Computational Complexity of Motion Planning of a Robot through Simple Gadgets

Seminar on Advanced Algorithms and Datastructures
Student Report

Matthias Roshardt

1 Introduction

1.1 The Problem

Robot motion planning is a broad category of problems which comprise at least one *actor* (the robot), an *environment* with a set of configurations and rules governing the transitions between configurations. Motion planning problems then specify a *initial configuration* and a *goal configuration* with the problem being finding a sequence of outputs (movements) for the actor that transforms the initial configuration into the goal configuration while optimizing some parameters. Many tasks in robotics map onto motion planning problems, which makes investigating their computational complexity both interesting and practically relevant.

In the titular paper we examine a discrete version of the motion planning problem with a single robot, stated in terms of *gadgets*.

Definition 1. A **gadget** consist of one or more *states*, one or more *locations* and a set of allowed *traversals*, which depend on and can change the state of the gadget.

More concretely, a gadget is characterized by its *state space*, so you can think of it as a mixed graph where each node is a (state, location)-tuple and the edges represent the traversals. For example, an edge from (s, a) to (s', b) means that if the gadget is in state s, the robot can go from location a to location b, which would unambiguously change the gadget's state to s'. We disallow 'spontaneous' state changes, i.e. the robot can't change the state without also changing locations.

Remark. Suppose we have a k-state gadget where for an $1 \leq i \leq k$ and $1 \leq j \leq k$ with $i \neq j$ the i-th and the j-th state are identical (i.e. for all locations a, b and all states s', edge $((s_i, a), (s', b))$ is in the state-space graph if and only if $((s_j, a), (s', b))$ is also present, or where $s' = s_j$, we have both $((s_i, a), (s_j, b))$ and $((s_j, a), (s_i, b))$ in the state-space graph). We will not consider this gadget to have k states, as two of its states can be collapsed into one, making it a $(k-1)$-state gadget.

Definition 2. A **system of gadgets** consists of a set of gadgets, their initial states, and *links* between disjoint pairs of locations forming a matching. The links do not depend on the state of either gadget and are always freely traversable. You may think of them as a set of locations that are each *shared* between exactly two gadgets. Gadgets are *local*, in that changing the state of one gadget does not change the state of any others in the system. Check Figure 14 on page 12 for an example of a system of gadgets.

Definition 3. A **gadget puzzle** consists of a system of gadgets, the robot's initial position and its goal position. The robot can move around by traversing gadgets, and move between gadgets via their links.

Given a gadget puzzle, the obvious accompanying decision problem is: *Is it solvable?* Before we delve into the details of this question, I'd like to draw attention to the fact that the space of all possible gadgets as defined here is quite vast and given the high expressiveness of graphs (in terms of computational reducibility from other problems), we could probably come up with a hardness proof rather quickly. Seeing this, the authors moved on and addressed the question: How much can we restrict the class of gadgets under consideration until solving the decision problem becomes easy (as in, being in P)? In other words: what is the *minimal* set of gadgets that guarantee puzzle hardness?

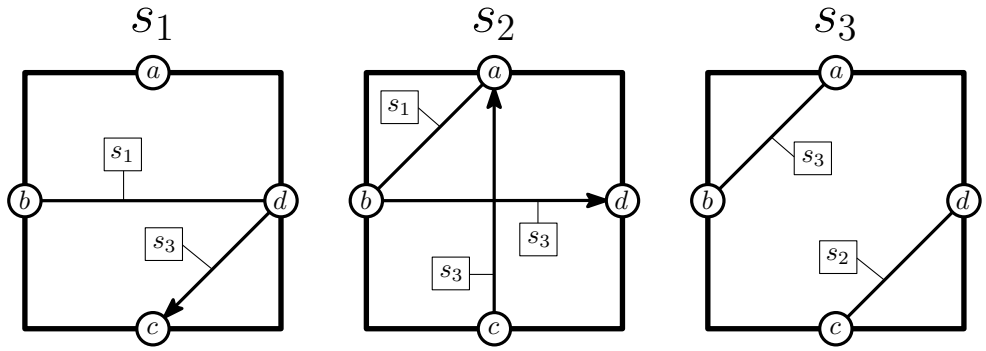

Figure 1: An example of a general gadget with three states (s_1, s_2, s_3) and four locations (a, b, c, d). The connectivity graph is pictured for each state. The edges are labeled to indicate the corresponding state transition (i.e. traversing an edge will change the gadget state to the state on the label). The thick lines around the gadget are not traversals, but serve to differentiate the interior of the gadget to the exterior, with the locations at the border. This separation becomes important when we deal with systems of multiple gadgets.

