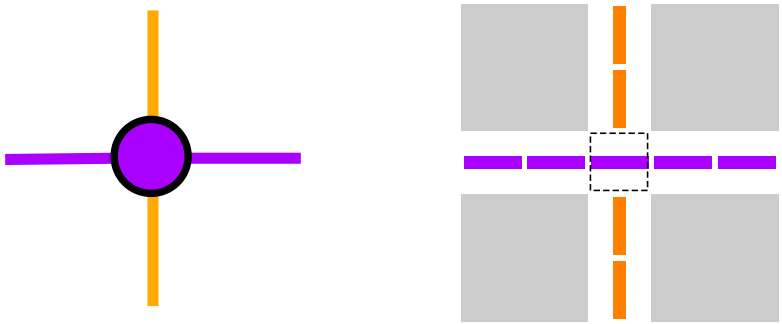


Figure 7 A degree-4 Subway Shuffle vertex embedded in Rush Hour. Note that, while this is not a valid Subway Shuffle vertex, all valid vertices are subsets of this vertex. Individual dashes represent cars. A line of cars of one color represents a Subway Shuffle edge of that color. The center boxed car represents the Subway Shuffle vertex.

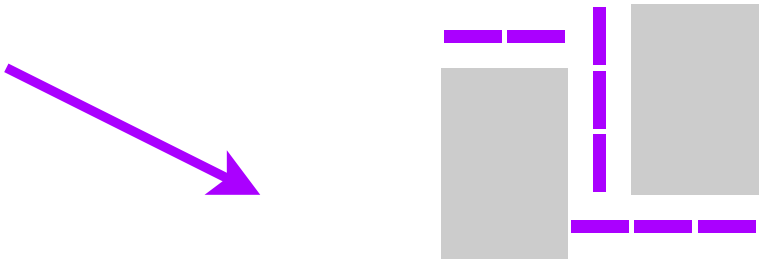
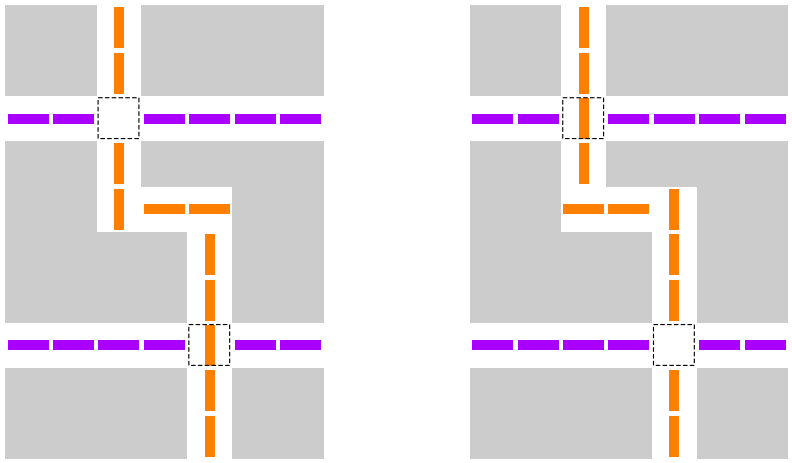


Figure 8 A Rush Hour simulation of a Subway Shuffle edge. This is a purple edge which points right.

We replace each vertex in our Subway Shuffle instance with a single car which is vertical if there is an orange token there, and horizontal if a purple token is there. Orange edges leading from a vertex attach to it as vertical rows of cars, and purple edges attach to a vertex as horizontal rows of cars. A degree-4 vertex with a purple token is depicted in Figure 7. Valid vertices can be embedded this way, with fixed blocks on the unused sides for lower degree vertices.

A Subway Shuffle edge is simulated by a path of cars which can make right-angle turns, allowing us to embed an arbitrary planar Subway Shuffle graph. The direction of a car at a turn in an edge defines which way the Subway Shuffle edge is oriented. A purple edge which points right is depicted in Figure 8. In order to maintain the directionality of edges, each edge must be simulated by a path with at least one turn.

