On the Computational Complexity of Portal and Other Games

Erik Demaine* Isaac Grosof* Jayson Lynch*

January 7, 2016

Abstract

This paper proves that push-pull block puzzles in 3D are PSpace-Complete to solve and push-pull block puzzles in 2D with thin walls are NP-Hard to solve. Push-pull block puzzles are a recreational motion planning problem, similar to Sokoban, that involves moving a 'robot' on a grid with obstacles. The obstacles cannot be traversed by the robot, but some can be pushed or pulled by the robot into adjacent squares. Thin walls prevent movement between two adjacent squares. This work follows in a long line of algorithms and complexity work on similar problems [Wil91] [DDO00] [Hof00] [DHH04] [DH01] [DO92] [DHH02] [Cul98] [DZ96] [Rit10]. The 2D push-pull block puzzle shows up in The Legend of Zelda giving another proof of hardness for the game [ADGV14]. This variant of block pushing puzzles is of particular interest to the authors because it is fully reversible, meaning the inverse of any action that has been taken is valid.

1 Introduction

Block pushing puzzles are a common puzzle type with perhaps the best known example being Sokoban. They have found their way into several popular video games including The Legend of Zelda, Pokemon, and Tomb Raider. Block pushing puzzles are a recreational embodiment of motion planning problems with movable obstacles. Since motion planning is such an important and computationally difficult problem, it can be useful to look at simplified models to try to get a better understanding of the larger problem. The push-pull model itself also doesn't seem entirely abstract, as one could imagine the constraints of a forklift in a warehouse bearing similarity.

A significant amount of research has gone into characterizing the complexity of block sliding puzzles. This includes PSPACE-completeness for well-known puzzles like sliding-block puzzles [HD05], Sokoban [Cul98, DZ96], the 15-puzzle [RW86], and Rush Hour [FB02]. Block pushing puzzles are a type of block sliding puzzle in which the blocks are moved by a small robot within the puzzles. This type of block sliding puzzle has gathered a significant amount of study. Variations include Sokoban [Cul98, DZ96], where blocks must reach specific targets, variations where multiple blocks can be pushed [DDO00, Hof00, DHH02], versions where blocks continue to slide after being pushed [DHH04, Hof00], versions where the robot can pull blocks [Rit10], and versions where the blocks are subject to gravity [Fri]. The problem of motion planning in an environment where blocks may be pushed and pulled is modeled in a general form in Gordon Wilfong's Motion Planning in the Presence of Movable Obstacles [Wil91]. There he shows a polynomial time algorithm

*MIT Computer Science and Artificial Intelligence Laboratory, 32 Vas- sar Street, Cambridge, MA 02139, USA

Name	*Push*	*Pull*	*Blocks*	*Fixed?*	*Path?*	*Sliding*	*Complexity*
Push-k	k	0	Unit	No	Path	min	NP-hard [DDO00]
Push-$*$	$*$	0	Unit	No	Path	min	NP-hard [Hof00]
PushPush-k	k	0	Unit	No	Path	Max	PSPACE-c. [DHH04]
PushPush-$*$	$*$	0	Unit	No	Path	Max	NP-hard [Hof00]
Push-kX	k	0	Unit	No	No-Cross	min	NP-c. [DH01]
Push-$*$X	$*$	0	Unit	No	No-Cross	min	NP-c. [DH01]
Push-1F	1	0	Unit	Yes	Path	min	NP-hard [DO92]
Push-kF	$k \geq 2$	0	Unit	Yes	Path	min	PSPACE-c. [DHH02]
Push-$*$F	$*$	0	Unit	Yes	Path	min	PSPACE-c. [DHH02]
Sokoban	1	0	Unit	Yes	Storage	min	PSPACE-c. [Cul98]
Sokoban$^+$	$k \geq 2$	1	2x1	Yes	Storage	min	PSPACE-c. [DZ96]
Motion Planning	k	1	L	Yes	Storage	min	NP-hard [Wil91]
Sokoban$(k, 1)$	$k \geq 5$	1	Unit	Yes	Storage	min	NP-hard [DZ96]
Pull-1	0	1	Unit	No	Storage	min	NP-hard [Rit10]
Pull-kF	0	k	Unit	Yes	Storage	min	NP-hard [Rit10]
PullPull-kF	0	k	Unit	Yes	Storage	Max	NP-hard [Rit10]
Push-1G[1]	1	0	Unit	Yes	Path	min	NP-hard [Fri]
Push-k Pull-lW	k	l	Unit	Wall	Path	min	**NP-hard** (§2.1)
3D Push-k Pull-lF	k	l	Unit	Yes	Path	min	**NP-hard** (§2.3)
3D Push-1 Pull-1W	1	1	Unit	Wall	Path	min	**PSPACE-c.** (§2.4)
3D Push-k Pull-kF	$k > 1$	$k > 1$	Unit	Yes	Path	min	**PSPACE-c.** (§2.4)

Table 1: Summary of Past and New Block Pushing Puzzle Results. $*$ refers to an unlimited number of blocks. F means fixed blocks are included. X means the robot cannot step on the same square more than once. G means the blocks are subject to gravity. W means thin walls are included. k and l are positive integers.

for motion planning with one movable object, NP-hardness for the general planning problem, and PSPACE-hardness for the storage problem. Figure 1 gives a summary of results on block pushing puzzles.

We add several new results showing that certain block pushing puzzles, which include the ability to push and pull blocks, are NP-hard or PSPACE-complete. We introduce *thin walls*, which prevent motion between two adjacent empty squares. We prove that all path planning problems in 2D with thin walls or in 3D, in which the robot can push k blocks and pull l blocks for all $k, l \in \mathbb{Z}^+$ are NP-hard. Our results are shown in the last four lines of Table 1. To prove these results we introduce two new abstract gadgets, the set-verify and the 4-toggle, and prove hardness results for questions about the legal state transitions of these gadgets.

2 Push-Pull Block Puzzles

In this section we prove several results about push-pull block puzzles. We show Push-1 Pull-1 in 3D with thin walls is PSPACE-complete2.4; Push-i Pull-j in 3D is PSPACE-complete for all positive integers $i, j \geq 22.4$; Push-k Pull-l in 3D is NP-hard for all positive integers $k, l2.3$; and Push-q Pull-r in 2D with thin walls is NP-hard for all positive integers $q, r2.1$. A summary of these results

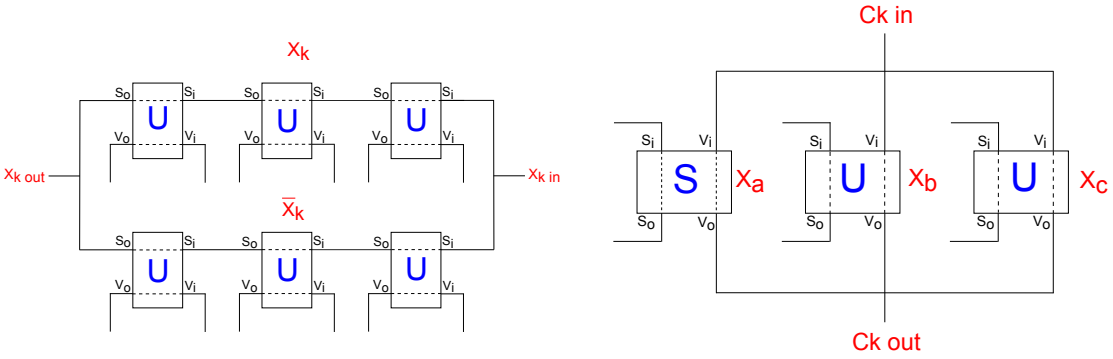
can be seen in Table 1.

