Journal of Information Processing Vol.0 1–14 (??? 1992)

[DOI: 10.2197/ipsjjip.0.1]

Regular Paper

PSPACE-completeness of Pulling Blocks to Reach a Goal

Joshua Ani[1,a)] Sualeh Asif[1,b)] Erik D. Demaine[1,c)] Yevhenii Diomidov[1,d)] Dylan Hendrickson[1,e)] Jayson Lynch[1,f)] Sarah Scheffler[2,g)] Adam Suhl[3,h)]

Received: xx xx, xxxx, Accepted: xx xx, xxxx

Abstract: We prove PSPACE-completeness of all but one problem in a large space of pulling-block problems where the goal is for the agent to reach a target destination. The problems are parameterized by whether pulling is optional, the number of blocks which can be pulled simultaneously, whether there are fixed blocks or thin walls, and whether there is gravity. We show NP-hardness for the remaining problem, Pull?-1FG (optional pulling, strength 1, fixed blocks, with gravity).

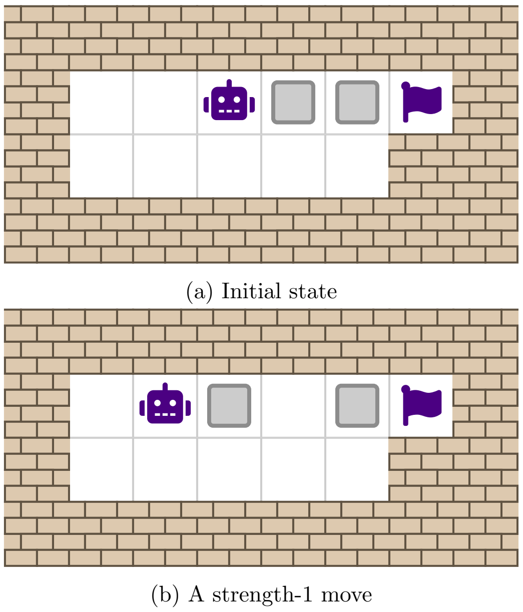
Keywords: motion planning, hardness, puzzles

1. Introduction

In the broad field of ***motion planning***, we seek algorithms for actuating or moving mobile agents (e.g., robots) to achieve certain goals. In general settings, this problem is PSPACE-complete [3, 15], but much attention has been given to finding simple variants near the threshold between polynomial time and PSPACE-complete; see, e.g., [13]. One interesting and well-studied case, arising in warehouse maintenance, is when a single robot with $O(1)$ degrees of freedom navigates an environment with obstacles, some of which can be moved by the robot (but which cannot move on their own). Research in this direction was initiated in 1988 [20].

A series of problems in this space arise from computer puzzle games, where the robot is the agent controlled by the player, and the movable obstacles are ***blocks***. The earliest and most famous such puzzle game is ***Sokoban***, first released in 1982 [19]. Much later, this game was proved PSPACE-complete [4, 13]. In Sokoban, the agent can ***push*** movable 1×1 blocks on a square grid, and the goal is to bring those blocks to target locations. Later research in ***pushing-block puzzles*** considered the simpler goal of simply getting the robot to a target location, proving various versions NP-hard, NP-complete, or PSPACE-complete [5, 9, 11].

In this paper, we study the Pull series of motion-planning problems [14, 16], where the agent can ***pull*** (instead of push) movable 1×1 blocks on a square grid. Fig. 1 shows a simple example. This type of block-pulling mechanic (sometimes together with a block-pushing mechanic) appears in many real-world video games, such as Legend of Zelda, Tomb Raider, Portal, and Baba Is You.


(a) Initial state

(b) A strength-1 move

Fig. 1: A pulling-block problem. The robot is the agent, the flag is the goal square, the light gray blocks can be moved, and the bricks are fixed in place. *Robot and flag icons from Font Awesome under CC BY 4.0 License.*

We study several different variants of Pull, which can be combined in arbitrary combination:

[1] MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA
[2] Boston University, Boston, MA, USA
[3] Algorand, Boston, MA, USA
a) joshuaa@mit.edu
b) sualeh@mit.edu
c) edemaine@mit.edu
d) diomidov@mit.edu
e) dylanhen@mit.edu
f) jaysonl@mit.edu
g) sscheff@bu.edu
h) asuhl@mit.edu

Problem	Forced	Strength	Features	Gravity	Our result	Previous best
PULL?-kF	no	$k \geq 1$	fixed blocks	no	PSPACE-complete [§2]	NP-hard [16]
PULL?-$*$F	no	∞	fixed blocks	no	PSPACE-complete [§2]	NP-hard [16]
PULL!-kF	yes	$k \geq 1$	fixed blocks	no	PSPACE-complete [§2]	
PULL!-$*$F	yes	∞	fixed blocks	no	PSPACE-complete [§2]	
PULL?-1FG	no	$k = 1$	fixed blocks	yes	NP-hard [§4]	
PULL?-1WG	no	$k = 1$	thin walls	yes	PSPACE-complete [§3.2]	
PULL?-kFG	no	$k \geq 2$	fixed blocks	yes	PSPACE-complete [§3.2]	
PULL?-$*$FG	no	∞	fixed blocks	yes	PSPACE-complete [§3.2]	
PULL!-kFG	yes	$k \geq 1$	fixed blocks	yes	PSPACE-complete [§3.3]	
PULL!-$*$FG	yes	∞	fixed blocks	yes	PSPACE-complete [§3.3]	

Table 1: Summary of our results.

(1) **Optional/forced pulls:** In PULL!, every agent motion that can also pull blocks must pull as many as possible (as in many video games where the player input is just a direction). In PULL?, the agent can choose whether and how many blocks to pull. Only the latter has been studied in the literature, where it is traditionally called PULL; we use the explicit "?" to indicate optionality and distinguish from PULL!.

(2) **Strength:** In PULL-k, the agent can pull an unbroken horizontal or vertical line of up to k pullable blocks at once. In PULL-$*$, the agent can pull any number of blocks at once.

(3) **Fixed blocks/walls:** In PULL-F, the board may have fixed 1×1 blocks that cannot be traversed or pulled. In PULL-W, the board may have fixed thin (1×0) walls; this is more general because a square of thin walls is equivalent to a fixed block. Thin walls were introduced in [7].

(4) **Gravity:** In PULL-G, all movable blocks fall downward after each agent move. Gravity does not affect the agent's movement.

Table 1 summarizes our results: for all variants that include fixed blocks or walls, we prove PSPACE-completeness for any strength, with optional or forced pulls, and with or without gravity, with the exception of PULL?-1FG for which we only show NP-hardness.

The only previously known hardness result for this family of problems is NP-hardness for both PULL?-kF and PULL?-$*$F [16]. In some cases, our results are stronger than the best known results for the corresponding PUSH (pushing-block) problem; see [14]. More complex variants PULLPULL (where pulled blocks slide maximally), PUSHPULL (where blocks can be pushed and pulled), and STORAGE PULL (where the goal is to place multiple blocks into desired locations) are also known to be PSPACE-complete [7, 14].

Our reductions are from Asynchronous Nondeterministic Constraint Logic (NCL) [13, 18] and planar 1-player motion planning [2, 10]. In Section 2, we reduce from NCL to prove PSPACE-hardness of all nongravity variants. In Section 3, we use the motion-planning-through-gadgets framework [10] to prove PSPACE-completeness of most variants with gravity, including all variants with forced pulling and variants with optional pulling and either thin walls or fixed blocks with $k \geq 2$. These reductions use two particular gadgets for 1-player motion planning, the newly introduced ***nondeterministic locking 2-toggle*** (a variant of the locking 2-toggle from [10]) and the ***3-port self-closing door*** (one of the self-closing doors from [2]). Although the latter gadget is proved hard in [2], for completeness, we give a more succinct proof in Appendix A.1. In Section 4, we prove NP-hardness for the one remaining case of PULL?-1FG, again reducing from 1-player planar motion planning, this time with an NP-hard gadget called the crossing NAND gadget [2].

2. Pulling Blocks with Fixed Blocks is PSPACE-complete

In this section, we show the PSPACE-completeness of all variants of pulling-block problems we have defined without gravity, namely PULL?-kF, PULL?-kW, PULL!-kF, and PULL!-kW for $k \geq 1$, and PULL?-$*$F, PULL?-$*$W, PULL!-$*$F, and PULL!-$*$W. We do this through a reduction from Nondeterministic Constraint Logic [13], which we describe briefly before moving on to the main proof.

