Subway Shuffle, 1×1 Rush Hour, and Cooperative Chess Puzzles: Computational Complexity of Puzzles

by

Josh Brunner

S.B., Computer Science and Engineering
MIT (2019)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2021

Certified by .....
Erik Demaine
Professor
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Subway Shuffle, 1×1 Rush Hour, and Cooperative Chess Puzzles: Computational Complexity of Puzzles

by

Josh Brunner

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Oriented Subway Shuffle is a game played on a directed graph with colored edges and colored tokens present on some vertices. A move consists of moving a token across an edge of the matching color to an unoccupied vertex and reversing the orientation of that edge. The goal is to move a token across a target edge. We show that it is PSPACE-complete to determine whether a particular target edge can be moved across through a sequence of Oriented Subway Shuffle moves. We show how this can be interpreted in the context of the motion-planning-through-gadgets framework, thus showing PSPACE-completeness of certain motion planning problems. In contrast, we show that polynomial time suffices to determine whether a particular token can ever move.

This hardness result is motivated by three applications of proving other puzzles hard. A fairly straightforward reduction shows that the puzzle game Rush Hour is PSPACE-complete when all of the cars are 1×1 and there are fixed immovable cars. We show that two classes of cooperative Chess puzzles, helpmates and retrograde Chess, are also PSPACE-complete by reductions from Oriented Subway Shuffle.

Thesis Supervisor: Erik Demaine
Title: Professor

Acknowledgments

I would like to thank my advisor, Erik Demaine, for introducing me to the study of the complexity of various puzzles and game and providing these problems in particular. I would like to thank my collaborators, especially Dylan Hendrickson, Lily Chung, and Julian Wellman, for many ideas toward solving these problems. Dylan especially has provided many of the ideas here and I spent countless hours staring at whiteboards and chess boards with him.

I would also like to thank all of the students in MIT's 6.892 Algorithmic Lower Bounds class, as that class and the community of researchers there is what inspired all of the work in this thesis.

Contents

Chapter 1

Introduction

Subway Shuffle is a puzzle game created by Hearn in 2006 [9] to try to show that the 1×1 variant of Rush Hour (defined below) is PSPACE-complete. A Subway Shuffle instance consists graph with colored edges, and each vertex may have a colored token on it or be empty. One vertex is designated as a target vertex, and one token is designated as a target token. A move consists of moving a token across an edge of the matching color. An instance is solvable if there is a sequence of moves that results in the target token occupying the target vertex. De Biasi and Ophelders showed this to be PSPACE-complete in 2015 [4], however the original question of PSPACE-completeness of 1×1 Rush Hour remained open.

In this thesis, we strengthen the results about Subway Shuffle. Our restricted version of Subway Shuffle is a sufficiently simple algorithmic problem that it can provide fairly clean reductions to show that a variety of problems that are otherwise hard to analyze, including a slight variant of the original 1×1 Rush Hour problem it was inspired by, are PSPACE-complete. We show this reduction as well as two other applications of reducing this problem to computational problems arising from other puzzles.

In Chapter 2, we first strengthen De Biasi and Ophelders's [4] results about the complexity of Subway Shuffle. We make several modifications to the original Subway Shuffle problem which allow it to be reduced to other problems more easily. First, we make the graph directed; tokens can only move along edges in the direction of

the edge, and, to maintain the reversibility of moves originally present in Subway Shuffle, we reverse the direction of the edge after a token has crossed it. We also greatly restrict the types of allowed vertices to only two types of vertices which have only two colors, degree at most 3, and where each vertex has at most two edges of a single color. We slightly modify the win condition to be moving any token across a target edge, instead of the win condition which was moving a specified token to a specified location. Finally, we also require there to be exactly one unoccupied vertex. Even under these restrictions, we show Subway Shuffle remains PSPACE-complete. This result originally appeared in [2] in collaboration with Lily Chung, Erik Demaine, Dylan Hendrickson, Adam Hesterberg, Adam Suhl, and Avi Zeff.

We give a new interpretation of the Subway Shuffle results in motion-planning-through-gadgets framework of [5, 6]. Because our reduction uses an instance of Subway Shuffle with only one unoccupied vertex, we can think of it as a 1-player motion planning problem where the unoccupied vertex is trying to move around the graph. From this, we show that the reconfiguration problem for a certain type of motion-planning gadget is PSPACE-complete.

In Chapter 3, we show that 1×1 Rush Hour with fixed immovable blocks is PSPACE-complete from a simple reduction from the Subway Shuffle variant in Chapter 2. Here 1×1 Rush Hour is a type of sliding block puzzle. Each block is a 1×1 square and is either labeled as horizontally moving, vertically moving, or immobile, and the goal is to slide the blocks around within an $n \times n$ grid to get a specific block to exit the grid. This problem originally inspired the creation of Subway Shuffle, and can be thought of as a Subway Shuffle variant where the graph is a subgraph of a grid graph, and vertical edges are one color while horizontal edges are a different color.

In Chapter 4 and Chapter 5 we show PSPACE-completeness for two different types of Chess puzzles. Helpmate and retrograde Chess are two types of Chess puzzles that are cooperative in nature. Instead of having White and Black competing, these puzzles ask whether gameplay could possibly produce a given result, which is equivalent to the two players (Black and White) cooperating to achieve the goal. This means the two players effectively act as a single player (in the sense that quantifiers

no longer alternate), placing the problem in PSPACE (see Lemmas 4.1 and 5.1).

It had previously been shown that it can take exponentially many moves to reach a desired configuration [14]. This naturally suggested that these problems are PSPACE-hard, which we show here via reductions from Subway Shuffle. These results originally appeared in [3] in collaboration with Erik Demaine, Dylan Hendrickson, and Julian Wellman.

Chapter 2

Subway Shuffle

In this chapter, we show that 2-color oriented Subway Shuffle is PSPACE-complete. To do so, we reduce from nondeterministic constraint logic (NCL), which is PSPACE-complete [10]. Our reduction is an adaptation of the proof in [4] in which the gadgets use only two colors (instead of four) and work in the oriented case. We also modify the win condition to be moving any token across a target edge, instead of moving a particular token to a target vertex. Both proofs use only a single unoccupied vertex.

We actually prove a slightly stronger result in Theorem 2.4: that 2-color oriented Subway Shuffle is PSPACE-complete even with a restricted vertex set. 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 2-1. Our proof of PSPACE-hardness will only use valid vertices. These results were published in [2], and were written in collaboration with Lily Chung, Erik Demaine, Dylan Hendrickson, Adam Hesterberg, Adam Suhl, and Avi Zeff. Presented here is a slight modification of those results, since those results used the original Subway Shuffle win condition of moving a target token to a target vertex. Section 2.4 is also a new result about how Subway Shuffle relates to motion planning problems.

2.1 Basics

We start with some basic definitions of Subway Shuffle, and in particular the specific variant that we show is PSPACE-hard.

Definition 2.1. 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 to have a token of the appropriate color move across a designated target edge. When we refer to Subway Shuffle, we are referring to the decision problem of, whether there a sequence of legal moves which moves a token across the target edge in a given Subway Shuffle instance.

Definition 2.2. 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 to have a token of the appropriate color move across a designated target edge.

Lemma 2.3. *Subway Shuffle is in PSPACE.*

Proof. We can solve Subway Shuffle 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 [13] they are in PSPACE. □

2.2 PSPACE-Hardness of Subway Shuffle

To show PSPACE-hardness of Subway Shuffle, we will be reducing from a variant of Constraint Logic. Nondeterministic Constraint Logic (NCL) is a graph game which is PSPACE-complete that was invented by Hearn to show that certain puzzles are computationally hard [9]. In NCL, there is a directed graph of red and blue edges with maximum degree 3. We call blue edges weight-2 edges and red edges weight-1

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

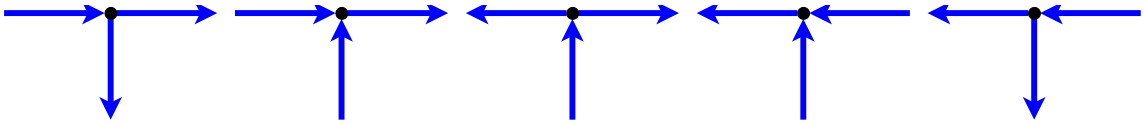


Figure 2-2: The five possible states that a protected OR vertex can be in. The only allowed transitions are between adjacent states.

edges. An vertex is *satisfied* if the edges pointing into it have total weight at least 2, and a configuration is *valid* if all vertices are satisfied. A "move" from a valid configuration is flipping the orientation of a single edge such that the configuration remains valid. They show that it is PSPACE-complete to determine whether there exists a sequence of moves which result in a target edge being flipped [9].

Theorem 2.4. *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 2.3. To show hardness, we reduce from planar NCL with AND and protected OR vertices. An OR vertex is one with three blue edges incident, and an AND vertex is one with two red edges and one blue edge incident. Hearn showed that NCL is PSPACE-complete if all vertices are of this form [9].

In constraint logic, a *protected OR vertex* is an OR vertex (one with three blue edges) such that two particular edges, due to global constraints, cannot simultaneously point towards the vertex. NCL is still PSPACE-complete when every OR vertex is protected [10]. Because of this global constraint, there are only five possible states that a protected OR vertex can be in, as shown in Figure 2-2. 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, 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 NCL graph has paths connecting vertex gadgets and edge gadgets as necessary to allow the bubble to visit each gadget. The structure of the reduction is similar to the reduction in [6] that shows motion planning with locking 2-toggles is PSPACE-complete.