2.1 2D Push-Pull with Thin Walls

In this section we prove that Push-k Pull-l in 2D with fixed blocks is NP-hard, for all positive k and l, if we include *thin walls*. Thin walls are a new, but natural, notion for block pushing puzzles, which prevent blocks or the robot from passing between two adjacent, empty squares, as though there were a thin wall blocking the path. These were needed because many of the gadgets depended greatly on having very tight corridors to ensure limited behavior. We will prove hardness by a reduction from Planar 3SAT.

Theorem 2.1. *Push-k Pull-l in 2D with thin walls is NP-hard.*

2.2 3SAT Construction



(a) A variable gadget representing X_k occurring in three clauses.

(b) A clause gadget with variables x_a, x_b, x_c.

Figure 1

We reduce from 3SAT and will be making use of the Set-Verify gadget to produce our literals. One significant difficulty with this model is the complete reversibility of all actions. Thus we need to take care to ensure that going backward at any point does not allow the robot to cheat in solving our 3SAT instance. The directional properties of the Set-Verify allow us to create sections where we know if the robot exits, it must have either reset everything to the initial configuration or put everything in another known state.

Our literals will be represented by Set-Verify gadgets, described in Section 2.2.1. They are considered true when the V_i to V_o traversal is possible, and false otherwise. Thus we can set literals to true by allowing the robot to run through the S_i to S_o passage of the gadget. This allows a simple clause gadget, shown in Figure 1b, consisting of splitting the path into three hallways, each with the corresponding verify side of our literal. We can then pass through if any of the literals is set to true and cannot pass otherwise.

The variables will be encoded by a series of passages which split to allow either the true or negated literals to be set, shown in Figure 1a. Once variables split into two hallways, we have to show that when you enter or exit a side of a hallway, all the literals are set to true at one end and false at the

other. Also, we cannot go back through the other hallway, so if we go backward, everything gets reset. Thus we cannot exploit the reversibility to set anything extra true.

The variables and clauses must be joined together to allow the robot to move between the gadgets in the correct order. We can connect these with simple, empty hallways, except in the cases where these hallways must cross. Although we reduce from Planar 3SAT, it is not known whether the problem remains hard if we further connect all of the variable nodes and then the edge nodes in a single path, as is required in our construction, while still ensuring that the graph is planar.

Therefore, in order to traverse all of the clauses and reach the goal location, the robot must traverse a set of variable hallways that encode an accepting assignment to the 3SAT problem, thus reducing 3SAT to 2D Push-Pull block puzzles.

2.2.1 Set-Verify Gadgets

There are four possible states of the Set-Verify gadget: Broken, Unset, Set, and Verified. These are depicted in Figure 2. In the Broken state, the only possible transition is $S_o \to S_i$, changing the state to Unset. This state can only be reached from Unset, and allows strictly less future transitions than Set, which can also be reached from Unset, so we will disregard it. In the Unset state, the $S_i \to S_o$ transition is the only possibility, changing the state to Set. In the Set state, the $S_o \to S_i$ transition is possible, changing the state back to Unset, as well as the $V_i \to V_o$ transition, which changes the state to Verified. Finally, from the Verified state, the only transitions possible are $V_o \to V_i$, changing the state back to Set, and $V_i \to V_o$, leaving the state as Verified.

For the Set-Verify gadget in the Unset state, the S_i entrance is the only one which allows the robot to move any blocks. From the S_i entrance they can traverse to S_o, and they can also pull the middle block down behind them. Doing so will allow a traversal from V_i to V_o. To traverse back from S_o to S_i, the robot must first traverse back from V_o to V_i. Then, when the robot travels back from S_o to S_i, they must push the middle block back, ensuring the V_i to V_o traversal is impossible. Further, access to any sequence of entrances will not allow the robot to alter the system to allow traversals between the V_i and S_i entrances.

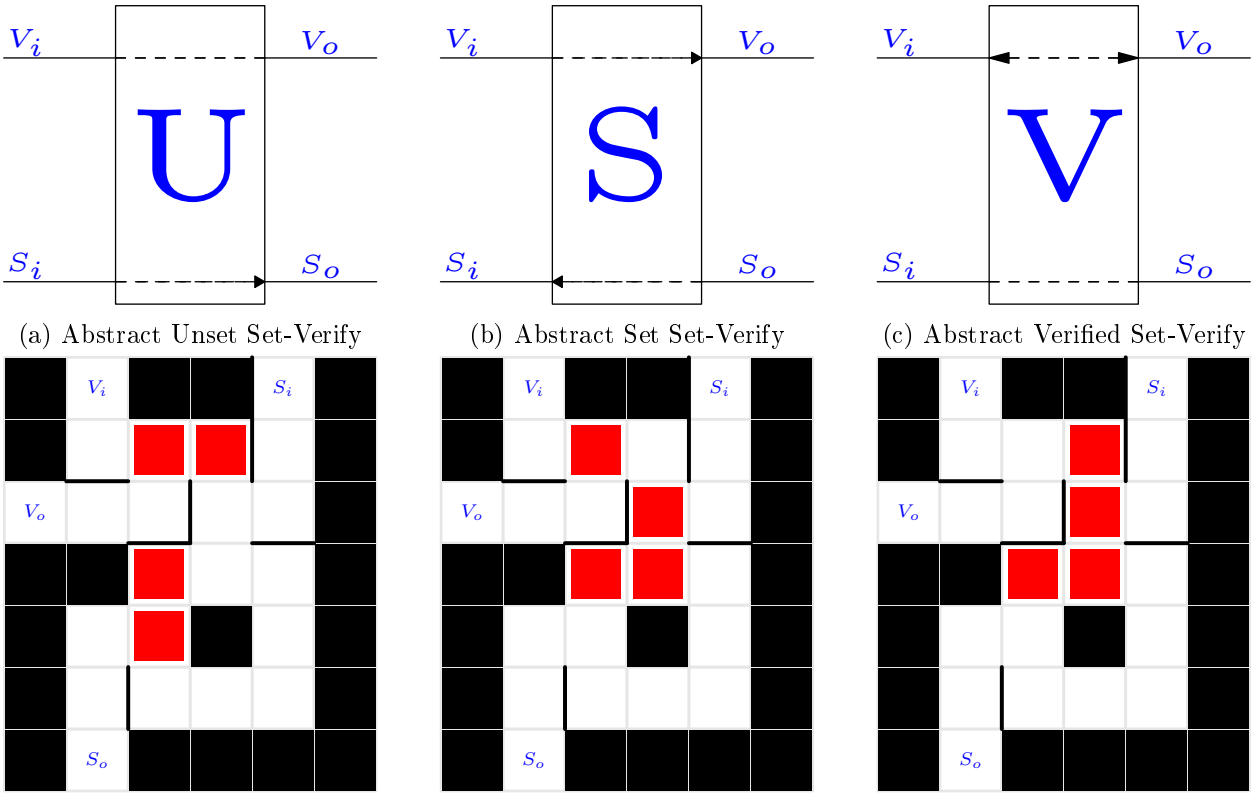
Since the Set-Verify gadget has no hallways with length greater than 3, any capabilities the robot may have of pushing or pulling more than one block at a time are irrelevant. Thus, the following proof will apply for all positive values of j and k in Push-j Pull-k.