2.1 Asynchronous Nondeterministic Constraint Logic

NONDETERMINISTIC CONSTRAINT LOGIC (NCL) takes place on ***constraint graphs***: a directed graph where each edge has weight 1 or 2. Weight-1 edges are called ***red***; weight-2 edges are called ***blue***. The "constraint" in NCL is that each vertex must maintain in-weight at least 2. A ***move*** in NCL is a reversal of the direction of one edge, while maintaining compliance with the constraint.

In ***asynchronous*** NCL, the process of switching the orientation of an edge does not happen instantaneously, but instead it takes a positive amount of time, and it is possible to be in the process of switching several edges simultaneously. When an edge is in the process of being reversed, it is not oriented toward either vertex. Viglietta [18] showed that this model is equivalent to the regular (synchronous) model, because there is no benefit to having an edge in the intermediate unoriented state. In this work, we only use the asynchronous NCL model; any mention of NCL should be understood to mean asynchronous NCL.

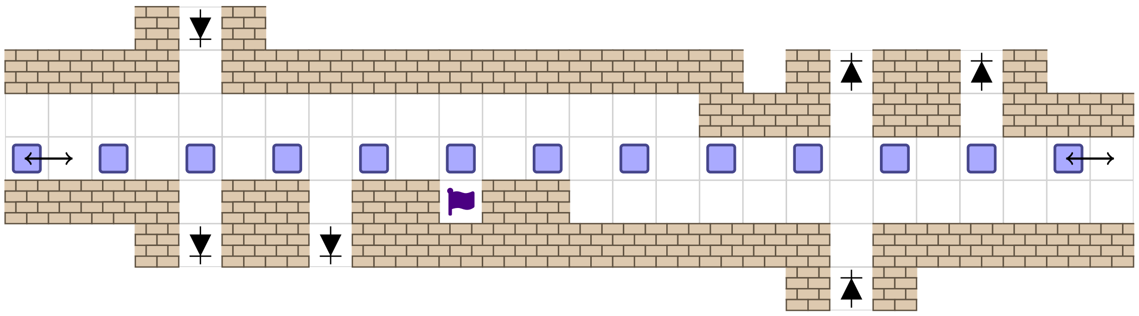


Fig. 3: Edge gadget for building NCL constraint graph in Pull?-kF. This edge encodes an NCL wire pointing to the right (opposing the blocks, which are moved to the left). This figure shows the winning edge, which if flipped allows reaching the goal; nonwinning edges are the same but without the flag.

An instance of Nondeterministic Constraint Logic consists of a constraint graph G and an edge e of G, called the ***target edge***. The output is yes if there is a sequence of moves on G that reverses the direction of e, and no otherwise. Nondeterministic Constraint Logic is PSPACE-complete, even for planar constraint graphs that have only two types of vertices: AND (two red edges, one blue edge) and OR (three blue edges). We will reduce from the planar, AND/OR, asynchronous version of NCL to show pulling-block problems without gravity PSPACE-hard. For more description of NCL, including a proof of PSPACE-completeness, the reader is referred to [13].

2.2 NCL Gadgets in Pulling Blocks

In order to embed an NCL constraint graph into Pull?-kF, we need three components, corresponding to NCL edges (which can attach to AND and OR gadgets in all necessary orientations, and that allows the player to win if the winning edge is flipped), AND vertices, and OR vertices. In each of these gadgets, we will show that if the underlying NCL constraint is violated, then the agent will be “trapped”, meaning that the state is in an ***unrecoverable configuration***, a concept used in several previous blocks games [4,13]. This occurs when the agent makes a pull move after which no set of moves will lead to a solution, generally because the agent has trapped itself in a way that no pull can be made *at all* (or only a few more pull moves may be made, and all of them lead to a state such that there are no more pull moves).

Diode Gadget. Before describing the three main gadgets, we describe a helper gadget, the ***diode***, shown in Fig. 2. The diode can be repeatedly traversed in one direction but never the other. It was introduced in [16].

In the next three sections, we describe the three main gadgets in turn.

2.2.1 Edge Gadget

The edge gadget, shown in Fig. 3, encodes a single edge (of either weight) from NCL into Pull?-kF. The blocks can shift by exactly one space; whether they are moved left or right (or up or down) corresponds to the NCL edge pointing right or left (or down or up) respectively. (Note that the orientation of the NCL edge ***opposes*** the direction of the blocks.) The ends of the wires will be in vertex gadgets, which are explained below.

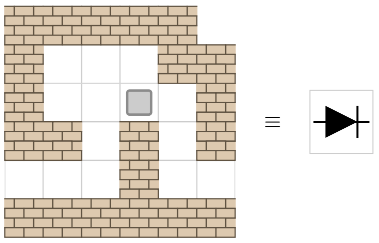


Fig. 2: Diode gadget, which can be repeatedly traversed from left to right but never from right to left. In diagrams to follow it will be represented by the diode symbol.

The diodes on the sides are to allow the agent to traverse between edges without going through a vertex gadget. The position and orientation of the diode gadgets prevent the agent from pulling a block out of the edge gadget without trapping itself.

The edge shown in Fig. 3 only goes in a straight line; it may turn corners via the corner gadget in Fig. 4. We fix small misalignment of wires at the gadgets, not on the wires.

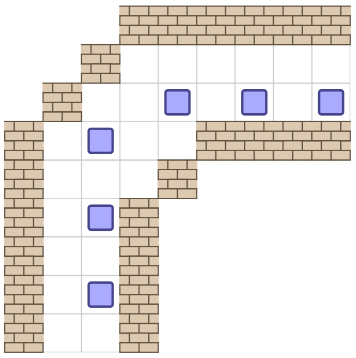


Fig. 4: Gadget for allowing wires to turn corners. Currently oriented with the NCL wire pointed left/down.

This wire gadget also allows encoding the win condition in the target edge: the finish tile can be in a small room coming out of the wall, blocked by the blocks’ current placement, as shown in Fig. 3; it can then be reached only if the edge can be reversed. (Without the flag, Fig. 3 shows an ordinary edge gadget.)

The agent could try to cheat by pulling a block out of the wire, by pulling a block up into the downward-facing diode at the top-left of the gadget, or symmetrically by pulling a block down into the upward-facing diode at the bottom-right of the gadget. If the agent does so, then we claim that the game enters an unrecoverable configuration. First, the agent cannot return to the wire gadget the way it came, because the block it just pulled is blocking the way for forward traversal. Second, the agent cannot traverse the adjacent diode because it points the wrong way. Therefore, the agent cannot pull blocks without reaching an unrecoverable configuration, except for the moves which correspond to reversing the NCL edge.

The player may partially reverse an NCL edge and exit before completing the reversal. This leaves a gap of two empty squares between consecutive movable blocks somewhere in the edge gadget. This is why we reduce from asynchronous NCL; while in this partially reversed state, the NCL edge is not oriented toward either vertex, and each vertex gadget behaves as though the edge were oriented away from it.

2.2.2 OR gadget

The OR gadget, shown in Fig. 5, consists of an area fully enclosed by walls except at three connections to edge gadgets. The agent can enter and exit the enclosed area through an edge connection when the blocks in the edge are pulled away from the OR gadget (i.e., when the NCL edge points in). When the edge blocks are pulled inward (i.e., NCL edge points out), the agent cannot escape the enclosed area through that edge gadget. Thus, when inside the enclosed area, the agent may pull an edge block in (i.e., start switching the NCL edge to point out), but if both other NCL edges already point out, the agent will be trapped inside the gadget. This enforces the constraint that at least one edge must point toward the OR vertex.

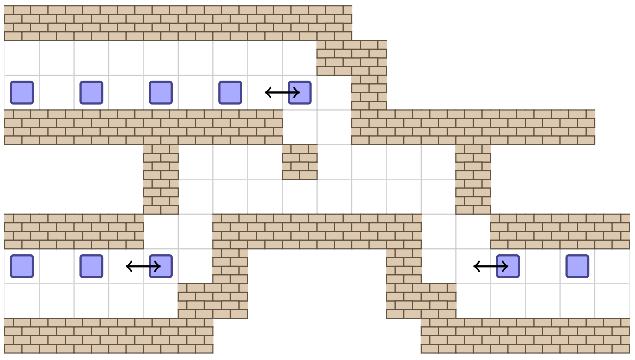


Fig. 5: Gadget for an NCL OR vertex. Currently, the left edge points out, the right edge points in, and the top edge points out.

