Agent Motion Planning as Block Asynchronous Cellular Automata: Pushing, Pulling, Suplexing, and More

MIT Hardness Group* Hayashi Ani† Josh Brunner† Erik D. Demaine†
Jenny Diomidova† Timothy Gomez† Della Hendrickson† Yael Kirkpatrick†
Jeffery Li† Jayson Lynch† Ritam Nag† Frederick Stock‡

Abstract

In this paper, we explore how agent reachability problems from motion planning, games, and puzzles can be generalized and analyzed from the perspective of block asynchronous cellular automata, inspired by asynchronous cellular automata, surface chemical reaction networks, and similar rewriting dynamical systems. Specifically, we analyze square grids with three or four cell types (states) that model motion planning with blocks: an agent which occurs uniquely, empty space, blocks, and (optionally) fixed walls. The agent can freely exchange with (walk through) empty space, and can interact with blocks according to a single local asynchronous 3-cell replacement rule; the goal is for the agent to reach a specified destination (reachability). This setting generalizes well-studied motion-planning problems such as Push-1F and Pull-1F, which are known to be PSPACE-complete. We analyze all 40 possible 3-cell replacement rules (22 that conserve blocks so are naturally in PSPACE, and 18 that create or destroy blocks so are naturally in NP), and except for a few open problems, characterize their complexity — ranging from L to NL to as hard as Planar Monotone Circuit Value Problem to P-complete to NP-complete to PSPACE-complete.

1 Introduction

Cellular automata have been studied extensively in computer science, along with many variations. A traditional ***cellular automaton*** [vN66, Gut91, Sut95, Kar05] is defined by a set of possible ***states*** for cells of a grid, and a local ***replacement rule*** for how the state of every cell evolves over time as a function of the states of its local neighborhood (including the cell itself). The most famous examples are Conway's Game of Life (2D) [BCG04] and Rule 110 (1D) [Coo04, NW06], which are each Turing-complete in certain senses.

In a traditional cellular automaton, all cells update in lock step, synchronized by a global clock. In ***asynchronous*** cellular automata [Fat13, DFM+17, GOT15, GMMR21, OT22, GDC+10], any cell can update at any time. Updating two neighboring cells in different orders can produce different results, as each cell's update can depend on the other cell's state. Thus the evolution of an asynchronous cellular automaton is nondeterministic. For example, asynchronous cellular automata capture the abstract Tile Assembly Model (aTAM) [Win98, ACG+02], where a cell can transition from empty to having a particular tile when certain other tiles are neighbors.

*Artificial first author to highlight that the other authors (in alphabetical order) worked as an equal group. Please include all authors (including this one) in your bibliography, and refer to the authors as "MIT Hardness Group" (without "et al.").

†MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, {joshuaa,brunnerj, edemaine,diomidova,tagomez7,della,yaelkirk,jeli,jaysonl,rnag}@mit.edu

‡University of Massachusetts, Lowell, MA 01854, USA, Frederick_Stock@student.uml.edu

Block cellular automata [Mar84, TM87, DL01] use a different kind of update rule: they define a replacement rule on a constant-size ***block*** of cells (e.g., two consecutive cells in 1D, or a 2×2 square of four cells in 2D) by specifying the new state of the entire block as a function of its old state. In a synchronous cellular automaton, block updates require partitioning the cells into blocks, updating each block in lock step, and then repeating with a shifted version of the partition so that all neighboring cells eventually interact.

Block asynchronous cellular automata. In this paper, we explore *block asynchronous* cellular automata, apparently for the first time. As in block cellular automata, the replacement rule specifies the new state of an entire constant-size block of cells as a function or relation of their old states. But unlike synchronous automata, we do not need a partition of cells into blocks. Instead, as in asynchronous cellular automata, the replacement rule can be applied nondeterministically to any one block at any time.

Block asynchronous cellular automata directly model many existing dynamical systems, with applications to DNA computing, modular robotics, and puzzles and games:

1. Surface Chemical Reaction Networks (sCRNs) [QW14, ABC$^+$23] are 2D block asynchronous cellular automata with rotatable 1×2 blocks. In other words, an sCRN specifies rules of the form $AB \to A'B'$, where A, B, A', B' are possible states, which can be applied to any two (horizontally or vertically) neighboring cells. There may be multiple rules of the form $AB \to \cdots$, so they form a relation instead of a function, adding to the nondeterminism of the system.

2. Friends-and-strangers graphs [DK21, Mil24] are graphical block asynchronous cellular automata with blocks defined by a graph edge. More precisely, for every two friends A, B, we have a ***swap rule*** $AB \to BA$ that can be applied to any two adjacent vertices of the graph.

3. The Fifteen Puzzle and related $n^2 - 1$ puzzles [RW90, DR18] are 2D block asynchronous cellular automata with rotatable 1×2 blocks. Here we have a swap rule $Ae \to eA$ for every state A, where e is one (uniquely occurring) state representing the empty space.

4. Modular pivoting robots [ADG$^+$21] are 2D or 3D block asynchronous cellular automata with rotatable constant-size blocks of side length up to 3. Specifically, the rules allow a robot to move from one position to another (similar to $Ae \to eA$ swap rules), but use a larger block to guarantee that certain neighboring cells are empty to avoid collisions during the motion.

5. The abstract Tile Assembly Model (aTAM) [Win98, ACG$^+$02] is a 2D block asynchronous cellular automaton with non-rotatable constant-size blocks involving a cell and its four cardinal neighbors. The replacement rule specifies how a tile can attach to a growing assembly, modifying only one cell but depending on the neighbors.

6. The puzzle game Lights Out! [FY13] is a 2D block asynchronous cellular automaton, where a block is a cell and its four cardinal neighbors, and the replacement rule flips the state of all five cells. Similarly, the generalization to graphs [FY13] is a graphical block asynchronous cellular automaton.

Our definition of block asynchronous cellular automata is inspired by these various applications, and seems natural to study more broadly. Many applications (e.g., 1–3 above) involve just two-cell blocks.

Our results. In this paper, we study the natural extension of block asynchronous cellular automata with *three-cell blocks*. Specifically, we are motivated by two well-studied agent-based motion-planning games involving reconfiguration of movable blocks, Push-1F and Pull?-1F, which are known to be PSPACE-complete [ACD$^+$22, AAD$^+$20]. In both games, an agent (the player) can traverse empty cells of a 2D grid of square cells, where each cell can be empty, contain a movable block, or be a fixed wall, and the goal is

for the agent to reach a particular cell. In Push-1F (which has the same dynamics as the famous Sokoban puzzle game), the agent can push a neighboring block, provided that the cell on the other side of the block is empty. In Pull?-1F, the agent can (but does not have to) pull a neighboring block, provided the cell on the other side of the agent is empty. We can model these games as block asynchronous cellular automata with rotatable 1×2 and 1×3 blocks and the following replacement rules:

Push-1F	**Pull?-1F**	
$Ae \to eA$	$Ae \to eA$	(agent A can move through empty space e)
$ABe \to eAB$	$eAB \to ABe$	(agent A can push/pull a block B into empty space e)

The rules do not involve fixed walls F, which captures the desired property that the agent and blocks cannot move into such cells.

The main goal of this paper is to characterize the complexity of ***reachability*** (whether the agent can reach a particular cell) in all block asynchronous cellular automata with a single three-cell block rule involving the same four states: agent A, empty e, movable block B, and fixed wall F. Notably, we require that the agent state A appears uniquely at all times in the configuration, as in Push-1F and Pull?-1F and other agent-based motion-planning problems, and that the fixed walls F are ***frozen*** (never change state). Push-1 was recently shown to be hard even without fixed walls F [MIT24], so we also consider the complexity with just three states $\{A, e, B\}$, forbidding fixed walls F. Finally, we assume that two-cell movement rule $Ae \to eA$ is always present, and characterize the behavior of an arbitrary single three-cell "game" rule.

We provide an almost complete complexity-theoretic landscape for the (single-agent) reachability problem. There are 40 possible single replacement rules for a three-cell block (subject to the constraints above), and two versions of each (fixed blocks allowed or forbidden). We decompose the rules into 22 "conservative" game rules that conserve the number of blocks, and 18 "bounded" game rules that create or destroy a block. Bounded rules naturally lead to polynomially bounded games, as the game rule can be applied a number of times at most linear in the playing area. Tables 1 and 2 summarize our results for conservative and bounded games respectively, which solve all but five of the 80 variations (two of which are equivalent). We also attempt to give each rule a name to provide intuition behind what the agent is doing, like Push-1 modeling an agent "pushing" a block.

More specifically, we show the following:

1. We start in Section 3 by analyzing what we call ***generalized swaps***. These game rules are very weak and in some sense reduce to a swap, so they can be solved in logarithmic space (L or NL), and some are complete for their respective classes.

2. Next, in Section 4, we analyze ***conservative rules*** where the total number of blocks does not change. This includes famous examples like Push-1 and new games such as Suplex-1 ($eAB \to BAe$), where the agent throws the block over its head into the cell behind it.[1] As shown in Table 1, we give a complete characterization for when fixed walls F are allowed, and leave only four open cases without fixed walls. Many cases are PSPACE-complete, while some are surprisingly in P.

3. Next, in Section 5, we cover ***bounded rules*** where the number of blocks monotonically increases or monotonically decreases. For these rules, we prove that each game is either NP-complete or in P. A few games are P-complete as well.

4. Finally, in Section 6, we provide initial results for ***bendy rules***, where the rule's block can be any path of three adjacent cells, not necessarily a 1×3 rectangle. This rule type is a natural extension of block asynchronous cellular automata (especially in the graphical view), and is motivated by implementations where rigidity is hard to enforce, such as DNA, as proposed in [QW14].

Rules	Game	Reachability-F	Reachability
$ABe \to eBA$	Leap	L-complete (Thm. 5)	L-complete (Thm. 5)
$eAB \to AeB$	Trivial	L (Cor. 4)	L (Cor. 4)
$AeB \to eAB$	Trivial	L (Cor. 4)	L (Cor. 4)
$AFe \to eFA$	Vault	L-complete (Thm. 5)	L (Cor. 4)
$AFB \to BFA$	Vault Swap	NL-complete (Thm. 5)	NL (Cor. 4)
$ABF \to BAF$	Push Swap at Fixed	NL (Cor. 4)	NL (Cor. 4)
$BAF \to ABF$	Pull Swap at Fixed	NL (Cor. 4)	NL (Cor. 4)
$ABe \to BAe$	Push Swap at Empty	NP-complete (Thm. 14)	NP-complete (Thm. 14)
$ABe \to BeA$	= Push Swap at Empty	NP-complete (Thm. 14)	NP-complete (Thm. 14)
$ABe \to eAB$	Push-1	PSPACE-complete [ACD+22]	PSPACE-complete [MIT24]
$ABe \to AeB$	= Push-1	PSPACE-complete [ACD+22]	PSPACE-complete [MIT24]
$eAB \to eBA$	Pull Swap at Empty	PSPACE-complete (Thm. 15)	PSPACE-complete (Thm. 15)
$AeB \to eBA$	= Pull Swap at Empty	PSPACE-complete (Thm. 15)	PSPACE-complete (Thm. 15)
$eAB \to ABe$	Pull?-1 [AAD+20]	PSPACE-complete [AAD+20]	*OPEN*
$AeB \to ABe$	= Pull?-1	PSPACE-complete [AAD+20]	*OPEN*
$eAB \to BAe$	Suplex-1	PSPACE-complete (Thm. 15)	P (Thm. 23)
$eAB \to BeA$	= Suplex-1	PSPACE-complete (Thm. 15)	P (Thm. 23)
$AeB \to BAe$	= Suplex-1	PSPACE-complete (Thm. 15)	P (Thm. 23)
$AeB \to BeA$	= Suplex-1	PSPACE-complete (Thm. 15)	P (Thm. 23)
$ABB \to BBA$	Swap-2	PSPACE-complete (Thm. 15)	*OPEN*
$ABB \to BAB$	Push Swap at Block	PSPACE-complete (Thm. 15)	*OPEN*
$BAB \to ABB$	Pull Swap at Block	PSPACE-complete (Thm. 15)	*OPEN*

Table 1: Our results for conservative game rules. "=" indicates that the rule is equivalent to the specified rule when combined with the movement rule.

The connection between cellular automata and agent-based motion planning can be seen in Langdon's Ant [Lan86] which was later considered from a computational complexity perspective [TH11, DHHL22]. Motion planning is also becoming of increased interest due to experimental constructions such as robots that walk along DNA origami lattices which perform tasks [TLJ+17].

2 Definitions

2.1 Block Asynchronous Cellular Automata

Definition 1 (States)**.** In this paper, we will use a size-4 ***state alphabet*** with the following state symbols and names:

A — the agent
B — a movable block
e — an empty space
F — a fixed wall (allowed only in some models)

Definition 2 (Board)**.** A ***board*** is an $x \times y$ grid graph where each vertex of the grid is assigned a single state.

[1] This name comes from the wrestling move where an opponent is thrown backward over ones head.