2.2.2 Crossover Gadgets

In this section we build up the needed one use crossover gadget from a series of weaker types of crossover gadgets.

Directed Destructive Crossover This gadget, depicted in Figure 3a, allows either a traversal from a to a' or b to b'. Once a traversal has occurred, that path may be traversed in reverse, but the other is impassable unless the original traversal is undone.

First, observe that transitions are initially only possible via the a and b entrances, since the transitions possible through a Set-Verify in the Set state can be entered through V_i and S_o, not S_i.

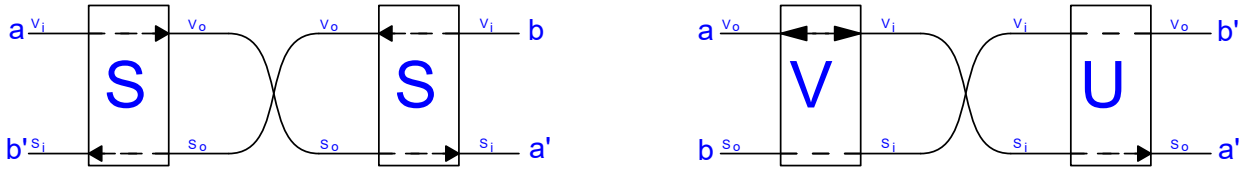


(a) Abstract Unset Set-Verify (b) Abstract Set Set-Verify (c) Abstract Verified Set-Verify

(d) Set-Verify, unset state (e) Set-Verify, set state (f) Set-Verify, verified state

Figure 2: Set-Verify Gadgets

Assume without loss of generality that the gadget is entered at a. This changes the state of the left Set-Verify to Unset. At this point, only the right V_i and left S_i transitions are passable. Taking the S_i transition reverts all changes to the original state. Taking the V_i transition changes the right Set-Verify to Verified, and completes the crossover. At this point, the only possible transition is to undo the transition just made, from a' back to a, restoring the original state. Thus, the only transition possibilities are as stated above.

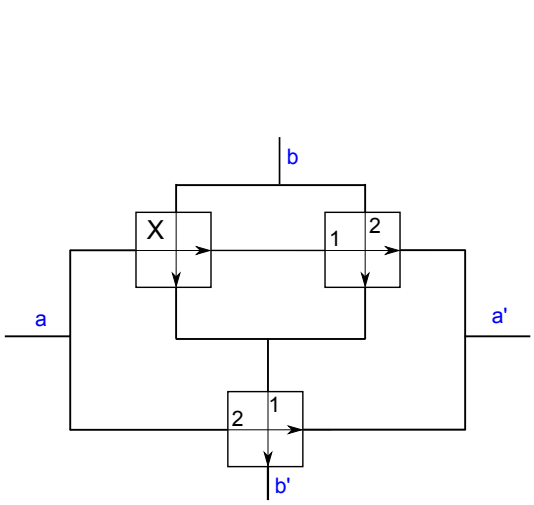


(a) The one-way destructive crossover constructed from two connected Set-Verify gadgets initialized in the set position.

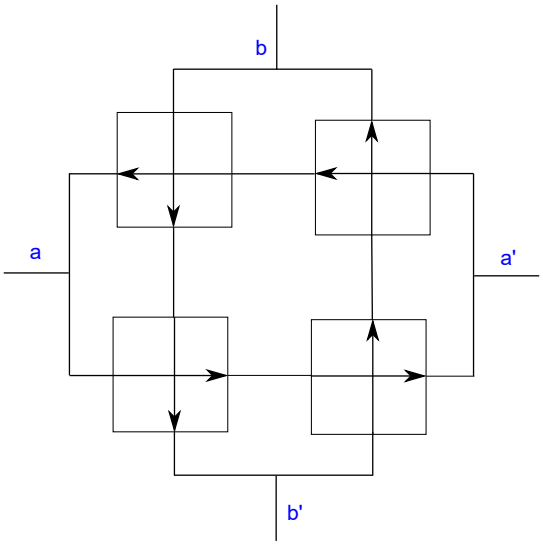
(b) The in-order one-way crossover constructed from two connected Set-Verify gadgets initialized in the verified and unset positions.

Figure 3: Two types of crossover gadgets

In-order Directed Destructive Crossover This gadget, depicted in Figure 3b allows a traversal from a to a', followed by a traversal from b to b'.



(a) The one use one-way crossover is constructed from a one-way destructive crossover and two in-order one-way crossovers.



(b) A full one use crossover constructed from four directed one use crossovers.

Figure 4: Composite crossover gadgets

Initially, no entrance is passable except for a, since V_o is passable only in the Verified state, and S_o is passable only in the Set state. Once the left $V_o \to V_i$ transition is made, the robot has 2 options. It can either change the left Set-Verify gadget's state to Set, or leave it as Verified. In either case, the S_i entrance on that toggle is impassable, since a S_i entrance may only be traversed in the Unset state. The only transition possible on the right crossover is $S_i \to S_o$, changing the state from Unset to Set. This completes the first crossing.

Now, there are at most 2 transitions possible: from a' back to a, undoing the whole process, or entering at b. Note that entering at b is only possible if the left Set-Verify is in the Set state, so let us assume that state change occurred. In that case, the left $S_o \to S_i$ transition may be performed, changing the left Set-Verify's state to Unset. At that point, the only possible transitions are back to b, or through the right Set-Verify's $V_i \to V_o$ transition, completing the second crossover.

One Use Directed Crossover The One Use Directed Crossover, depicted in Figure 4a, is the gadget needed for our proof. It allows a traversal from a to a' followed by a traversal from b to b', or from b to b' and then a to a'.

It is constructed out of an In-order Directed Crossover gadget and a Destructive Directed Crossover, as shown in Figure 4a. The a to a' traversal is initially passable, and goes through both gadgets, blocking the destructive crossover but leaving the in-order crossover open for the b to b' traversal. If the a to a' traversal does not occur, the b to b' traversal is possible via the destructive crossover.