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 NCL 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 2-3. 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 2-4. 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 2-3 (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 2-5. Whenever the bubble is not visiting the vertex gadget, either the blue (weight 2) edge or both red (weight one)

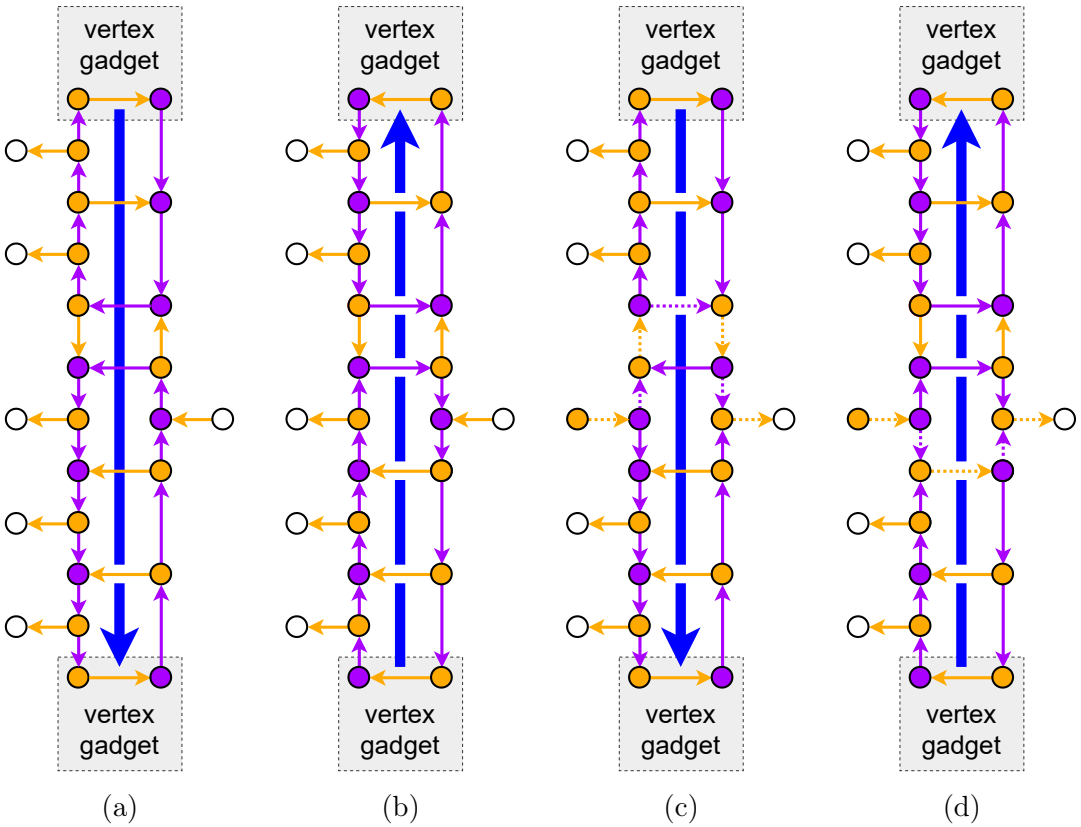


Figure 2-3: The edge gadget for 2-color oriented Subway Shuffle, shown (a) directed down and unlocked, (b) directed up and unlocked, (c) directed down after the bubble has passed through, and (d) directed up after the bubble has passed through. This gadget is based on the edge gadget in [4].

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 2-6. 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 to the five possible states of a NCL 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 NCL 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

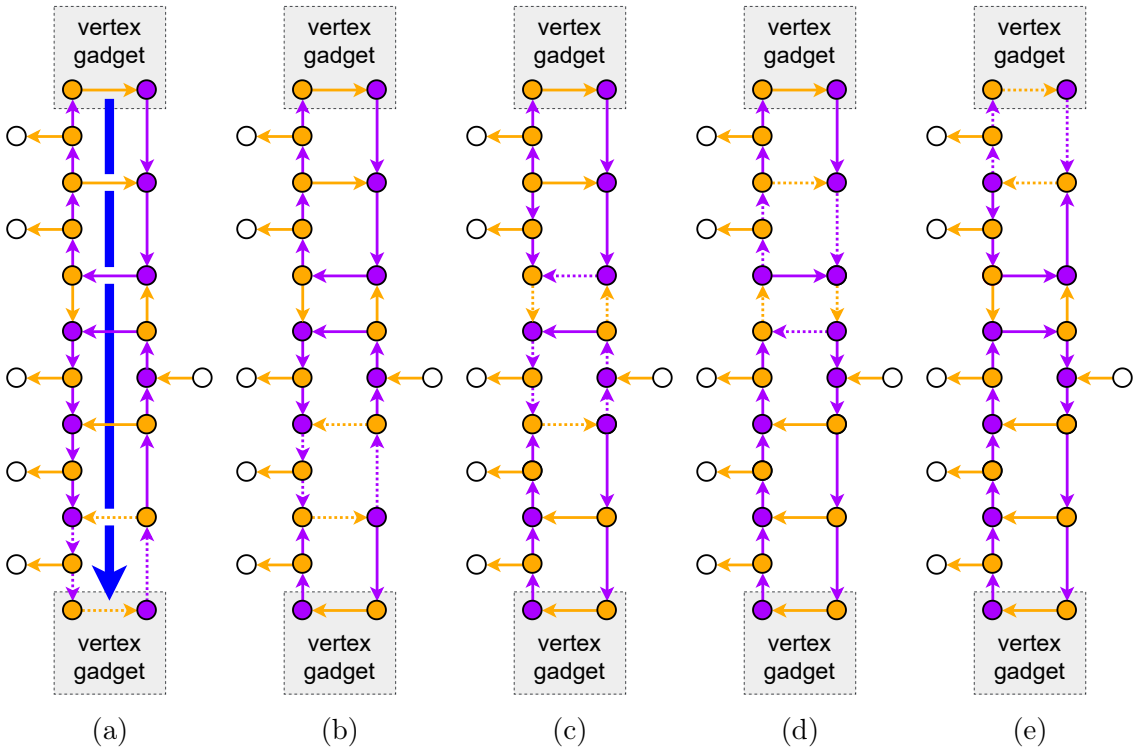
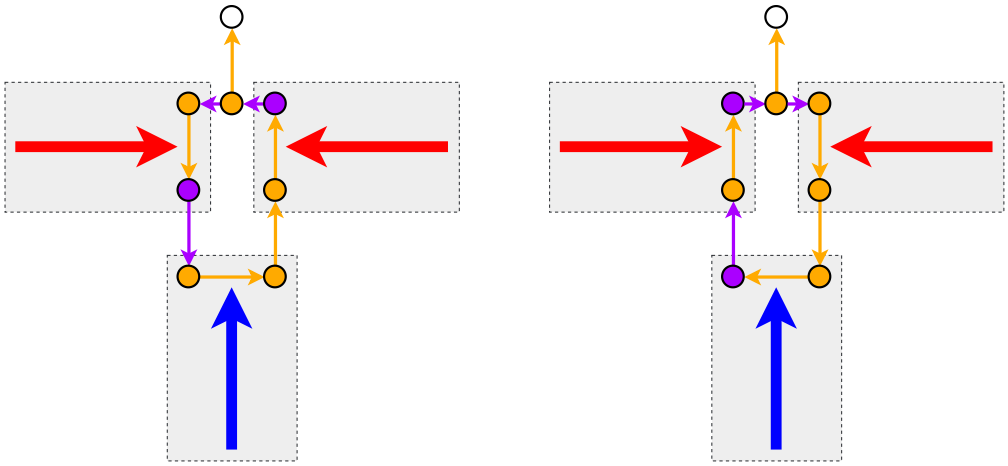


Figure 2-4: The five cycles that the bubble rotates to flip the orientation of an edge gadget. For each cycle, the bubble enters at the white vertex, goes around the dotted cycle, and leaves where it entered. Note that the colors and orientation of the edge and vertices that connect to the vertex gadget 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.

state, both the middle and right edges are locked. Finally, in the fifth state, only the right edge is locked. The fives states and the transitions between them are shown in Figure 2-6. 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 2-7. 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, causing



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

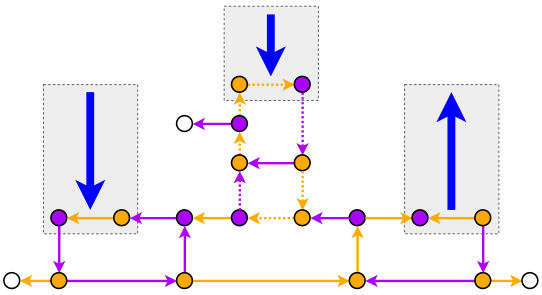
(b) All edges oriented in, with both red edges locked.

Figure 2-5: The AND vertex gadget for 2-color oriented Subway Shuffle. This gadget is based on the AND vertex gadget in [4].

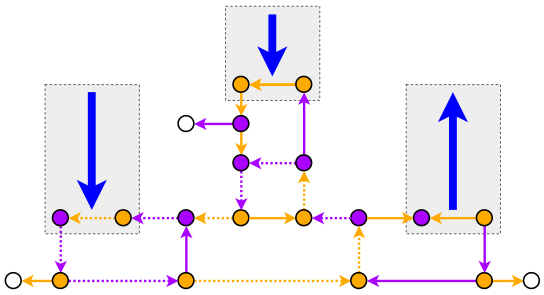
a purple token to be present at the bottom middle vertex. Finally, the bubble can enter at the bottom entrance and cross the target edge, since there is now a purple token there.

To allow the bubble to reach every gadget, we connect the entrances and exits of gadgets which are on the same face of the NCL 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 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.

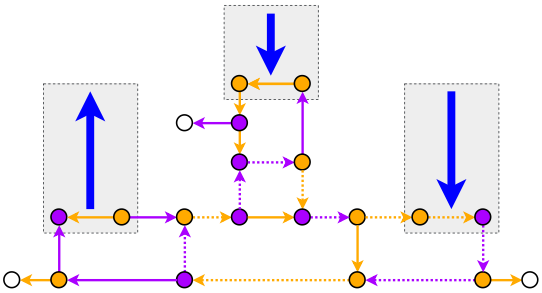
Now we show how the gadgets prevent any moves other than the moves outlined above that simulate the NCL instance. First we consider the edge gadget. It is easy to check that while rotating any of the dotted cycles in an edge gadget, there are only two legal moves other than continuing the cycle. The 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 2.6, this is never useful. The other legal move is



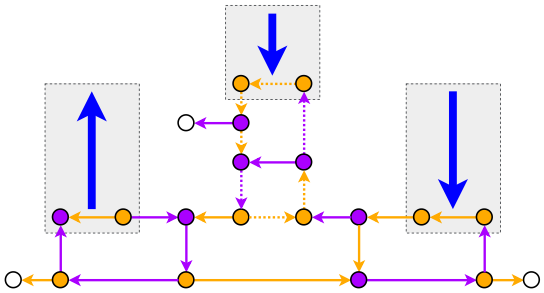
(a) State 1: The left edge is locked. The middle edge is unlocked and pointing in. The right edge is pointing out.



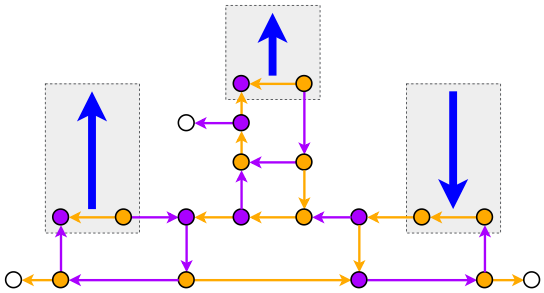
(b) State 2: The left and middle edges are locked. The right edge is pointing out.