Rules	Game	Reach-F	Reach
$ABB \to Aee$	Push Smash	$\longrightarrow$	NP-c (Thm 26)
$ABB \to eBA$	Leap Crush	$\longrightarrow$	NP-c (Thm 29)
$ABB \to BAe$	Swap Zap	$\longrightarrow$	NP-c (Thm 29)
$ABB \to AeB$	Push Merge	$\longrightarrow$	NP-c (Thm 25)
$ABB \to ABe$	Pull Merge	$\longrightarrow$	NP-c (Thm 28)
$BAB \to eAe$	Suplex Smash	$\longrightarrow$	NP-c (Thm 25)
$BAB \to BAe$	Suplex Merge	$\longrightarrow$	NP-c (Thm 25)
$BAB \to eBA$	Clap Merge	$\longrightarrow$	NP-c (Thm 30)
$eAB \to BBA$	Trail Swap	$\longrightarrow$	NP-c (Thm ??)
$ABe \to BAB$	Trail Push	$\longrightarrow$	NP-c (Thm 30)
$eAB \to eAe$	Battering Ram	P-c (Thm. 32)	NL-c (Thm. 35)
$ABe \to Aee$	Knock Over	in NL (Thm. 38)	in NL (Thm. 38)
$ABe \to BBA$	Trail Leap	$\longrightarrow$	L-c (Thm. 5)
$AFB \to AFe$	Zap through Walls	in P (Thm. 32), PMCVP-hard (Cor. 33)	N/A
$AFB \to eFA$	Vault Crush	NL-c (Thm. 5)	N/A
$ABF \to AeF$	Push into Wall	in NL	N/A
$FAB \to FAe$	Wall Suplex	in NL	N/A
$AFe \to BFA$	Trail Vault	L-c (Thm. 5)	N/A

Table 2: Our results for bounded game rules. "$\longrightarrow$" in the F column indicates that the complexity is the same as the right column, as hardness without fixed walls is a stronger result. "N/A" in the right column indicate that the problem is trivial as the rule cannot be applied without fixed walls.

Definition 3 (Subconfiguration)**.** A ***subconfiguration*** of size k is a sequence of k distinct vertices on a board. A ***connected subconfiguration*** is a sequence of k distinct vertices where each vertex is adjacent to the next vertex in the sequence. A ***linear subconfiguration*** is a connected subconfiguration where all vertices lie either in the same row or same column on the board.

We reconfigure a subconfiguration on a board using a "rule":

Definition 4 (Rule)**.** A ***rule*** of size k is an ordered pair from $\{A, B, e, f\}^k \times \{A, B, e, f\}^k$. A rule is applied to a board by replacing a length k linear subconfiguration that matches the left side of the rule with the states of the right side of the rule. Rules can be applied regardless of reflection and rotation.

Definition 5 (Rule System)**.** A ***rule system*** is a binary relation over $\bigcup_k \{A, B, e, f\}^k$ specifying rules that can be applied.

Definition 6 (Agent Based Rule System)**.** A rule system is ***agent-based*** if every rule in the rule system contains exactly one A on both the left and right hand sides of the rule.

When $k = 2$, this model is equivalent to a 4 species Surface Chemical Reaction Network [ABC+23]. In the real world, surface chemical reaction networks have a difficult time distinguishing between 3 states in a straight line versus 3 states in an L. This then gives rise to a second class of rule, a "bendy" rule.

Definition 7 (Bendy Rule)**.** A ***bendy rule*** of size k is an ordered pair $\{A, B, e, f\}^k \times \{A, B, e, f\}^k$ where a size-k (connected) subconfiguration is replaced by a new subconfiguration of the same size and shape. **[It would be good to draw up a figure, and wording needs work - wording reworked, but figures for linear and bendy subconfigurations would still be helpful]** xxx

Note that bendy rules do not require linear subconfigurations; the states can instead "bend" making an L shape. When $k = 2$, bendy rules are identical to normal rules. In this paper, a rule will not be bendy unless otherwise specified.

We study the ***Single Agent Reachability Problem***. This is characterized by tracking a special agent state which the rules and starting configuration ensure only one can exist at a time. If a node becomes the state A we say it is occupied by the agent.

Definition 8 (Single Agent Reachability Problem)**.** Given a board B with a single agent state A, an agent-based rules system R, and a target location t, does there exist a sequence of rules applied to subconfigurations which causes the node at t to be in state A?

This problem is commonly studied in the context of block pushing puzzles, and therefore we use a naming convention found in block pushing literature. For example, given the rule $ABe \to eAB$, we call Single Agent Reachability problem Push-1 if we do not allow fixed walls F, Push-1F if we allow fixed walls. and Push-1W if we allow ***thin walls***, which can be thought of as removing edges from the board (changing whether we have a full grid, an induced subgraph, or a general subgraph of the grid). We include the name of each rule in the tables. In the future, when we refer to the Single Agent Reachability problem by a rule r we really mean the Single Agent Reachability Problem using the set of rules consisting of r and the ***movement rule***, $\{r, Ae \to eA\}$. For example, in Table 1, the results for the rule $ABe \to eBA$ are in fact results for the Single Agent Reachability Problem for the set of rules $\{ABe \to eBA, Ae \to eA\}$. This is because in typical pushing block puzzles the agent is allowed to move freely unless it comes in contact with a fixed wall or movable block, where in the latter case the agent can then either push or pull the block. Hence, any ruleset that does not include the movement rule is an incomplete representation of these puzzles.

2.2 Gadgets

Almost all of our hardness results reduce from the motion-planning-through-gadgets model [DGLR18, DHL20]. This framework was introduced by a series of papers [DGLR18, DHL20], and attempts to generalize motion planning problems where an autonomous agent attempts to navigate an environment to get between two distinct points.

A ***gadget*** consists of a finite set of ***states***, a finite set of ***locations***, and a finite set of ***transitions*** of the form $(q, a) \to (r, b)$, meaning that, when the gadget is in state q, an agent can enter at location a and exit at location b while changing the gadget's state to r. Given a system of such gadgets, with locations connected together by a graph, the ***reachability*** problem asks whether the agent can start at one specified location and reach another through a sequence of gadget traversals and connections between gadgets.

As pushing-block puzzles are motion-planning problems at their core, we make extensive use of this framework to show the vast majority of our rules are NP-hard or PSPACE-hard.

[To-Do: I think it would be good to introduce the types of gadgets we use, and mention what hardness they give. Also add citations if any come from things other than the first two papers.] xxx

3 Generalized Swaps

We start by defining a generalized swap and prove that all generalized swaps are in NL and all symmetric generalized swaps are in L. We also prove that many generalized swaps are L-hard and NL-hard. For hardness, we reduce from non-planar graph reachability and thus we need a crossover gadget.

3.1 Rules in L and NL

Definition 9 (Generalized Swap)**.** Let P denote a subconfiguration that a rule is being applied to. A set of rules is a generalized swap if for each rule:

1. The only vertices that change state are the start s and ending location t of an agent, and
2. For all other vertices $p \in P \setminus \{s, t\}$ which do not contain a fixed wall F, adding or removing a block B from p does not impact the ability of the agent to move from s to t, though perhaps using different rules from the set.

Adding or removing blocks may prevent movement through other positions in P, however. A generalized swap is ***symmetric*** if the rule with the agent starting at t and moving to s is also a generalized swap.

Theorem 1. *Single Agent Reachability with a generalized swap is in NL. If the generalized swap is symmetric, it is in L.*

Proof. This problem can be reduced to $s \to t$ reachability in a digraph. For each cell, we have a corresponding vertex, and for each pair of cells a and b, we have an edge $a \to b$ if there is a rule that allows the transition between the agent in cell a to the agent in cell b with all other cells in the starting state. We prove that this reduction works in each direction:

Lemma 2. *If the agent can go from s to t in the game, then t is reachable from s in the digraph.*

Proof. We proceed by strong induction on the length of the path the agent took. In the base case, $s = t$, and clearly t is reachable from s in the digraph. Otherwise, the agent took a path from s to some cell t', and then a single move $t' \to t$. If, during the traversal of its path from s to t', the state of t changed, then the agent must have crossed t in their path to t', as the only way to move a block is to step into its location (by Generalized Swap condition 1). In that case, by the inductive hypothesis, t is reachable from s in the digraph. Otherwise, t still has its starting state. By condition 2, since the agent can (after $s \to t'$) make a single move from t' to t, they can also do so in the game's starting state (if they were to start at t'). This means we have a path $s \to t'$ and an edge $t' \to t$ in the digraph, forming a path $s \to t$. □

Lemma 3. *If t is reachable from s in the digraph, then the agent can go from s to t in the game.*

Proof. We proceed by strong induction on the length of the path in the digraph. As before, the base case is trivial. Otherwise, the digraph path can be partitioned into a path $s \to t'$ and a single edge $t' \to t$. By the induction hypothesis, the agent can move from s to t' in the game. If the state of t changed during the travel, the agent stepped on t and we are done. Otherwise, the agent is on t' and the state of t is the starting state. Note that the agent can make a single move $t' \to t$ if the game was in the starting state (except with the agent on t'). By condition 2, since the state of t did not change, it is still possible to go from t' to t, giving a path in the game from s to t. □

This completes the reduction. The reduction can be done in logarithmic space and reachability in a digraph is in NL. Furthermore, if the rule is symmetric, then we may treat edges as undirected and can thus solve graph reachability in deterministic logarithmic space, implying membership in L. □

Corollary 4. *The following games are in L under both the non-bendy setting and the bendy setting:*

- *Trivial ($eAB \to AeB$ and $AeB \to eAB$)*
- *Leap-F ($ABe \to eBA$)*
- *Vault-F ($AFe \to eFA$)*

- *Trail Leap-F (ABe → BBA)*
- *Trail Vault-F (AFe → BFA)*

The following games in in NL under both the non-bendy setting and the bendy setting:

- *Vault Swap-F (AFB → BFA)*
- *Vault Crush-F (AFB → eFA)*
- *Push into Wall-F (ABF → eAF)*
- *Wall Suplex-F (BAF → eAF)*
- *Fixed Wall's Push Swap-F (ABF → BAF)*
- *Fixed Wall's Pull Swap-F (BAF → ABF)*

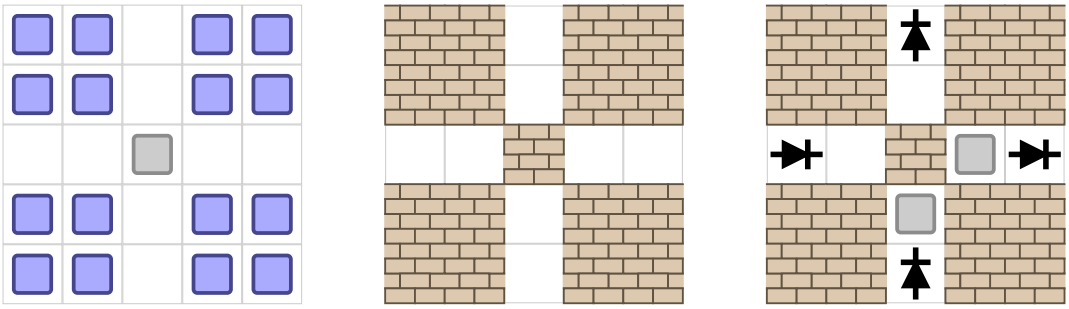
Proof. Leap-F and Trail Leap-F are generalized swaps as you may remove the middle block from each, and then use the movement rule to travel across. Vault-F and Trail Vault-F are as well, since the only block relevant to the movement that is not s or t is a fixed wall F. These four rules are symmetric since the agent could start in t and we still can apply the rule to move the agent from t to s.

Vault Swap-F and Vault Crush-F are generalized swaps for a similar reason as Vault-F although it is not symmetric. For Push into Wall-F, Fixed Wall's Push Swap-F, and Fixed Wall's Pull Swap-F, if we remove the single block from the rule, the agent is still free to move into its final position. While Wall Suplex-F itself does not fit the definition of a generalized swap, this game is equivalent to having $BAF \to AeF$ as the game rule, as one application of the movement rule to eAF reaches AeF. $BAF \to AeF$ is a generalized swap.

The arguments for membership in L or NL work regardless if the setting is bendy or non-bendy as the reasoning does not depend on whether or not the rules can only be applied to linear subconfigurations. The main difference is that there will be additional edges in the digraph for the bendy setting. □

3.2 Path Reductions

For L-hardness and NL-hardness we must provide a crossover gadget; We distinguish if the crossover gadget is directed or undirected, which influences the hardness of the ruleset.



(a) Leap ($ABe \to eBA$)
Trail Leap ($ABe \to BBA$)

(b) Vault ($AFe \to eFA$)
Trail Vault ($AFe \to BFA$)

(c) Vault Swap ($AFB \to BFA$)
Vault Crush ($AFB \to eFA$)

Figure 1: Undirected crossovers in (a) and (b), and a directed crossover in (c)

Theorem 5. *The following games are complete for their corresponding complexity classes:*

- *Leap (ABe → eBA) is L-complete.*
- *Vault-F (AFe → eFA) is L-complete.*
- *Trail Leap (ABe → BBA) is L-complete.*
- *Trail Vault-F (AFe → BFA) is L-complete.*
- *Vault Crush-F (AFB → eFA) is NL-complete.*
- *Vault Swap-F (AFB → BFA) is NL-complete.*

Proof. We reduce from (directed) graph reachability using the crossover gadgets in Figure 1. As the first two crossover gadgets are undirected, we only get L-completeness for the first four games; since the last crossover gadget is directed, we get NL-completeness for the last two games. □

4 Conservative Games

We present the complete set of "conservative" rules on three nodes. A rule is ***conservative*** if and only if:

1. The number of movable blocks present on the board before the rule is applied is equal to the number of movable blocks after, and
2. The number of fixed walls is the same before and after applying the rule, and fixed walls do not change position.