This gadget can vary in size: if some edge gadgets are slightly misaligned, the middle part can be made bigger or smaller to accommodate—the interior of the gadget needs only to be fully enclosed except at the incident edge gadgets.

Lemma 2.1. *The OR gadget enforces exactly the constraints of an NCL OR vertex.*

Proof. The NCL OR constraint is that at least one edge must point into the vertex at all times. If the constraint is satisfied, then at least one wire of blocks is pushed out, and the agent can escape through that edge gadget. If the agent tries to violate the constraint, then the first move it must take in order to make the last edge point away from the vertex is to pull the last block of the corresponding edge gadget inward. This puts the gadget into an unrecoverable condition: the agent is now trapped in the OR gadget. □

2.2.3 AND gadget

An NCL AND vertex has two red (weight 1) edges and one blue (weight 2) edge. Its constraint is that the blue edge may point outward only if both red edges point inward. Our AND vertex gadget in Pull?-kF is shown in Fig. 6.

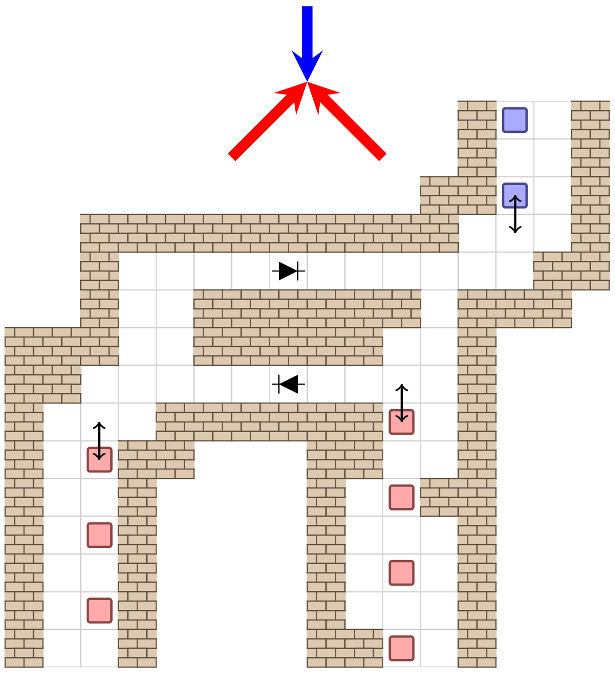


Fig. 6: Gadget for an NCL AND vertex, currently with all three edges pointing in. The lower diode allows traversal from right to left, and is blocked if the right red edge is pointing away; the upper diode allows traversal from left to right. If the (top) blue edge is pointing in, the agent can escape through that edge gadget. To escape the AND gadget after making the blue edge point away by pulling the bottommost blue block down, both red edges must be pointing in, so that the agent can go through the bottom diode and escape through the left red edge.

Like the OR gadget, the AND gadget traps the agent inside the gadget if the agent tries to violate the NCL constraint. Two of the edge connections, one red and one blue, are like those of the OR gadget, allowing the agent to escape the gadget into the edge if the blocks have been pulled outward (i.e., if the NCL edge points inward). The remaining red edge connection is different: the agent can never escape into this edge. Instead, when this edge's blocks are pulled outward (i.e., when the NCL edge points inward), it unblocks a path allowing the agent to traverse from the blue-exit side of the gadget to the red-exit side of the gadget.

An agent inside the gadget trying to pull the blue edge block inward (i.e., start switching the blue NCL edge to point outward) is trapped on the blue-exit side of the gadget unless this special red edge has its blocks pulled outward (i.e., the red NCL edge points inward); even then, the agent is still trapped inside the gadget unless the other red edge also has its blocks pulled outward to allow escape (i.e., the other red NCL edge also points inward). Thus an agent trying to switch the blue NCL edge outward is trapped unless both red NCL edges point inward, enforcing the AND condition. This is illustrated in Fig. 7.

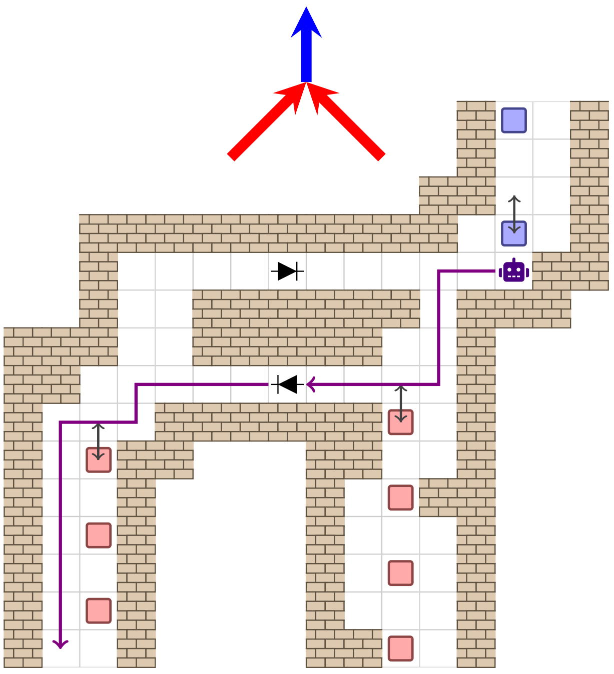


Fig. 7: The agent has just started to flip the blue NCL edge outward by pulling a blue block inward. Both red NCL edges are pointed inward, so the agent can traverse the lower diode and escape out the left red edge. Note that if either red NCL edge were pointed outward, escape would be impossible.

The AND gadget contains two reusable one-way gadgets. The lower diode is blocked if the right red edge points away, trapping the agent if the blue edge also points away, but allowing the agent to traverse from right to left and escape if both red edges point in. The upper diode allows the agent to travel from the red-exit side to the blue-exit side regardless of the state of the third edge; this is necessary to ensure the agent can escape out the blue exit (if the blue edge points in) after flipping either red edge to point away, as illustrated in Fig. 8.

As with the OR gadget, if the incident edge gadgets are aligned with different parity, this gadget can be expanded slightly to accommodate the edge gadgets.

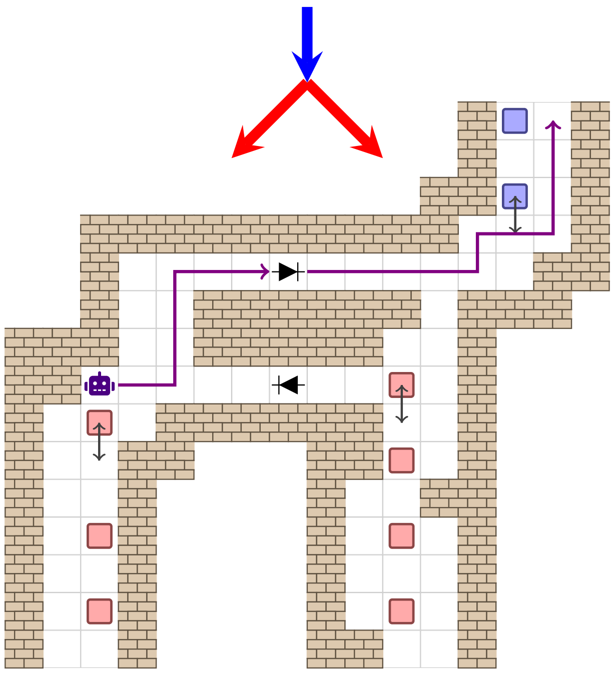


Fig. 8: The agent has just started to flip the left red NCL edge outward by pulling the red block inward. Because the blue NCL edge points inward, the agent can traverse the top diode and escape out the blue edge.

Lemma 2.2. *The AND gadget enforces exactly the constraints of an NCL AND vertex.*

Proof. The NCL AND constraint is that either the blue edge (top) or both red edges (bottom left and right) point toward the vertex. If both red edges are pointing in, then the agent can pull the bottom blue block in and then escape through the left red edge gadget by going through the bottom diode. If the blue edge is pointing in, then the agent can always escape through the blue edge gadget. The agent can orient the left red edge out by entering through the red-exit, going through the top diode, and leaving through the blue exit; and it can orient the right red edge out by entering and leaving through the blue exit.

The agent can attempt to violate the constraint by making both the blue edge and at least one red edge point away from the AND vertex. We consider the two red edges separately. First, if both the left red edge and the blue edge point out, then both exit points from the gadget are blocked, so the agent is trapped. Second, suppose that the constraint becomes violated by both the blue edge and the right red edge pointing away. Then the agent has just pulled either the bottom blue block on the top red block into the AND gadget, and the agent is in the right side of the gadget. The agent is trapped: the blue exit is blocked by the blue edge pointing away, the bottom diode is blocked by the red edge pointing away, and the top diode cannot be traversed from right to left. □

2.3 Proof of PSPACE-completeness

We first observe that every pulling-block problem we consider is in PSPACE.