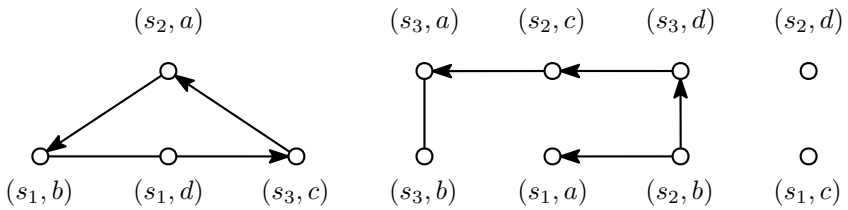

Figure 2: The state space graph of the same gadget as described in Figure 1. The state space graph is better suited for precise statements about gadgets, but the connectivity graph representation is better suited for illustrations, and will be used from here on out.

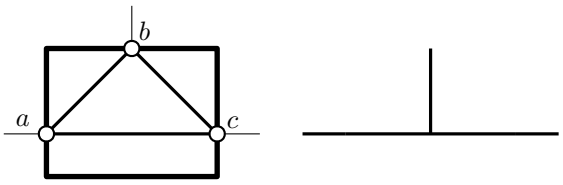

Figure 3: **BH**, A very simple non-deterministic gadget. Its three locations are labeled. It has a single state, and the connectivity in that state is given on the left. Since we use **BH** a lot later on, we use the representation on the right. Whenever you see lines fork in a diagram, there is a **BH** gadget at the intersection.

Note that there are multiple ways of reducing this problem in complexity, and much of the prior work is based on different reduction approaches. As we will see later, even with very tight restrictions on the types of gadgets we allow in the puzzles the problem is PSPACE-complete.

Remark. From now on, we will refer to the decision problem of whether a gadget puzzle is solvable simply as *motion planning*.

1.2 Gadget Terminology

1.2.1 Restrictions

In the following, we look at the aforementioned restrictions on gadgets that are chosen in the paper. First, let's define some characteristics of gadgets.

Definition 4. A gadget is **deterministic** if and only if its state-space graph has maximum out-degree ≤ 1 (i.e. a robot entering the gadget at (s, a) can go to at most one location b, changing the state in an unambiguous way to s', for arbitrary locations a, b and states s, s').

Definition 5. A gadget is **reversible** if and only if its state-space graph contains the reverse of every edge (i.e. any traversal by the robot can be undone immediately).

In the context of the paper we only consider gadgets that are both deterministic and reversible, with a single exception: The *Branching Hallway* **BH** (Figure 3). It has three locations, a single state, and its locations are fully connected (meaning the robot can move freely between the three locations).

Remark. In any of the constructions later on we always allow the use of **BH**, unless explicitly stated otherwise.

Furthermore, all gadgets besides **BH** fulfill the following definition:

Definition 6. A **k-tunnel** is a gadget with $2k$ locations which are connected in a perfect matching. Each of these k connections (called *tunnels*), can be *open* (traversable in both directions), *closed* (not traversable), or *directionally open* (traversable from one end to the other, but not vice versa), depending on the state. Traversing a tunnel may of course change the state of the k-tunnel gadget.

Definition 7. A tunnel *within* a k-tunnel is **trivial** if for all states it is open and traversing it does not change the state or for all states it is closed.

We will not consider gadgets containing trivial tunnels, as one can easily split trivial tunnels off and represent them in a different way (see Figure 4). This will in turn simplify our analysis.

To recap the restrictions we have made on the gadgets under consideration: with the exception of the **BH** gadget, *all* gadgets in the paper are *deterministic*, *reversible* k*-tunnels* (for any k) which contain only *non-trivial* tunnels. We add two final restrictions: first, we only consider gadgets with *two or less states*; second, we require that the systems of gadgets are *planar*, meaning you can draw the gadget locations and inter-gadget links in the plane without crossings. Note that this is not required for the interiors of gadgets, so it is possible to have crossing tunnels within gadgets. This restriction may seem somewhat arbitrary, but it simplifies certain proof steps when using the simulation technique later on. Luckily, even with planar systems we can retain the semantics of the unrestricted problem as crossings can be modeled with the crossover **XO** gadget. Also note that one gadget type may correspond to multiple planar gadgets, depending on the layout of the locations in the plane (see Figure 5 which I will explain shortly).

