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 game, and 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 wall prevent movement between two adjacent squares. This work follows in a long line of algorithms and complexity work on similar problems [21] [5] [16] [9] [8] [10] [7] [3] [11] [19]. The 2D push-pull block puzzle shows up in a number of video games, including The Legend of Zelda, thus implying a number of other results. 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

This work builds on a long line of work on the mathematics of games and puzzles, primarily, the recent work in the computational complexity of video games. We also give a new result showing that block pushing puzzles, like Sokaban, which also include the ability to pull blocks are NP-Hard. This result also proves a number of other games, which have embedded push-pull block puzzles, are NP-Hard.

Games have and continue to be a source of interesting problems for mathematics and computer science. Game playing continues to be a benchmark in the field of AI. The study of economic games has become a large field with applications in online auctions [4], network analysis [17], and voting [1]. Combinatorial games have lead to algebraic insights such as Conway's Surreal Numbers [2] and the Sprague-Grundy Theorem [20] [14]. More recently, games have been studied from an algorithmic perspective and can be seen as models of computation in examples like Constraint Logic [15] and Conway's Game of Life [18]. For a survey of algorithmic combinatorial game theory see Demaine and Hearn's paper [6]. Understanding the computational and algorithmic aspects of games may lead to better understanding of algorithms and problems that share similar properties. For example, certain asymmetries or algebraic properties may be more intuitively obvious in games, such as the relationship between two-player games and quantified boolean formulas []. In this vein, we are particularly interested in the push-pull block model because it is fully reversible, meaning any sequence of moves in the puzzle can be undone. Reversibility is fundamentally linked to quantum computation and the thermodynamics of computation; thus understanding the difference between reversible and irreversible computation is very relevant.

Finally, many games and puzzles can also be seen as simplified models of problems in the real world. Thus, solving these problems gives us partial understanding and tools to attack more difficult questions. For example, push-pull block puzzles can be seen as a simplified model of robotic forklifts operating in a warehouse. Complexity results in these very simplified models help us understand what aspects of problems makes them hard and allows us to start differentiating between the complexity that arises from the combinatorics vs the geometry of path planning problems. Similarly, path-planning in dynamic graphs is a complicated problem which occurs every day from traffic to packet routing.

1.1 Sliding Blocks Related Work

A significant amount of research has gone into characterizing the complexity of sliding block puzzles. This includes PSpace-Completeness for well known puzzles like Sokaban [3] and Rush Hour [12]. We are specifically interested in block pushing puzzles, which involve a 'robot' that is able to move on a grid and is able to move blocks adjacent to it, typically by 'pushing' or 'pulling' the block. Further, we only address the path planning problem, in which the robot wants to get from a given location A to a target location B, rather than the storage problem in which the movable blocks must reach some final configuration. Figure 1 gives a summary of results on block pushing puzzles.

The problem of motion planning in an environment where blocks may be pushed and pulled is modeled in a very general form in Gordon Wilfong's Motion Planning in the Presence of Movable Obstacles, where he shows a polynomial time algorithm for motion planning with one movable object, NP-Hardness for the general planning problem, and PSpace-Hardness for the storage problem. [21] His problem, however, deals with continuous motion and polygonal walls and blocks; making the model significantly different from the 1x1 blocks on a grid that is considered for most block pushing puzzles. Marcus Ritt paper [19] addresses the model in which blocks can only be pulled. We primarily build off of Dor and Zwick's work which generalizes Sokoban to include block pulling. They show the block storage problem which includes 2x1 blocks which can be pushed or pulled is PSpace-Complete, and the unit-size block storage where the robot can push at five or more blocks at a time and pull one is NP-Hard [11]. We introduce *thin walls*, which prevent motion between two adjacent empty squares. We prove that all path planning problemsn 2D with thin wall or in 3Ds, in which the robot can push k blocks and pull l blocks for all $k, l \in \mathbb{Z}^+$ are NP-Hard. As with many of these problems, closing the gap between NP and PSpace remains open.