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



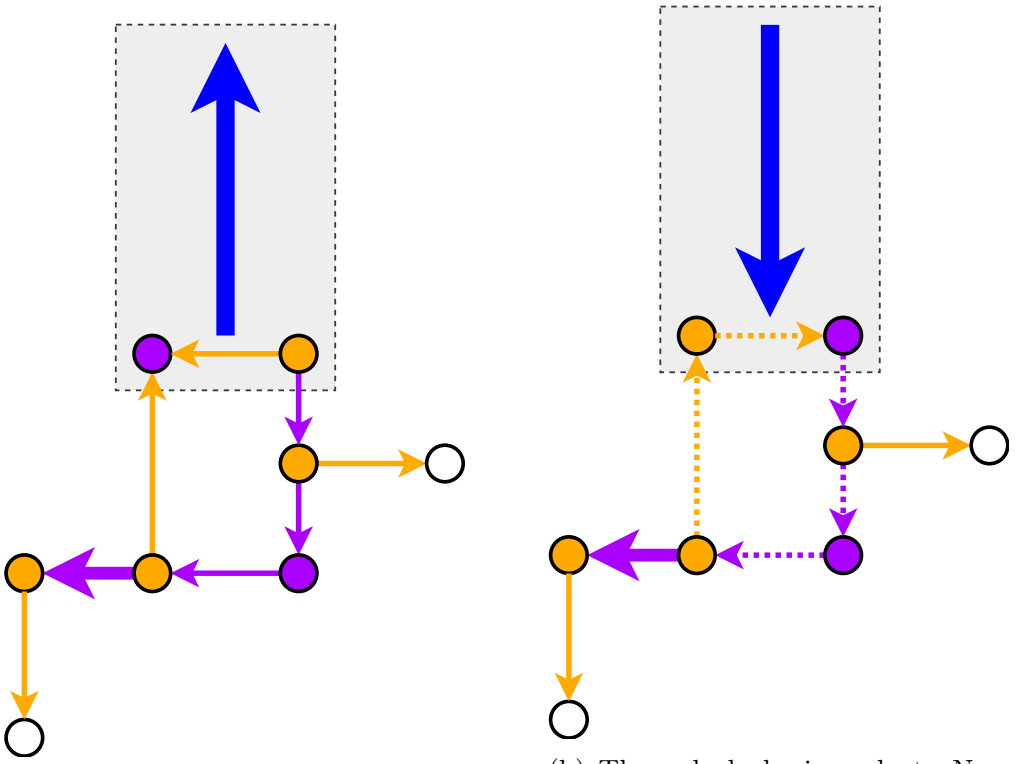
(d) State 4: The left edge is pointing out. The middle and right edges are locked.



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

Figure 2-6: 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 edges are locked in; the other unlocked edges can be either pointing in or out in each state.

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 2.5.



(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 2-7: The win gadget for 2-color oriented Subway Shuffle. The target NCL 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 token across the bottom left target Subway Shuffle edge. This gadget is based on the FINAL gadget in [4].

Lemma 2.5. *It is never useful for the bubble to enter an edge gadget directly from a vertex gadget through the shared vertices.*

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 2.6. *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 2.5), does not allow you to do anything. If we look at all of the partial edge configurations as shown in Figure 2-4, 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 2.5.

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 2.5.

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 2.5 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.

Lemma 2.7. *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. □

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 2.7. 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.

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 2.7 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 connecting gadget entrances, use

only valid vertices, so it is still PSPACE-hard with only valid vertices. □

2.3 Two-Orange One-Purple Subway Shuffle

We now show a slight strengthening of our result that Subway Shuffle is PSPACE-complete. Even when every vertex has at most two orange edges and at most one purple edge, Subway Shuffle is still PSPACE-complete . In our Subway Shuffle reduction, some vertices have two orange edges incident while others have two purple edges incident. We can reduce the vertex set to eliminate vertices with two purple edges by simulating them using Lemma 2.8.

Lemma 2.8. *In Subway Shuffle with exactly one bubble (empty vertex), a vertex with two purple edges can be simulated with using vertices with at most one purple edge and at most two orange edges.*

Proof. Given an instance of Subway Shuffle, every vertex with two purple incident edges can be replaced with one with two orange incident edges as shown in Figure 2-8. Note that while we need to transform vertices with one or two purple outgoing edges, we do not need to worry about vertices with zero purple outgoing edges. This is because a purple token on a vertex with zero purple outgoing edges can never move, so the entire vertex is stuck and can safely be ignored. It is easy to check that the set of legal moves is almost the same in every configuration. The only difference is that in Figure 2-8(d), the purple token can leave the vertex twice through each of the two outgoing edges; however the second purple token that leaves does not allow any further moves except moving the purple token back into place. With the assumption that only one vertex is ever empty, this situation is never useful, so this replacement perfectly simulates the original vertex. □

Corollary 2.9. *Subway Shuffle is PSPACE-complete even when every degree three vertex has exactly two orange and one purple edge incident.*

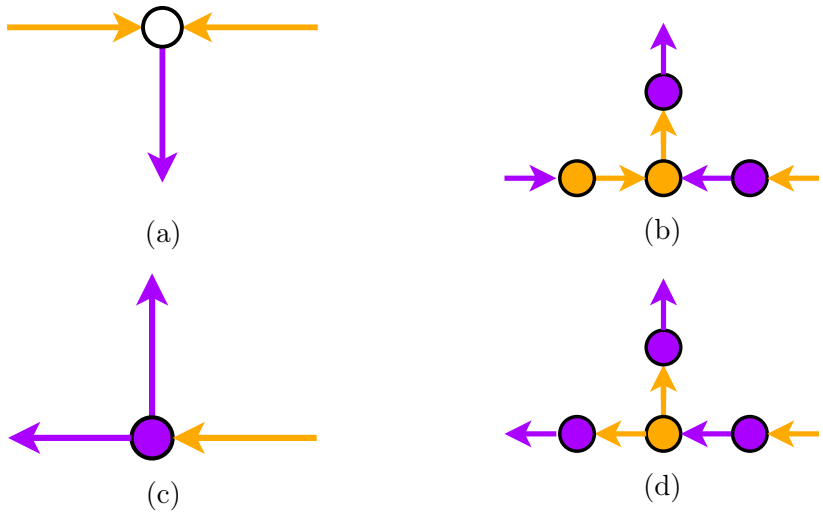


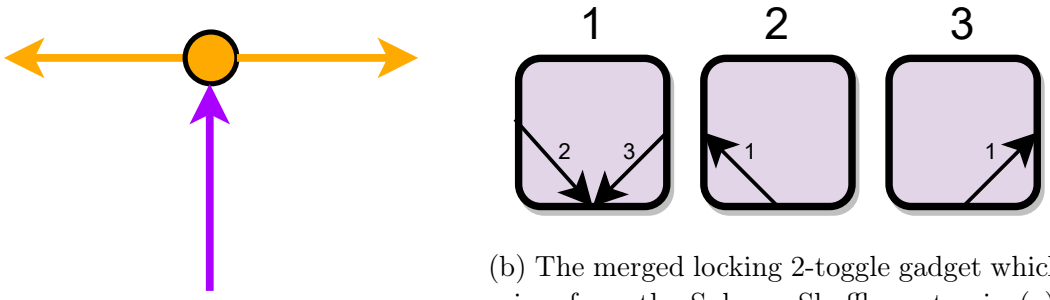
Figure 2-8: The color changing gadget. (a) and (b) show the transformation for vertices with one incoming purple edge, and (c) and (d) show the transformation for vertices with no incoming purple edges. In both cases, the vertex behaves identically after the change, and using this technique we can give every degree-3 vertex two orange edges.

2.4 Motion Planning

Subway Shuffle with a single bubble has a natural interpretation in the motion-planning-through-gadgets framework of [6, 5]. In 1-player motion planing, a single agent is traversing a graph of gadgets.

A *gadget* in the framework has several *locations* and several *states*. The current state of the gadget determines the available (directed) traversals between locations. Each traversal is labeled with one of the gadget's states; when the agent crosses that traversal, the gadget changes to that state. Figure 2-9 (b) is an example of a gadget with 3 states and 3 locations. The bubble in Subway Shuffle will correspond to the agent in the motion-planning-through-gadgets framework, and the vertices in Subway Shuffle will correspond to specific gadgets.

In Subway Shuffle, because every move can be undone, the gadgets it corresponds to will all be *reversible* as defined in [6]. Furthermore, because every time an edge is traversed it flips orientation, it is never possible to traverse any path twice consecutively in the same direction. In motion planning, the simplest gadget that satisfies these conditions is the *1-toggle*. The 1-toggle is simply an edge which can only be



(a) The Subway Shuffle vertex that corresponds to the merged locking 2-toggle in state 1.

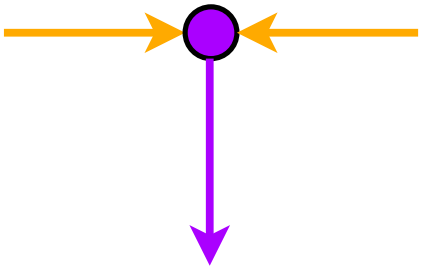
(b) The merged locking 2-toggle gadget which arises from the Subway Shuffle vertex in (a). From state 1 the agent can traverse either path from a side location to the bottom location. From each of the other states the only available traversal is returning the way you came.

Figure 2-9: The Subway Shuffle vertex which has one orange edge pointing outward when the bubble is present and the corresponding gadget.

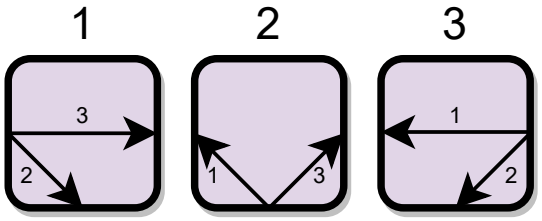
crossed in one direction, and when the agent does so, the direction of the edge is reversed. In Subway Shuffle, a path of edges all oriented in the same direction is thus equivalent to a 1-toggle.

From Lemma 2.8, we can reduce any Subway Shuffle instance to one with only two kinds of degree 3 vertices: those with one of the two orange edges pointing out when the bubble is present, and those with zero of the two orange edges pointing out when the bubble is present. By considering all possible states a vertex can be in, we arrive at two kinds of gadgets which represent these two vertex types. To represent a Subway Shuffle instance with gadgets, we think of the bubble (agent) as generally existing on a Subway Shuffle edge, and a move is moving the bubble through a vertex to another Subway Shuffle edge. Thus, the states we consider will be what we would normally consider to be in the middle of a move in Subway Shuffle. In particular, this means that we can assume that every vertex is always occupied by a token.

First we consider the vertex which has one orange edge pointing out when the bubble is present. Figure 2-9 shows the gadget's possible traversals and states. When there is an orange token present, there is only one possible state of the vertex, which has both orange edges pointing out. In this case, there are two possible traversals of the vertex. This corresponds to state 1 in the figure. The bubble can enter through the left orange edge and exit through the purple edge, or the bubble can perform the



(a) The Subway Shuffle vertex that corresponds to the fork in state 2.



(b) The fork gadget which corresponds to the Subway Shuffle vertex in (a). From any state, the agent has two possible traversals from one location. After any traversal the new state is the one where the agent can enter from the location they just left.