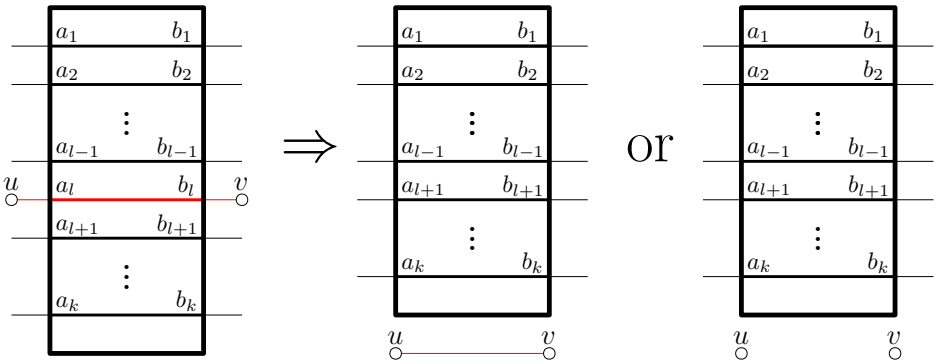

Figure 4: Pictured on the left is a k-tunnel gadget which contains a trivial tunnel, namely the one connecting a_l and b_l (for an arbitrary $1 \leq l \leq k$). The left end location of the trivial tunnel (a_l) is linked to a location in another gadget (u). The same holds for the other end (b_l is linked to v). We distinguish two cases. First, if the l-th tunnel is always open and does not change the state of the k-tunnel gadget, we can simply delete the l-th tunnel and link u and v directly. The overall behavior of the system is the same (pictured in the middle). Second, if the tunnel is always closed, we can also delete it from the gadget and leave u and v separated.

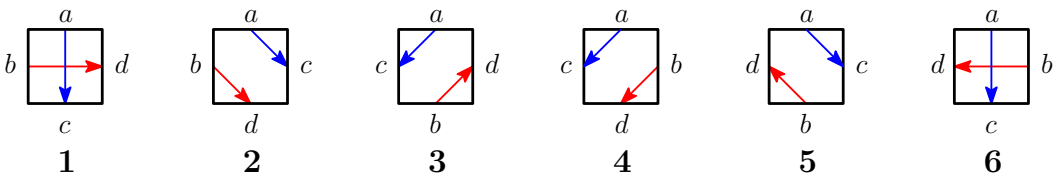

Figure 5: The six not-so-unique layouts of the locations of a 2-tunnel gadget in the plane.

1.3 2-Tunnels

In this section, we examine all possible 2-tunnels that fit our earlier restrictions. As we will see later, 2-tunnels are almost all we need to construct the more complex gadget systems which are used in the hardness proof. First, let's examine what kind of tunnels we could encounter in a k-tunnel gadget.

Tripwire This tunnel is always traversable, but traversing it switches the gadget's state.

Lock If the gadget is in the *locked* state, this tunnel is closed; in the *unlocked* state, it is open. Traversing the tunnel does not change the state (otherwise, the gadget would not be reversible).

Toggle This tunnel is always traversable in one (and only one) direction which depends on the state. Traversing the tunnel switches the state.

Note that with the exception of the toggle, these would not be valid 1-tunnel gadgets, as they would contain trivial tunnels.

Planar layouts A k-tunnel gadget generally corresponds to multiple planar *layouts*. Pictured in Figure 5 are the possible layouts for a 2-tunnel. We label the four locations a, b, c and d. The first tunnel leads from a to c, the second from b to d. For easier comprehension, we represent the gadget as a square where each location sits in the middle of a side (whereas later on, we generally place the locations on two sides). The following results would still apply if we drew the gadget as a disk (or any other shape) and placed the locations on the border.

First, we note that there are $(2k)! = 4! = 24$ ways to assign locations to the four sides of the square (list locations counter-clockwise starting from the top, count number of permutations of the resulting sequence). However, many of these layouts will turn out to be identical in our context.

We consider two group symmetries on the square: *rotation* and *reflection about an axis*. Suppose we have four locations v_1, v_2, v_3, v_4 distributed in the plane, and suppose we have linked them to a, b, c, d in some fashion without crossings in the plane. When considering rotation (by 1, 2, or 3

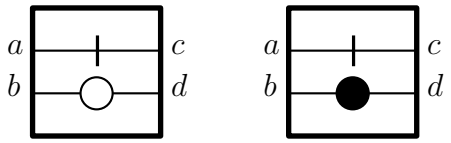

Figure 6: Non-crossing wire-lock (**NWL**) in its two states. Locations a and c are connected by a tripwire tunnel, b and d by a lock. The lock is open on the left figure and closed on the right. In this gadget, one could switch between the left and right state by traversing the tripwire (in either direction).

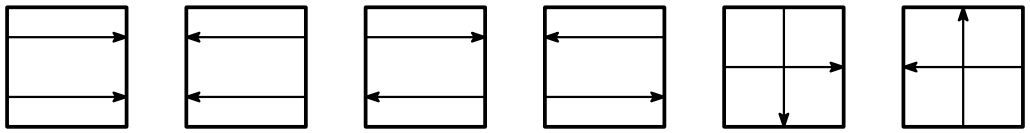

Figure 7: All planar Planar variants of the toggle-toggle gadget: parallel (**P2T**), anti-parallel (**AP2T**), crossing (**C2T**), pictured left to right in groups of two, corresponding to the two states. Rotations are not considered here. The arrows denote the directions of the toggles (arrows always indicate toggle tunnels from here on out). We note a departure from the convention of labeling the gadget locations. I trust the reader to identify these based on the intersection of the gadget border with its tunnel lines.

quarters of a turn), we notice that we can always maintain the links without crossings. We consider two layouts identical if we can exchange one with the other and link to v_1, v_2, v_3, v_4 such that the system behaves the same without the links crossing in the plane. Gadgets in our context are therefore rotation-invariant. This collapses our number of unique layouts to $(2k-1)! = 3! = 6$. We have labeled the six classes of layouts above.