[check against Erik's notes for completeness] xxx

[might be more updating, check 6.890] xxx

2 Push-Pull Block Results [rough draft - real diagrams and better arguments for gadget safety pending]

In this section we prove that Push-1 Pull-1 with fixed blocks is NP-Hard if we include *thin walls*. Thin walls are a new, but natural, notion for pushing block 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 also note that being able to push or pull more blocks does not impact the gadgets, thus proving hardness for Push-k Pull-l for all k and l.
We will prove hardness by a reduction to 3SAT. Clauses can be formed by splitting hallways, each with a check that the corresponding variable was set true. Our literals are composed of the Set-Verify gadget listed below. A variable is a split hallway with one side going to all the true literals and the other going to all of the false literals. Program flow, and the fact that going backwards removes the pass-ability from the Set-Verify gadgets ensures variables are only set to be true or false.

Name	*Push*	*Pull*	*Block Size*	*Fixed?*	*Path?*	*Sliding*	*Complexity*
Push-k	k	0	Unit	No	Path	min	NP-Hard [5]
Push-*	*	0	Unit	No	Path	min	NP-Hard [16]
PushPush-k	k	0	Unit	No	Path	Max	PSpace-Comp. [9]
PushPush-*	*	0	Unit	No	Path	Max	NP-Hard [16]
PushPushPush-k	k	0	Unit	No	Path	Max	PSpace-Comp. []
PushPushPush-*	*	0	Unit	No	Path	Max	NP-Hard []
Push-kX	k	0	Unit	No	No-Cross	min	NP-Comp. [8]
Push-*X	*	0	Unit	No	No-Cross	min	NP-Comp. [8]
Push-1F	1	0	Unit	Yes	Path	min	NP-Hard [10]
Push-kF	k≥ 2	0	Unit	Yes	Path	min	PSpace-Comp. [7]
Push-*F	*	0	Unit	Yes	Path	min	PSpace-Comp. [7]
Sokoban	1	0	Unit	Yes	Storage	min	PSpace-Comp. [3]
Sokoban$^+$	k≥ 2	1	2x1	Yes	Storage	min	PSpace-Comp. [11]
	k[1]	1	Polygon	Yes	Storage	min	NP-Hard [21]
Sokoban(k,1)	k≥ 5	1	Unit	Yes	Storage	min	NP-Hard [11]
Pull-1	0	1	Unit	No	Storage	min	NP-Hard [19]
Pull-kF	0	k	Unit	Yes	Storage	min	NP-Hard [19]
PullPull-kF	0	k	Unit	Yes	Storage	Max	NP-Hard [19]
Push-1G[2]	1	0	Unit	Path	min	Yes	NP-Hard [13]
PushPull-kW	j	k	Unit	Wall	Path	min	**NP-Hard**
3DPushPull-kF	j	k	Unit	Yes	Path	min	**PSpace-Comp.**

Table 1: Summary of Block Pushing Puzzle Results

2.1 Reversible One-Way Gadget

When in the initial configuration A the gadget only permits a traversal from s_0 to s leaving the gadget in configuration B. Configuration B only permits traversals in the opposite direction, from s to s_0. Note, when the robot leave this gadget, the only possible configurations are A and B. These gadgets or similar structures will be used several times within more complicated structures.

2.2 Set-Verify Gadgets

For the Set-Verify gadget, the S entrance is the only one which allows the robot to move any blocks. From the S entrance they can traverse to S_0, and they can also pull the middle block down behind them. Doing so will allow a traversal from V to V_0. To traverse back from S_0 to S, the robot must first traverse back from V_0 to V. Then, when the robot travels back from S_0 to S, they must push the middle block back, ensuring the V to V_0 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 and S entrances.