Table 1 summarizes our results. There are several rules which are "congruent" to other rules. Two rules R_1, R_2 are ***congruent*** if applying the movement rule to either the left or right hand states of R_1 can produce identical states to R_2. If the movement rule is congruent to a rule R, then R is ***trivial***. **[is congruent symmetric? If not I think we should rename it.]** xxx **[When we use the term congruent we also allow movement rules so we need congruent to apply to rule systems as well as rules.]** xxx

4.1 Congruent Rules

Observation 6. *The movement rule is congruent to the rules AeB → eAB and eAB → AeB.*

Proof. If we have the arrangement *AeB*, one application of the movement rule returns the state *eAB*. The movement rule is congruent to *AeB → eAB*.

Given the arrangement *eAB*, one application of the movement rule returns *AeB*. The movement rule is congruent to *eAB → AeB*. □

Observation 7. *ABe → AeB is congruent to Push-1 (ABe → eAB).*

Proof. Given the arrangement *ABe*, one application of *ABe → AeB* gets us to *AeB*, and then one movement turns the output into *eAB*. □

Observation 8. *ABe → BeA is congruent to Push Swap at Empty, or ABe → BAe.*

Proof. After applying this rule to *ABe* we get *BeA*, and then one movement rule sends *BeA* to *BAe*. □

Observation 9. *eAB → BeA, AeB → BAe, and AeB → BeA are equivalent to Suplex-1 (eAB → BAe).*

Proof. Note that $eAB \to BeA$ differs from Suplex-1 on its output, but one movement on BeA turns it into BAe. Further, $AeB \to BAe$ differs from Suplex-1 on its input, but one movement on AeB transforms it to eAB. Finally, note that $AeB \to BeA$ differs from Suplex-1 on its input and output states. The arguments from the two previous rules show that we can transform both the input and output to the states of Suplex-1. □

Observation 10. *$AeB \to eBA$ is congruent to Pull Swap at Empty, or $eAB \to eBA$.*

Proof. Note that $AeB \to eBA$ differs from Pull Swap at Empty by its input. We can transform AeB to eAB with one movement. □

Observation 11. *$AeB \to ABe$ is equivalent to Pull?-1 ($eAB \to ABe$).*

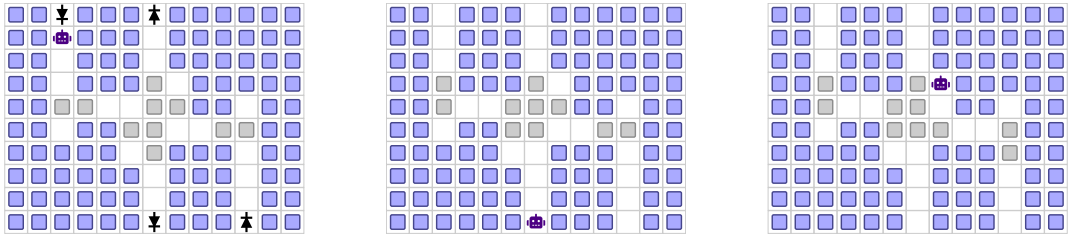
Proof. Note that $AeB \to ABe$ differs from Pull?-1 on its input, but we can transform AeB to eAB with one movement rule. □

4.2 Rules in NP

Lemma 12. *Single-Agent Reachability for Push Swap at Empty is NP-hard.*

Proof. In Figure 2 we show an antiparallel matched dicrumbler gadget for this rule. This gadget has two opposing paths, and traversing one prevents and subsequent traversals of either path. The possible paths through this gadget are from the upper left to lower middle and lower right to upper middle. Traversing either path will move the gadget to a state where no traversal is possible. The correctness of this gadget was additionally verified via computer.

Antiparrel matched dicrumblers suffice for NP-hardness. □



(a) Initial state **(b)** First traversal (left to down) **(c)** Fail at second traversal (right to up)

Figure 2: An antiparallel matched dicrumbler gadget for the Push Swap at Empty ($ABe \to BAe$) model

[need a definition and description of a NAND gadget and the mapping from Figure 2 to the gadget locations] xxx

[The table says NP-C but this only shows hardness. Where is the containment proof? (I'd guess the table has a typo but the section heading also explicitly says in NP.)] xxx

[There is a containment proof on Coauthor; it should probably be added/included here. - Jeffery] xxx

Lemma 13. *The shortest solution to any instance of Single Agent Reachability for Push Swap at Empty visits no cell twice.*

Proof. Suppose there is a shortest path P from s to t that visits at least one cell twice. Take the minimum suffix of P in which only one cell is visited twice. Let this suffix be;

$$s \to a_1 \to \cdots \to a_m \to s \to b_1 \to \cdots \to b_n$$

So cell s is repeated but no other cell in the suffix is repeated. We claim that we can produce a shorter suffix contradicting the assumption P is a shortest path. If $s \to b_1 \to \cdots \to b_n$ can be traversed without the traversal of $s \to a_1 \to \cdots \to a_m$ then we could replace our current suffix with $s \to b_1 \to \cdots \to b_n$. Therefore, suppose this is not the case. No cell of $\{b_1, b_2, \ldots, b_n\}$ is visited by the traversal of $s \to a_1 \to \cdots \to a_m$ so it cannot be the case that there is a block at b_j that we need to remove by visiting some a_i so we can traverse $s \to b_1 \to \cdots \to b_n$. Otherwise this would imply that $a_i \in \{b_1, b_2, \ldots, b_n\}$. Therefore if $s \to b_1 \to \cdots \to b_n$ is not a valid shorter suffix, there is some block in a cell adjacent to some b_j that prevents the application of $ABe \to BAe$. Therefore there is a cell a_i which is adjacent to b_j, and a_i initially has some block we need to remove. But if this were the case, the agent could take the pair a_i, b_j with maximum j and construct a shorter suffix $s \to a_1 \to \cdots \to a_i \to b_j \to \cdots \to b_n$. Since j is maximal no traversal after b_j will require the traversal of any a_i, so $b_j \to \cdots \to b_n$ must be valid.

□

Therefore if a solution exists, the shortest solution is bounded by the size of the board, which means it is a polynomial length certificate.

Theorem 14. *Single-Agent Reachability for Push Swap at Empty is NP-complete.*

4.3 PSPACE-complete Rules

[again need gadget figures and mappings from constructions to gadgets.] xxx

Theorem 15. *Single-agent Reachability for the following rules is PSPACE-complete:*

- *Swap-2F (ABB → BBA)*
- *Pull Swap at Empty (eAB → eBA)*
- *Suplex-1F (eAB → BAe)*
- *Pull Swap at Block-F (BAB → ABB)*
- *Push Swap at Block-F (ABB → BAB)*

Proof. We construct a Locking-2-Toggle (L2T) gadget for each rule:

Swap - 2F Figure 4 demonstrates the L2T gadget for this model. An agent can traverse from either the bottom left to upper left corner or bottom right to top right. Either of these traversals locks the gadget such that the gadget only admits the opposite traversal.

Pull Swap at Empty Figure 5 shows a L2T for this model. The blue tiles can either be fixed walls or movable blocks. both will work as the agent is unable to affect two blocks in a row. This gadget is traversed either bottom to top or right to left. Either of these traversals lock the gadget and then the opposite traversal becomes the only traversal admitted.

Suplex-1F Figure 6 presents a L2T for Suplex-1F. This gadget can either be traversed from the bottom to the right edge, or from the left to the top edge. Either of these traversals lock the gadget and only the opposite traversal is then possible.

Pull Swap at Block-F Figure 7 gives a L2T for this model. It can be traversed from the top left to bottom left or top right to bottom right (following the arrows). Again, either traversal will lock the gadget, only allowing the opposite traversal.

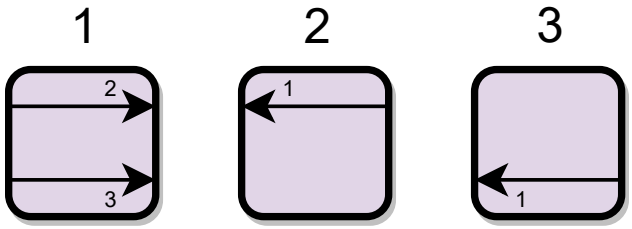
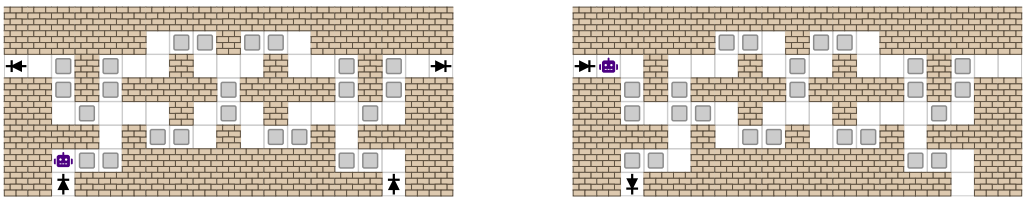


Figure 3: The Locking 2-Toggle (L2T) gadget and its states from the motion planning framework. The numbers above indicate the state and when a traversal happens across the arrows, the gadget changes to the indicated state.

Push Swap at Block-F A L2T is given in Figure 8, this gadget can be traversed from the bottom left to bottom right, or top right to top left. Either traversal will lock the gadget only admitting the opposite traversal.

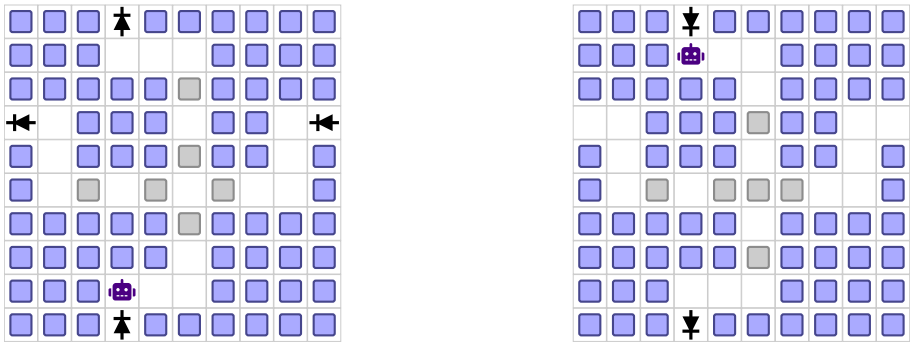
Correctness of these gadgets was also verified programmatically, to check that no other traversals are possible.

Locking-2-Toggles suffice for PSPACE-hardness; as these rules are in PSPACE, we also get PSPACE-completeness. □



(a) Initial state **(b)** Locked state

Figure 4: An L2T gadget for the Swap-2F ($ABB \rightarrow BBA$) model



(a) Initial state **(b)** Locked state

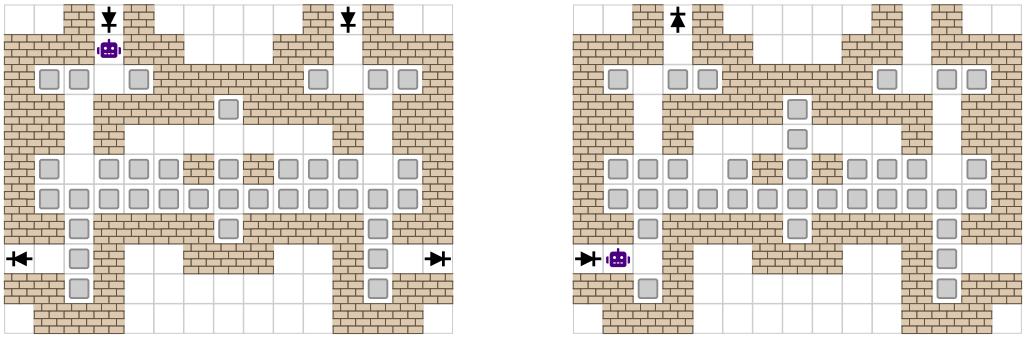
Figure 5: An L2T gadget for the Pull Swap at Empty ($eAB \rightarrow eBA$) model

4.4 Polynomial Time Rules

Here we show agent reachability Suplex-1 with no fixed walls is in P. Recall that Suplex-1 is the rule $eAB \rightarrow BAe$ (or equivalently, $eAB \rightarrow BeA$). We will work with the version with the rule $eAB \rightarrow BAe$. We call applications of the this rule (which does not move the agent) ***suplex moves*** to distinguish it from