We will now consider reflections (either horizontal, vertical or diagonal). The results of the reflection are the same for all three axes: we find that the reflected type 1 layout corresponds to the rotated type 6 layout, and vice versa. The same holds for layouts 2 and 4, as well as 3 and 5. We will call 1 and 6 *crossing* layouts, 2 and 4 *parallel* layouts and 3 and 5 *anti-parallel* layouts, for obvious reasons.

For now, we have assumed that the directions of the tunnels matter. We see that this is not always warranted: whereas *toggles* are directed, *tripwires* and *locks* are not. Consider the case where we have two toggle tunnels: first, if we have a type 1 or 6 (crossing) layout, we can switch between the two by traversing the gadget (i.e. changing the state corresponds to reflection). The same holds for the parallel and anti-parallel layouts. What's more, there is even a rotational symmetry within the crossing layouts and within the parallel layouts because the two tunnels are of identical type. We conclude that the toggle-toggle 2-tunnel really has three layouts instead of 24.

On the other hand, consider the case where we have two undirected tunnels (e.g. tripwire-lock). There, the anti-parallel and parallel layouts collapse into one (even without state changes), which we call the *non-crossing* layout. Finally, if we have one directed and one undirected tunnel in the gadget, state changes again collapse the parallel and anti-parallel layouts, as the reader may verify.

With all the above in mind, we can now look at all distinct 2-tunnel gadgets. Combinatorics tells us that there are six ways of pairing up the above tunnel types. Two of these pairs, namely *Tripwire-Tripwire* and *Lock-Lock* are made up of trivial tunnels, so we ignore them. We are left with *Tripwire-Lock*: **NWL** (non-crossing), **CWL** (crossing); *Tripwire-Toggle*: **NWT**, **CWT**; *Toggle-Lock*: **NTL**, **CTL**; *Toggle-Toggle*: **P2T** (parallel), **AP2T** (anti-parallel), **C2T** (crossing). We see that 2-toggle gadgets are the only ones with three layouts, for reasons outlined above.

2 Proving Hardness

2.1 The Takeaway

The following theorem states the main result of the paper.

Theorem 1. *Motion planning in puzzles consisting of any* deterministic, reversible, 2-state, k-tunnel *(for any k)* planar gadget *(and branching hallways) is PSPACE-complete if and only if the gadget*

has two toggle tunnels, a toggle tunnel and a tripwire tunnel, a toggle tunnel and a lock tunnel or a tripwire tunnel and a lock tunnel. Motion planning with all other such gadgets is in P.

Remark. The attentive reader may have noticed that the problem as stated before did not explicitly include an imput size n. I will depart from the paper and define n to be the number of gadgets in the puzzle. In the paper, the input size definition is rather awkward and relies on prior work by the authors. The origins of gadget analysis in this context is from 2D grid-based puzzle games (e.g. *Sokoban*), where the player (the robot) may move certain obstacles around the grid either by pushing or pulling to get to the goal. Therefore, the obvious size parameter is the size of the grid which houses the puzzle. Because the gadgets take up multiple 2D tiles, this sets an upper limit on the number of gadgets in the puzzle (particularly, there can't be more gadgets than grid tiles). In our context where the space in which the puzzle exists is more abstract, it would not make sense to refer to the size of a 'grid'. This redefinition does not change the hardness of the problem, as gadgets take up a constant amount of space in a 2D grid setting, meaning you could 'rasterize' our puzzles of size n such that they require at most cn tiles of space, for some constant $c \in \mathbb{R}$ [2].
However, using the original input size definition also puts an upper bound on the number of locations in the puzzle (since each location takes up a tile and each location can be part of at most two gadgets). In the formulation of Theorem 1, this is remedied with the specification of a parameter k for the size of the tunnel gadgets. In the PSPACE containment proof below however, we must address this.
As a second remark, the formulation of Theorem 1 implies that only instances of a single type of gadget (besides **BH**) are used to construct the puzzles.

In the following sections we will break this theorem down by looking at the structure of the proof, examining the main techniques used in it and illustrating the latter by actually proving parts of the theorem.