There are three possible states of the Set-Verify: Unset, Set, and Verified. In the Unset state, the $S \to S_0$ transition is the only possibility, changing the state to Set. In the Set state, the $S_0 \to S$ transition is possible, changing the state back to Unset, as well as the $V \to V_0$ transition, which

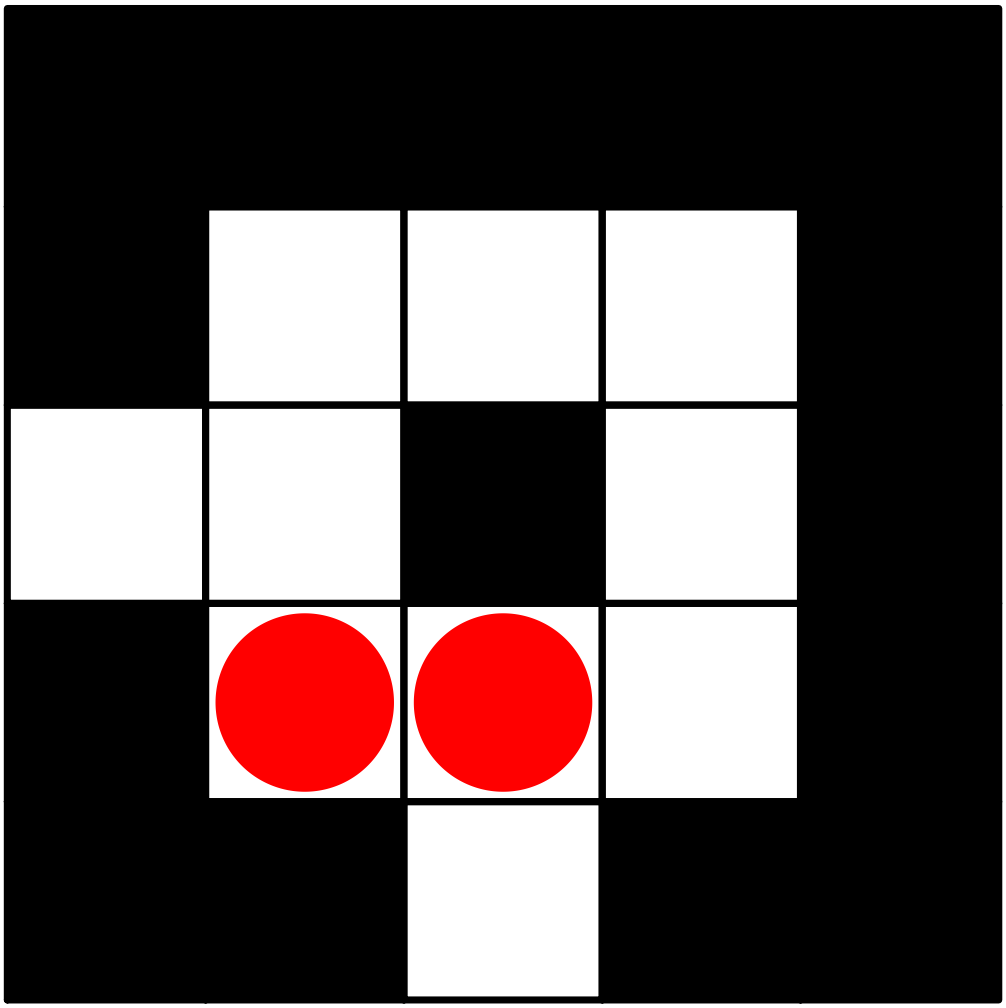


Figure 1: Reversible one-way gadget.

changes the state to Verified. Finally, from the Verified state, the only transitions possible are $V_0 \to V$, changing the state back to Set, and $V \to V_0$, leaving the state as Verify.