Figure 2-10: The Subway Shuffle vertex which has both orange edges pointing inward when the bubble is present and the corresponding gadget.

symmetric traversal from the right. If it traverses from the left, then the result is a state with both orange edges pointing rightward (state 2 in the figure). The only possible traversal from this state is going back the way the bubble came through the purple edge and out the left edge. This is symmetrically true for the other traversal, which leads to state 3 in the figure. We call this gadget the *merged locking 2-toggle* gadget.

Now we consider the vertex which has all edges pointing inward when the bubble is present. Figure 2-10 shows the gadget's possible traversals and states. Whenever a token is present, there will only be one edge pointing outward from the vertex, and thus the only way to enter the vertex will be from that edge. Once the bubble is in the vertex, it can leave through any of the edges since all three point inward. Thus there are three possible states, corresponding to which edge is pointing inward. In each state, the bubble can enter in from the outward pointing edge and leave through either of the other edges. The resulting state is one with the edge the bubble left now pointing outward, and the other edges pointing inward. We call this gadget the *fork* gadget.

Finally, we should consider vertices of degree less than 3. For degree 2 vertices, the only relevant kind is one with exactly one edge pointing inward when a token is present. If both are pointing inward, then the vertex can never be accessed by the

bubble, so it can be ignored. If neither are pointing inward, then while the bubble can access it, the bubble can never cross it; the bubble must instead always undo its last move if it ever arrives. A vertex with exactly one edge pointing inward is simply a 1-toggle. When the bubble arrives from the correct side, it can traverse the vertex and leave through the other edge, flipping the direction of the edges and thus reversing the 1-toggle; if the bubble is on the wrong side, it cannot enter the vertex at all.

In one form the motion-planning-through-gadgets framework [5], we assume the locations of the gadgets are connected by a matching of undirected edges, and the agent traverses edges of the graph to go between the different locations of different gadgets. Because Subway Shuffle has directed edges which flip every time they are used, we cannot simply have an undirected edge connecting our gadgets. Instead, we insist that they are *1-toggle protected*, meaning that every connection between gadgets has a 1-toggle on that edge. 1-toggle protected motion planning was introduced in [1].

In motion planning problems, the most common decision problem is *reachability*: Given a network of gadgets, their initial states, and an initial location of the agent, can a particular location be reached by the agent? In Subway Shuffle, this corresponds to the question: given a Subway Shuffle instance with a single bubble, which has a target *vertex* instead of a target edge, is there a sequence of moves which vacates that vertex? Surprisingly, while the original formulation of Subway Shuffle is PSPACE-complete, this problem is in NL.

Lemma 2.10. *A vertex is reachable if and only if either the bubble starts there or it has a outgoing edge of the same color as the token on this vertex that points to a different reachable vertex.*

Proof. First, suppose a vertex is reachable. Consider a sequence of moves in which the bubble reaches this vertex. The last move made must have been the bubble moving from an adjacent vertex along an edge of the appropriate color and orientation to this vertex. Thus, that vertex is a reachable vertex which satisfies the conditions in the

lemma.

Now suppose that a vertex has a outgoing edge of the same color as the token on this vertex that points to a different reachable vertex. Consider the sequence of moves that vacates that vertex. From that position, we can make an additional move to move the token from the target vertex to this one. Thus, the bubble can reach this vertex. □

Theorem 2.11. *There is a polynomial-time (in fact, NL) algorithm for deciding whether the bubble can reach a target location in Subway Shuffle.*

Proof. Consider only edges which point away from a vertex with a token of the appropriate color there. From Lemma 2.10, if and only if there is a directed path on these edges from the target location to the bubble, then the target location is reachable. This gives a natural correspondence between reachability in Subway Shuffle and connectivity in directed graphs. It is known that connectivity in a directed graph is solvable in NL, so reachability in Subway Shuffle is solvable in NL. □

In order to apply the hardness result of Subway Shuffle, we need to ask about a different decision problem we call *one-gadget reconfiguration*. Specifically, since using a target edge in Subway Shuffle corresponds to performing a specific traversal of a gadget, we will use "Can a specific gadget ever reach a target state?".

At this point, we have a straightforward reduction from Subway Shuffle to motion planning on a 1-toggle protected graph with these gadgets. Because 1-toggles can be simulated with merged locking 2-toggles by ignoring one of their inputs, we also have that motion planning on an undirected graph with these gadgets is PSPACE-hard. Typically, in motion planning we think of an edge being a sort of "hallway" between two gadgets. Thus, it is natural to allow *branching hallways*, which act as hyperedges that connect several gadgets' locations together. In this model, we can simulate the fork gadget with three 1-toggles all connected by a branching hallway. Thus, we get that one-gadget reconfiguration motion planning with only merged locking 2-toggles and branching hallways is PSPACE-complete.

Notably, this transformation of a Subway Shuffle instance to a motion planning instance is easily done in either direction. To reduce a Subway Shuffle instance to a motion planning instance, simply replace each vertex with the corresponding motion-planning gadget. To reduce a motion-planning instance to a Subway Shuffle instance, again replace each gadget with the corresponding vertex. This doesn't immediately give a valid Subway Shuffle instance though, because the colors of the edges coming from two adjacent gadgets might not agree. However, since we insisted that our motion-planning problem is 1-toggle protected, we know that every edge contains a one-toggle, and a 1-toggle can be implemented with a degree 2 vertex with any combination of colors for its edges, so we can always choose appropriate colors to make adjacent gadgets agree.

From this, we get that reachability is easy with these gadgets in 1-toggle protected motion planning without branching hallways. We can in fact strengthen our result about reachability to remove the restriction that the graph is 1-toggle protected, and allow branching hallways that connect several gadgets' locations together.

Theorem 2.12. *It is NL-complete to decide whether the agent can reach a given location in a motion planning instance consisting of merged locking 2-toggles and branching hallways.*

Proof. We give a proof analogous to that in Theorem 2.11. We identify locations that are connected by a hallway, since trivially if we can reach one then we can reach the other by walking down the hallway. Now we can prove a lemma analogous to Lemma 2.10.

Lemma 2.13. *A location x is reachable if and only if there is another reachable location y and a traversal from y to x in some gadget's initial state, or x is the starting location.*

Proof. First, suppose that x is reachable and not the starting location. Consider any sequence of traversals that result in reaching x, and the last traversal in particular. This traversal is from another reachable location y to x, so the only thing we need to show is that this traversal was present in the gadget's initial state. Suppose for

contradiction that it was not. Then this gadget must not be in its initial state, so it must have been traversed earlier in the sequence. Since the merged locking 2-toggle only has three locations, and we know that x was not reached previously, this traversal must have been (in some direction) between y and the third location. However, if we look at the states of a merged locking 2-toggle, we can see that there is never a traversal between two locations in which the resulting state has a traversal pointing toward the third location. Thus, there is no traversal earlier in the sequence that could have led to the gadget being in this state, which is a contradiction, so there must have initially been a traversal from y to x.

Now suppose that y is reachable and there is a traversal from y to x in some gadget's initial state, and again consider any sequence of traversals that result in reaching y. We show that, at the end of this sequence, this gadget must be in its initial state or x is reachable. Consider the gadget's initial state. In every state of a merged locking 2-toggle, only one of its locations has traversals which exit the gadget at that location. Thus, since we know our gadget's initial state had a traversal from y to x, every traversal of the gadget in its initial state ends at x. If this gadget was never traversed in this sequence, then it must be in its initial state. Otherwise, consider the first time our traversal sequence traverses this gadget. Then this traversal must have resulted in the agent arriving at x, so x is reachable. Thus either x is reachable or the gadget is in its initial state. If the gadget is in its initial state, then simply traversing the y to x traversal after reaching y shows that x is reachable. □

With this lemma, for determining reachability we can assume that gadgets never change state since Lemma 2.13 only cares about the initial states of gadgets. At this point, reachability is equivalent to directed connectivity in the graph with a vertex for each location and a directed edge for each traversal in a gadget's initial state between locations. Since directed connectivity is NL-complete, reachability is also NL-complete. □

Chapter 3

1×1 Rush Hour

3.1 Problem Definition and Prior Work

Rush Hour is a puzzle game about sliding cars in a 6×6 grid to try and get a specific car to exit the grid. The complexity of Rush Hour was first analyzed by Flake and Baum in 2002 [8]. 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 [16] strengthened this result to use just 1×2 cars (which again can move only in their long direction), using Nondeterministic Constraint Logic (NCL). Demaine and Hearn [10, 9] simplified this proof, and proved analogous results for triangular Rush Hour, again using NCL. In 2016, Solovey and Halperin [15] proved that Rush Hour is also PSPACE-complete with 2×2 cars and immovable 0×0 (point) obstacles.

In 2002, Demaine, Hearn, and Tromp asked whether Rush Hour is still PSPACE-complete even if all of the cars are 1×1. These cars do not have a natural orientation, so each car instead is explicitly either a horizontal moving car or a vertical moving car. The same techniques that worked to prove Rush Hour hard with longer cars were not easily adapted to this variant because they fundamentally relied on particular interactions in which cars partially blocked each other, which is impossible with 1×1 cars. Tromp and Cilibrasi [16] exhaustively searched all small cases of 1×1 Rush

Hour, and found that the solution length appeared to grow exponentially, suggesting that the problem may be PSPACE-hard.

In this chapter, we show that 1×1 Rush Hour with immovable fixed blocks 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. This reduction originally appeared in [2] in collaboration with Lily Chung, Erik Demaine, Dylan Hendrickson, Adam Hesterberg, Adam Suhl, and Avi Zeff.

3.2 PSPACE-hardness

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.

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 3-1. 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 3-2. 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

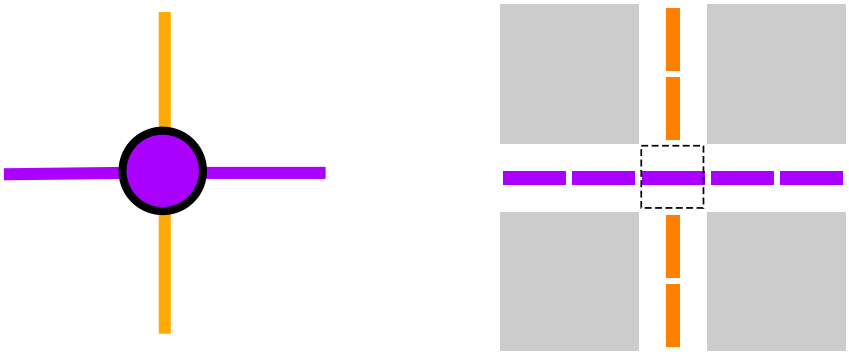


Figure 3-1: A degree-4 Subway Shuffle vertex embedded in Rush Hour. Note that, while this is not a valid Subway Shuffle vertex, all valid vertices are subsets of this vertex. Individual dashes represent cars. A line of cars of one color represents a Subway Shuffle edge of that color. The center boxed car represents the Subway Shuffle vertex.