2.2 Proof Outline

The proof proceeds in four steps:

1. Prove that motion planning is in PSPACE
2. Prove that a gadget puzzle solving with two specific types of gadgets (namely *2-toggle-locks* **2TL** and *crossovers* **XO**) is PSPACE-Hard by reduction from the canonical PSPACE-complete problem *True Quantified Boolean Formula*
3. Prove that all other gadgets characterized in the theorem are capable of *simulating* **2TL**s and **XO**s
4. Prove the second part of the theorem, namely that motion planning with simpler gadgets is in P

As an appetizer, let us go through the first step.

Lemma 2. *Motion planning with gadgets comprising at most k locations and at most l states (for fixed arbitrary $k, l \in \mathbb{N}$) is in PSPACE.*

Proof. The entire state of the system can be described by the current state of the gadgets and the location of the robot. In a puzzle of n gadgets, we have $\mathcal{O}(k \cdot n)$ locations and $\mathcal{O}(l \cdot n)$ states, which puts a bound of $\mathcal{O}((k \cdot l \cdot n)^2) = \mathcal{O}(n^2)$ on the size of the representation of all gadgets in the puzzle (this bound includes the $\mathcal{O}(k \cdot n)$ links between the gadgets). We can non-deterministically search for a path to the goal position which only involves changing the robot's location and the state of the gadgets. Therefore, the problem is in NPSPACE and by Savich's theorem [3] also in PSPACE (because PSPACE = NPSPACE). □

2.3 3QSAT Reduction

We prove hardness of motion planning by a reduction from *True Quantified Boolean Formula* (also known as TQBF, or in our case 3QSAT), the canonical PSPACE-complete problem.

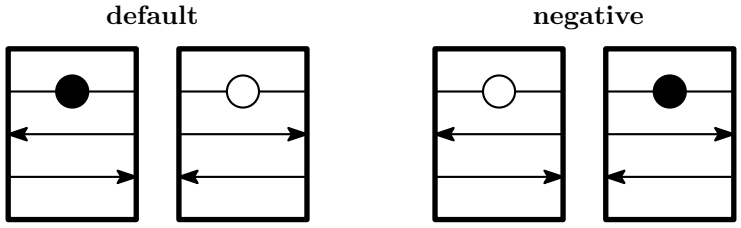

Figure 8: All states and layouts of 2-toggle-lock gadgets encountered in the 3QSAT reduction. The left half of the figures corresponds to the *default* layout and the other half to the *negative* layout. These designations coincide with the use of the **2TL**s as literals in the formula. In the formula, passing through a section is equivalent to fulfilling a predicate. The default layout corresponds to a positive literal: 0 in its initial state (on the left), 1 in the other state. In the initial state the lock is closed and the robot cannot pass the literal. The inverse is true for the other state. The negative layout corresponds to a negative literal by the same mechanism. For both layouts, we will refer to the left state as the 0-state and the right one as the 1-state.

Definition 8. 3QSAT is the following decision problem.
Given a set of variables $\{x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_n\}$ and a Boolean formula $\phi(x_1, \ldots, x_n, y_1, \ldots, y_n)$ in conjunctive normal form with exactly three variables per clause, is the quantified Boolean formula $\forall y_1 \exists x_1 \forall y_2 \exists x_2 \cdots \forall y_n \exists x_n \; \phi(x_1, \ldots, x_n, y_1, \ldots, y_n)$ true?

In my opinion, this section is the most interesting part of the proof. The following construction has been presented in [4], although not very clearly. The following is my attempt to flesh out that description.

The idea is to build a gadget puzzle which forces the robot to demonstrate a solution to the 3QSAT problem in order to reach the goal. The general layout of the puzzle comprises three parts: first, the *quantifier chain*, second, the *formula*, and third, a *locking mechanism* which prevents the robot from sneaking back into the quantifier chain and changing variables it's not supposed to.
Before going into detail on the construction of these parts, let's examine our building blocks: the 2-toggle-lock **2TL** (see Figure 8) and the crossover **XO**. The latter is somewhat uninteresting and we only use it implicitly when we need some inter-gadget links to cross in the plane. The **2TL** variant we consider is a 3-tunnel gadget with two anti-parallel toggle tunnels and one lock tunnel such that no tunnels intersect. Note that in the construction we sometimes only need either the two toggle tunnels or the lock and a toggle tunnel. If you see 2-tunnel gadgets in the illustrations, it is because the unused tunnel has been omitted. Now, let's examine the parts of the puzzle one after the other.

2.3.1 The Formula

Literals We model literals as single **2TL** gadgets (see Figure 8). Negated literals are represented by mirroring the gadget and switching the initial state. The literal gadgets appear along the quantifier chain, exactly at the point in the chain where they are quantified. There is one literal gadget in the chain for every time that literal appears in the formula. The truth value of the literal is represented by its lock (open = true, closed = false).