Lemma 2.3. *Every pulling-block problem defined in Section 1 is in* PSPACE.

Proof. The entire configuration while playing on instance of a pulling-block problem can be stored in polynomial space (e.g., as a matrix recording whether each cell is empty, a fixed block, a movable block, the agent's location, or the finish tile). There is a simple nondeterministic algorithm which guesses each move and keeps track of the configuration using only polynomial space, accepting if the agent reaches the goal square. Thus the problem is in NPSPACE, so by Savitch's Theorem [17] it is also in PSPACE. □

Theorem 2.4. Pull?-*k*F *and* Pull!-*k*F *PSPACE-complete for* $k \geq 1$ *and* $k = *$.

Proof. Lemma 2.3 gives us containment in PSPACE. For PSPACE-hardness, we reduce from asynchronous NCL (as defined in Section 2.1).

Given a planar AND/OR NCL graph, we construct an instance of Pull?-*k*F or Pull!-*k*F as follows. First, embed the graph in a grid graph. Scale this grid graph by enough to fit our gadgets; 20×20 suffices. At each vertex, place the appropriate AND or OR vertex gadget. Place edge gadgets in the appropriate configuration along each edge, using corner gadgets on turns. Adjust the vertex gadgets to accommodate the alignment of the edge gadgets incident to them. Finally, place the goal tile in the edge gadget corresponding to the target edge so that it is accessible only if the target edge is flipped, and place the agent on any empty tile.

The agent can walk through edge gadgets to visit any NCL edge or vertex, and by Lemmas 2.1 and 2.2, flip edges in accordance with the rules of NCL. Ultimately, it can reach the goal tile if and only if the target edge of the NCL instance can be reversed.

In our construction, the agent never has the opportunity to pull more than 1 block at a time. Thus the reduction works for Pull?-*k*F for any $k \geq 1$, including $k = *$. In addition, the agent never has to choose not to pull a block when taking a step, so the reduction works for Pull!-*k*F as well as Pull?-*k*F. □

Corollary 2.5. Pull?-*k W and* Pull!-*k W are* PSPACE-*complete for* $k \geq 1$ *and* $k = *$.

Proof. A fixed block can be simulated using four thin walls drawn around a single tile, so our constructions can be built using thin walls instead of fixed blocks. Formally, this is a reduction from Pull?-*k*F to Pull?-*k*W and a reduction from Pull!-*k*F to Pull!-*k*W. □

3. Pull?-*k*FG is PSPACE-complete for $k \geq 2$ and Pull!-*k*FG is PSPACE-complete for $k \geq 1$

In this section, we show PSPACE-completeness results for most of the pulling-block variants with gravity. In Section 3.1, we introduce and prove results about ***1-player motion planning*** from the motion-planning-through-gadgets framework introduced in [8], which will be the basis for the later proofs. In Section 3.2, we show PSPACE-completeness for Pull?-*k*FG with $k \geq 2$, for Pull?-∗FG, for Pull?-*k*WG with $k \geq 1$, and for Pull?-∗WG. In Section 3.3, we show PSPACE-completeness for Pull!-*k*FG with $k \geq 1$, and for Pull!-∗FG. The one case missing from this collection is Pull?-1FG, which we prove NP-hard later in Section 4.

3.1 1-player Motion Planning

1-player motion planning refers to the general problem of planning an agent's motion to complete a path through a series of gadgets whose state and traversability can change when the agent interacts with them. In particular, a ***gadget*** is a constant-size set of locations, states, and traversals, where each traversal indicates that the agent can move from one location to another while changing the state of the gadget from one state to another. A system of gadgets is constructed by connecting the locations of several gadgets with a graph, which is sometimes restricted to be planar. The decision problem for 1-player motion planning is whether the agent, starting from a specified stating location, can follow edges in the graph and transitions within gadgets to reach some goal location.

Our results use the known results that 1-player planar motion planning is PSPACE-complete for the following gadgets:

(1) The ***locking 2-toggle***, shown in Fig. 9, is a three-state two-tunnel reversible deterministic gadget. In the ***middle state***, both tunnels can be traversed in one direction, switching to one of two ***leaf states***. Each leaf state only allows the transition back across that tunnel in the opposite direction, returning the gadget to the middle state. Traversing one tunnel "locks" the other side from being used until the prior traversal is reversed. 1-player planar motion planning with locking 2-toggles was shown PSPACE-complete in [10]. In Section 3.1.1, we strengthen the result in [10] by showing that 1-player motion planning with locking 2-toggle remains hard even if the initial configuration of the system has all gadgets in leaf (locked) states.

(2) The ***nondeterministic locking 2-toggle***, shown in Fig. 10, is a four-state gadget where each state has two transitions, each across the same tunnel. The top pair of states each allow a single traversal downward, and allow the agent to choose either of the two bottom states for the gadget. Similarly, the bottom pair of states each allow a single traversal upward to one of the top states. We can imagine this as being similar to the locking 2-toggle if the tunnel to be taken next is guessed ahead of time: the bottom state of the locking 2-toggle is split into two states which together allow the same traversals, but only if the agent picks the correct one ahead

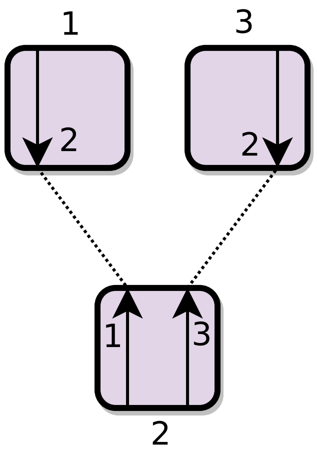


Fig. 9: State space of the locking 2-toggle [10].

of time.

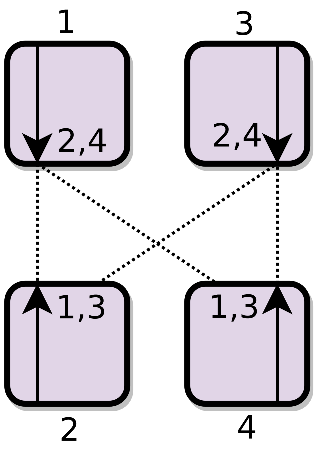


Fig. 10: State space of the nondeterministic locking 2-toggle.

In Section 3.1.1, we show that 1-player motion planning with the nondeterministic locking 2-toggle is PSPACE-complete.

(3) The ***door gadget*** has three directed tunnels called ***open***, ***close***, and ***traverse***. The traverse tunnel is open or closed depending on the state of the gadget and does not change the state. Traversing the open or close tunnel opens or closes the traverse tunnel, respectively. 1-player motion planning with door gadgets was shown PSPACE-complete in [1] and explored more thoroughly (in particular, proved hard for most planar cases) in [2].

(4) The ***3-port self-closing door***, shown in Fig. 11, is a gadget with a tunnel that becomes closed when the agent traverses it and a location that the agent can visit to reopen the tunnel. It has an ***opening port***, which opens the gadget, and a ***self-closing tunnel***, which is the tunnel that closes when traversed.

In Appendix A.1, we prove that 1-player planar motion planning with the ***3-port self-closing door*** is PSPACE-complete. A more general result on self-closing doors can be found in [2], but we include this more succinct proof for completeness and conciseness.

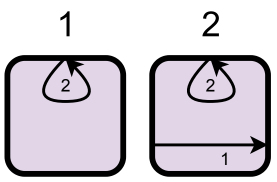


Fig. 11: State space of the 3-port self-closing door, used in the PULL!-kFG reduction.

3.1.1 Nondeterministic Locking 2-toggle

In this section, we prove that 1-player motion planning with the nondeterministic locking 2-toggle is PSPACE-complete. We also show that 1-player motion planning with the locking 2-toggle remains PSPACE when the gadgets are restricted to start in leaf states.

We use the construction shown in Fig. 12 to show simultaneously that locking 2-toggles starting in leaf states can simulate a locking 2-toggle starting in a nonleaf state, and nondeterministic locking 2-toggles can simulate a locking 2-toggle. This construction consists of two nondeterministic locking 2-toggles and a 1-toggle. A ***1-toggle*** is a two-state, two-location, reversible, deterministic gadget where each state admits a single (opposite) transition between the locations and these transitions flip the state. It can be trivially simulated by taking a single tunnel of a locking 2-toggle or nondeterministic locking 2-toggle.