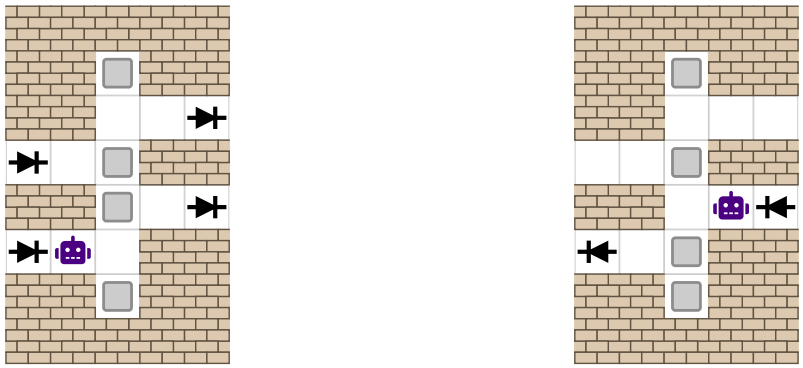
(a) Initial state **(b)** Locked state

Figure 6: An L2T gadget for the Suplex-1F ($eAB \rightarrow BAe$) model



(a) Initial state **(b)** Locked state

Figure 7: An L2T gadget for the Push Swap at Block-F ($BAB \rightarrow ABB$) model



(a) Initial state **(b)** Locked state

Figure 8: An L2T gadget for the Pull Swap at Block-F ($ABB \rightarrow BAB$) model

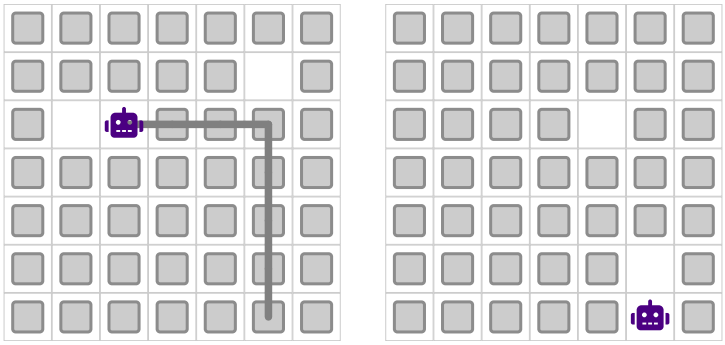
moving the agent with the movement rule. We call empty locations ***holes***; because the agent needs a hole behind it in order to move blocks, keeping track of the location of holes will be key to our algorithm. Call a location ***reachable*** if it is possible for the agent to ever reach that location, and call a move a ***turn*** if it is in a different direction than the previous move (i.e., horizontal after a vertical or vice versa). We will think of the path the agent takes as a sequence of ***segments*** and turns, where a segment is the sequence of consecutive moves in the same direction. Most of the time, when following a path, the agent will alternate between using suplex moves to clear the space in front of them, and then moving into the newly cleared space. Using these ideas we show Suplex-1 without fixed blocks is in P.

First, we note with a couple of observations about how Suplex-1 moves work.

Observation 16. *If the agent is adjacent to an empty location in the same row, then the agent can reach anywhere in that row.*

Observation 17. *Whenever the agent turns, it must leave a hole behind in the row/column it came from, and it can only turn if there is a hole adjacent to its location in the adjacent row/column from where it was moving.*

From these we see that a sequence of suplex-1 moves tends to look like the agent dragging an hole behind itself, leaving behind a hole and picking up a new one each time it turns. We will call these two holes the ***turnpike*** associated with that turn. We say that the turnpike moves one space diagonally after the turn is made, and player has two options as to which direction the turnpike moves. Either they leave behind the hole in the space they just came from or the space immediately past this turn in the direction they were previously going in. See Figure 9.



(a) The path before the turn **(b)** The state after the turn

Figure 9: An example of a turn in Suplex-1. Notice how the turnpike hole has move one space diagonally down and left after traversing the turn.

We say that a path is ***simple*** if it satisfies the following:

- does not contain two segments in the same row or column
- does not contain two segments in adjacent rows or columns

We will now give a sufficient condition for an agent to be able to traverse a simple path. The intuition is that all the agent needs is to start next to a hole pointing in the right direction, and at each turn in the path there needs to be an adjacent hole in the row/column of the next segment of the path.

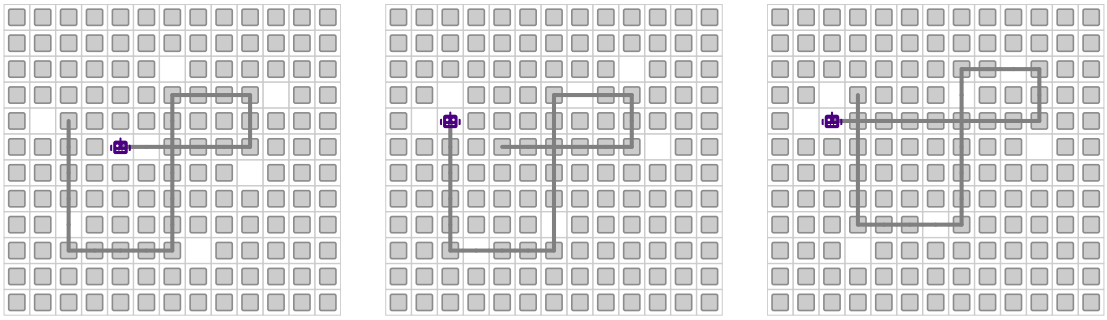
Lemma 18. *Let P be a simple path which starts at the agent's location. The agent can traverse the path if the following is true for the agent's start location and of each turn t at location (x, y):*

- *If the next segment after t is horizontal, there is a hole at $(x+1, y)$ or $(x-1, y)$*
- *If the next segment after t is vertical, there is a hole at $(x, y+1)$ or $(x, y-1)$*

Proof. The agent can always move in the same direction as their last move, since whenever the agent moves, it must leave a hole in the cell directly behind it. Thus, as long as all of the turns are legal moves, the path is traversable. As long as there is a hole directly behind the agent, the agent can move in that direction, dragging the hole behind them. If there is a hole in front of the agent, the agent can simply move directly in that direction, after which there is a hole behind them. The conditions guarantee that there is a hole either directly in front of or behind the agent at each turn, so the agent can always make the turn move.

We also need to ensure that the holes are not filled by some move earlier in the path before they are used. Because the path is simple, it cannot cross one of these holes, nor can it use the same hole for two different turns, so it's never the case that the path fills one of the holes before it is needed. □

An example of a simple path which the agent can traverse is in Figure 10.



(a) A simple path P **(b)** The state C_1 after traversing P **(c)** The path Q from C_1.

Figure 10: An example of a configuration that satisfies Lemma 19, and the paths the agent takes to reach two rows up. Notice how Q just P shifted up and to the right 1 tile.

Lemma 19. *Let C_0 be the initial configuration. Suppose there is a traversable simple path P that starts with a horizontal move in row y and ends horizontally adjacent to a hole in row $y+1$. Then there is a reachable configuration C_1 and simple path Q such that Q is traversable from C_1, and Q ends horizontally adjacent to a hole in row $y+2$. This results in a configuration with the agent horizontally adjacent to a hole in row $y+2$.*

Proof. Q will be exactly P, but all of the moves are shifted one unit in the positive x and y directions. To get to C_1, we will have the agent take the path P, but possibly using some additional suplex moves. At each turn in P, there are two possible locations the turnpike can end up. We make a suplex move if necessary so that the turnpike always ends up in the one that is further in the positive x and y directions. Because the path P was simple, neither of these locations could have been touched by any moves later in P, so this choice leaves all of the turnpikes from P unfilled. Figure 10 gives an example of a configurations C_0 and C_1, and a paths P and Q.

Now, when we do Q, all the turnpikes form P (and the hole in row $y+1$) will be exactly in the locations needed for Lemma 18 to show that Q is a traversable path. In particular, for each turn at location (x, y) in Q, the turnpike from the corresponding turn in P will be at $(x, y+1)$ if the next segment is vertical, or $(x+1, y)$ if the next segment is horizontal. The last segment of P must have been vertical, so the last move along P can leave a hole in row $y+2$, which the agent will be adjacent to at the end of Q. □

By starting at C_1 and tracing P backwards, we can use this lemma to shift down by one row as well. By repeated application of this lemma, if we can shift up by one row, we can shift up or down by any number of rows. This leads to the following stronger corollary.

Corollary 20. *Suppose there is a traversable simple path P that starts with a horizontal move in row y and ends horizontally adjacent to a hole in row $y+1$. Then the agent can reach everywhere.*

Now, we strengthen our result to handle cases where the agent can reach an adjacent row via an arbitrary (not necessarily simple) path.

Lemma 21. *Suppose there is* any *sequence of moves P that starts with a horizontal move in row y and ends in a configuration with the agent horizontally adjacent to a hole in row $y+1$. Then the agent can reach anywhere.*

Proof. If the P is a simple path, then we can apply Corollary 20 and be done. Thus we assume P is not a simple path.

We will now prove this by induction on the length of P. There are two cases based on whether there are segments s_1 and s_2 such that s_1 and s_2 are in adjacent rows/columns.

If there are two segments s_1 and s_2 in P that are in adjacent rows or columns, then consider the subpath that starts with s_1 and up to but not including s_2. Then this path starts with a horizontal move in some row y' and ends in an adjacent row with a hole in that row adjacent to the agent. This subpath is a shorter path that matches the conditions of our inductive hypothesis, so the agent can reach anywhere.

The other case is when there is no s_1 and s_2 in adjacent rows/columns. The only other way P can be not simple is if there are segments s_1 and s_2 in the same row or column. In this case, we shortcut: we remove the portion of the path between s_1 and s_2, and since both in the same row, we can simply merge them into one longer segment. The merging is possible by Observation 16. The remainder of the path is still valid since for each turn, the hole adjacent to the agent which allowed the turn was not on the shortcut path since that would require there to be segments on adjacent rows. After shortcutting, we have a shorter path that satisfies the conditions of our inductive hypothesis, so the agent can reach anywhere. □

Now we further strengthen this to any adjacent reachable locations.

Lemma 22. *Let $\ell_0 = (x, y), \ell_1 = (x, y + 1), \ell_2 = (x, y + 2)$ be three adjacent colinear locations. Suppose the agent starts at ℓ_0, and ℓ_1 is reachable. Then ℓ_2 is reachable.*

Proof. If ℓ_1 is reachable from ℓ_0 via a vertical move, then the result is trivial: just move another space in that direction. Otherwise, consider the path from ℓ_0 to ℓ_1. It must start and end with a horizontal move. That means that at the end, there is an adjacent hole to ℓ_1 when the agent is in ℓ_1. Then by Lemma 21 the agent can reach everywhere, in particular including ℓ_2. □

Finally, we give a full characterization of exactly where the agent can reach. Let S be the smallest set of locations containing the agent that satisfies the following conditions:

- If $\ell_0 \in S$ and ℓ_1 is an empty location adjacent to ℓ_0, $l_\ell \in S$.
- If $\ell_0, \ell_1 \in S$ are adjacent locations in the same row, then every location in that row is in S.

S can be easily computed greedily by starting from the agent's start location and adding elements to S that satisfy either of the above conditions. Now we show that this is in fact a complete characterization of the set of reachable locations.

Theorem 23. *The set of locations S as defined in Lemma 22 is exactly the set of reachable locations. Thus Suplex-1 ($eAB \rightarrow BAe$) is in P.*

Proof. We can see that every location in S is reachable by repeated application of Lemma 22. Now we need to show that every reachable location is in S. We will show by induction this is true. Our inductive hypothesis is that if the agent is at a location ℓ after i steps, then ℓ and every adjacent empty location is in S. On their $i + 1$th step, the agent either moves into an adjacent empty space or the agent makes a suplex move. If the agent made a suplex move, then the agent's location didn't change, and a single new location adjacent to the agent became empty. By our inductive hypothesis ℓ is in S and the location across from our new empty location must have previously been empty and thus in S. The second condition of S implies that this new location is in S.

If the agent moves on step $i + 1$, then it's new location ℓ' is in S by the inductive hypothesis. For each adjacent empty location, either it was empty initially or it was not. If it was empty initially, then by the first condition of S it must be in S. If it was not, then the agent must have made a suplex move adjacent

to that location at some earlier timestep. In that case, the agent's location, and the location the suplexed block was moved to both must have been in S by our inductive hypothesis. Then ℓ' is in the same row or column as two other locations in S, so it must also be in S.

Thus, every reachable location is in S. □

5 Bounded Games

In this section we study what we call "bounded" games. In a ***bounded game***, the total number of movable blocks monotonically increases or decreases with each application of the game rule. We can divide the bounded games into two types: ***Zap***, which deletes blocks in each rule application, and ***Trail***, which adds blocks. The results are outlined in Table 2. We then prove the remaining bounded games are all easy via a greedy algorithm. Finally we investigate which of these games are P-complete. We start with the proof bounded games are all in NP.