One Use Crossover Four Directed Crossovers can be combined, as shown below, to create a crossover that can be traversed in any direction. This is not necessary for our proof but is shown for general interest. Unfortunately, the inability to go through this gadget multiple times in the same direction without first going back through means it likely isn't suitable for a PSPACE-completeness

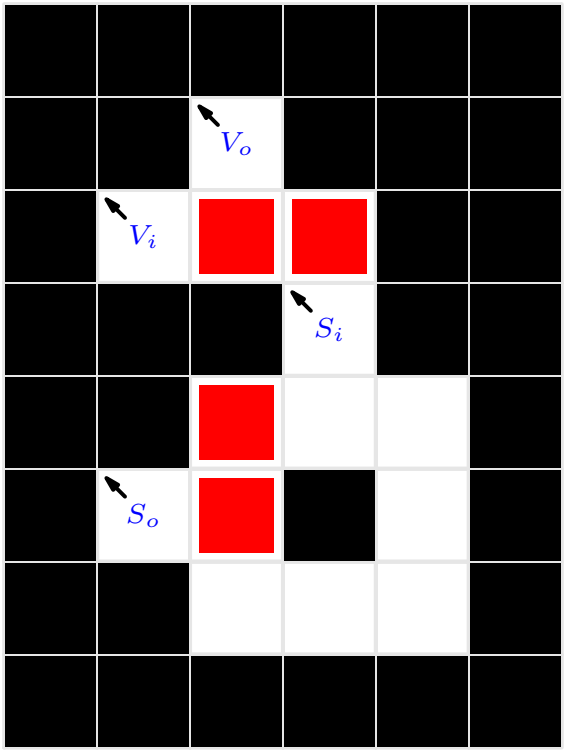


Figure 5: A Set-Verify gadget where the entrances and exits extend upward, notated by the diagonal arrows. This gadget is in the unset state.

reduction.

2.3 3D Push-Pull is NP-hard

In this section we prove that 3D Push-k Pull-l with fixed blocks is NP-hard, for all positive k and l. All of the hard work was done in the previous section. Here we will simply show how we can use the additional dimension to tweak the previous gadgets to build them without thin walls. We reduce from 3SAT, constructing our variables from chains of 3D Set-Verify gadgets, and our clauses from the verify side of the corresponding 3D Set-Verify gadget.

Theorem 2.2. *3D Push-k Pull-l with fixed blocks is NP-hard, for all positive k and l.*

Proof. We follow the proof of Theorem 2.1 using a modified Set-Verify gadget, shown in Figure 5. As before, in the unset state the only possible traversal is S_i to S_0. This traversal allows the top right bock to be pulled down, moving the gadget into the set state. From here the V to V_0 traversal is possible, as well as going back through the S_0 to S pathway. However, the S to S_0 traversal is not possible. For more detail refer to Section 2.2.1. We do note that the cyclic ordering of the entrances in the 3D Set-Verify is different from that of the 2D Set-Verify, however this is not important as we no longer need to construct crossovers.

Variables are composed of hallways of 3D Set-Verify gadgets connected S_0 to S, one for each clause in which the variable appears, as in Figure 1a. Clauses are composed of three 3D Set-Verify gadgets connected in parallel as in Figure 1b. The details of these constructions follow those in Section 2.2

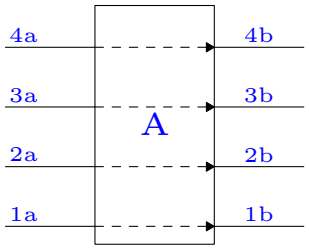
This completes the reduction from 3SAT. In addition, we note that all blocks are in hallways of length at most 3, thus the gadgets still function as described for any positive push and pull values. □

2.4 PSPACE

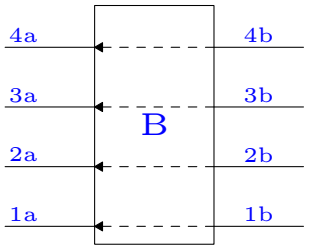
In this section we show the PSPACE-completeness of 3D push-pull puzzles with equal push and pull strength. We introduce a gadget called the 4-toggle and use it to simulate Quantified Boolean Formulas [GJ79]. We construct the 4-toggle gadget in 3D push-pull block puzzles, completing the reduction. In particular we prove 3D Push1-Pull1 with thin walls is PSPACE-complete and 3D Push$i$-Pull$j$, for all positive $i = j$, is PSPACE-complete. A gap between NP and PSPACE still remains for 3D puzzles with different pull and push values, as well as 2D puzzles.

2.4.1 Toggles

We define an n-toggle to be a gadget which has n internal pathways and can be in one of two internal states, A or B. Each pathway has a side labeled A and another labeled B. When the toggle is in the A state, the pathways can only be traversed from A to B and similarly in the B state they can only be traversed from B to A. Whenever a pathway is traversed, the state of the toggle flips.



(a) 4-Toggle in state A. Here we can only follow the paths from left to right.

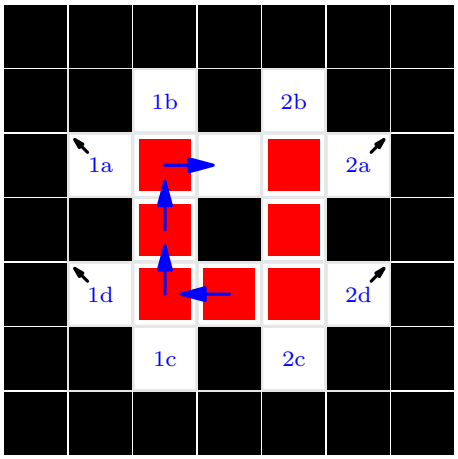


(b) 4-Toggle in state B. The direction of the paths through the toggle are flipped.

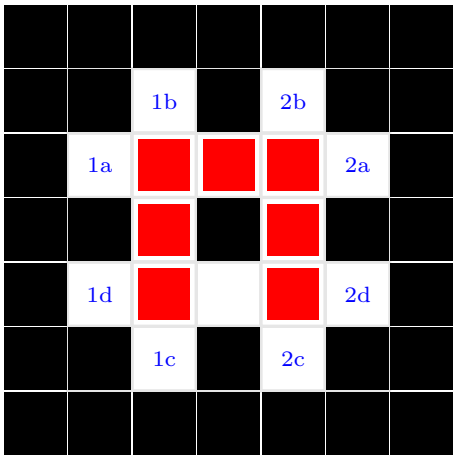
Figure 6

Figure 7a acts as a 2-toggle. The locations $1a$, $1d$, $2a$, and $2d$, are all entrances and exits to the 2-toggle, while $1b$ connects directly to $1c$, and $2b$ connects directly to $2c$. Notice that there is a single block missing from the ring of eight blocks. When the missing block is on top, as diagrammed, it will represent state A, and when it is on the opposite side, we call it state B. Notice that in state A, it is impossible to enter through entries $1d$ or $2d$. When we enter in the $1a$ or $2a$ sides, we can follow the moves in the series of diagrams to exit the corresponding $1b$ or $2b$ side, leaving the gadget in the B state. One can easily check that the gadget can only be left in either state A, B, or a broken state as seen in Figure 7c. Notice that in the broken state, every pathway except the one just exited is blocked. If we enter through that path, it is in exactly the same state as if it had been in an allowed state and entered through the corresponding pathway normally. For example, in the diagram one can only enter through $1b$ and after doing so it is the same as entering in path $1b$ on a 2-toggle in state B. Thus the broken state is never more useful for solving the puzzle and can be safely ignored.