Theorem 3.1. *1-player planar motion planning with the nondeterministic locking 2-toggle is* PSPACE*-complete.*

Proof. In the construction shown in Fig. 12, the agent can enter through either of the top lines; suppose they enter on the left. Other than backtracking, the agent's only path is across the bottom 1-toggle, then up the leftmost tunnel, having chosen the state of the nondeterministic locking 2-toggle which makes that tunnel traversable.

Now the only place the agent can usefully enter the construction is the leftmost line. The agent can only go down the leftmost tunnel, up the 1-toggle, and out the top right entrance, again making the appropriate nondeterministic choice when traversing the left gadget.

Symmetrically, if (from the unlocked state) the agent enters the top right, they must exit the bottom right, and the next traversal must go from the bottom right to the top right and return the construction to the unlocked state. Thus this construction simulates a locking 2-toggle. □

If we instead build the above construction with locking 2-toggles in leaf states, then all three of the locking 2-toggles used are in leaf states (the 1-toggle is one tunnel of a locking 2-toggle). A very similar argument as the nondeterministic locking 2-toggle construction shows this gadget also simulates a locking 2-toggle. Thus, given a 1-player motion planning problem with locking 2-toggles, we can replace all of the locking 2-toggles in nonleaf states with this gadget to obtain an instance where all starting gadgets are in leaf states.

Corollary 3.2. *1-player motion planning with the locking*

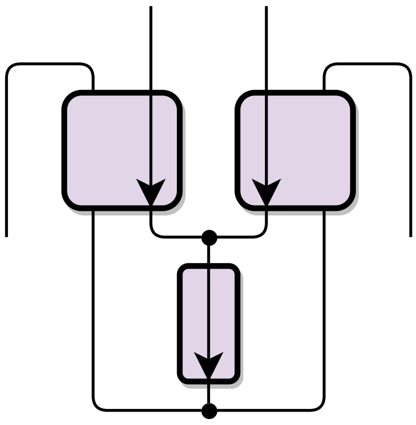


Fig. 12: Constructing a locking 2-toggle from a nondeterministic locking 2-toggle. It is currently in the unlocked state. The nondeterministic locking 2-toggles are in leaf states (top states in Fig. 10).

2-toggle where all of the locking 2-toggles start in leaf states is PSPACE*-complete.*

3.2 Pull?-*k*FG

In this section, we show that several versions of pulling-block problems with optional pulling and gravity are PSPACE-complete by a reduction from 1-player motion planning with nondeterministic locking 2-toggles, shown PSPACE-hard in Section 3.1.1.

We begin with a construction of a 1-toggle, and then use those and an intermediate construction to build a nondeterministic 2 toggle.

1-toggle. A ***1-toggle*** is a gadget with a single tunnel, traversable in one direction. When the agent traverses the tunnel of a 1-toggle, the direction that the tunnel can be traversed is flipped to the other direction, meaning that the agent must backtrack and return the way it came in order to be able to traverse it the first way again.

Our 1-toggle construction in Pull?-kFG for $k \geq 2$ is shown in Fig. 13. In the state shown, it can only be traversed from left to right by pulling both blocks to the left. This traversal flips the direction that the gadget can be traversed—it can now only be traversed from right to left.

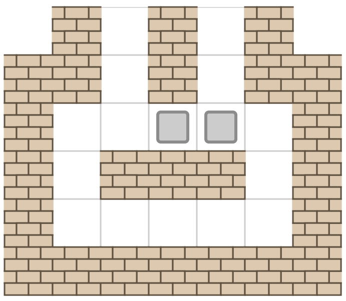


Fig. 13: 1-toggle in Pull?-2FG.

Nondeterministic Locking 2-toggle. Our construction of a nondeterministic locking 2-toggle, shown in Fig. 14, uses two 1-toggles plus a connecting section at the top.

The configuration shown in Fig. 14 is a leaf state. The

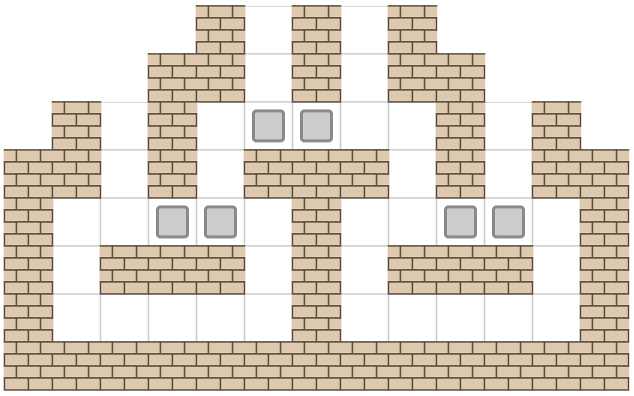


Fig. 14: Locking 2-toggle in Pull?-2FG.

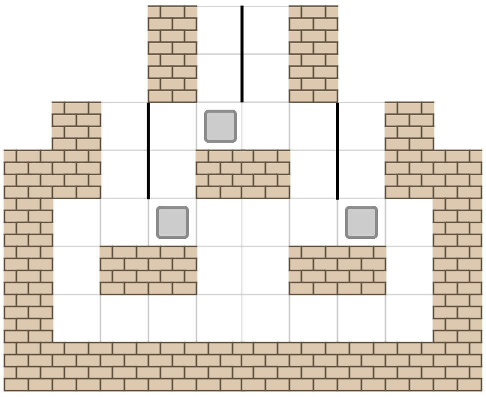


Fig. 15: Locking 2-toggle in Pull?-1WG.

right tunnel is traversable from top right to bottom right. If the agent traverses that tunnel, it can choose whether to pull the top pair of blocks to the right (because pulling is optional), corresponding to the nondeterministic choice in the nondeterministic locking 2-toggle. Both 1-toggles will be in the state where they can be traversed from bottom (outside) to top (inside). One of these paths will be blocked by the top pair of blocks and the other will be traversable, depending on whether the agent chose to pull those blocks. Traversing the traversable path then puts the gadget in a leaf state, either the one shown or its reflection.

It is possible for the agent to pull only one block instead of two, but this can only prevent future traversals, so it never benefits the agent.

Theorem 3.3. Pull?-*kFG is* PSPACE*-complete for* $k \geq 2$ *and* $k = *$.

Proof. Lemma 2.3 gives containment in PSPACE. For hardness, we reduce from 1-player planar motion planning with the nondeterministic locking 2-toggle, shown PSPACE-hard in Theorem 3.1. We embed any planar network of gadgets in a grid, and replace each nondeterministic locking 2-toggle with the construction described above in the appropriate state. The resulting pulling-block problem is solvable if and only if the motion planning problem is.

This reduction works for Pull?-kFG for any $k \geq 2$ including $k = *$, because the player only ever has the opportunity to pull 2 blocks at a time. This proof requires optional pulling because the player must choose whether to pull blocks while traversing a nondeterministic locking 2-toggle. □

Corollary 3.4. *Pull?-kWG is* PSPACE*-complete for* $k \geq 1$ *and* $k = *$.

Proof. With thin walls, the tunnels can be separated by a thin wall instead of a fixed block, which means that only one block is required in each of the toggles. This is shown in Fig. 15. The rest of the proof follows in the same manner, demonstrating PSPACE-completeness of Pull?-kWG for $k \geq 1$. □

3.3 Pull!-kFG

In this section, we show PSPACE-completeness for pulling-block problems with forced pulling and gravity, using a reduction from 1-player planar motion planning with the 3-port self-closing door, shown PSPACE-hard in Theorem A.1.1.

Theorem 3.5. Pull!*-kFG is* PSPACE*-complete for* $k \geq 1$ *and* $k = *$.

Proof. Lemma 2.3 gives containment in PSPACE. We show PSPACE-hardness by a reduction from 1-player planar motion planning with the 3-port self-closing door. It suffices to construct a 3-port self-closing door in Pull!-kFG.

First, we construct a diode, shown in Fig. 16. The agent cannot enter from the right. If the agent enters from the left, it must pull the left block to the left to advance. If it pulls the left block left and then exits, they still cannot enter from the right, so doing so is useless. The agent then advances and is forced to pull the left block back to its original position. The agent then must pull the right block left to advance, and must actually advance because the way back is blocked. As the agent exits the gadget, it is forced to pull the right block back to its original position. Therefore, the agent can always cross the gadget from left to right and never from right to left, simulating a diode.