Theorem 24. *All bounded games are in NP.*

Proof. If reachability is possible, then consider the shortest sequence of moves that allows the agent to reach the target location, which can be given as a certificate. It's clear that we can verify that this sequence works in polynomial time by following the moves in the sequence and seeing if we end up at the target location. We claim that the length of this sequence is polynomial. Indeed, since the game rule monotonically changes the number of blocks on the board, there can only be a polynomial number of applications of the game rule in the certificate. Furthermore, in between each application of the game rule, there can only be a polynomial number of applications of the movement rule, as otherwise the agent visits some vertex twice without applying the game rule in between, contradicting the fact that the sequence of moves is the shortest possible (as we can cut out the loop from the sequence of moves and get a shorter sequence of moves). Therefore, the overall length of the sequence must be polynomial, meaning that the certificate is polynomial-length. This means that reachability is in NP. □

5.1 NAND Gadgets

In many of these games we get NP-hardness by building a NAND gadget. This gadget has two tunnels which both start open. Traversing either tunnel closes the other and leaves the first open. We use two variants: ***anti-parallel*** and ***crossing***. There are other slight variations on this such as whether or not tunnels are directed, and whether or not the tunnels can be reused after. The second distinction has a different name in previous literature. When tunnels can be reused the gadget is referred to as a ***NAND***, then they both close after a traversal it is a ***Matched Dicrumbler***. In all of these cases the behavior of the gadget remains the same the problem is NP-hard. The first is shown in Figure 11 For the last reduction we introduce a new version of the NAND called the ***Delayed NAND***. This NAND adds an extra initial state where the gadget must be "activated" by a button before solving. It is of note that all of these NP-complete reductions do not use fixed blocks.

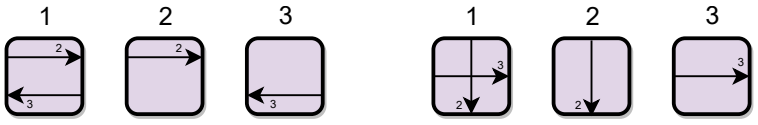
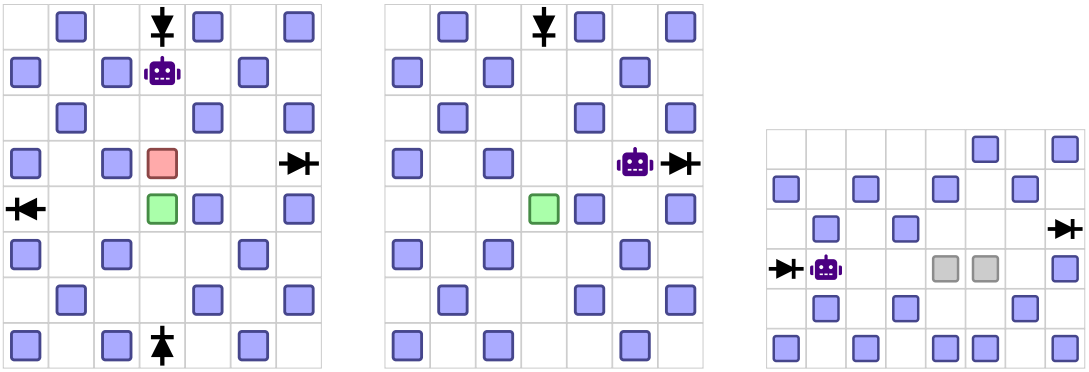


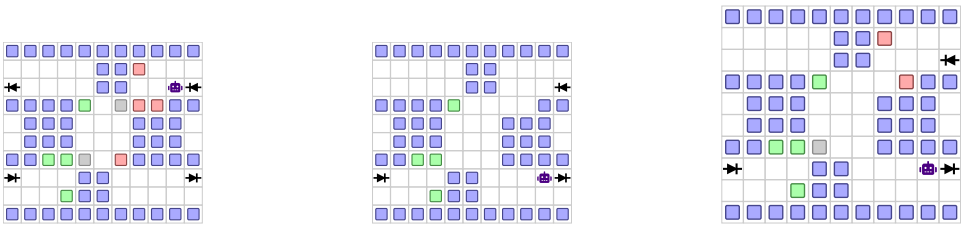
Figure 11: Antiparallel (Left) and Crossing (Right) NAND gadget shown. This gadgets are drawn with all transitions directed however some of our gadgets are undirected or mixed, i.e., having both types. In all of these settings the motion planning problem is NP-complete.

5.1.1 Anti-Parallel



(a) Initial state **(b)** Traversed state **(c)** Diode

Figure 12: Antiparallel NAND and Diode for Push Merge ($ABB \rightarrow AeB$)



(a) Initial State **(b)** Sup. Smash ($BAB \rightarrow eAe$) final **(c)** Sup. Merge ($BAB \rightarrow BAe$) final

Figure 13: Antiparallel NAND for Suplex Smash ($BAB \rightarrow eAe$) and Suplex Merge ($BAB \rightarrow BAe$)

Theorem 25. *The following games are NP-complete:*

- *Push Merge (ABB → AeB)*
- *Bendy Push Merge-F (ABB → AeB)*
- *Suplex Smash (BAB → eAe)*
- *Bendy Suplex Smash-F (BAB → eAe)*
- *Suplex Merge (BAB → BAe)*
- *Bendy Suplex Merge-F (BAB → BAe)*
- *Bendy Clap Merge-F (BAB → eBA)*

Proof. We can build an anti-parallel NAND gadget for each of these rules; the anti-parallel NANDs are shown in Figures 12, **??**, 13. In the gadgets, the color of the blocks indicate which blocks get deleted in one traversal or the other (gray blocks get deleted in both traversals). In the figures, the agent starts at the opening of the red tunnel.

In Push Merge (Figure 12), the agent may enter from north or south, then delete the top or bottom block to open the side tunnels. If the agent enters from the east or west tunnel they may delete the block and cause unintended behavior. To prevent this, we build the one-way gadget in Figure 12(c) (where the agent

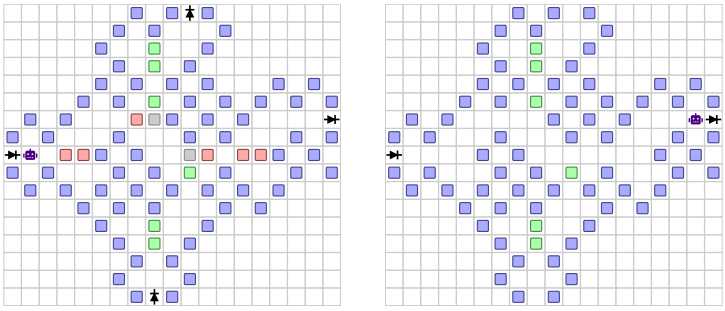
cannot traverse from $b \to a$ until the agent has traversed $a \to b$ first) and add it to output of each tunnel. Also in the traversed state (Figure 12(b)) the block adjacent to the opposite (green/red) can be zapped. This allows to go to the other input tunnel, however the output tunnel is still unreachable. We can also add a one-way in front of each input tunnel so the agent can't cross unless the other side is already reachable. The resulting gadget behaves as a directed NAND, which is NP-hard.

For the Suplex Smash and Suplex Merge gadget, in order to enter the center, we must delete one of the gray blocks. Then in order to exit we must "suplex" the the block in the way of the exit into the opposite gray block. If we enter the unused input after traversing we can still delete the pair of input blocks. This allows an additional traversal in Suplex Smash however this gadget is equivalent to a NAND because the set of reachable locations. We must make the paths thicker in order to prevent the agent from deleting blue blocks to create additional paths.

The gadgets so far still work in the bendy setting, although the blocks that we don't want the agent to interact with (i.e., the blue blocks) must now be fixed walls.

Anti-parallel NANDs suffice for NP-hardness; as these rules are in NP, we also get NP-completeness. □

5.1.2 Crossing



(a) Initial state **(b)** Traversed State

Figure 14: Crossing NAND for Push Smash ($ABB \to Aee$).

Theorem 26. *The following rule is NP-complete:*

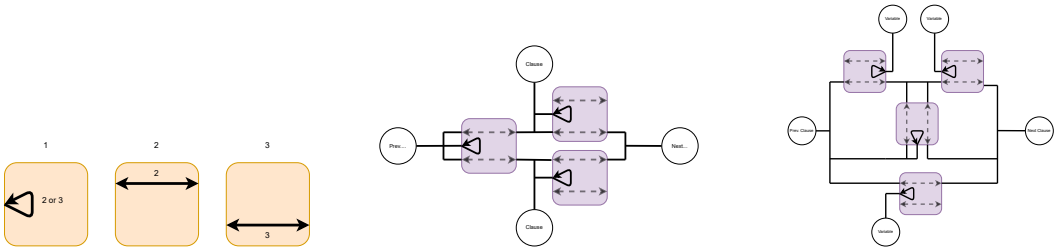
- *Push Smash ($ABB \to Aee$)*

Proof. We build a Crossing NAND gadget, shown in Figure 14. As demonstrated by the figure, one traversal left to right leaves that traversal open for the future, however the removal of some key blocks in the middle of the gadget prevent a later top to bottom traversal. This gadget is symmetric so the opposite is also true if the initial traversal was top to bottom. Crossing NANDs suffice for NP-hardness; as these rules are in NP, we also get NP-completeness. □

5.2 Delayed NAND

Pull Merge requires a special gadget called a ***delayed NAND*** (Figure 15). This is a NAND where the gadget has an additional button which sets the state of the gadget to top or bottom open.

Theorem 27. *Agent Reachability with the Delayed NAND is NP-complete.*



(a) Delayed NAND **(b)** Variable Gadget **(c)** Clause Gadget

Figure 15: (a) Delayed NAND state diagram (b) Variable gadget, the agent moves from left to right through the gadget and is able to set the variables. (c) The clause gadget. In order to traverse from left to right through the gadget one of the doors must have been opened.

Proof. We reduce from Linked Planar 3SAT [Pil19]. This is a special case of Planar 3SAT where we are also given a cycle that visits all the variables, then all the clauses. In addition we assume the "sided" condition which means that positive appearances of a variable appear one side of the gadget while negative appears are on the opposite side.

The gadgets are shown in Figure 15. The agent enters a variable gadget and must chose up or down corresponding to true or false. The agent may then go to press buttons in the clause gadget which the literal appears. Once the agent has set all the variables then it starts traversing through clause gadgets. The agent may only cross the clause if at least one of the buttons in the outer three gadgets.

If there exists a satisfying assignment to the formula then the agent may set all the variables gadgets to the bits denoted by the assignment. This will allow the agent to open all the clause gadgets and reach the target. If the agent can reach the target then it must have pressed buttons in a way to satisfy each clause which represents an assignment. □

(a) Initial state **(b)** Setup state

Figure 16: Construction for Pull Merge (*ABB* → *ABe*). The agent deletes the red blocks to open the right tunnel.

Theorem 28. *Pull Merge (ABB → ABe) is NP-complete.*

Proof. We build a delayed NAND gadget for Pull Merge; see Figure 16. As we have shown that delayed NAND gadgets suffice for NP-hardness in Theorem 27, we get NP-hardness for Pull Merge; as Pull Merge is in NP, we also get NP-completeness. □

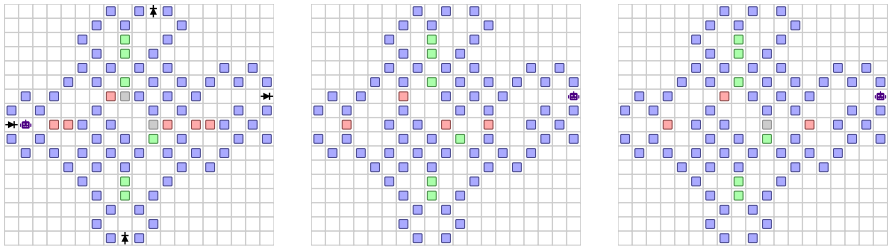
5.3 Matched Crumblers

We present NP-hardness for five models using varaints of Matched Crumbler gadgets. These are gadgets that have two traversals and by conducting one traversal, the gadget becomes impassible. We use crossing matched crumblers and antiparallel matched dicrumblers. In a crossing matched crumbler, there are two traversals that, if one traced with a curve, cross. In a antiparallel matched dicrumbler, the two traversals are uncrossing but face in opposing directions.

5.3.1 Crossing Matched Crumblers

(a) Initial state **(b)** Traversed state

Figure 17: Crossing matched crumber for the Trail Swap ($eAB \rightarrow BBA$) model



(a) Initial state **(b)** Swap Zap ($ABB \rightarrow BeA$) **(c)** Leap Crush ($ABB \rightarrow eBA$)

Figure 18: Crossing NANDs for Swap Zap ($ABB \rightarrow BeA$) and Leap Crush ($ABB \rightarrow eBA$), (b) and (c) show the state of the gadget after traversal.

Theorem 29. *The following models are NP-complete:*

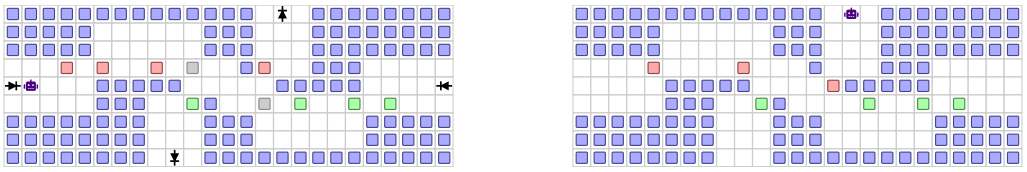
- *Swap Zap* ($ABB \rightarrow BeA$)
- *Leap Crush* ($ABB \rightarrow eBA$)
- *Trail Swap* ($eAB \rightarrow BBA$)

We construct crossing matched crumblers, as shown in Figures 17 and 18. In the gadget for Trail Swap (Figure 17) one traversal in either direction will leave the middle of the gradet filled with blocks preventing a future traversal.
Due to the similarity of the rules for Swap Zap and Leap Crush, we are able to build one gadget for both (Figure 18). A left to right traversal is shown for both rules, these traversals leave both paths untraversable, as they remove key blocks (shown in gray) that are needed to leave an entrance, or enter an exit.