To make a move, suppose the bubble is currently at a vertex. To move a token in from an adjacent vertex, a car from the connecting edge is moved in. Then cars from that edge are all moved one space toward the initial vertex, until finally we can move the car in the second vertex out. Note that this process reverses the orientation of the edge as desired. If the edge was pointed in the correct direction, then this process will succeed; if the edge is oriented in the wrong direction, then this process will fail when we try to turn a corner in the edge. Similarly it is impossible to move a token along an edge of the opposite color, because it will be unable to move out of its vertex. An example of a single Subway Shuffle move where an orange token is moved up along an orange edge embedded in Rush Hour is shown in Figure 9.



(a) Before moving the orange token up.

(b) After moving the orange token up.

Figure 9 Moving a single orange token in Subway Shuffle when simulated by Rush Hour in Figure 8.

No other useful actions can be taken. If the bubble is not currently at a vertex, then there are at most two possible moves. One of them would just be undoing the previous move, and the other would be continuing the process of moving a token along an edge. When the bubble is at a vertex, moving any adjacent car into the vertex is the same as starting the process of moving a Subway Shuffle token along the corresponding edge.

The win condition of a Rush Hour instance is allowing the marked car to escape the grid. The win gadget needs to be specified more precisely because Subway Shuffle tokens do not correspond exactly to Rush Hour cars. Also, we want to make sure that everything can fit within a grid so our win condition is actually located near the edge.

Our win gadget is depicted in Figure 10. The win condition is the circled car reaching the star. The boxed cars represent Subway Shuffle vertices. In order to win, first the boxed orange car directly in front of the circle car must leave by rotating this cycle. This represents the marked token in the Subway Shuffle vertex moving to the middle vertex along the bottom of the win gadget. Then, the leftmost orange line must be moved down one space, clearing the way for the marked car to leave.

In Rush Hour, because winning requires a car leaving the grid, we must also take care to make sure that the win gadget is at the boudary of our construction, and not somewhere buried in the middle. To do this, we conisder the CL edge which is part of the win gadget. Since the CL graph is planar, we can consider one of the faces that this edge is a part of, and make this face the "outside" face. Now our win gadget is at the boundary, which is what we needed.

5 Open Problems

In this paper, we have shown that 1×1 Rush Hour with fixed blocks is PSPACE-complete, solving Tromp and Cilibrasi's open problem [17]. It remains whether the assumption of fixed blocks can be eliminated, and thereby solve the open problem of Hearn, Demaine, and Tromp [7, 17]. We note that it is impossible to perfectly simulate a fixed block using Rush Hour cars, since for any arrangement of cars in a region, there must be at least one point along the

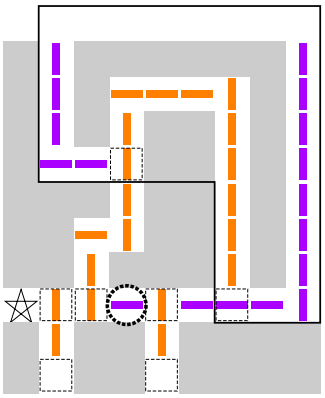


Figure 10 The cycle of the subway shuffle win gadget embedded in Rush Hour. The goal is to get the circled car to the star. The boxed cars are the vertices in the Subway Shuffle win gadget. The two lines of purple cars extending upward are the purple edges of the connected edge gadget. Everything inside the solid black line is part of the connecting edge gadget.

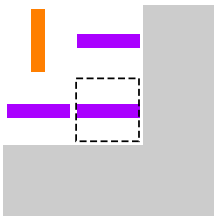


Figure 11 Let the gray area be accessible by the bubble. Then the boxed car is at the corner of the boundary of the accessible region, and regardless of its orientation it must also be accessible by the the bubble.

boundary of the region that, if it were empty, a car can exit the region. For a single bubble, it gets worse than that. Let a space be ***accessible*** if the bubble can ever reach that space. By Theorem 5.1, the accessible region is always a rectangle. Since we can ignore anything inaccessible, we can just assume that everywhere in the entire Rush Hour grid is accessible. Because the bubble can get everywhere, it seems impossible to modify the gadgets in our proof in any simple way to constrain the bubble from wandering freely inside and between the cycles in gadgets.