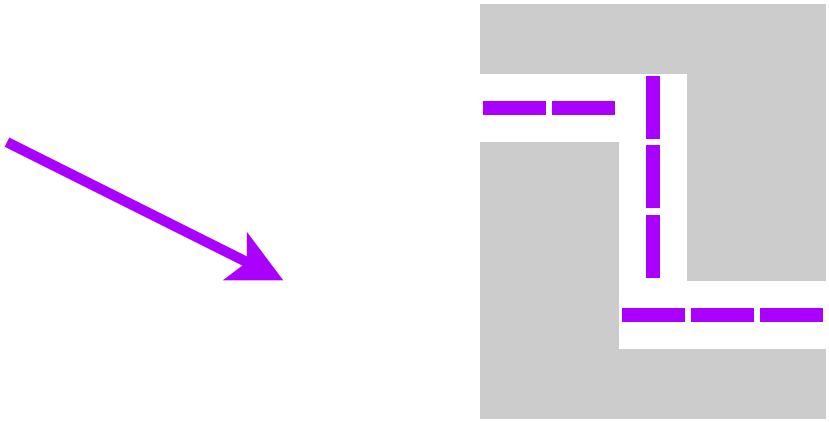
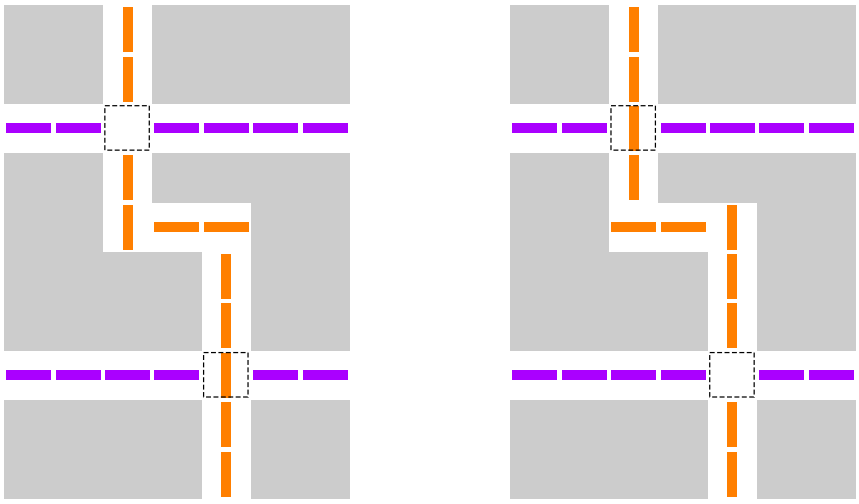


Figure 3-2: A Rush Hour simulation of a Subway Shuffle edge. This is a purple edge which points right.

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



(a) Before moving the orange token up.

(b) After moving the orange token up.

Figure 3-3: Moving a single orange token in Subway Shuffle when simulated by Rush Hour in Figure 3-2.

moved up along an orange edge embedded in Rush Hour is shown in Figure 3-3.

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 3-4. The win condition is the circled car reaching the star. The boxed cars represent Subway Shuffle vertices. The two empty boxes in the bottom of the figure correspond to the two entrances to the win gadget in Subway Shuffle in Figure 2-7. 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 purple

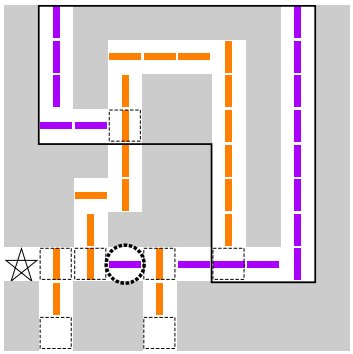


Figure 3-4: The cycle of the Subway Shuffle win gadget embedded in Rush Hour. The goal is to get the circled car to the star. The boxed cars are the vertices in the Subway Shuffle win gadget. The two lines of purple cars extending upward are the purple edges of the connected edge gadget. Everything inside the solid black line is part of the connecting edge gadget.

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. Moving the marked car between the two adjacent boxed spaces corresponds to exactly when the target purple edge in the Subway Shuffle instance is used.

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 boundary of our construction, and not somewhere buried in the middle. To do this, we consider the NCL edge which is part of the win gadget. Since the NCL 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.

Chapter 4

Helpmate

In *helpmate* Chess puzzles [12, 17], the goal is to find any legal sequence of Chess moves, with both Black and White cooperating, that leads to Black getting checkmated. In addition to being a popular form of Chess puzzle, helpmate problems naturally arise in regular games of Chess, as FIDE's[1] "dead-reckoning" rule says that any position without a helpmate is automatically a draw:

> "The game is drawn when a position has arisen in which neither player can checkmate the opponent's king with any series of legal moves. The game is said to end in a 'dead position'." [7, Article 5.2.2]

In practice, this condition is often checked when a player runs out of time, in which case that player loses if and only if they could ever possibly be checkmated from the current position (i.e., the helpmate problem has a solution) [7, Article 6.9]. In this chapter, we prove that characterizing such non-dead positions is PSPACE-complete. Amusingly, this result implies that it is PSPACE-complete to decide whether a given game position is already a draw (*draw-in*-0) for Chess.

This result originally appeared in [3] in collaboration with Erik Demaine, Dylan Hendrickson, and Julian Wellman.

[1] The International Chess Federation (FIDE) is the governing body of international Chess competition. In particular, they organize the World Chess Championship which defines the world's best Chess player. All top-level Chess competitions (not just FIDE's) follow FIDE's rules of Chess [7].

4.1 Overview

Lemma 4.1. *Helpmate is in PSPACE.*

Proof. An $n \times n$ Chess position takes only polynomial (in n) space to record. A nondeterministic polynomial-space machine can guess a sequence of moves, accepting when it achieves checkmate; thus the problem is in NPSPACE = PSPACE. □

In the remainder of this chapter, we prove that Helpmate is PSPACE-complete by reducing from max-degree-3 two-color oriented Subway Shuffle which we proved PSPACE-hard in Chapter 2.

Theorem 4.2. *Helpmate is PSPACE-complete.*

The structure of the reduction is to use a line of pieces of one type to represent an edge in the Subway Shuffle graph, where using the edge involves moving every piece in the line one space. A vertex is represented by a square where pieces from three different edges can move to. The two colors of Subway Shuffle are represented by which piece type is present in the vertex. All of the moving pieces involved in the reduction are white; black will be given a gadget to pass their turn with. In many of the figures, we label the relevant pieces that can move in red; all of the red pieces are white in the actual Chess position.

In order to make sure that players cannot make moves outside of the reduction, all of the edge and vertex gadgets are walled in with walls of bishops and pawns that are completely stuck.

4.2 Gadgets

We start with the *edge gadget*. To represent an edge, we simply use a line of bishops, shown in Figure 4-1. To move a token along the edge, move all of the bishops one space in that direction. The net effect will be a bishop entering one end and another bishop leaving the other end, representing a token moving.

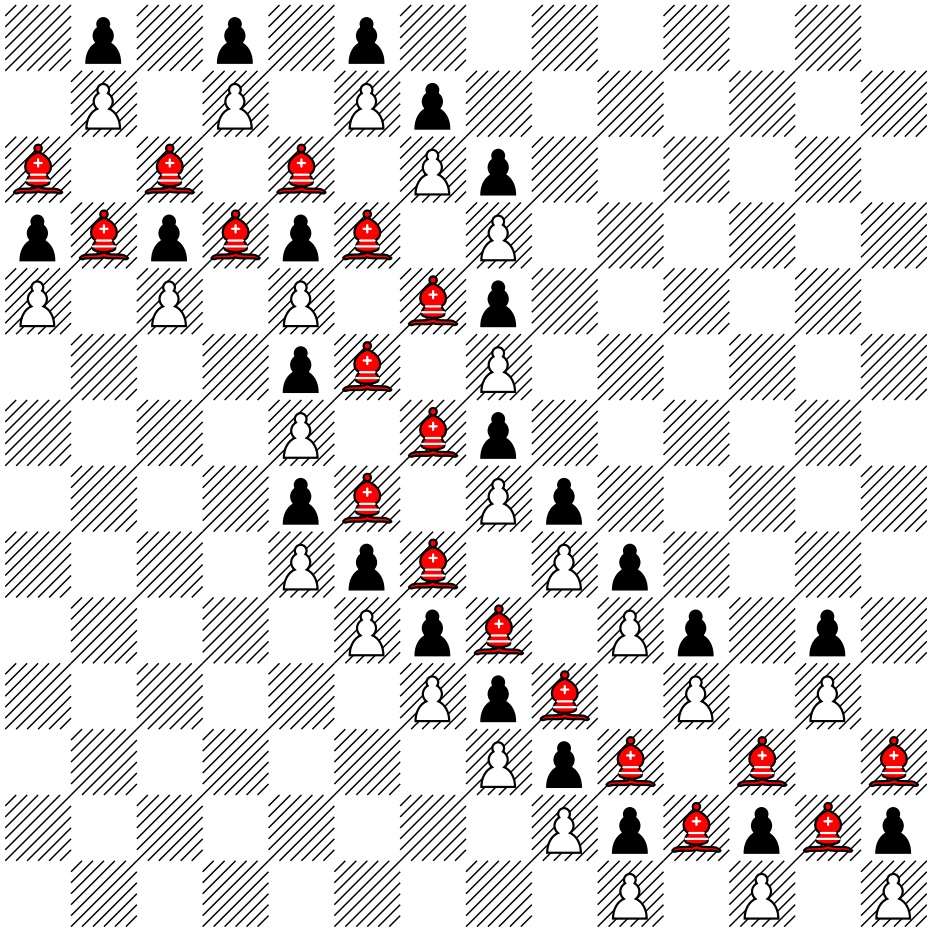
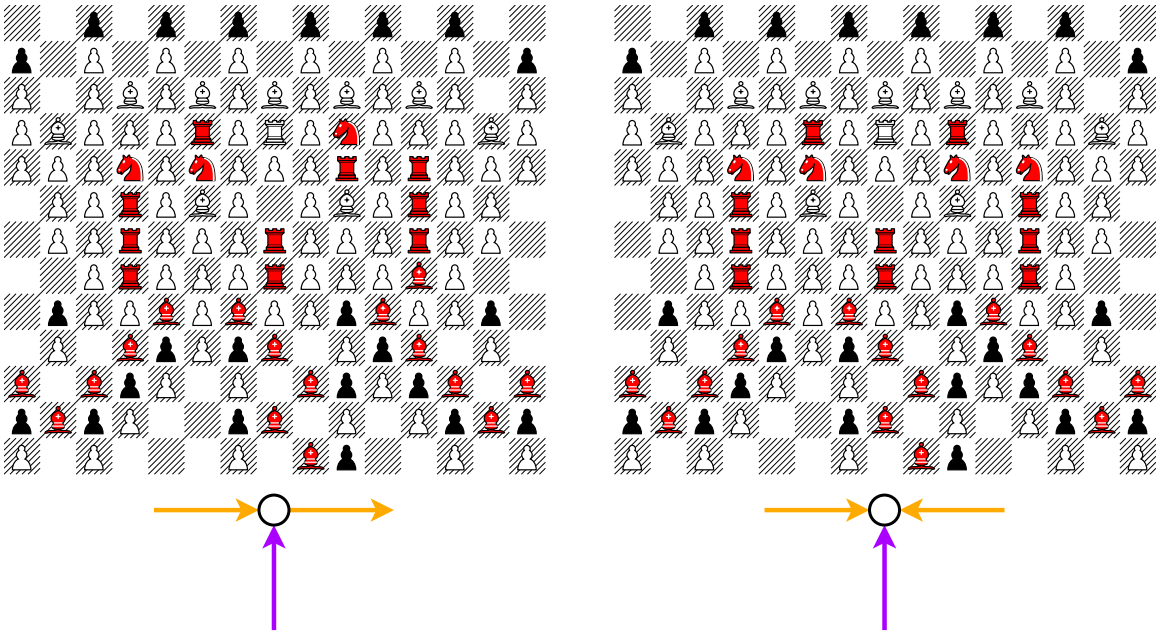


Figure 4-1: Our edge gadget. Since pawns care about their orientation, the gadget looks slightly different when the edge runs vertically compared to horizontally. This figure shows what both look like and how it can turn. Red represents movable white pieces.