5.3.2 Antiparallel Matched Dicrumblers

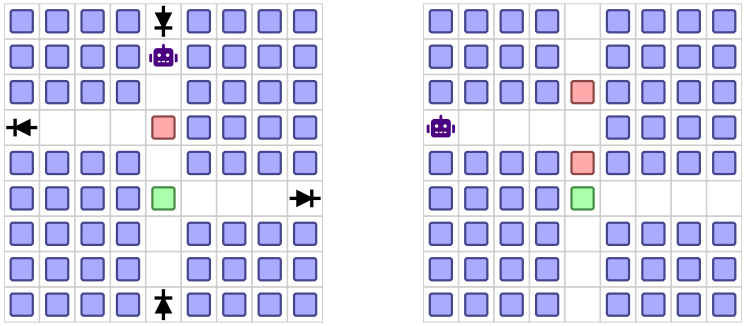
Theorem 30. *The following games are NP-complete:*

- *Clap Merge (BAB → eBA)*
- *Trail Push (ABe → BAB)*



(a) Initial state **(b)** Traversed state

Figure 19: Antiparallel matched dicrumbler gadget for the Clap Merge ($BAB \longrightarrow eBA$) model



(a) Initial state **(b)** Traversed State

Figure 20: Antiparallel matched dicrumbler for the Trail Push ($ABe \longrightarrow BAB$) model

Proof. For each of these gadgets we build an antiparallel matched dicrumbler, show in Figures 19 and 20. In Figure 19 we present a traversal from the left entrance to the top exit. This traversal blocks the left endtrance and the top and bottom exits. If an agent chooses to enter from the right entrance after such a traversal they will therefore be stuck in the middle of the gadget. This gadget is symmetric and so the same behavior occurs when the right entrance is traversed to the bottom exit. This is precisely the behavior of an antiparallel matched dicrumbler.

The Trail Push gadget functions identically. □

5.4 More Easy Cases

First we will describe a greedy algorithm which gives membership in P for a few games. We only include rules here that aren't covered by log-space algorithms described in Section 3.

Theorem 31. *The following games are in P:*

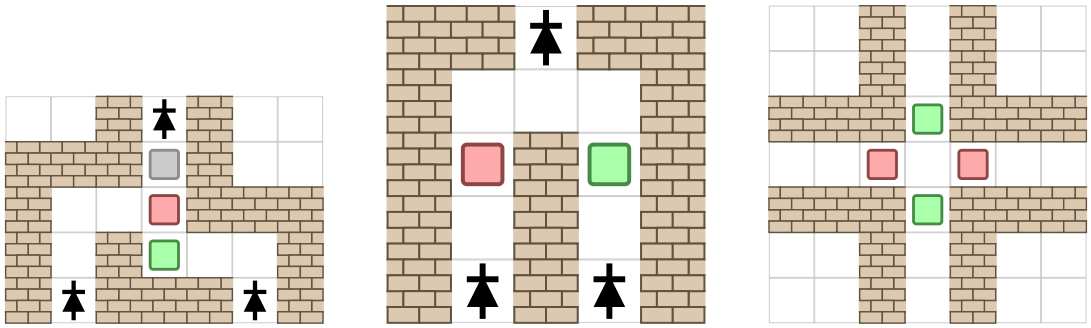
- *Battering Ram-F ($eAB \longrightarrow eAe$)*
- *Knock over-F ($ABe \longrightarrow Aee$)*
- *Zap Through Walls-F ($AFB \longrightarrow AFe$)*

Proof. We consider a greedy algorithm which repeatedly applies the rule to destroy all movable blocks that are able to be destroyed. For these games, deleting a block only ever helps the agent. This also generalizes to any rule where there is only one block on the left side and the agent can delete it.

□

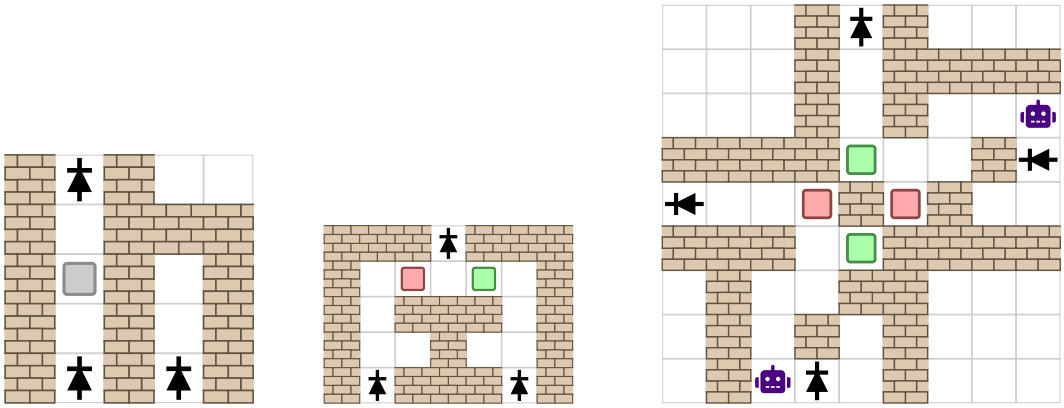
Theorem 32. *The following games are P-complete:*

- *Battering Ram-F ($eAB \longrightarrow eAe$)*



(a) AND **(b)** OR **(c)** Crossover

Figure 21: Battering Ram-F (*eAB* → *eAe*) P-complete Reduction



(a) AND gadget **(b)** OR gadget **(c)** Crossover gadget for Multiple Agents

Figure 22: Zap Through Walls-F (*AFB* → *AFe*) P-complete Reduction

- *Zap Through Walls-F (AFB → AFe) in 3D*
- *Zap Through Walls-F (AFB → AFe) with Multiple Agents*

Proof. We reduce from Monotone CVP. For each rule, we construct an AND gadget, an OR gadget and a crossover gadget. The gadgets for Battering Ram are shown in Figure 21. The gadgets for Zap Through Walls-F are shown in Figure 22. The crossover uses multiple agents species *A*. We modify the construction so each wire contains a single agent isolated in it's room. In 3D we get crossovers easily. □

5.4.1 Zap-Through-Walls

Corollary 33. *Zap-Through-Walls-F (AFB → AFe) is PMCVP-hard.*

Proof. Using only the AND and OR gadgets from Figure 22 we can reduce from Planar Monotone Circuit Value Problem (a problem known to be in NC^3). This reduction can be done in log space. Thus, if Zap-Through-Walls is in NL, then so is PMCVP. □

Theorem 34. *Bendy Zap Through Wall (AFB → AFe) is PMCVP-hard in 2D and P-complete in 3D.*

Proof. The gadgets for Zap Through Wall in the non-bendy setting still work in the bendy setting. □

5.4.2 Battering Ram

Here, we give a result for Battering Ram ($eAB \rightarrow eAe$) with no fixed blocks:

[Would a figure help here?] xxx

Theorem 35. *Battering Ram is NL-complete.*

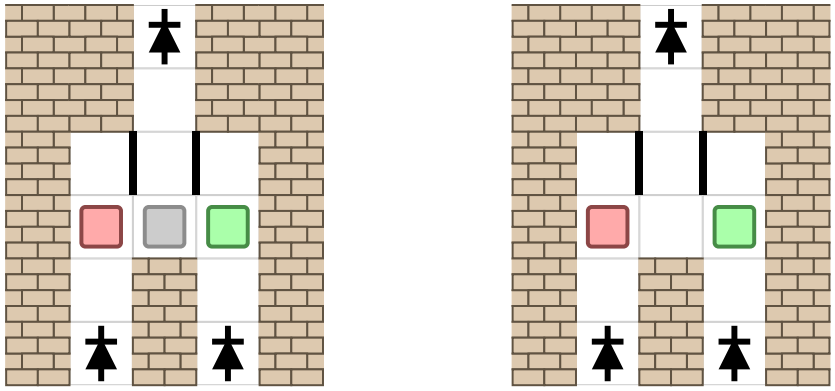
Proof. Note that if the agent is adjacent to an empty space e it may delete all the blocks in that row or column. We can then treat any instance of Battering Ram as equivalent to a special kind of directed graph. Given an instance of Battering Ram we construct graph G with a node r_i for each i^{th} row, c_j for each column, and a special start node s. W.L.O.G assume $s = 0, 0$. We add the following edges:

- An edge from r_i to c_j if there's an empty cell in either $i-1, j$ or $i+1, j$,
- An edge from c_j to r_i if there's an empty cell in either $i, j-1$ or $i, j+1$,
- An edge from s to c_j if there's an empty cell in either $-1, j$ or $1, j$,
- An edge from s to r_i if there's an empty cell in either $i, -1$ or $i, 1$.

The agent can reach the target node if and only if there's a path in this graph, thus implying membership in NL. For hardness we note that any bipartite graph can be encoded by reserving the reduction described above. We can assign each node 3 rows or columns.**[what does this previous sentence mean? - Jeffery]** xxx Then for each edge $r_i \rightarrow c_j$ we add an empty space to $i+1, j$. We gain NL-hardness since any directed graph can be made bipartite by adding extra nodes between each edge. □

5.4.3 Knock Over

Knock Over ($ABe \rightarrow Aee$) is of interest because the problem is P-complete if the problem is in 3D, PMCVP-hard in 2D with thing walls, but is in NL in 2D with just fixed blocks. This is the only section we have where -W vs -F makes a distinction so we specify this in our Theorems. Note that Theorem 31 holds for -W.



(a) AND gadget **(b)** OR gadget

Figure 23: Knockover-W ($ABe \rightarrow Aee$) gates. In 3D the middle tunnel goes through plane $z = 2$

Theorem 36. *Knock Over-W ($ABe \rightarrow Aee$) is PMCVP-hard.*

Proof. We utilize the AND and OR gadgets shown in Figure 23. This allows for us to reduce from Planar Monotone Circuit Value Problem. □

Theorem 37. *Knock Over-F (ABe → Aee) in 3D is P-complete.*

Proof. Imagine the output of the AND is pointed upward in the third dimension This preserves the key detail of the gadget which is that the empty space behind the other blocks is disconnected from the output wire. We can also use the third dimension to cross over wires allowing us to reduce from Non-Planar Monotone SAT. □

Theorem 38. *Knock Over-F (ABe → Aee) is in NL.*

Proof. Let D be a set of rule applications which solves the game by deleting the fewest amount of blocks. We will argue that we may ignore many blocks to reach an instance which is easy.

For an block B_t we will call it horizontally ***sandwiched*** if it has a neighboring blocks B_e, B_w to the east and west. Define vertically sandwiched in the same way. Assume we must delete B_t while standing next to it horizontally to each a position e_n directly above B_t. In order to do this we must delete both B_e and B_w vertically. This means the points above and below these blocks must be empty so we know we have $e_{nw} and e_{ne}$ empty blocks which are adjacent to e_n. This means we did not have to delete B_t in the first place as we can walk from e_{nw} to e_n.

Thus there exists a solution which does not delete any sandwiched block from the direction it is sandwiched. For now assume the target location does not contain a sandwiched block. From this we can then construct the following graph,

- For each cell c not containing F, add vertices c_v, c_h and edges $c_v \to c, c_h \to c$. If c is empty, i.e., starts with e or A, add edges to it from each of its neighbors without a F.

- For each cell b with a block B_b "horizontally" surrounded by cells a and c: If both are empty, add edges $a \to b_h$ and $c \to b_h$. If a is a block and c is empty, add edges $a_v \to b_h$ and $c \to b_h$. If a is empty and c is a block, add edges $a \to b_h$ and $c_v \to b_h$. If both are blocks, or either of them is a wall, do nothing. Perform the same for vertical neighbors.

If there is a path in this graph to the target location then the agent may follow the path. If the path goes through a cell b which contains a block to start it must have gone through the node b_v or b_h. We know this block is not sandwiched. In order to enter this node we must have reached cell a which allows us to delete the block then enter that location. If there is a solution to the game we know there there exists a solution which does not delete any sandwiched block. Thus each possible block deletion and movement is represented by an edge in the graph. Finally it remains to handle the case where the target location is a sandwiched blocked. In order to do this we can replace the target location with a fixed wall then ask if we can reach both blocks which sandwich it. If those blocks are sandwiched we repeat this. We only would need to repeat this twice as if it was nested further the target location would be in the center a 3×3 box of blocks which cannot be reached.