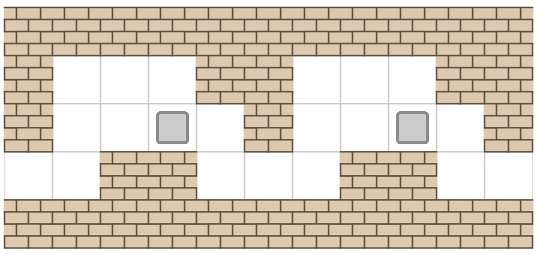
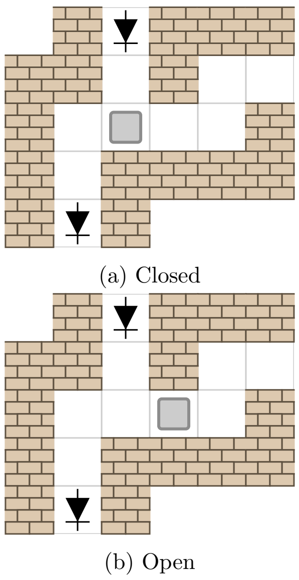


Fig. 16: A diode in Pull!-kFG.

Using this diode, we then construct a 3-port self-closing door, shown in Fig. 17; the diode icons indicate the diode shown in Fig. 16. The bottom is exit-only. In the closed state, the agent should not enter from the top because it would become trapped between a block and the wrong end of a diode. The agent can enter from the right, pull the block 1 tile right, and leave, opening the gadget. In the open state, the agent can enter from the top and exit out the bottom, and is forced to pull the block back to its original position, closing the gadget. So this construction simulates a 3-port self-closing door.


(a) Closed

(b) Open

Fig. 17: A 3-port self-closing door in Pull!-kFG.

Because the player never has the opportunity to pull multiple blocks, this reduction works for all $k \geq 1$ including $k = *$. □

4. Pull?-1FG is NP-hard

In this section, we show NP-hardness for Pull?-1FG by reducing from 1-player planar motion planning with the crossing NAND gadget from [2]. A ***crossing NAND gadget*** is a three-state gadget with two crossing tunnels, where traversing either tunnel permanently closes the other tunnel. 1-player planar motion planning with the crossing NAND gadget is shown NP-hard in [2, Lemma 4.9] based on the constructions in [5, 12] which originally reduce from Planar 3-Coloring.

Theorem 4.1. Pull?*-1FG is* NP*-hard.*

Proof. We reduce from 1-player planar motion planning with the crossing NAND gadget [2, Lemma 4.9]. First we first construct a "single-use" one-way gadget, shown in Fig. 18. This gadget can initially can be crossed in one way, but then becomes impassable in both directions.

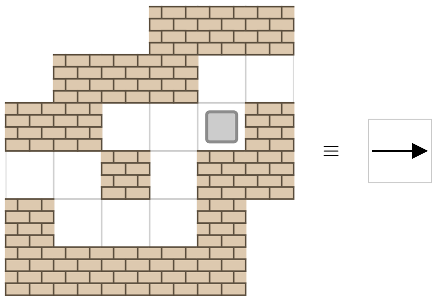


Fig. 18: Single-use one-way gadget in Pull?-1FG that initially allows traversal from left-to-right and then prevents traversal in both directions.

Fig. 19 shows our construction of the crossing NAND gadget. Single-use one-way gadgets enforce that the agent must enter through one of the top paths. The agent must pull two blocks to enter the gadget; these blocks end up stacked in the vertical tunnel on top of the block below. The agent cannot exit via the bottom tunnel underneath its entry tunnel: the agent can pull one block into the slot on the bottom, and then can pull one block one square, but that still leaves the third block of the stack blocking off the exit path. The agent cannot exit via the other top path, because it is blocked by the single-use one-way gadget. The only path remaining is for the agent to cross diagonally by pulling the single block in the lower layer into the slot, revealing a path to the exit opposite where the agent entered. After leaving, both the entry tunnel and exit tunnel are impassable because the single-use one-way gadgets have become impassable. If the agent later enters via the other entry tunnel, the agent will be trapped, because it will not be able to leave via the tunnel that was "collapsed" in the initial entry. □

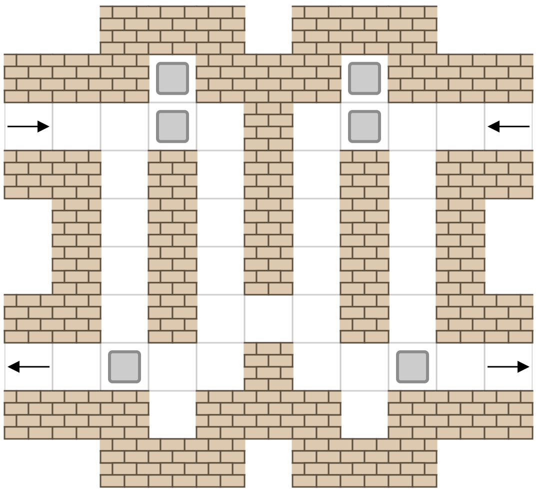


Fig. 19: Crossing NAND gadget in Pull?-1FG allowing traversal either from the top-left to the bottom-right, or from the top-right to the bottom-left. After being traversed once, the entire gadget becomes impassable in any direction.

We leave open the question of whether Pull?-1FG is in NP or PSPACE-hard.

5. Open Problems

There are several open problems remaining related to the pulling-block problems considered in this paper.

(1) What is the complexity of Pull?-1FG (the last remaining problem in Table 1)? We leave a gap between NP-hardness and containment in PSPACE.

(2) What is the complexity of pulling-block puzzles without fixed blocks (say, on a rectangular board)? With block pushing, one can generally construct effectively fixed blocks by putting enough blocks together. This technique no longer works in the block-pulling context.

(3) Do all of these variants remain PSPACE-hard when we ask about storage (can the player place blocks covering some set of squares?) or reconfiguration (where blocks are distinguishable and must reach a desired configuration) instead of reachability? The storage question for Pull?-kFG for $k \geq 1$ and Pull?-$*$FG has been proved PSPACE-hard [14].

(4) What about the studied variants applied to Push-Pull (where blocks can be pushed and pulled) and PullPull (where blocks must be pulled maximally until the robot backs against another block)? Standard versions are proved PSPACE-complete in [7, 14], but variations with mandatory pulling, gravity, and/or no fixed blocks all remain open.

Acknowledgments This work was initiated during open problem solving in the MIT class on Algorithmic Lower Bounds: Fun with Hardness Proofs (6.892) in Spring 2019. We thank the other participants of that class for related discussions and providing an inspiring atmosphere. We also thank the anonymous referees for helpful comments.

Appendix

A.1 3-port Self-Closing Door

Ani et al. [2] proved PSPACE-completeness of 1-player planar motion planning with many types of self-closing door gadgets and all of their planar variations. For completeness, we give a proof specific to the 3-port self-closing door gadget in this section. Our proof is more succinct because it does not consider other variants of the gadget. The reduction is from 1-player motion planning with the door gadget from [1].

Theorem A.1.1. *1-player planar motion planning with the 3-port self-closing door is* PSPACE*-hard.*

Proof. We will show that the 3-port self-closing door planarly simulates a crossover, which lets us ignore planarity. We will then show that the 3-port self-closing door simulates the door gadget. Because 1-player motion planning with the door gadget is PSPACE-hard [1], so is 1-player motion planning with the 3-port self-closing door, and because it simulates a crossover, so is 1-player planar motion planning with the 3-port self-closing door. Along the way, we will construct a self-closing door with multiple door and opening ports as well as a diode.

Diode. We can simulate a diode (one-way tunnel which is always traversable) by connecting the opening port to the input of the self-closing tunnel. The agent can always go to the opening port and then through the self-closing tunnel, but can never go the other way because the self-closing tunnel is directed.

Port Duplicator. The construction shown in Fig. A·2 simulates a self-closing door with two equivalent opening ports. If the agent enters from the top, it can open only one of the upper gadgets, then open the lower gadget, and then must exit the same way it came. Note, this same idea can be used to construct more than two ports, which will be needed later.