Clauses Recall that the formula is in conjunctive normal form with exactly three literals per clause. The robot should be able to pass through a clause if and only if it is true under the current variable assignment. We model the conjunctive clauses by linking the disjunctive clauses sequentially. This way, the only way for the robot to pass the formula-section of the puzzle is by passing every one of the disjunctive clauses one after the other. Disjunctive clauses are modeled as a sequence of two branching hallways **BH**, such that each of the three 'dangling' ends is linked to the lock tunnel of one of the three literals in the clause. Behind the literals, the paths are again unified by **BH** gadgets (Figure 9). This way, the robot can pass if and only if one of the literals is true under the current assignment. Note that we have enough literals such that each literal gadget is linked to exactly one disjunctive clause, which is crucial for the correctness of the construction.

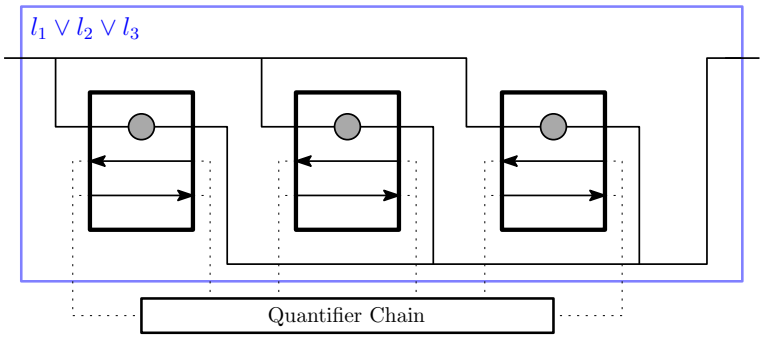

Figure 9: Representation of a disjunction with three literals. The locks in the literal gadgets are gray to indicate that they have no fixed assignment. The locks will be open or closed depending on the robot's movement through the quantifier chain. Note that the literals can be far apart in the puzzle (somewhere in the quantifier chain) and not next to each other as pictured here. This is because in the quantifier chain, the literals are grouped by the index of the variable they represent, which is generally false in the formula.

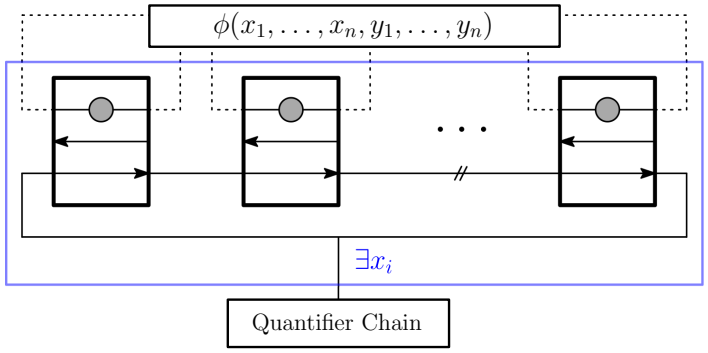

Figure 10: Existential quantifier system. Central to the construction is the chain of literals. The chain can be entered from both sides, ensuring that the variable assignment can be switched freely.

2.3.2 The Quantifier Chain

We model the quantifiers as a series of gadget systems with the robot's starting position at the beginning and its goal position at the end. There are exits along the quantifier chain which all lead to the formula. The only way to re-enter the quantifier chain is to pass the formula and enter at the starting position. All in all, we want to force the robot to cycle through the quantifier chain and the formula once for each of the 2^n possible assignments to $y_1, \cdots, y_n$ as required by the universal quantifiers, such that either the robot gets stuck in the formula or it completes all iterations and can proceed to the goal. As it passes through the quantifier chain, the robot can choose values for the variables $x_1, \cdots, x_n$.

Existential Quantifiers Same as in the description of the formula, there is an existential quantifier after each universal quantifier in the quantifier chain. An existential quantifier is modeled simply as a branching hallway on the way from the previous universal quantifier to the next one, such that the branch points to the literal section. There, the robot encounters all literal gadgets of the variable being currently quantified (Figure 10), one linked to the next into a chain by their bottom toggle (or the upper toggle, it does not matter which as long as it is the same for all literals in the chain). Both ends of the chain are then linked to a **BH**, where the 'dangling' end points back to the quantifier chain. This way, the robot, when passing from one universal quantifier to the next, can change the assignment of the current variable by making a pass through the literal section (which then inverts the truth value of every literal in the literal chain).

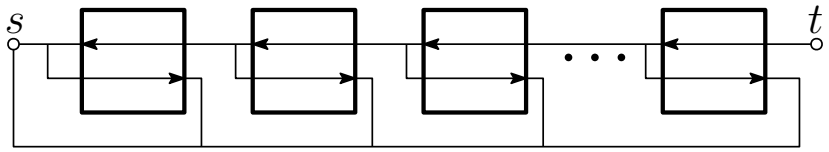

Figure 11: Counter puzzle with n 2-toggle-lock gadgets.