□

6 Bendy Rules

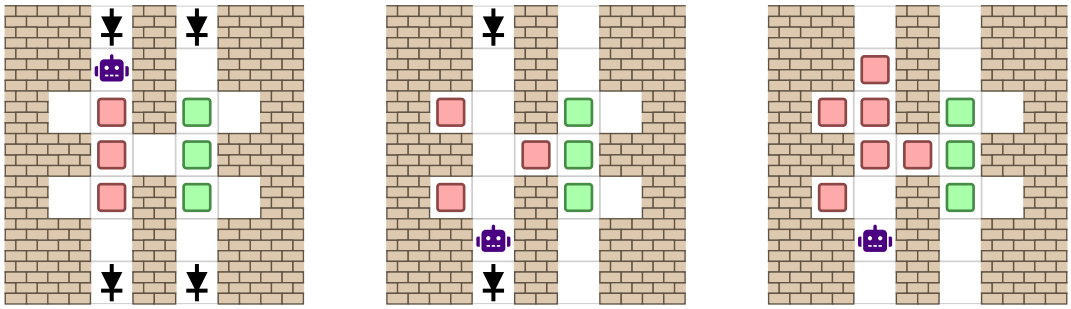
In this section, we consider length-3 bendy rules not considered in earlier sections. Recall that ***bendy rules*** are rules that can be applied in an L-shape. For example, for Push-1 ($ABe \to eAB$), allowing the rule to be bendy allows for an agent to push a block around a corner.

6.1 Other Conservative Bendy Results

Here, we obtain partial results for some conservative bendy rules not mentioned in earlier sections:

Theorem 39. *Bendy Push-1F (ABe → eAB) is NP-hard.*

Proof. We can construct an undirected NAND gadget; see Figure 24. The paths are the two columns with movable blocks. Traversing one path forces the other to close, as the middle movable block must be pushed into the middle empty square once one traversal has been completed. Undirected NAND gadgets are sufficient for NP-hardness. □

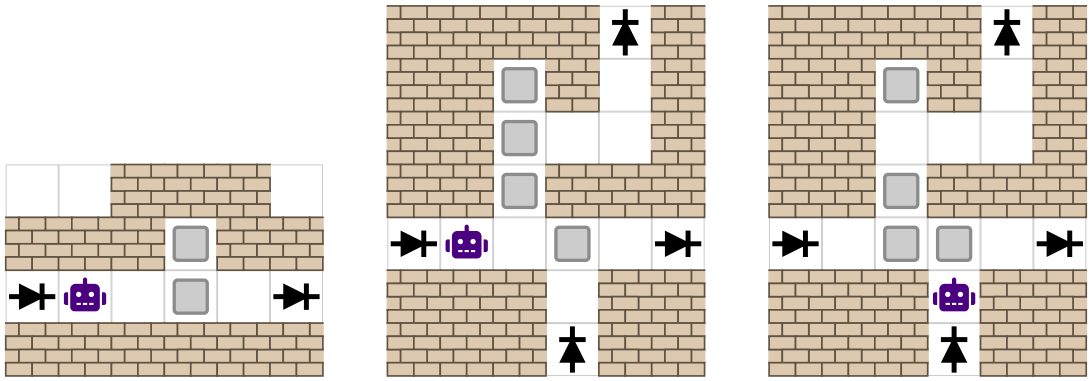


(a) Initial state **(b)** Bendy Push-1F traversal **(c)** Bendy Trail Push-F traversal

Figure 24: An undirected NAND gadget for Bendy Push-1F (*ABe* → *eAB*) and an undirected matched crumblers gadget for Bendy Trail Push-F (*ABe* → *BAB*).

Theorem 40. *Bendy Push Swap at Block-F (ABB → BAB) is NP-hard.*

Proof. We can construct a one-button door and an undirected crumbler; see Figure 25. For the one-button door, the traversal path is from the bottom entrance to the right exit, and the other path (which opens the traversal path) is from the left entrance to the top exit. One-button doors and undirected crumblers are sufficient for NP-hardness. □

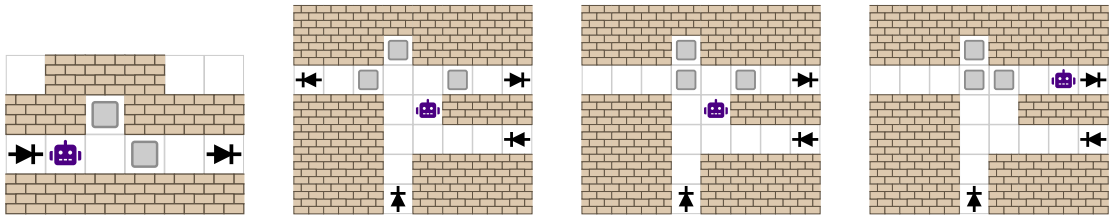


(a) **(b)** **(c)**

Figure 25: Gadgets for Bendy Push Swap at Block-F (*ABB* → *BAB*). (a) shows the undirected crumbler, and (b) and (c) show the one-button door when inactive and when active, respectively.

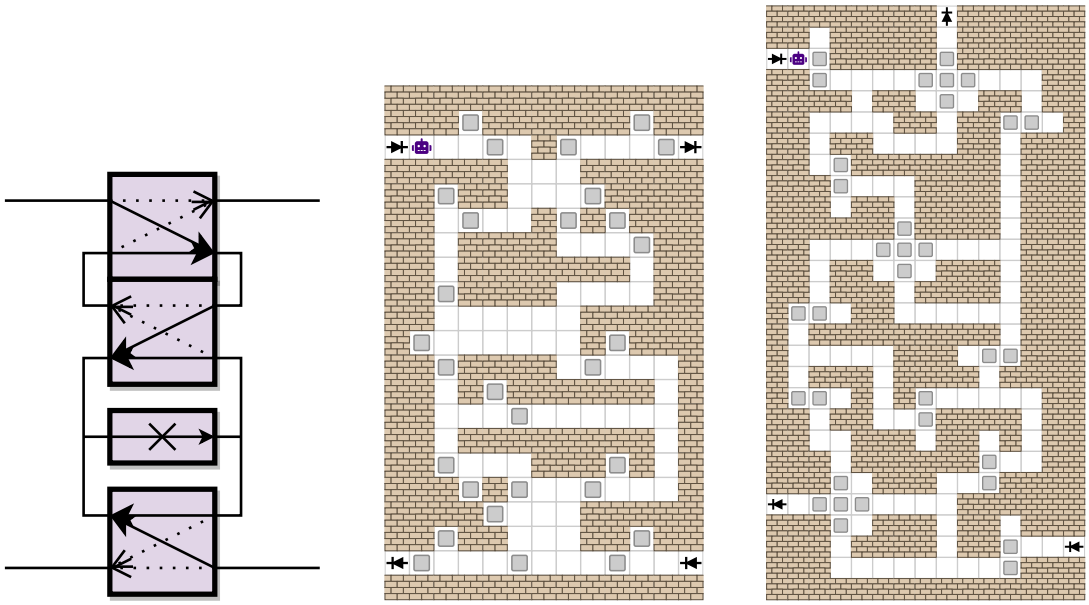
Theorem 41. *Bendy Pull Swap at Block-F (BAB → ABB) is NP-hard.*

Proof. We can construct an antiparallel matched dicrumbler using a directed crumbler and a "two-use gadget". More formally, by ***two-use gadget***, we mean a gadget that has two one-use entrances and two one-use exits (made one-use by directed crumblers), where one exit must always be used before the other. See Figure 26 for the two-use gadget for this rule (in this case, the left exit must be used before the right exit) and Figure 27 to see how the directed crumbler and two-use gadget combine to form an antiparallel matched dicrumbler. Antiparallel matched dicrumblers are sufficient for NP-hardness. □



(a) **(b)** First traversal **(c)** Second traversal **(d)** After second traversal

Figure 26: Gadgets for Bendy Pull Swap at Block-F ($BAB \longrightarrow ABB$). (a) and (b-d) show the directed crumbler and two-use gadget, respectively.



(a) **(b)** Bendy Pull Swap at Block-F **(c)** Bendy Trail Swap-F

Figure 27: This shows how a directed crumbler and a two-use gadget can be used to simulate an antiparallel matched dicrumbler, along with applications to Bendy Pull Swap at Block-F ($BAB \longrightarrow ABB$) and Bendy Trail Swap-F ($eAB \longrightarrow BBA$).

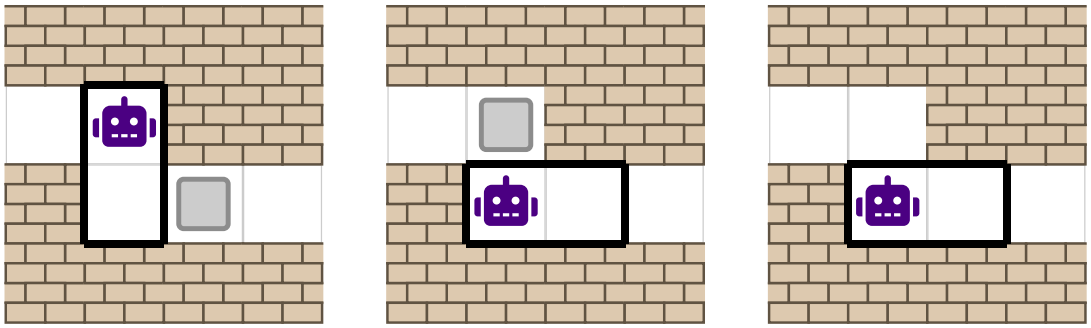
Here, we note a particularly surprising easiness result:

Theorem 42. *Bendy Suplex-1F ($AeB \longrightarrow BAe$ and congruent) is in L.*

Proof. The key observation is that if the agent has access to a 1×2 rectangle (i.e., if the agent is adjacent to an empty square at any given point), then the agent can swap this 1×2 rectangle, which contains the agent, with any movable block they can reach (pivoting around the middle square), as in Figure 28, so movable blocks become irrelevant. Thus, if the agent has no starting moves (i.e., there are no empty spaces around the agent), then the agent cannot go anywhere; otherwise the problem becomes planar undirected reachability, which is in L. □

6.2 Other Bounded Bendy Results

Here, we obtain some results (some partial, some full) for bounded bendy rules that are not mentioned in earlier sections:



(a) Before **(b)** After for Bendy Suplex-1F **(c)** After for Bendy Battering Ram-F

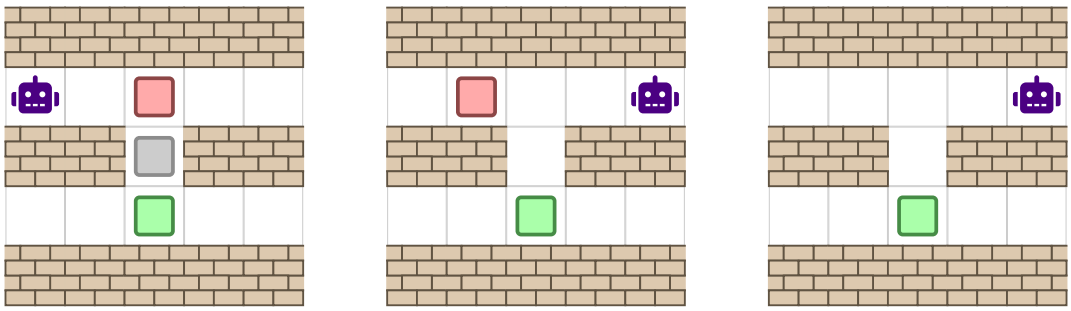
Figure 28: Illustration of the argument for Bendy Suplex-1F (*AeB* → *BAe*) and Bendy Battering Ram-F (*AeB* → *eAe*)

Theorem 43. *Bendy Trail Push-F (ABe → BAB) is NP-complete.*

Proof. We can construct an undirected matched crumblers gadget; see Figure 24. The paths are the two columns with movable blocks. Traversing one path forces the other to close, as the middle movable block must be pushed into the middle empty square once one traversal has been completed. Undirected matched crumblers gadgets are sufficient for NP-hardness; as the rule is bounded, we can conclude NP-completeness. □

Theorem 44. *Bendy Push Smash-F (ABB → Aee) and Bendy Swaplex-F (ABB → BAe) are NP-complete.*

Proof. We use the gadget in Figure 29, which serves as an undirected NAND gadget for Bendy Push Smash-F and an undirected matched crumbler for Bendy Swaplex-F. The paths are the two rows containing a single movable block with no fixed walls to the left or right of it. Traversing one path causes the middle (gray) movable block to disappear, which closes the other path (and the current path, in the case of Bendy Swaplex-F). Both of these are sufficient for NP-hardness; as these rules are bounded, we can conclude NP-completeness. □



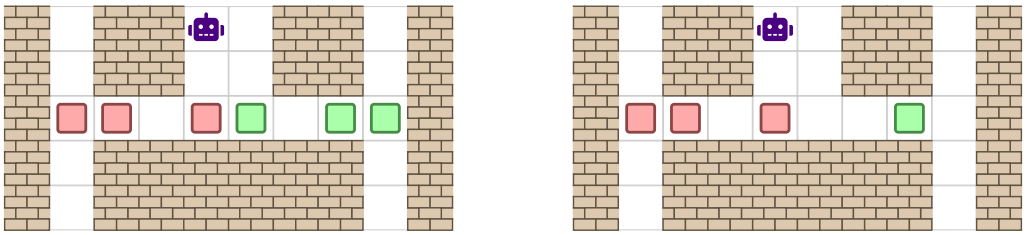
(a) Initial state **(b)** Bendy Swaplex-F traversal **(c)** Bendy Push Smash-F traversal

Figure 29: The gadget for Bendy Push Smash-F (*ABB* → *Aee*), serving as an undirected NAND, and Bendy Swaplex-F (*ABB* → *BAe*), serving as an undirected matched crumbler.