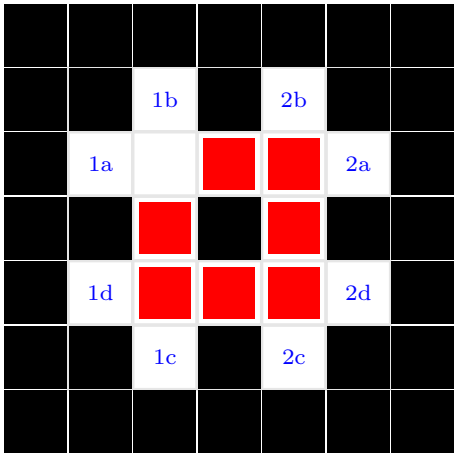
To construct a 4-toggle we essentially take two copies of the 2-toggle, rotate them perpendicular to



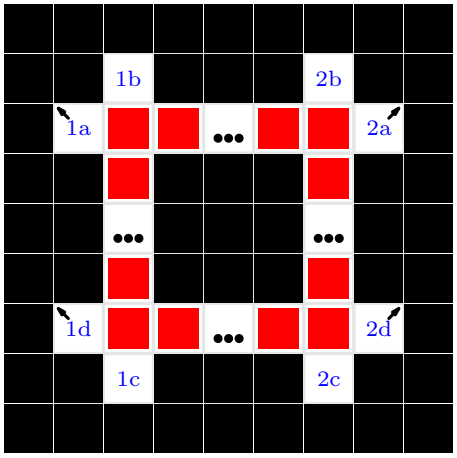
(a) 2-Toggle in state A. The arrows indicate the transition to state B.



(b) 2-Toggle in state B.



(c) 2-Toggle in one of four broken states.



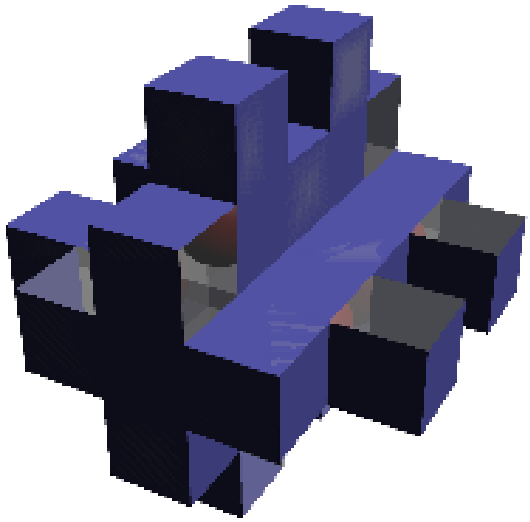
(d) Construction of a 2-toggle when the robot can push or pull multiple blocks.

Figure 7: 2-Toggles constructed in a push-pull block puzzle.

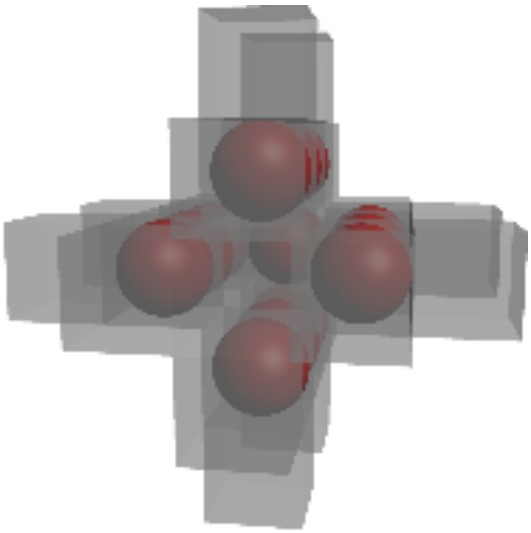
each other in 3D, and let them overlap on the central axis. See Figure 9a. We still interpret the lack of blocks in the same positions as in the 2-toggle as states A or B. Now we have four different paths which function the same as the ones described above. Similar arguments show the broken states of the 4-toggle also don't matter.

2.5 Locks

A lock is a gadget consisting of a 2-toggle and a separate pathway. Traversing the separate pathway can only be done in a single direction, which is dependent on whether the 2-toggle is in state A or B, and the traversal does not change the internal state of the 2-toggle. The 2-toggle functions exactly as described above. This gadget can be implemented using a 4-toggle by connecting the $3B$ and $4B$ entrances of the 4-toggle with an additional corridor, as shown in 9b. Traversing the resultant full pathway, from $3A$ to $3B$ to $4B$ to $4A$, is possible only if the initial state of the 4-toggle is A, and will leave the 4-toggle in state A. In addition, a partial traversal, such as from $3A$ to $3B$ and

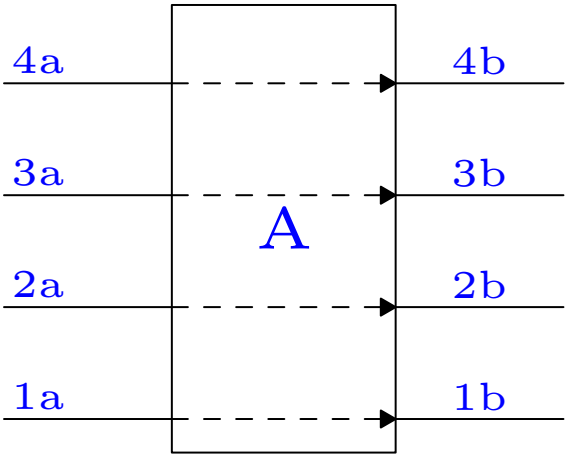


(a) Diagram of a 4-toggle showing impassible surfaces.

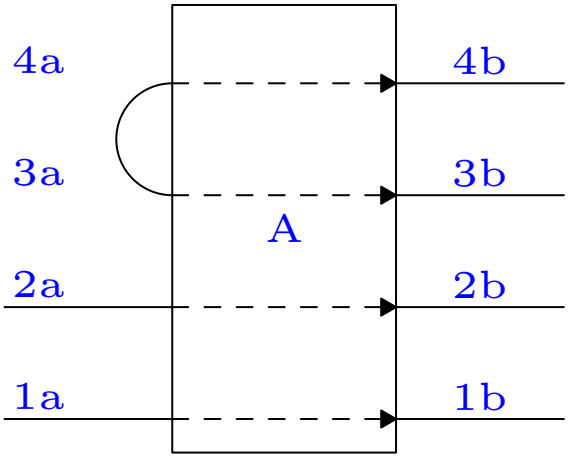


(b) Diagram of the internals of a 4-toggle.

Figure 8



(a) Representation of a 4-toggle in state A.



(b) Diagram of a lock. The $1a$ to $2a$ traversal is only possible in state A and returns the toggle to state A.

Figure 9

back to $3A$, does not change the internal state. The two unaffected pathways of the toggle, 1 and 2, continue to function as a 2-toggle.