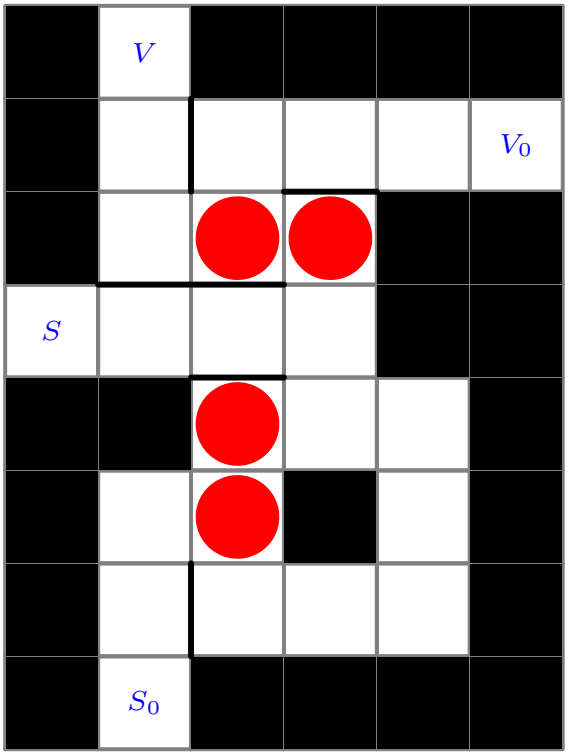
2.3 Directed Destructive Crossover

Initially this gadget allows either a traversal from a to a' or b to b'. Once a traversal has occurred, that path may be traversed freely, 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 state Set can be entered through V and S_0, not S.

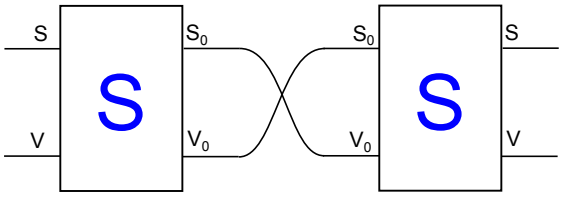
Assume without loss of generality that the gadget is entered at a. This changes the state of the left Set-Verify to Unset. At this point, the right V transition is passable, as well as the left S transition, and no others. Taking the S transition reverts all changes to the original state. Taking the V 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.

Figure 2: Set-Verify Gadget

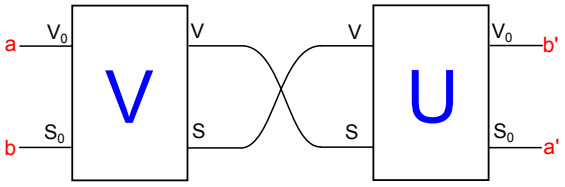


(a) 2D Set-Verify with walls.

Figure 3: Two types of crossover gadgets



(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.

2.4 In-order One-way Crossover

Initially, no entrance is passable except for a, since V_0 is passable only in state Verifed, and S_0 is passable only in state Set. Once the left $V_0 \to V$ transition is made, the robot has 2 options. It can either change the left Set-Verify's state to Set, or leave it as Verified. In either case, the S entrance

on that toggle is impassable, since a S entrance may only be traversed in state Unset. The only transition possible on the right crossover is $S \to S_0$, 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 occured. In that case, the left $S_0 \to S$ 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 $V \to V_0$ transition, completing the second crossover.

2.5 Directed Crossover

Using the One-way Destructive Crossover and the In-order One-way Crossover, we can construct a Directed Crossover which allows Arbitrary traversals from A to A' and B to B'. Notice, that in either path, one will activate one destructive crossover, but in doing so will toggle the in-order crossover, always ensuring that there is a path in the other direction, no matter which one is crossed first. This allows us to cross each way once, in either order, which is sufficient for our hardness proof.

2.6 Crossover

Four Directed Crossovers can be combined, as shown below, to create a crossover that can be traversed in any direction.

2.7 3-SAT Construction

We plan to reduce from 3-SAT 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 3-SAT instance. The directional properties of the Reversible One-Way and 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. They are considered true when the V to V_0 traversal is possible, and false otherwise. Thus we can set literals to true by allowing the robot to run through the S to S_0 passage of the gadget. This implies a very simple clause gadget, 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 true and cannot be passed otherwise.