Theorem 45. *Bendy Pull Merge-F (ABB → ABe) is NP-complete.*

Proof. We construct a delayed NAND gadget (similar to the non-bendy version) in Figure 30. The button is the middle section of the gadget; the two traversal paths are the two columns containing exactly one movable block with no fixed walls above or below it. There are two ways to set the button, each of which

correspond to clearing one of the two movable blocks at the center of the gadget. As mentioned in Theorem 27, this is sufficient for NP-hardness; as this rule is bounded, we can conclude NP-completeness. □

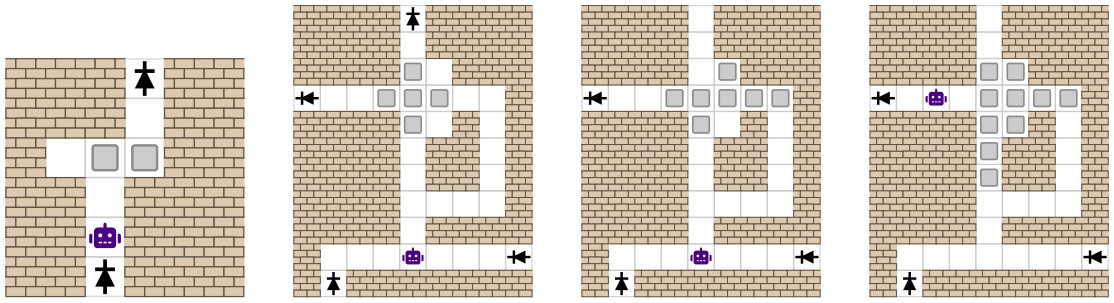


(a) Initial state **(b)** Setup state

Figure 30: A delayed NAND gadget for Bendy Pull Merge-F (*ABB* → *ABe*).

Theorem 46. *Bendy Trail Swap-F (eAB → BBA) is NP-complete.*

Proof. We can construct an antiparallel matched dicrumbler using a directed crumbler and a two-use gadget (refer to the Bendy Pull Swap at Block-F construction for a description of the two-use gadget). See Figure 31 for the two-use gadget (in this case, the top exit must be used before the left exit) and Figure 27 for how the directed crumbler and two-use gadget combine to form an antiparallel matched dicrumbler. Antiparallel matched dicrumblers are sufficient for NP-hardness; as this rule is bounded, we can conclude NP-completeness. □



(a) **(b)** First traversal **(c)** Second traversal **(d)** After second traversal

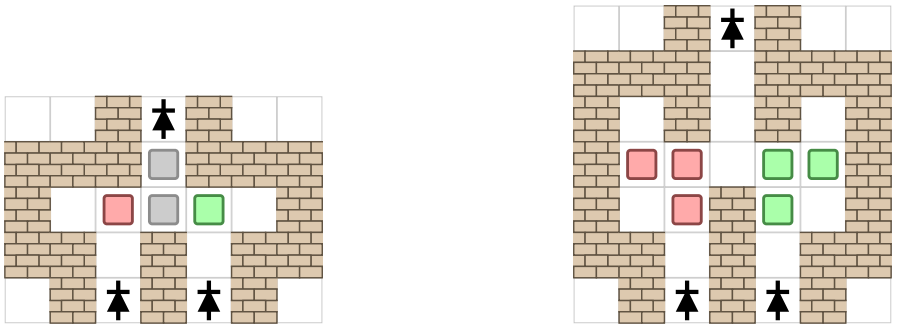
Figure 31: Gadgets for Bendy Trail Swap-F (*eAB* → *BBA*). (a) and (b-d) show the directed crumbler and two-use gadget, respectively.

Theorem 47. *Bendy Knock-Over-F (ABe → Aee) is PMCVP-hard in 2D and P-complete in 3D.*

Proof. We can construct an AND gadget and an OR gadget; see Figure 32. The AND gadget requires both the red and the green movable blocks to be cleared in order to traverse to the top port; the OR gadget only requires the movable blocks of one of the colors (either red or green) to be cleared in order to traverse to the top port. These are sufficient for showing PMCVP-hardness in 2D and P-completeness in 3D. □

Theorem 48. *Bendy Battering Ram-F (AeB → eAe) is in L.*

Proof. The key observation, similar to Bendy Suplex-1F, is that if the agent has access to a 1×2 rectangle (i.e., if the agent is adjacent to an empty square at any given point), then the agent can destroy any movable block they can reach, so movable blocks become irrelevant. See Figure 28 for an illustration. Thus, if the agent has no starting moves (i.e., there are no empty spaces around the agent), then the agent cannot go anywhere; otherwise the problem becomes planar undirected reachability, which is in L. □



(a) (b)

Figure 32: The AND and OR gadgets for Bendy Knock-Over-F ($ABe \to Aee$).

7 Future Directions

Many other variants of Push-1F and Pull?-1F have been introduced and studied. Some of them require further extenions to our block asynchronous cellular automata framework. For example, Pull!-1 [AAD+20] forces the agent to pull a block when it moves away from a block. We can model this by replacing the $Ae \to eA$ swap with the more restrictive $eAe \to Aee$ as the movement rule, in addition to the rule $eAB \to ABe$ representing the pull. It would be interesting to consider the other games with this movement rule. Pull-1W [AAD+20] allows thin 0×1 walls. Thin walls can be modeled as missing edges in the graph of cells. Games like Push-$*$, where the agent can push any number of blocks in front of it, can be modeled as an infinite set of game rules $AB^ie \to eAB^i$ for all $i > 0$. PushPull-1 is a game that would have two game rules, $\{ABe \to eAB, eAB \to ABe\}$, or as a single reversible rule $ABe \leftrightarrow eAB$. The puzzle game Sokoban can be modeled with the same rules as Push-1, expect that the goal is to reach a target configuration of blocks. It would be interesting to study the other puzzles we have introduced in these variants as well.

References

[AAD+20] Hayashi Ani, Sualeh Asif, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, Jayson Lynch, Sarah Scheffler, and Adam Suhl. PSPACE-completeness of pulling blocks to reach a goal. *Journal of Information Processing*, 28:929–941, 2020.

[ABC+23] Robert M. Alaniz, Josh Brunner, Michael Coulombe, Erik D. Demaine, Jenny Diomidova, Timothy Gomez, Elise Grizzell, Ryan Knobel, Jayson Lynch, Andrew Rodriguez, Robert Schweller, and Tim Wylie. Complexity of Reconfiguration in Surface Chemical Reaction Networks. In Ho-Lin Chen and Constantine G. Evans, editors, *Proceedings of the 29th International Conference on DNA Computing and Molecular Programming (DNA 29)*, volume 276 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:18, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[ACD+22] Hayashi Ani, Lily Chung, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Pushing blocks via checkable gadgets: PSPACE-completeness of Push-1F and Block/Box Dude. In *Proceedings of the 11th International Conference on Fun with Algorithms (FUN 2022)*, 2022.

[ACG+02] Len Adleman, Qi Cheng, Ashish Goel, Ming-Deh Huang, David Kempe, Pablo Moisset De Espanes, and Paul Wilhelm Karl Rothemund. Combinatorial optimization problems in self-

assembly. In *Proceedings of the 34th Annual ACM Symposium on Theory of Computing*, pages 23–32, 2002.

[ADG+21] Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Della H. Hendrickson, Adam Hesterberg, Matias Korman, Oliver Korten, Jayson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots. In *Proceedings of the 37th International Symposium on Computational Geometry*, 2021.

[BCG04] Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. What is Life? In *Winning Ways for Your Mathematical Plays*, volume 4. A K Peters, 2nd edition, 2004.

[Coo04] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1), 2004.

[DFM+17] Alberto Dennunzio, Enrico Formenti, Luca Manzoni, Giancarlo Mauri, and Antonio E. Porreca. Computational complexity of finite asynchronous cellular automata. *Theoretical Computer Science*, 664:131–143, 2017.

[DGLR18] 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)*, 2018.

[DHHL22] Erik D. Demaine, Robert A. Hearn, Dylan Hendrickson, and Jayson Lynch. PSPACE-completeness of reversible deterministic systems. In *Proceedings of the 9th Conference on Machines, Computations and Universality (MCU 2022)*, pages 91–108, Debrecen, Hungary, August–September 2022.

[DHL20] Erik D. Demaine, Della H. Hendrickson, and Jayson Lynch. Toward a general complexity theory of motion planning: Characterizing which gadgets make games hard. In *Proceedings of the 11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*, pages 62:1–62:42, 2020.

[DK21] Colin Defant and Noah Kravitz. Friends and strangers walking on graphs. *Combinatorial Theory*, 1, 2021.

[DL01] Jérôme Durand-Lose. Representing reversible cellular automata with reversible block cellular automata. *Discrete Mathematics & Theoretical Computer Science Proceedings*, AA: Discrete Models: Combinatorics, Computation, and Geometry (DM-CCG 2001), January 2001.

[DR18] Erik D. Demaine and Mikhail Rudoy. A simple proof that the (n^2-1)-puzzle is hard. *Theoretical Computer Science*, 732:80–84, 2018.

[Fat13] Nazim Fates. A guided tour of asynchronous cellular automata. In *Proceedings of the International Workshop on Cellular Automata and Discrete Complex Systems*, pages 15–30. Springer, 2013.

[FY13] Rudolf Fleischer and Jiajin Yu. A survey of the game "Lights Out!". In *Space-Efficient Data Structures, Streams, and Algorithms: Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday*, pages 176–198. Springer, 2013.

[GDC+10] Neil Gershenfeld, David Dalrymple, Kailiang Chen, Ara Knaian, Forrest Green, Erik D. Demaine, Scott Greenwald, and Peter Schmidt-Nielsen. Reconfigurable asynchronous logic automata. *ACM SIGPLAN Notices*, 45(1):1–6, 2010.

[GMMR21] Eric Goles, Diego Maldonado, Pedro Montealegre, and Martín Ríos-Wilson. On the complexity of asynchronous freezing cellular automata. *Information and Computation*, 281:104764, 2021.

[GOT15] Eric Goles, Nicolas Ollinger, and Guillaume Theyssier. Introducing freezing cellular automata. In *Proceedings of the 21st International Workshop on Cellular Automata and Discrete Complex Systems (AUTOMATA 2015)*, volume 24, pages 65–73, 2015.

[Gut91] Howard Gutowitz, editor. *Cellular Automata: Theory and Experiment.* MIT Press, 1991.

[Kar05] Jarkko Kari. Theory of cellular automata: A survey. *Theoretical Computer Science*, 334(1):3–33, 2005.

[Lan86] Christopher G. Langton. Studying artificial life with cellular automata. *Physica D: Nonlinear Phenomena*, 22(1-3):120–149, 1986.

[Mar84] Norman Margolus. Physics-like models of computation. *Physica D: Nonlinear Phenomena*, 10(1):81–95, 1984.

[Mil24] Aleksa Milojević. Connectivity of old and new models of friends-and-strangers graphs. *Advances in Applied Mathematics*, 155:102668, 2024.

[MIT24] MIT Hardness Group, Josh Brunner, Lily Chung, Erik D. Demaine, Jenny Diomidova, Della Hendrickson, and Jayson Lynch. Pushing blocks without fixed blocks via checkable gizmos: Push-1 is PSPACE-complete. Manuscript under submission, 2024.

[NW06] Turlough Neary and Damien Woods. P-completeness of cellular automaton Rule 110. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP 2006)*, pages 132–143, Venice, Italy, July 2006. Springer.

[OT22] Nicolas Ollinger and Guillaume Theyssier. Freezing, bounded-change and convergent cellular automata. *Discrete Mathematics & Theoretical Computer Science*, 24(Automata, Logic and Semantics), 2022.

[Pil19] Alexander Pilz. Planar 3-SAT with a clause/variable cycle. *Discrete Mathematics & Theoretical Computer Science*, 21(Discrete Algorithms), 2019.

[QW14] Lulu Qian and Erik Winfree. Parallel and scalable computation and spatial dynamics with DNA-based chemical reaction networks on a surface. In *Proceedings of the DNA Computing and Molecular Programming: 20th International Conference (DNA 20)*, volume 8727, page 114, Kyoto, Japan, September 2014. Springer.

[RW90] Daniel Ratner and Manfred Warmuth. The $(n^2 - 1)$-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10:111–137, 1990.

[Sut95] Klaus Sutner. On the computational complexity of finite cellular automata. *Journal of Computer and System Sciences*, 50(1):87–97, 1995.

[TH11] Tatsuie Tsukiji and Takeo Hagiwara. Recognizing the repeatable configurations of time-reversible generalized langton's ant is PSPACE-hard. *Algorithms*, 4(1):1–15, 2011.

[TLJ+17] Anupama J. Thubagere, Wei Li, Robert F. Johnson, Zibo Chen, Shayan Doroudi, Yae Lim Lee, Gregory Izatt, Sarah Wittman, Niranjan Srinivas, Damien Woods, et al. A cargo-sorting DNA robot. *Science*, 357(6356):eaan6558, 2017.

[TM87] Tommaso Toffoli and Norman Margolus. Ii.12: The Margolus neighborhood. In *Cellular Automata Machines: A New Environment for Modeling*, pages 119–138. MIT Press, 1987.

[vN66] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[Win98] Erik Winfree. *Algorithmic self-assembly of DNA*. PhD thesis, California Institute of Technology, 1998.