Universal Quantifiers The universal quantifier systems are different from the existential systems in that the robot must be forced to go through every variable assignment at least once (of which there are two per variable). The idea here is to construct a binary counter from **2TL** gadgets, such that the state of the gadgets (i.e. the bits) represent the values of $y_1, \cdots, y_n$. Before every incrementation of the counter, we force the robot through the formula. Once the robot has counted through all representable numbers, we can be sure that the formula really is true for all possible assignments. In a **2TL** gadget, we define the *0-state* as the state in which the top toggle points to the left (and by consequence the bottom toggle to the right), and the other state as the *1-state* (Figure 8). Let the robot's starting position be at the very left. We now chain n **2TL**s together, such that the starting position links to the left ends of the toggles (see Figure 11). We link the top toggle's right end to the left ends of the next **2TL** in the same fashion. The remaining unlinked tunnel end (bottom right) is wired back to the starting location (these are the aforementioned *exits* of the quantifier chain). The rightmost **2TL** has it's top right location wired to the goal and it's bottom right location back to the start same as all the others. Let all gadgets be in the 0-state at the start.

We now let the robot solve the puzzle. First it can only go through gadget 1, namely via its bottom toggle. This switches gadget 1 to the 1-state. The robot is back at the start. It can now move through gadget 1 via its top toggle and then through gadget 2 via its bottom toggle (or revert the previous move, which serves no purpose). As the robot returns to the start, gadget 1 is in the 0-state and gadget 2 is in the 1-state. As we can see, if we interpret the n gadgets as a binary number where the leftmost gadget represents the least significant bit, every time the robot runs a cycle through the puzzle, that number is incremented by 1. Once all gadgets are in the 1-state (i.e. after $2^n - 1$ iterations), the robot may pass through gadget n and finish the puzzle.
We can now simply expand this counting apparatus to incorporate all the necessary structures for the problem. We will call the gadgets currently present in the chain *universal gadgets*. Let the state of the i-th universal gadget represent the truth value of y_{n-i+1} for all $1 \leq i \leq n$ (i.e. the leftmost gadget represents the last universally quantified variable in the 3SAT formula). To complete the functionality of the universal quantifier systems, consider for every universal gadget the literals associated with that gadget. Similarly to the literal chains in the existential quantifiers, we chain the literal gadgets together, but this time by linking to *both* the top and bottom toggles (Figure 12). The chain is then inserted right of the universal gadget such that the top right location first goes through the top toggles of the chain before linking to the next universal gadget. Analogously, the bottom right location is wired to the bottom toggles in the chain before leading back to the start. All we need to do now is collect the exit pathways leading out of the universal gadget systems (meaning the universal gadget plus the associated literal chain) with branching hallways and leading them through the formula construction as described earlier. Finally, we insert the existential systems on the top right pathways out of the universal systems (before they branch and lead into the next universal system).

However, we're not quite done yet. Two major problems remain.

The final link As per our current construction, the universal gadgets are initially all in the 0-state (i.e. allowing exit into the formula section but not traversal to the next quantifier system). Also, after the robot has performed $2^n - 1$ cycles through the puzzle, the robot can simply breeze through all universal (and existential) systems, particularly through the rightmost one, and go straight to the goal. This entails an obvious and a less obvious issue, which we will both fix with a single adjustment. Obviously, the robot does not check if ϕ is true under the very last variable assignment before reaching the goal. But even worse, if we start out with all universal gadgets in the 0-state, the robot does not get to choose a value for the existentially quantified variables before making a first pass through the formula section. We solve these problems by adding a final universal gadget between the n-th

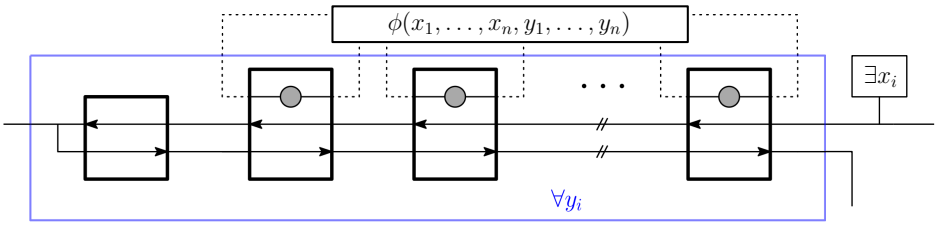

Figure 12: Complete universal system (in blue box). There are three links leading in and out of the system. The robot comes in from the left. On the right side, the lower exit leads through the formula and back to the start of the quantifier chain. The upper exit leads past an existential system which the robot may choose to go into (but must eventually come back out of), and then into the next universal system (where it would enter from the left).