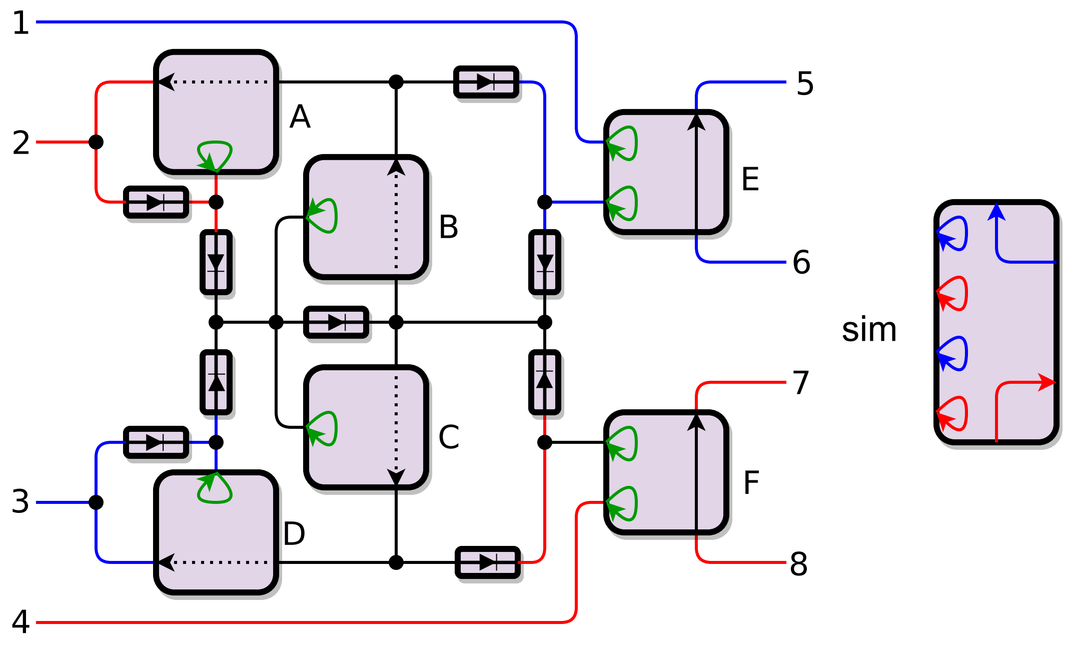


Fig. A·1: 3-port self-closing door simulating the gadget on the right, where each port opens the door of the same color (the top and third-from-top open the top door, and the others open the bottom door).

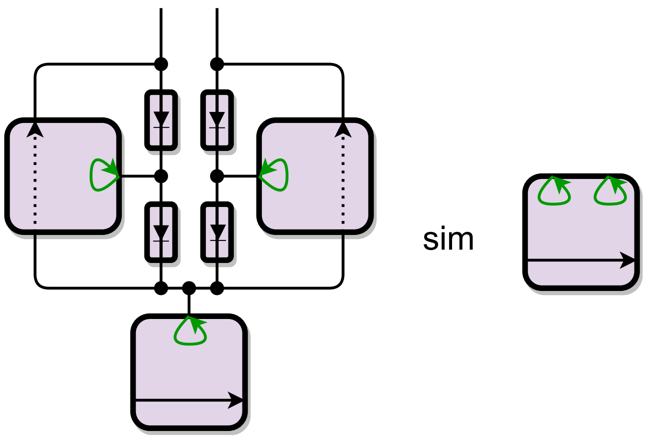


Fig. A·2: 3-port self-closing door simulating a version of it that has 2 opening ports. Opening ports are shown in green. A dotted self-closing tunnel is closed, and a solid self-closing tunnel is open.

We use these to simulate an intermediate gadget composed of two of self-closing doors each connected to two opening ports in a particular order arrangement, shown in Fig. A·1. If the agent enters from port 1 or 4, it will open door E or F, respectively, and then leave. If the agent enters from port 2, it can open doors A, B, and C. If it then traverses door B and opens door E, it will get stuck because both B and D are closed. So the agent cannot open door E and exit. Instead, it can traverse doors B and A, ending up back at port 2 with no change except that door C is open. Entering port 2 or 3 always gives the agent an opportunity to open door C, so leaving door C open does not help. So the only useful path after entering port 2 is to traverse door C. The agent is then forced to go right and can open door F. Then it is forced to traverse door B. Again if the agent opens door E, it will be stuck, so the agent traverses door A instead and returns to port 2, leaving door F open. Similarly, if the agent enters from port 3, the only useful thing it can do is open door E and return to port 3.

Crossover. This intermediate gadget can simulate a directed crossover, shown in Fig. A·3. If the agent enters at the top left, it can open the left door on the top gadget, open both doors on the bottom gadget, and then exit the bottom right while closing all three opened doors. If the agent opens both doors on the top gadget it will get stuck. Similarly if the agent enters the bottom left, all it can do is exit the top right. The directed crossover can simulate an undirected crossover, as in Fig. A·4 and shown in [6].

Door Duplicator. Now, we use this crossover to simulate a gadget with two self-closing doors controlled by the same opening port, as shown in Fig. A·5. This gadget has two states, open and closed. Both doors are either open or closed and going through either door closes both of them. The construction is similar to the construction for the port duplicator, but goes through a tunnel instead.

Door Gadget. Finally, we triplicate the opening port by adding a third entrance to the construction in Fig. A·2 similar to the other two, and use these ports to simulate a door gadget as shown in Fig. A·6. Recall the whole three-port two-door gadget has only two states, open and closed. The agent can open both doors from any of the open ports and going across either self-closing door will close both doors. If the agent enters from port O, it can open the doors and leave. If the agent enters from port T_0 and the gadget is open, the agent can traverse the door and then reopen it using the third port. The agent then leaves at port T_1. If the agent enters from port C_0, it can open the gadget and then must traverse the bottom tunnel and leave at port C_1,

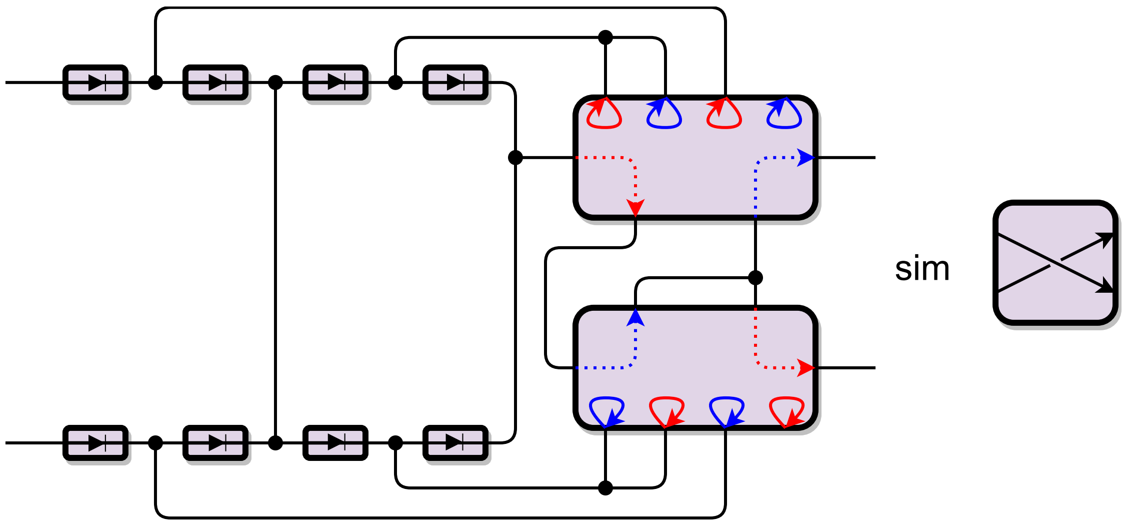


Fig. A·3: 3-port self-closing door simulating a crossover.

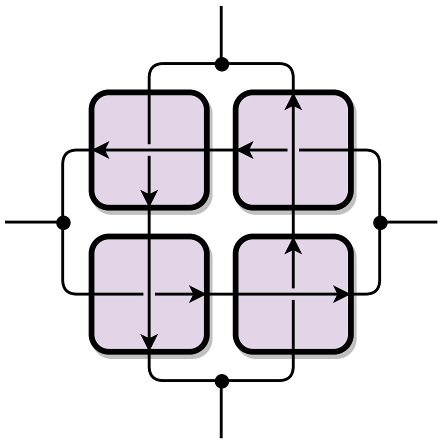


Fig. A·4: Directed crossover simulating an undirected crossover.