Now we move on to the *vertex gadget*. There are two cases for a Subway Shuffle vertex: a vertex which can have two edges of one color both pointing into the vertex when it is empty, and a vertex which has one edge pointing in and one pointing out of the same color when it is empty. Note that a vertex which has all of the edges of a color pointing out does not make sense because a token of that color could never reach the vertex, so those edges are provably unusable. We implement both of these with the same vertex gadget, shown in Figure 4-2. This gadget has three edges coming out of it from the left, right, and bottom. The left and right edges are orange, and the bottom edge is purple. Which type of vertex the gadget represents depends on which red knights are present in the middle. The empty square in the middle is the vertex



(a) An empty vertex with one orange edge pointing in and one pointing out.

(b) An empty vertex with both orange edges pointing in.

Figure 4-2: The vertex gadget. The two edges coming out of the sides are orange edges, and the middle edge coming out of the bottom is purple. Which of two red knights which threaten the center empty space are present determines whether the orange edges are pointing in or out.

square. When it contains a knight, it represents a vertex occupied with an orange token, and when it contains a rook, it represents a vertex occupied with a purple token. To use the gadget, white moves all of the red pieces one step away from the vertex along one of the edges until the vertex square becomes empty. This represents a token leaving the vertex along that edge. Then white moves one of the red pieces that can move into the vertex square and continues moving all of the pieces along that path of red pieces; this represents moving a token into this vertex along that edge.

We use the argument about color changing from Lemma 2.8 to allow all three edges leaving the vertex to be bishop lines. The transition from the rooks in the gadget to the bishop edges is essentially this color-changing. All of the bishops are on dark squares, and our edges can be routed to connect arbitrary squares of the same color, so we will not need to worry about parity issues with connecting different

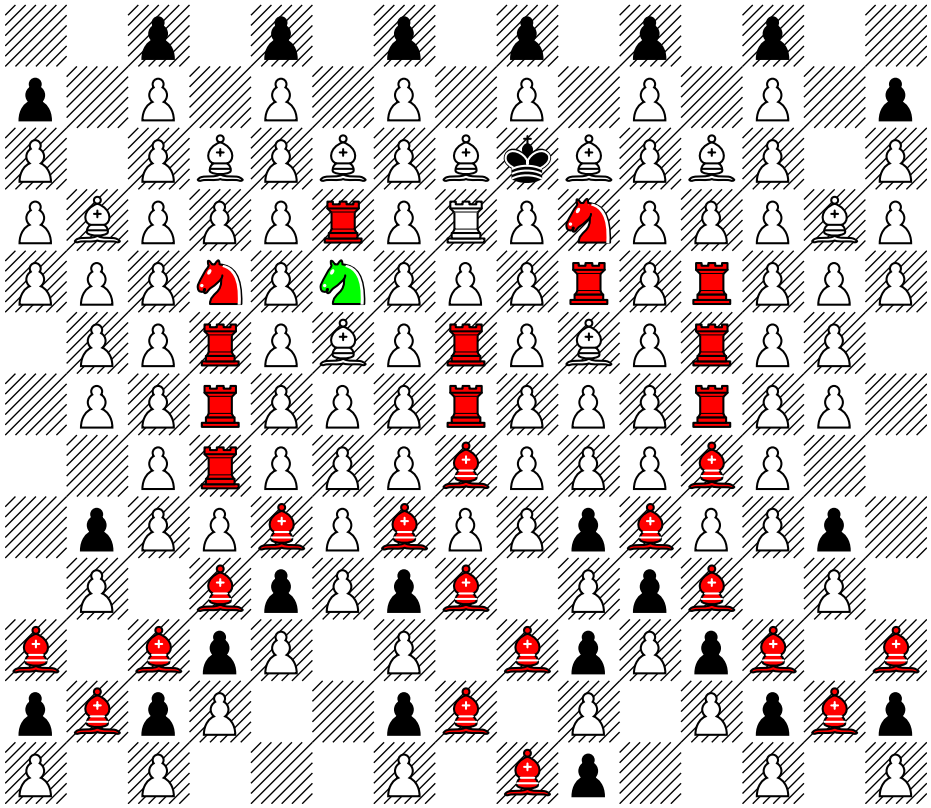


Figure 4-3: The win gadget. The black king is completely stuck; if it gets checked it will be immediately checkmated. The white knight highlighted in green is the only piece ever capable of accomplishing this. In order to do so, it must first move to the vertex, and then from there move to the right edge.

vertices.

Now we have to implement the Subway Shuffle target edge. This means making a *win gadget* which checks whether a particular edge is used. In order for an edge to be used, a piece must leave the vertex at the tail of the edge to move along that edge. Our win gadget is a modified version of the vertex gadget which allows white to checkmate if they can get a knight (representing an orange token) to leave the vertex along a specified edge. Our win gadget is depicted in Figure 4-3. Note that it is identical to the vertex gadget except for the replacement of one of white's pawns with a black king.

Lastly, we have a *do-nothing gadget*. The entire puzzle is solved by white; all of the pieces that can move in the gadgets are white's and the helpmate in question is white trying to checkmate black. A legal Chess game, however, must have alternating

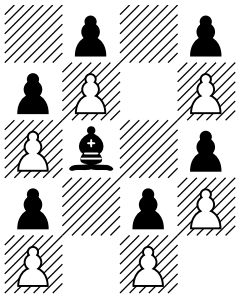


Figure 4-4: Do-nothing gadget, which allows black to pass forever.

moves by each side. Thus, we need to give black something to do. The gadget in Figure 4-4 accomplishes this, by giving black a trapped bishop that they can (and must) move back and forth in between white's moves.

It is worth noting that the positions that result from this reduction are reachable from a starting position, provided we make the board size a polynomial in the size of the Subway Shuffle instance large enough to have enough pieces and pawns to make the gadgets. Any extra pieces can capture each other prior to beginning to construct the position. We can also lock the white king away in a cage similar to the do-nothing gadget.

4.3 Correctness

Now we show that the only way white can ever checkmate black is by solving the Subway Shuffle problem and using the gadgets as they are intended to be used.

For the edge gadget, it is easy to check that no piece can move except the red bishops can move one space along the edge when it is in use.

For the vertex gadget, there is one other move white can try, but it does not do anything productive. White can try moving one of the pawns below the rook columns up one space when the rook above it moves up. This results in an immediately stuck position, so it is never useful for white to do. There are no other moves white can legally make outside of the reduction.

Chapter 5

Retrograde Chess Puzzles

Continuing our theme of cooperative Chess puzzles from Chapter 4, we now show that *retroChess* puzzles are PSPACE-complete. In retroChess puzzles, the goal is to determine whether a given position is legally reachable from the initial position. In order to analyze these puzzle, we first need to give a generalization of an initial position. For this, we use an $n \times n$ board where the first rank is all white non-pawn pieces with one king, and the second rank is all white pawns. Black's initial position is a mirror of white's on their back two ranks. The exact composition and ordering of the back rank is not relevant to our reduction; we only require that it has sufficiently many of each of the non-pawn piece types.

This result originally appeared in [3] in collaboration with Dylan Hendrickson, Erik Demaine, and Julian Wellman.

5.1 Overview

Lemma 5.1. *RetroChess puzzles are in PSPACE.*

Proof. An $n \times n$ Chess position takes only polynomial (in n) space to record. A nondeterministic polynomial-space machine can guess a sequence of moves, accepting when it reaches the target position; thus the problem is in NPSPACE = PSPACE. □

Theorem 5.2. *RetroChess puzzles are PSPACE-complete.*

First, by Lemma 5.1, retroChess puzzles are in PSPACE. Now we show that they are PSPACE-hard by reducing from the same problem, max-degree-3 two-color oriented Subway Shuffle, as in the previous chapter. As in the previous chapter, the basic structure will have white solving an instance of Subway Shuffle while black effectively passes their turn. But this time, rather than making moves, white will "undo" moves, which allows for pawns to move backwards or pieces to be uncaptured. If white succeeds, the win gadget will allow a piece to escape from the walls of the reduction. Once this hole appears, it will let more pieces forming the walls of the gadgets to start leaving, eventually unravelling all of the gadgets. At this point once the pieces are spread out, it is easy for the players to find a sequence of moves that could get there from the starting position.

Before we describe the gadgets, we will first make some observations about how moves work in retrograde puzzles. Instead of thinking about moves that can be made from a position, we will think about moves that could have just been done; we call these *unmoves*. All Chess pieces other than pawns unmove the same way that they move. Pawns are different, and all captures are different as well. A piece is never captured in an unmove; to undo a capture, a piece will unmove and the captured piece appears in its place. This means that, unlike in the checkmate reduction before, we will not have to worry about pieces being capturable, so the color of non-pawn pieces in the reduction is irrelevant.

Another important distinction is that pawns do not need a piece to be able to uncapture, so any pawn can always move diagonally backward to uncapture a piece unless the space it would unmove to is occupied. Due to Corollary 5.4 below, this will make walling our gadgets much harder than before since every Chess piece can unmove to some square to its left and some square to its right. This means that we cannot have any isolated gadgets in the middle of the board; in order for any block of pieces to be stuck, the block must extend to both the left and right edges of the board. This results in needing an additional gadget, a terminator that we attach to the ends of the construction which anchors everything to the edge of the board.