A synchronized lock block is a gadget consisting of a 2-toggle and any number of separate pathways. As in the lock, the 2-toggle functions as described above. Each pathway may be set up to be passable in either direction if the toggle is in state A and impassable in either direction if the toggle is in state B (type A), or passable in either direction if the toggle is in state B and impassable in either direction if the toggle is in state A (type B). This is implemented using one lock per non-toggling pathway needed. As shown in Figure 10, there are 2 pathways of the entire synchronized lock block

system: the 1 pathway, and the 2 pathway. A lock whose pathway will create a type A synchronized lock pathway is placed so that its pathways are in the same direction as the synchronized lock block's pathways, A entrance in the A direction and B entrance in the B direction. Its state matches the synchronized lock block's state. A type B lock is placed in the opposite direction, with its B entrance in the synchronized lock block's A direction and A entrance in the B direction. Its state is also opposite the synchronized lock block's state.

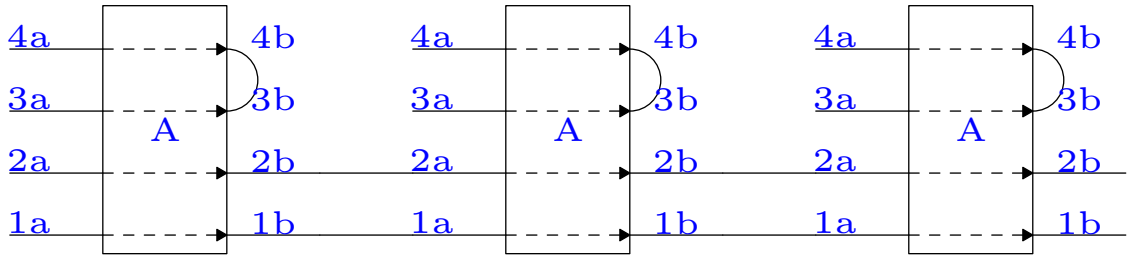


Figure 10: A synchronized lock block chain.

When the synchronized lock block is traversed, all of the internal locks' states flip, rendering the synchronized lock block passable in the opposite direction, and switching the passability and impassibility of all of the external pathways.

2.5.1 Binary Counter

Universal quantifiers must iterate through all possible combinations of values that they can take. In this section we construct a gadget that runs though all the states of its subcomponents as the robot progresses through the gadget. This construction will serve as the base for our universal quantifiers.

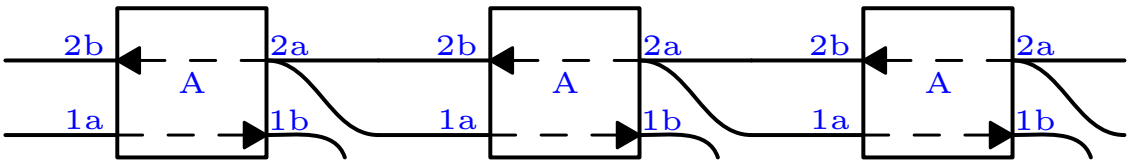


Figure 11: The central portion of a three bit binary counter made from 2-toggles.

We now define a binary counter. The binary counter has a fixed number of internal bits. Whenever the binary counter is traversed in the forwards direction, the binary number formed by the internal bits increases by one and the robot leaves via one of the exits. If the binary counter is traversed in the reverse direction, the internal value is reduced by one. If the binary counter is partial traversed, but then the robot leaves via its initial entrance, the internal value does not change.

The binary counter is implemented as a series of 2-toggles, as shown in Figure 11. The entrance pathway is connected to the 2-toggle's $1A$ and $2B$ entrances. The $1B$ exit from the 2-toggle will exit from the entire binary counter. The $2A$ exit will continue on to the next 2-toggle, attaching to that toggle's $1A$ and $2B$ entrances. This will continue for every toggle down the line, except that the last toggle's $2A$ exit signals an overflow and exits from the counter.

To see that this produces the desired effect, identify a toggle in state A as a 0 bit, and a toggle in state B as a 1 bit. Let the entrance toggle's bit be the least significant bit, and the final toggle be the most significant. When the robot enters the binary counter in the forwards direction, it will

flip the state of every toggle it passes through. When it enters a toggle that is initially in state B, and thus whose bit is 1, it will flip the state/bit and proceed to the next toggle, via the $2B-2A$ pathway. When it encounters a toggle that is initial in state A / bit 0, it will flip the state/bit and exit via the $1A-1B$ pathway. Thus, the overall effect on the bits of the binary counter is to change a sequence of bits ending at the least significant bit from 01..11 to 10..00. This has the effect of increasing the value of the binary counter by one.

If the robot approaches this apparatus from the exit side, there are three possibilities. If the robot attempts to enter via a $1B$ pathway whose toggle is in state A / 0, the toggle is impassable and the robot makes no progress. If the robot enters via a $1B$ pathway whose toggle is in state B / 1, after traversal the robot will have 2 options: To return in the direction it came, or to continue through the next toggle's $2A-2B$ pathway. The latter is only possible if the next toggle (the one just less significant than the entrance) is in state A / 0. Upon traversing that toggle, the robot will again be able to return from where it came, or progress in the same fashion if the next toggle is in state A / 0. This will continue until the robot either turns back, or exits out the main entrance.

Thus, if and only if the robot enters via the least significant toggle which is set to 1, the robot will be able to leave out the main entrance. If the robot enters any other way, it will be forced to return via the path it entered by, and undo all changes it has made.

The transformation on the bits caused by the only possible successful reverse traversal is to change 10..00 to 01..11, resulting in a decrement operation, as desired.

2.6 Existential Quantifiers

We now define an existential gadget. An existential gadget is like a synchronized lock block, except that instead of a 2-toggle, it has a single pathway which is always passable in both directions, and upon traversing the pathway the robot may or may not change the internal state of the synchronized lock block, as it chooses. The variable is considered true if the 4-toggles in the lock block are in state A and false if they are in state B.

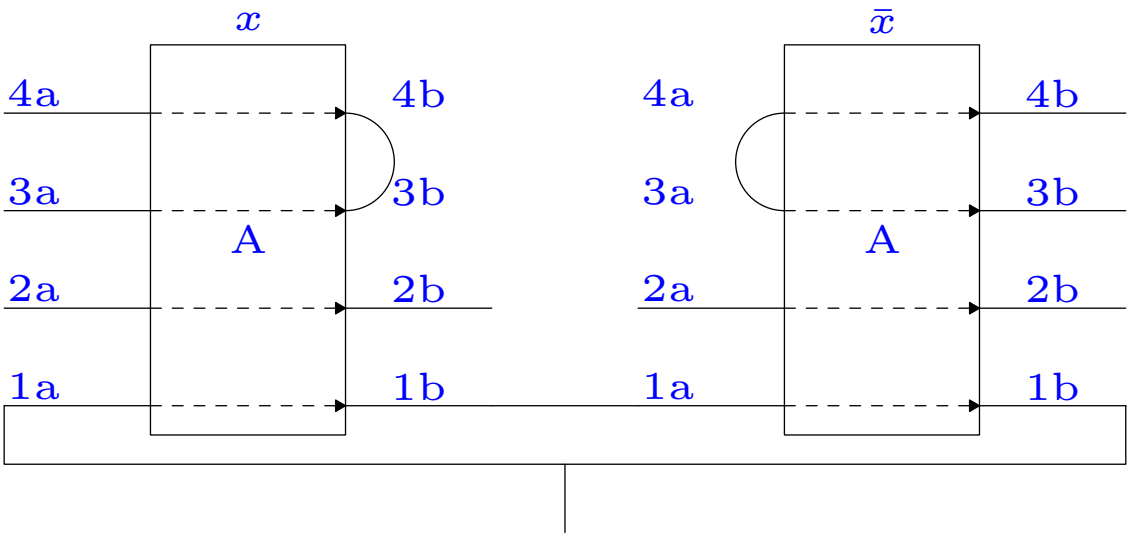


Figure 12: An existential gadget is simply a lock block with a number of copies equal to the number of times the corresponding variable occurs in the formula.