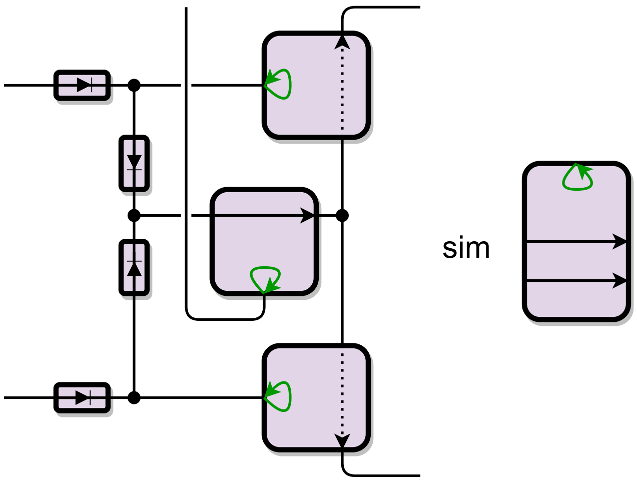


Fig. A·5: 3-port self-closing door simulating a gadget with 2 self-closing tunnels.

closing the gadget. □

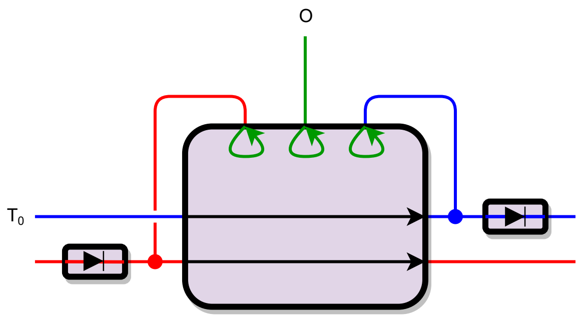


Fig. A·6: Simulation of the door gadget in [1] using a gadget with 3 opening ports and 2 self-closing tunnels.

References

[1] Aloupis, G., Demaine, E. D., Guo, A. and Viglietta, G.: Classic Nintendo Games are (Computationally) Hard, *Theoretical Computer Science*, Vol. 586, pp. 135–160 (2015).

[2] Ani, J., Bosboom, J., Demaine, E. D., Diomidov, Y., Hendrickson, D. and Lynch, J.: Walking through Doors is Hard, even without Staircases: Proving PSPACE-hardness via Planar Assemblies of Door Gadgets, *Proceedings of the 10th International Conference on Fun with Algorithms (FUN 2020)*, Favignana, Italy (2020).

[3] Canny, J.: Some algebraic and geometric computations in PSPACE, *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, Chicago, Illinois, pp. 460–469 (online), available from ⟨https://doi.org/10.1145/62212.62257⟩ (1988).

[4] Culberson, J.: Sokoban is PSPACE-complete, *Proceedings of the International Conference on Fun with Algorithms*, Elba, Italy, pp. 65–76 (1998).

[5] Demaine, E. D., Demaine, M. L., Hoffmann, M. and O'Rourke, J.: Pushing Blocks is Hard, *Computational Geometry: Theory and Applications*, Vol. 26, No. 1, pp. 21–36 (2003).

[6] Demaine, E. D., Demaine, M. L. and O'Rourke, J.: Push-Push and Push-1 are NP-hard in 2D, *Proceedings of the 12th Annual Canadian Conference on Computational Geometry (CCCG 2000)*, Fredericton, New Brunswick, Canada, pp. 211–219 (2000).

[7] Demaine, E. D., Grosof, I. and Lynch, J.: Push-Pull Block Puzzles are Hard, *Proceedings of the 10th International Conference on Algorithms and Complexity*, Lecture Notes in Computer Science, Vol. 10236, Athens, Greece, pp. 177–195 (2017).

[8] Demaine, E. D., Grosof, I., Lynch, J. and Rudoy, M.: Computational Complexity of Motion Planning of a Robot through Simple Gadgets, *Proceedings of the 9th International Conference on Fun with Algorithms (FUN 2018)*, La

Maddalena, Italy, pp. 18:1–18:21 (2018).

[9] Demaine, E. D., Hearn, R. A. and Hoffmann, M.: Push-2-F is PSPACE-Complete, *Proceedings of the 14th Canadian Conference on Computational Geometry*, Lethbridge, Canada, pp. 31–35 (2002).

[10] Demaine, E. D., Hendrickson, D. and Lynch, J.: Toward a General Theory of Motion Planning Complexity: Characterizing Which Gadgets Make Games Hard, *Proceedings of the 11th Conference on Innovations in Theoretical Computer Science*, Seattle, Washington (2020). arXiv:1812.03592.

[11] Demaine, E. D., Hoffmann, M. and Holzer, M.: PushPush-*k* is PSPACE-Complete, *Proceedings of the 3rd International Conference on Fun with Algorithms (FUN 2004)*, Isola d'Elba, Italy, pp. 159–170 (2004).

[12] Friedman, E.: Pushing blocks in gravity is NP-hard, Unpublished manuscript (2002). https://www2.stetson.edu/~efriedma/papers/gravity.pdf.

[13] Hearn, R. A. and Demaine, E. D.: *Games, Puzzles, and Computation*, A K Peters/CRC Press (2009).

[14] Pereira, A. G., Ritt, M. and Buriol, L. S.: Pull and PushPull are PSPACE-complete, *Theoretical Computer Science*, Vol. 628, pp. 50–61 (2016).

[15] Reif, J. H.: Complexity of the mover's problem and generalizations, *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, San Juan, Puerto Rico, pp. 421–427 (online), DOI: 10.1109/SFCS.1979.10 (1979).

[16] Ritt, M.: Motion planning with pull moves, arXiv:1008.2952 (2010). https://arXiv.org/abs/1008.2952.

[17] Savitch, W. J.: Relationships between nondeterministic and deterministic tape complexities, *Journal of Computer and System Sciences*, Vol. 4, No. 2, pp. 177–192 (1970).

[18] Viglietta, G.: Partial Searchlight Scheduling is Strongly PSPACE-complete, *Proceedings of the 25th Canadian Conference on Computational Geometry*, Waterloo, Canada, (online), available from ⟨http://cccg.ca/proceedings/2013/papers/paper_2.pdf⟩ (2013).

[19] Wikipedia: Sokoban, https://en.wikipedia.org/wiki/Sokoban.

[20] Wilfong, G.: Motion planning in the presence of movable obstacles, *Annals of Mathematics and Artificial Intelligence*, Vol. 3, No. 1, pp. 131–150 (1991). Originally appeared at SoCG 1988.

Joshua Ani is an undergraduate student at MIT studying computer science, expecting to graduate in 2021. His research interests are computer graphics and computational complexity of motion planning.

Sualeh Asif is an undergraduate student at MIT expecting to graduate in 2022. His research interests include computational complexity, efficient computing and topics at the intersection of computing and number theory.

Erik D. Demaine received a B.Sc. degree from Dalhousie University in 1995, and M.Math. and Ph.D. degrees from the University of Waterloo in 1996 and 2001, respectively. Since 2001, he has been a professor in computer science at the Massachusetts Institute of Technology. His research interests range throughout algorithms, from data structures for improving web searches to the geometry of understanding how proteins fold to the computational difficulty of playing games. In 2003, he received a MacArthur Fellowship as a "computational geometer tackling and solving difficult problems related to folding and bending— moving readily between the theoretical and the playful, with a keen eye to revealing the former in the latter". He cowrote a book about the theory of folding, together with Joseph O'Rourke (*Geometric Folding Algorithms*, 2007), and a book about the computational complexity of games, together with Robert Hearn (*Games, Puzzles, and Computation*, 2009). With his father Martin, his interests span the connections between mathematics and art.

Yevhenii Diomidov is a graduate student in computer science at MIT studying computational geometry and origami, and the computational complexity of games and puzzles under Erik Demaine. Yevhenii received a B.Sc. degree in mathematics and physics from MIT in 2019.

Dylan Hendrickson is a graduate student in computer science at MIT studying the computational complexity of motion planning problems under Erik Demaine. Dylan received a B.Sc. degree in mathematics and physics from MIT in 2019.

Jayson Lynch received a Ph.D. from MIT 2020 for work on the computational complexity of motion planning problems under Erik Demaine. Jayson is now a Postdoctoral Researcher at the University of Waterloo continuing to do work on computational geometry and origami, graph algorithms, resource efficient computing, and the computational complexity of games and puzzles.

Sarah Scheffler is a graduate student studying cryptography at Boston University working with Prof. Mayank Varia. Her research interests include zero-knowledge proofs, secure messaging, and topics at the intersection of computer science and law.

Adam Suhl received a B.S. in Mathematics from MIT in 2016 and is currently a Ph.D. student at UC San Diego. His interests include cryptography, complexity, and theoretical computer science.