Lemma 5.3. *If the five nearest spaces in either file (column) immediately adjacent*

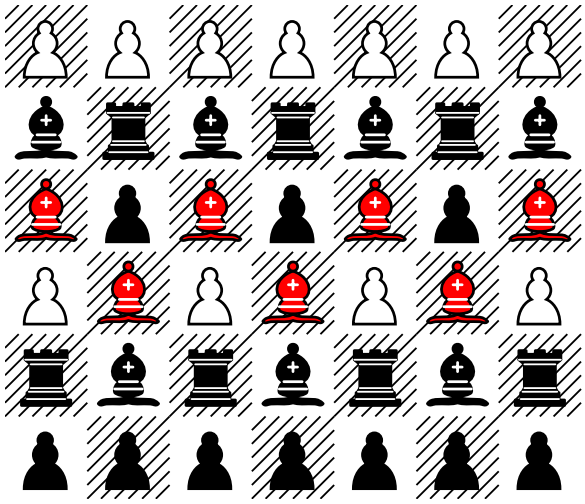


Figure 5-1: The edge gadget.

to a piece are empty, and the piece is not in the first two or last two ranks, then that piece can unmove into that file, leaving an empty space where it came from.

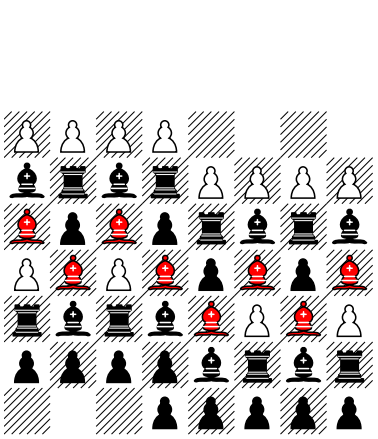
Proof. We simply look at each piece and note that every Chess piece can unmove into a square in the immediately adjacent file. For every non-pawn piece, it simply unmoves there and leaves an empty space immediately. For a pawn, it must uncapture to do this, which it can do because it is not in the first two or last two ranks. It can uncapture a non-pawn piece, and that piece can then unmove into the empty file immediately, leaving a hole. □

Corollary 5.4. *If a region of the board which does not include the first two or last two ranks has at least one piece and has an empty file adjacent to it, then a piece can unmove (possibly with multiple unmoves) to escape the region.*

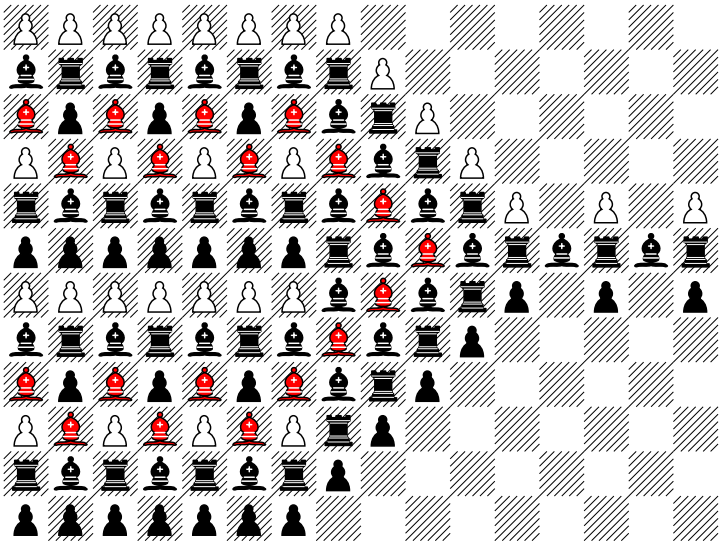
Proof. Without loss of generality, let the empty file be on the left. Consider the leftmost piece in the region. Then the conditions of Lemma 5.3 are satisfied, so the piece can unmove into that file. From here the piece can continue unmoving until it leaves the region. □

5.2 Gadgets

Now we describe the gadgets in our reduction.



(a) A shifted edge.



(b) A U-turn gadget for edges.

Figure 5-2: Edge routing gadgets: shift and U-turn.

First is the *edge gadget* shown in Figure 5-1. Like in the previous section, this gadget uses a line of bishops each of which move one space to represent the movement of a token. To keep the bishops locked in, we use a repeating pattern of pawns, rooks, and bishops across the top and bottom of the edge. Because pawns care about the orientation of the board, we cannot actually make vertical edges. Instead, we use the *turn* and *shift gadgets* shown in Figure 5-2; if you wiggle an edge back and forth with turns and shifts you can make it travel vertically up the board.

At the edge of the turn gadget, we have the *terminator gadget*, shown in Figure 5-3. As previously stated, because every piece is capable of unmoving to both adjacent files, we need a terminator gadget which connects these loose ends to the edge of the board. We have all of our terminator gadgets terminate either on the first rank of the board or on any other gadget. The terminator gadget in Figure 5-3 terminates on the first rank. To have one terminate on another gadget, it simply runs (diagonally) into the wall of pawns on either side of any of our gadgets, including another terminator. We will have only a single terminator gadget on each side of the construction terminate on the first rank, and all others will terminate on another gadget.

Now we describe the *vertex gadget*, shown in Figure 5-4. This vertex has a similar structure to the vertex gadget from the previous section, with potentially two knights

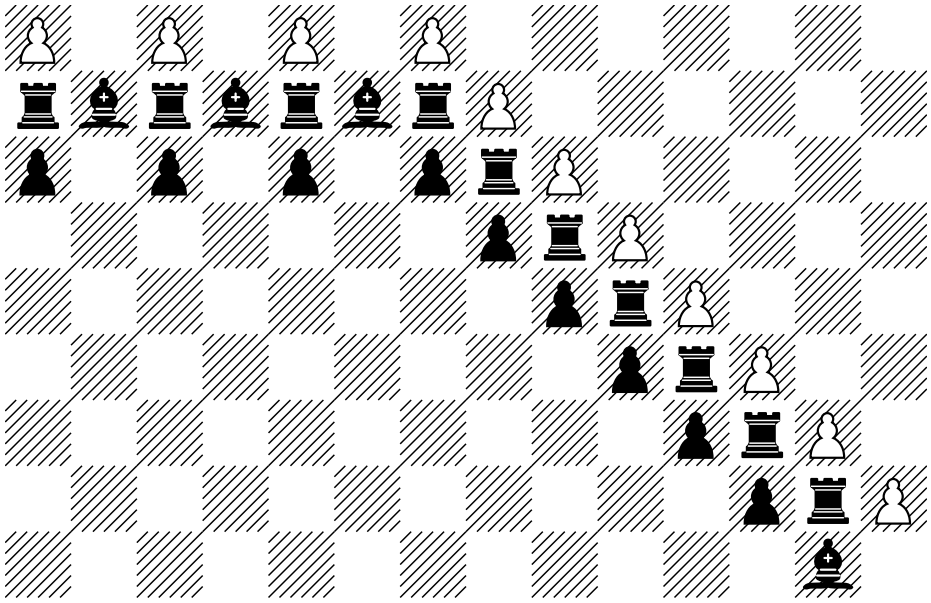


Figure 5-3: Terminator gadget that connects the loose ends of gadgets to the bottom rank. Unlike other figures, here we care that the first rank of this figure is the actual first rank of the Chess board. In particular, this means that the white pawn on the second rank cannot unmove, and that allows us to prove that everything is stuck.

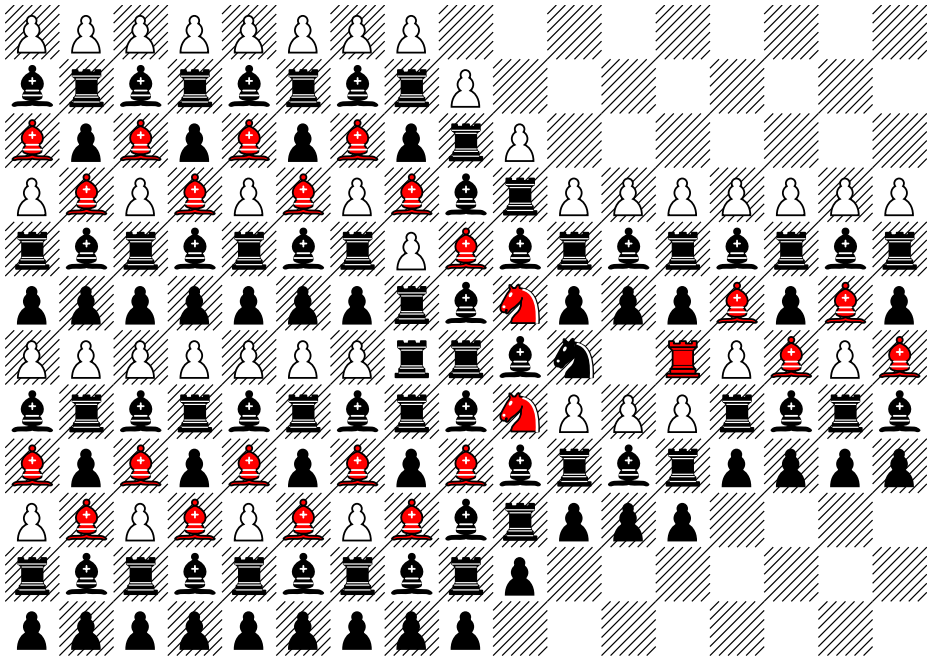


Figure 5-4: The vertex gadget. The empty square in the middle is the vertex; whether it is occupied by a knight or a rook determines the color of the token at this vertex.

and a rook representing the three tokens that can move in from connecting edges into the vertex. The two edges connected by a knight to the vertex are the orange edges;

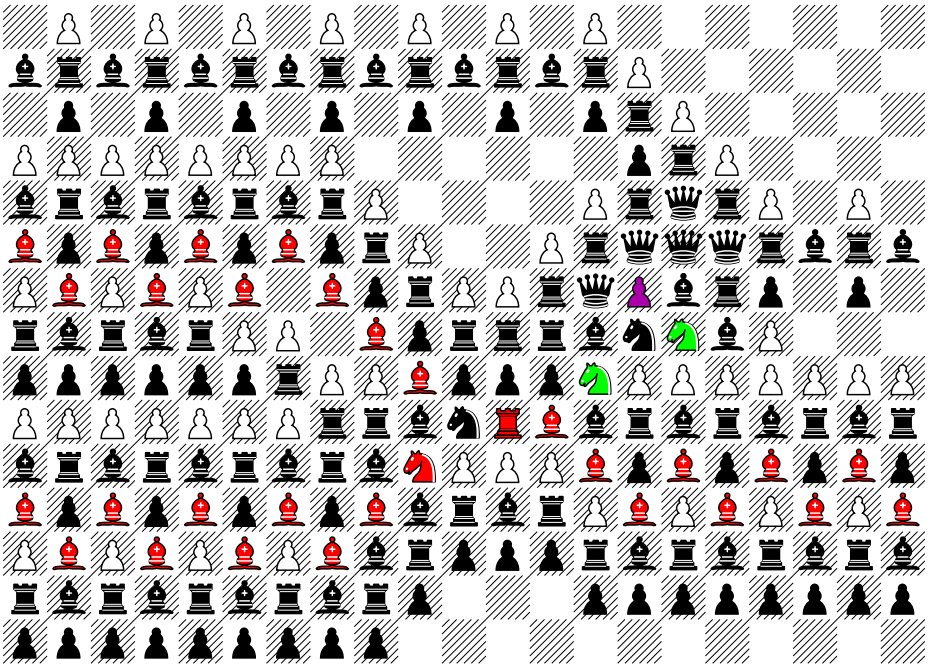


Figure 5-5: The win gadget. The purple pawn is a white pawn. If a knight leaves the vertex by the top edge, it allows the two white knights highlighted in green to follow it. Then the purple pawn can uncapture a knight where a knight was, which can move away, starting the unravelling process.

the rook is the purple edge. When both knights are present, we get a vertex which has both orange edges pointing into the vertex. If only one knight is present and the other is replaced by a bishop, only the edge with the knight points into the vertex and the other orange edge points out of the vertex.