▶ **Theorem 5.1.** *In any* 1×1 *Rush Hour instance with no fixed blocks with only a single "bubble," the set of accessible spaces is a rectangle.*

Proof. The accessible region is clearly connected. If it is not a rectangle, there must be a corner on the boundary of the accessible region where two accessible spaces are adjacent to the same inaccessible space, as in Figure 11. Then regardless of its orientation, the car in this inaccessible space must be able to move into one of these two accessible spaces, and thus is also accessible. This is a contradiction, so the accessible region must be a rectangle. ◀

References

1. Marzio De Biasi and Tim Ophelders. Subway Shuffle is PSPACE-complete. Manuscript, February 2015. URL: http://www.nearly42.org/cstheory/subway-shuffle-is-pspace-complete/.
2. Erik D. Demaine, Robert A. Hearn, and Michael Hoffmann. Push-2-F is PSPACE-complete. In *Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG 2002)*, pages 31–35, Lethbridge, Alberta, Canada, August 12–14 2002.
3. Erik D. Demaine and Mikhail Rudoy. A simple proof that the $(n^2 - 1)$-puzzle is hard. *Theoretical Computer Science*, 732:80–84, July 2018.
4. Gary William Flake and Eric B. Baum. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270(1–2):895–911, 2002.
5. Martin Gardner. Sliding-block puzzles. In *Martin Gardner's Sixth Book of Mathematical Diversions from Scientific American*. W. H. Freeman and Company, 1971. Republished by MAA, 2001.
6. Robert A. Hearn. *Games, Puzzles, and Computation*. PhD thesis, Massachusetts Institute of Technology, 2006. URL: http://erikdemaine.org/theses/bhearn.pdf.
7. Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1–2):72–96, October 2005. Originally appeared at ICALP 2002.
8. Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. AK Peters/CRC Press, 2009.
9. Patentarcade.com. Case update: Rubin v. Apple Inc. Blog post, 7 July 2011. URL: http://patentarcade.com/2011/07/new-case-rubin-v-apple-inc.html.
10. Daniel Ratner and Manfred Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10:111–137, 1990. URL: http://users.soe.ucsc.edu/~manfred/pubs/J15.pdf.
11. Don Rubin. The Parking Lot. http://www.donrubin.com/parking_lot.html, 2012.
12. Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970. doi:10.1016/S0022-0000(70)80006-X.
13. Jerry Slocum and Dic Sonneveld. *The 15 Puzzle*. Slocum Puzzle Foundation, 2006.
14. Kiril Solovey and Dan Halperin. On the hardness of unlabeled multi-robot motion planning. *The International Journal of Robotics Research*, 35(14):1750–1759, November 2016. doi:10.1177/0278364916672311.
15. James A. Storer. Tokyo Parking / Rush Hour. Jim Storer Puzzles Home Page, 2015. URL: https://www.cs.brandeis.edu/~storer/JimPuzzles/ZPAGES/zzzTokyoParking.html.
16. ThinkFun. The evolution of ThinkFun's Rush Hour. Blog post, February 2018. URL: http://info.thinkfun.com/stem-education/the-evolution-of-thinkfuns-rush-hour.
17. John Tromp and Rudi Cilibrasi. Limits of Rush Hour Logic complexity. *arXiv preprint cs/0502068*, 2005. URL: https://arXiv.org/abs/cs/0502068.
18. Stephen A. Wagner. Manipulable puzzle. U.S. Patent D395,468, June 1998. URL: https://patents.google.com/patent/USD395468S/en?oq=d395468.