The variables will be encoded by a series of passages which split to allow either the true or negated literals to be set. We begin each hall with a Reversible One-Way gadget, primarily to handle the case of having no literals of that type to set. Once

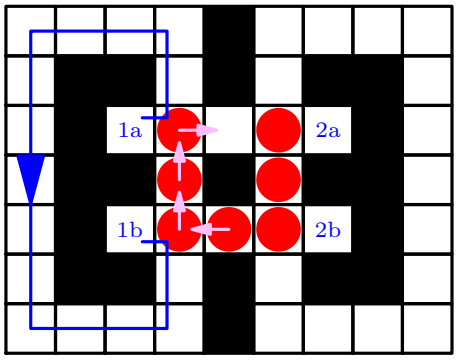
Variables split into two hallways. We have to show that when you enter or exit a side of a hallway, all the literals are set true at one end and false at the other. Also, we cannot go back through the other hallway, so if we go backward, everything gets reset. Thus we cannot exploit the reversibility to set anything extra true.

3 PSpace

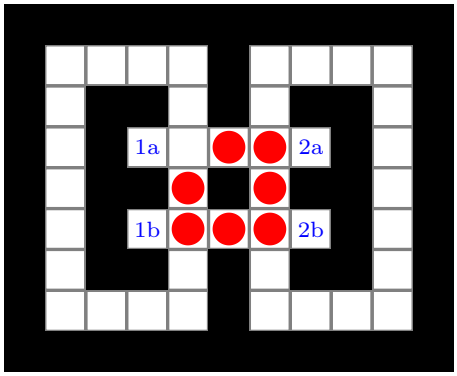
[Introduction Needed] xxx

3.1 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) 2-Toggle in state A



(b) 2-Toggle in a broken state

Figure 4a acts as a 2-Toggle. 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 $1b$ or $2b$. 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 and 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 **??**. Notice, 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 useful for solving the puzzle and can be safely ignored.

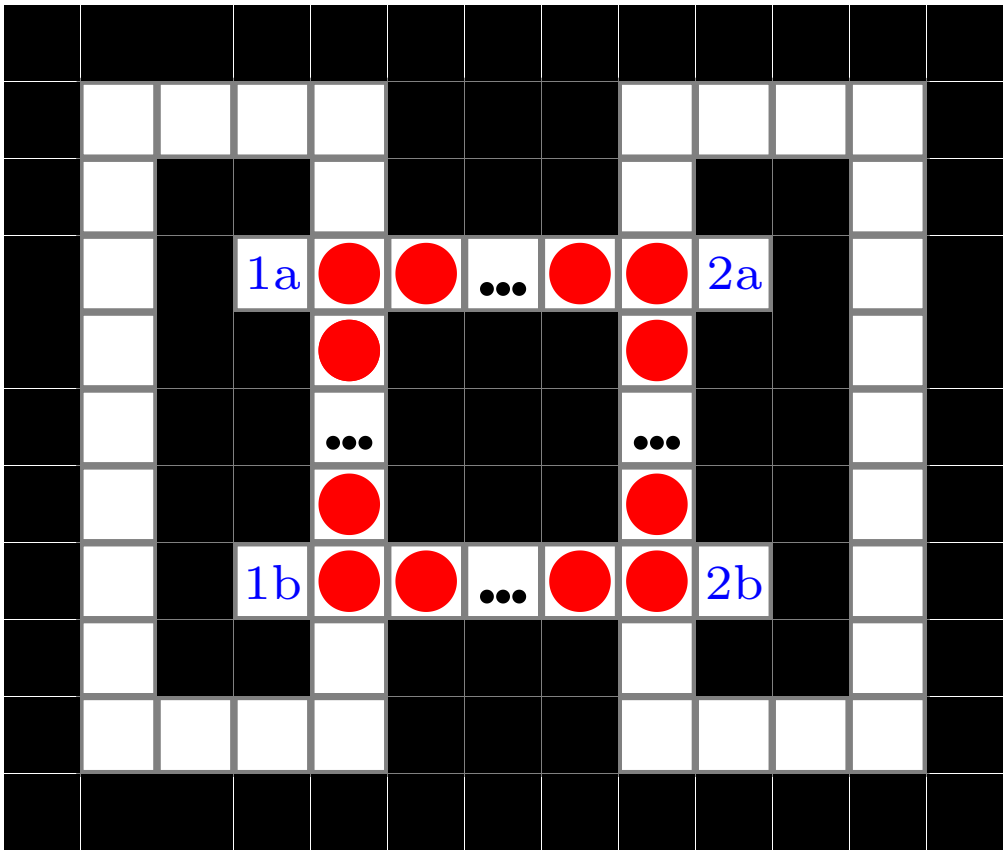
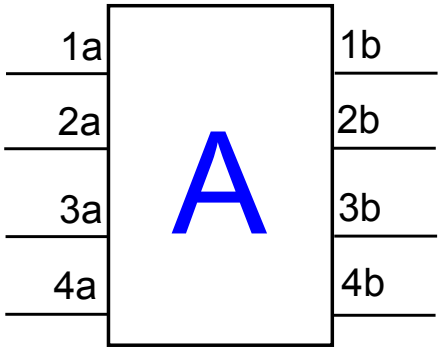