As shown in Figure 12, an existential gadget consists of a synchronized lock block and a pathway with access to all four pathways of the 2-toggle component of the synchronized lock block. Upon traversing the main pathway, the robot may choose to traverse the 2-toggle any number of times, leaving it in either state, as desired.

2.6.1 Quantifier Chain

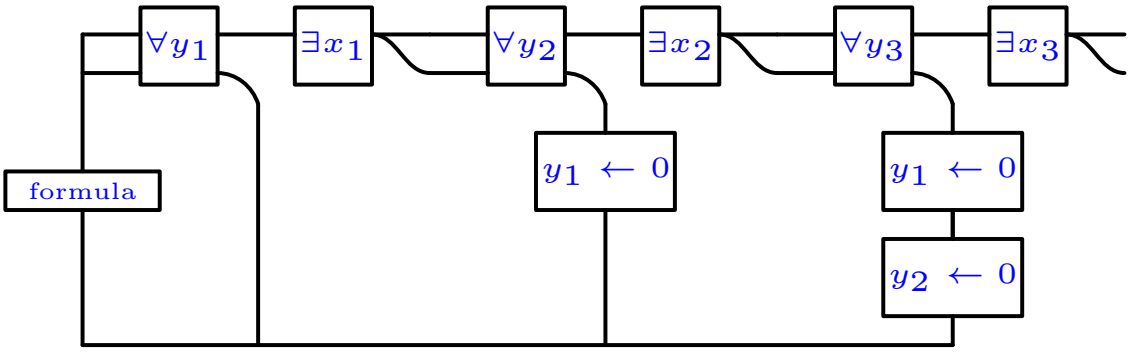


Figure 13: A segment of the quantifier chain. Each lock is actually a synchronized lock chain with length equal to the number of times the corresponding variable appears in the formula.

We now define a quantifier chain, as shown in Figure 13. A quantifier chain implements a series of alternating existential and universal variables, as well as external literal pathways, which may be traversed if and only if their corresponding variables are set to a pre-specified value.

Traversing the quantifier chain repeatedly in the primary direction will cycle the universal variables through all 2^n possible settings. Upon each traversal, an initial sequence of the universal variables will have their values flipped. During the traversal, the robot will have the option to set a series of corresponding existential variables to whatever value they wish. These comprise the existentials nested within the universal variables whose values were flipped.

Traversing the quantifier chain in the reverse direction is only possible if the robot enters via the lowest order universal toggle whose setting is 1. The traversal will go back one setting in the sequence of possible settings of the universal variables, and allow the robot to set all existential variables corresponding to altered universal variables arbitrarily. No other existential variables can be changed.

There is also a special exit, the overflow exit, which can only be reached after all of the universal variable settings have been traversed. This is the target location for the robot.

A quantifier chain is implemented much like a binary counter, with some additions. Every universal variable will be represented by a synchronized lock block, where each individual lock will serve as a literal. The 2-toggles which are governing the progression through the synchronized lock blocks are hooked up in the same manner as the 2-toggles in a binary counter gadget. This forces the synchronized lock blocks to be set to the corresponding values in the simulated binary counter.

The next addition is the existential variables, which consist of existential gadgets placed just after the $2A$ exits of each universal variable, and just before the $1A$ and $2B$ entrances of the the next universal variable, as shown in Figure 13.

One potential flaw in the apparatus as described so far is that a robot could enter via an exit

corresponding to a highly significant universal variable set to 1, alter its paired existential variable, and then leave via the original entrance. This must not happen for the existential counter to work properly, so a series of lock-chain pathways are added to each of the exits from the quantifier. Recall from the description of the binary counter that the only possible reverse traversal of the counter should be via entering at the lowest significance variable set to 1. To prevent entry at higher-significance variables set to 1, we will add a series of locks to that exit which are only passable if all lower-significance universal variables are set to 0.

This prevents the undesired high-significance existential alteration problem mentioned above, as it is now impossible to enter the gadget via the higher-significance universal variables which are set to 1, since that entryway will have at least 1 closed lock on it and thus be impassable. However, this addition does not affect the desired forward or backward transition, because the entrance/exit in question will have all of its synchronized lock block external pathways in the passable state.

2.6.2 Clause Gadget

We construct a clause gadget by putting the lock pathways of the three 4-toggles in the lock chains representing the corresponding variables in parallel, as we did with Set-Verify gadgets in Figure 1b. Each of these paths can be traversed only if the corresponding variable has been set. Since they are in parallel, only one needs to be passable for the robot to be able to continue on to the next clause.

2.6.3 Beginning and End Conditions

The overall progression of the robot through the puzzle starts with the quantifier chain. The robot increments the universal variables and sets the appropriate existential variables arbitrarily, then traverses the formula gadget to verify that the formula is true under that setting. The process then repeats.

At the beginning of this procedure, the robot must be allowed to set all of the existential variables arbitrarily. To ensure this, we will set up the quantifier gadget in the state 01..11, with all variables set to 1 except the highest order one. The highest order variable will be special, and will not be used in the $3CNF$ formula. The initial position of the robot will be at the entrance to the quantifier gadget. This will allow the robot to flip every universal in the quantifier gadget, from 01..11 to 10..00, and accordingly set every existential variable arbitrarily. To force the robot to go forward through the quantifier gadget instead of going backwards through the clause chain, we will add a literal onto the end of the formula gadget which is passable if and only if the highest order variable is set to 1.

After this set up, the robot will progress through the loop consisting of the quantifier gadget and the formula gadget, demonstrating the appropriate existential settings for each assignment of the universal quantifiers.

After progressing through every possible state of the universal quantifiers, the universals will be in the state 11..11. At this point, the robot may progress through the quantifier gadget and exit via its special pathway, the carry pathway of the highest order bit. This special pathway will lead to the goal location of the puzzle. Thus, only by traversing the quantifier - formula loop repeatedly, and demonstrating the solution to the TQBF problem, will the robot be able to reach the goal. The robot may reach the goal if and only if the corresponding quantified boolean formula is true.

3 Conclusion

In this paper we proved a few hardness results about variations of block pushing puzzles which also allow the robot to pull blocks. Along the way we analyzed the complexity of two new, simple gadgets adding useful new tools with which to attack future problems. The results themselves are obviously of interest to game and puzzle enthusiasts, but we also hope the analysis leads to a better understanding of motion planning problems more generally and the techniques we developed allow us to better understand the complexity of problems.