5.3 Win Gadget and Self-Destruction

Finally we have the *win gadget* shown in Figure 5-5. Here we have modified the vertex gadget to add an extra hole a knight's move away from the top orange edge's connection to the vertex. When the top edge is used by having a knight enter it from the vertex, it can hop into this hole. This creates a second hole, which will be key to allowing the construction to unravel. Protruding from the top left and top right of the gadget are two terminator gadgets. The unravelling of these will be crucial to winning.

Normally, the green knight just to the right of the vertex is capable of unmoving

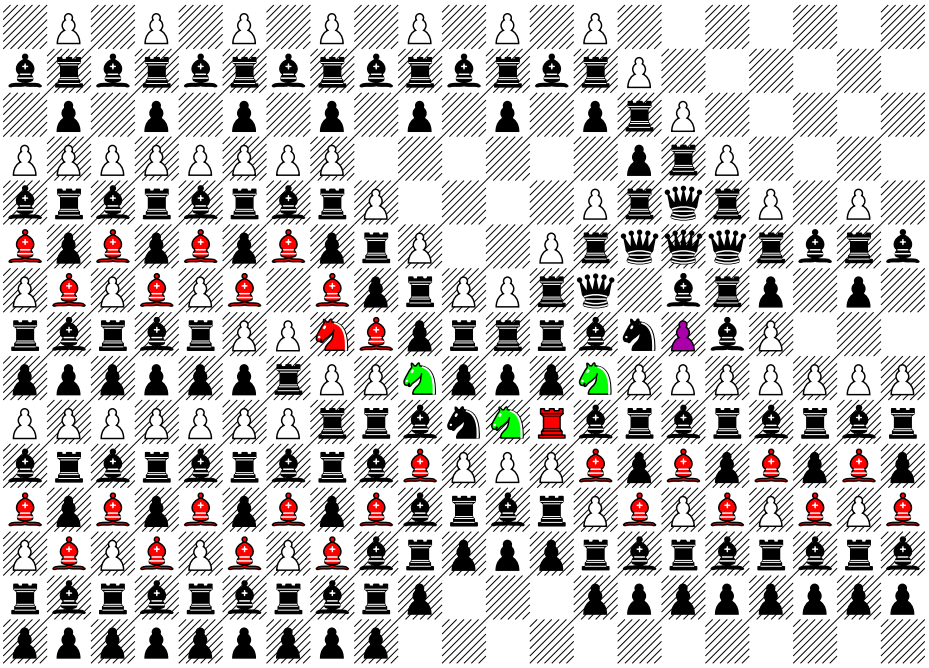


Figure 5-6: The win gadget after the first few unmoves to begin the unravelling are made. From here, the queens can leave allowing the terminator gadget in the top right to start unravelling.

to the vertex when it is empty. After this the second green knight can follow it unmoving into the square it just left. This then allows the purple (white) pawn to uncapture the square the knight was on. However, regardless of which piece the pawn uncaptures, the new piece is completely stuck and incapable of moving.

However, if there were a second hole for the green knights to jump into, then once the green knights unmove again, the purple pawn uncaptures a knight, which can then unmove to where one of the knights was. This is shown in Figure 5-6. At this point, the queens and rooks can shift around to let the queens start escaping. From here, everything begins to unravel. This is where the two terminator lines come in. Once a few queens leave, both of these lines can start to unravel.

We choose a layout of the Subway Shuffle instance such that the win vertex is the furthest north vertex, and these two terminator lines are on the outside face of the graph. We extend these lines very far away from the rest of the construction, which is possible because the choice of layout implies no other part of the construction is in as high a rank as win vertex. We have any other terminator lines from U-turn

gadgets which have not already terminated on another gadget terminate on these two terminator lines. Only these two terminators will eventually reach the first rank of the board, as shown in Figure 5-3.

Once these two lines have unravelled, it is now the case that our construction is in a region in the middle of the board with no pieces on either side of it. This means we can repeatedly apply Corollary 5.4, until every piece has left the construction. It is fairly easy once all of the pieces are free in the middle of the board to find a sequence of unmoves to send them home.

5.4 Counting Pieces

Now we need to do a piece counting argument, to show that the position is even plausible. One property of a starting Chess position is that it has only one pawn of each color in each file. Not only this, but pawns also cannot stray too far from their starting file. In particular, a pawn on rank n must come from a file at most n files away from its current position. Unfortunately, our construction can have many pawns in each file, and furthermore is constrained in how far it can be away from the bottom edge of the board due to the terminator gadgets.

However, the terminator gadget has the property that along the horizontal part, it has a white pawn (and similarly black pawn) density of only one pawn every two files. If we stretch the horizontal part far enough, and put the construction on a sufficiently far forward rank, we can use this to get the pawn density below one pawn per file. As long as this area with low pawn density is at a higher rank on the board than the number of files it is wide, with enough uncaptures the pawns can sort themselves into one pawn per file. The number of uncaptures required is at most quadratic in the number of pawns.

We also need to check the number of non-pawn pieces. To make sure that the board has the right amount of pieces, we simply have the board be much larger than our construction. Our pawns will need to make a large number of uncaptures during the unravelling, and to handle this we will have the board be much larger than

the number of pieces in our construction. It is always possible to keep uncapturing additional pieces and pawns so we do not need to worry about having too large of a board.

Finally, every legal Chess position needs to have one king of each color. Since we do not use kings anywhere in the construction, and their ability to roam free does not allow the players to unravel the position without solving the Subway Shuffle instance, we simply put the two kings in their home positions. This also ensures that both players always have legal unmoves allowing white and black to alternate making unmoves as is required in Chess.

Chapter 6

Future Work

In this thesis, we have shown that Subway Shuffle is PSPACE-complete and some applications of this result to other puzzles and motion planning.

The initial motivation for the restrictions in Subway Shuffle was showing that 1×1 Rush Hour is PSPACE-hard. Initially, we had wanted to allow any vertex of degree at most 3; however, it was difficult to represent degree 3 vertices all of one color in Rush Hour. Can these vertices can be simulated by the other degree 3 vertices? The motion planning interpretation gives a potentially useful framework to investigate this, as there has been recent work on what kinds of motion-planning gadgets can simulate each other [11].

For Rush Hour, it still remains open whether this puzzle remains PSPACE-complete without fixed blocks. Our proof relies heavily on cars being blocked by walls. 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 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 6.1 below, 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

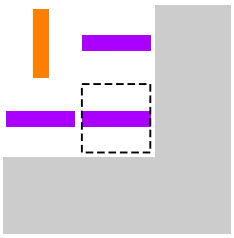


Figure 6-1: Let the gray area be accessible by the bubble. Then the boxed car is at the corner of the boundary of the accessible region, and regardless of its orientation it must also be accessible by the the bubble.

gadgets.

Theorem 6.1. *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 6-1. 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. □

For both helpmate and retrograde Chess, we note that pawns being a piece with very low mobility are crucial to both proofs. Are either of these problems still computationally difficult if the position has no pawns? This would not allow pieces to be enclosed in a small region, similar to the accessibility issue with fixed blocks in Rush Hour. Thus, it will like take substantially different techniques to show hardness in a context without pawns.

Bibliography

[1] Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Dylan H. Hendrickson, Adam Hesterberg, Matias Korman, Oliver Korten, Jayson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots, 2020.

[2] Josh Brunner, Lily Chung, Erik D. Demaine, Dylan Hendrickson, Adam Hesterberg, Adam Suhl, and Avi Zeff. 1×1 Rush Hour with fixed blocks is PSPACE-complete. In *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, pages 7:1–7:14, La Maddalena, Italy, September 28–30 2020.

[3] Josh Brunner, Erik D. Demaine, Dylan Hendrickson, and Julian Wellman. Complexity of retrograde and helpmate chess problems: Even cooperative chess is hard. In *Proceedings of the the 31st International Symposium on Algorithms and Computation (ISAAC 2020)*, pages 33:1–33:14, Hong Kong, December 14–18 2020.

[4] Marzio De Biasi and Tim Ophelders. Subway Shuffle is PSPACE-complete. Manuscript, February 2015. `http://www.nearly42.org/cstheory/subway-shuffle-is-pspace-complete/`.

[5] Erik D. Demaine, Isaac Grosof, Jayson Lynch, and Mikhail Rudoy. Computational complexity of motion planning of a robot through simple gadgets. In *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, pages 18:1–18:21, La Maddalena, Italy, June 13–15 2018.

[6] Erik D. Demaine, Dylan Hendrickson, and Jayson Lynch. Toward a general theory of motion planning complexity: Characterizing which gadgets make games hard. In *Proceedings of the 11th Conference on Innovations in Theoretical Computer Science (ITCS 2020)*, pages 62:1–62:42, Seattle, Washington, January 12–14 2020.

[7] FIDE. FIDE Laws of Chess. `https://handbook.fide.com/chapter/E012018`, 2018.

[8] 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.

[9] Robert A. Hearn. *Games, Puzzles, and Computation.* PhD thesis, Massachusetts Institute of Technology, 2006.

[10] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* AK Peters/CRC Press, 2009.

[11] Dylan Hendrickson. Gadgets and gizmos: A formal model of simulation in the gadget framework for motion planning. Master's thesis, Massachusetts Institute of Technology, 2021.

[12] David Hooper and Kenneth Whyld. *The Oxford Companion to Chess.* Oxford University Press, 2nd edition, 1992.

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

[14] Yaroslav Shitov. Chess God's number grows exponentially. arXiv:1409.1530, 2014. `https://arxiv.org/abs/1409.1530`.

[15] 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.

[16] John Tromp and Rudi Cilibrasi. Limits of Rush Hour Logic complexity. *arXiv preprint cs/0502068*, 2005. `https://arXiv.org/abs/cs/0502068`.

[17] Wikipedia. Helpmate. `https://en.wikipedia.org/wiki/Helpmate`, 2020.