Figure 5: Construction of a 2-toggle when the robot can push or pull multiple blocks.



(a) Diagram of a 4-toggle

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

To construct a 4-Toggle we essentially take two copies of the two toggle, rotate them perpendicular to each other in 3D, and let them overlap on the central axis. See Figure **??**. We still interpret the lack of blocks in the same positions as 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.

4 Locks

[lock figure] xxx

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, based 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 **??**. 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 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 **??**, 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.

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.

4.1 Binary Counter

[binary counter figure] 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. xxx

The binary counter is implemented as a series of 2-toggles, as shown in **??**. 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.

4.2 Existential

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.

[existential figure] xxx

As shown in **??**, 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.

4.3 Quantifier Chain

[individual quantifier figure, and figure for them all hooked up.] xxx

We now define a quantifier chain, as shown in **??**. A quantifier chain implements a series of alternating existential and universal variables, as well as external literal pathways, which may 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. At that time, the robot will have the option to set a corresponding length sequence of existential variables arbitrarily, consisting of the existentials nested within the

universal variables whose values were flipped.

Traversing the quantifier chain in the reverse direction is only the 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.

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-toggle 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 are 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 **??**.

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, as shown in **??**.

This prevents the undesired high-significance existential alteration problem mentioned above, because it is now impossible to enter the gadget via the higher-significance universal variables which are set to 1, because that entryway will have at least 1 closed lock on it, and so 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.

4.4 Formula

We now define a clause gadget, as shown in **??**. A clause gadget is a single pathway which is passable if and only if at least one of 3 literals are passable. Recall from above that a literal is a pathway which is passable if and only if a specified variable has a specified setting.

The formula gadget consists of a series of clauses, as shown in **??**. The formula is thus only passable if every clause is passable, and thus if the corresponding $3CNF$ formula is true.

4.5 Beginning and End Conditions

The overall progression of the robot through the puzzle will start with the quantifier chain, incrementing the universal variables and setting the appropriate existential variables arbitrarily; then to traverse the formula gadget to verify that the formula is true under that setting; and repeating.

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 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.

5 Conclusion

6 Open Questions

6.1 Block Pushing

- Close NP, PSpace gap.
- No thin walls
- All movable blocks
- Push-Push Pull (seen in Catherin-not quite since you can get on them, also Zelda?)

7 Acknowledgments

References

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

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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(12):895 – 911, 2002.

[13] Erich Friedman. Pushing blocks in gravity is np-hard.

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

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

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

[17] Christos 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. ACM.

[18] Paul Rendell. A turing machine in conway's game life. *http://www. cs. ualberta. ca/˜ bulitko F*, 2, 2001.

[19] Marcus Ritt. Motion planning with pull moves. *CoRR*, abs/1008.2952, 2010.

[20] Richard Sprague. Uber mathematische kampfspiele. *Tôhoku Math. J*, 41:438–444, 1935.

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