This work leads to many open questions to pursue in future research. For Push-Pull block puzzles, we leave a number of NP to PSPACE gaps, as is common for many other variations of the problem. One would hope to directly improve upon the results here to show tight hardness results for 2D and 3D push-pull block puzzles. One might also wonder if the gadgets used, or the introduction of thin walls might lead to stronger results for other block pushing puzzles. We also leave open the question of push-pull block puzzles without fixed walls. Notice that a single 3×3 block of clear space allows the robot to reach any point, making gadget creation challenging.

There are also interesting questions with regard to the abstract gadgets used in the proof. Are 2-toggles or 3-toggles sufficient to prove NP-hardness or PSPACE-hardness? Can one construct crossovers out of toggles? Are Set-Verify gadgets sufficient for PSPACE-hardness? Is this model of abstraction useful for capturing more agent-based games and puzzles? Finally, one would hope to use these techniques to show hardness for other problems.

References

[ACJ+10] David Arthur, Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach. The complexity of flood filling games. In Paolo Boldi and Luisa Gargano, editors, *Fun with Algorithms*, volume 6099 of *Lecture Notes in Computer Science*, pages 307–318. Springer Berlin Heidelberg, 2010.

[ADGV14] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (NP-)hard. In *Proceedings of the 7th International Conference on Fun with Algorithms (FUN 2014)*, Lipari Island, Italy, July 1–3 2014.

[BDD+02] Therese C. Biedl, Erik D. Demaine, Martin L. Demaine, Rudolf Fleischer, Lars Jacobsen, and J. Ian Munro. The complexity of Clickomania. In R. J. Nowakowski, editor, *More Games of No Chance*, pages 389–404. Cambridge University Press, 2002. Collection of papers from the MSRI Combinatorial Game Theory Research Workshop, Berkeley, California, July 24–28, 2000.

[ben14] bennyscube. YouTube channel. `https://www.youtube.com/user/bennyscube/featured`, May 2014.

[Col13] Andrew M Colman. *Game theory and its applications: in the social and biological sciences*. Psychology Press, 2013.

[Con01] John H. Conway. *On numbers and games (2nd ed.)*. A K Peters, 2001.

[Cor04] G. Cormode. The hardness of the Lemmings game, or oh no, more NP-completeness proofs. In *Proceedings of Third International Conference on Fun with Algorithms*, pages 65–76, 2004.

[Cul98] J. C. Culberson. Sokoban is PSPACE-complete. In *Proceedings International Conference on Fun with Algorithms (FUN98)*, pages 65–76, Waterloo, Ontario, Canada, June 1998. Carleton Scientific.

[DDO00] Erik D. Demaine, Martin L. Demaine, and Joseph O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *Proceedings of the 12th Annual Canadian Conference on Computational Geometry (CCCG 2000)*, pages 211–219, Fredericton, New Brunswick, Canada, August 16–18 2000.

[DH01] Erik D. Demaine and Michael Hoffmann. Pushing blocks is NP-complete for noncrossing solution paths. In *Proceedings of the 13th Canadian Conference on Computational Geometry (CCCG 2001)*, pages 65–68, Waterloo, Ontario, Canada, August 13–15 2001.

[DH09] Erik D. Demaine and Robert A. Hearn. Playing games with algorithms: Algorithmic combinatorial game theory. In Michael H. Albert and Richard J. Nowakowski, editors, *Games of No Chance 3*, volume 56 of *Mathematical Sciences Research Institute Publications*, pages 3–56. Cambridge University Press, 2009.

[DHH02] Erik D. Demaine, Robert A. Hearn, and Michael Hoffmann. Push-2-F is PSPACE-complete'. In *Proceedings of the 14th Canadian Conference on Computational Geometry (CCCG 2002)*, pages 31–35, Lethbridge, Alberta, Canada, August 12–14 2002.

[DHH04] Erik D. Demaine, Michael Hoffmann, and Markus Holzer. PushPush-k is PSPACE-complete. In *Proceedings of the 3rd International Conference on Fun with Algorithms (FUN 2004)*, pages 159–170, Isola d'Elba, Italy, May 26–28 2004.

[DHLN03] Erik D. Demaine, Susan Hohenberger, and David Liben-Nowell. Tetris is hard, even to approximate. In *Proceedings of the 9th International Computing and Combinatorics Conference (COCOON 2003)*, pages 351–363, Big Sky, Montana, July 25–28 2003.

[DO92] A. Dhagat and J. O'Rourke. Motion planning amidst movable square blocks. In *Proceedings of the 4th Canadian Conference on Computational Geometry (CCCG 1992)*, 1992.

[dVV03] Sven de Vries and Rakesh V. Vohra. Combinatorial auctions: A survey. *INFORMS J. on Computing*, 15(3):284–309, July 2003.

[DZ96] D. Dor and U. Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4), 1996.

[FB02] Gary William Flake and Eric B. Baum. Rush Hour is PSPACE-complete, or why you should generously tip parking lot attendants. *Theoretical Computer Science*, 270(1-2):895 – 911, 2002.

[For10] Michal Forisek. Computational complexity of two-dimensional platform games. In *Proceedings International Conference on Fun with Algorithms (FUN 2010)*, pages 214–227, 2010.

[Fri] Erich Friedman. Pushing blocks in gravity is NP-hard. http://www2.stetson.edu/~efriedma/papers/gravity.pdf.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[Gru39] Patrick M Grundy. Mathematics and games. *Eureka*, 2(6-8):21, 1939.

[HD05] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1):72–96, 2005.

[HD09] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation*. A. K. Peters, Ltd., Natick, MA, USA, 2009.

[Hof00] M. Hoffman. Push-* is NP-hard. In *Proceedings of the 12th Canadian Conference on Computational Geometry (CCCG 2000)*, Lethbridge, Alberta, Canada, 2000.

[Joh12] Nathaniel Johnston. The complexity of the puzzles of Final Fantasy XIII-2. arXiv:1203.1633, 2012.

[Kay00] Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[KPS08] Graham Kendall, Andrew J. Parkes, and Kristian Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

[Pap01] Christos H. Papadimitriou. Algorithms, games, and the internet. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, STOC '01, pages 749–753, New York, NY, USA, 2001.

[Ren] Paul Rendell. A Turing machine in Conway's game life. http://www.cs.unibo.it/~babaoglu/courses/cas00-01/papers/Cellular_Automata/Turing-Machine-Life.pdf.

[Rit10] Marcus Ritt. Motion planning with pull moves. arXiv:1008.2952, 2010.

[RW86] Daniel Ratner and Manfred K Warmuth. Finding a shortest solution for the n× n extension of the 15-PUZZLE is intractable. In *AAAI*, pages 168–172, 1986.

[Sch78] Thomas J. Schaefer. On the complexity of some two-person perfect-information games. *Journal of Computer and System Sciences*, 16(2):185–225, 1978.

[Spr35] Richard Sprague. Über mathematische Kampfspiele. *Tôhoku Math. J*, 41:438–444, 1935.

[Vig12] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! arXiv:1201.4995, 2012.

[Vig15] Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.

[Wal14] Toby Walsh. Candy Crush is NP-hard. arXiv:1403.1911, 2014.

[Wil91] Gordon Wilfong. Motion planning in the presence of movable obstacles. *Annals of Mathematics and Artificial Intelligence*, 3(1):131–150, 1991.