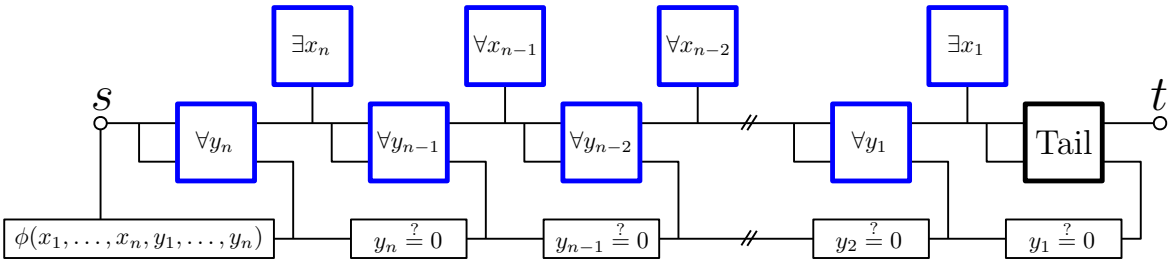

Figure 13: Schema of the full TQBF puzzle.

universal gadget and the goal position by linking the n-th universal system to that gadget (let's call it the *tail gadget*) on the left and the goal on the top right, while the bottom right is led back to the formula (Figure 13). We also change the initial states of all the universal gadgets to the 1-state, while the tail gadget is set to the 0-state.
Now let's revisit what happens when we let the robot solve the puzzle. From the start, the robot can go through the entire quantifier chain and set all of the existentially quantified variables. It then passes through the lower toggle of the tail gadget, switching it to the 1-state (i.e. the exit is now open) while all the universal gadgets have been switched to the 0-state.

2.3.3 The Locking Mechanism

One final problem remains: there is nothing stopping the robot, after leaving the quantifier chain and before entering the formula section, to just re-enter the chain through one of the exits and changing one of the earlier (or in the puzzle later) existentially quantified variables. If you recall the semantics of the 3QSAT-problem, you will see why this is a problem: after having 'decided' on a value for x_i we are not allowed to change the assignment of any x_j for all $0 \leq j \leq i$ while we cycle through the assignments to y_l for all $i < l \leq n$. We prevent this by locking the exits. Roughly speaking, we want to make sure that the robot can only go back into the quantifier chain and change those existentially quantified variables which it could have set during the current cycle anyway. The reader can verify that for all $1 \leq i \leq n$, the robot could have set the assignment to x_i if and only if after exiting the quantifier chain, the variables y_j are set to 0 for all $i < j \leq n$. We implement this condition by adding to each universal system, somewhere in the literal chain, another literal of the variable and then leading the exit paths sequentially through the lock tunnels, such that in order to go down in variable indices the robot has to prove one after the other that the next higher indexed universally quantified variable is indeed 0. This concludes our reduction.

2.4 Gadgets Simulate Gadgets

The major technique used in the proof of Theorem 1 is *simulation*. The following definitions assume that any system of gadgets made up of deterministic reversible gadgets is itself deterministic and reversible (adapted from [1], Lemma 2.2, by induction).

Definition 9. The **state** of a system of n gadgets is the n-tuple $(s_1, ..., s_n)$ where s_i is the state of the i-th gadget in the system.

Definition 10. A gadget g and a system of gadgets S are **equivalent** if and only if there is a bijective map f_L between the locations of g and a subset L of the locations in S as well as a bijective map f_S between the states of g and a subset of states of S such that there is a one-to-one correspondence between the legal transitions between locations in g and legal paths between the mapped locations in L. For instance if S and g are equivalent, the robot can move from (s, a) to (s', b) in g if and only if the robot can move through S from $f_L(a)$ in state $f_S(s)$ to $f_L(b)$ such that the system is left in state $f_S(s')$ at the end. In other words S is equivalent to g if they "behave the same".

Definition 11. A set of gadgets A **simulates** a gadget g if and only if there is a *system of gadgets* S containing a constant number of instances of gadgets exclusively from A, which is equivalent to g.

Remark. For the sake of readability, I will be sloppy with the above notation by stating that "gadget g simulates gadget h" even though it should be "$\{g\}$ simulates h".

Remember that our goal is to prove that puzzles with **NWL**, **CWL**, **NWT**, **NTL**, **CTL**, **P2T**, **AP2T** and **C2T** gadgets are PSPACE-Complete, which we can expand to all k-tunnels which contain them as a 'sub-gadget'. At this point we know that puzzles with 2-toggle-locks and **XO** are PSPACE complete. All that said, here is how we perform step 3 from the proof outline described earlier.

1. Show that gadget puzzles with **AP2T** are PSPACE-complete by showing that it can simulate 2-toggle-locks and **XO**, specifically by proving:
   a) **AP2T** simulates **C2T**
   b) **C2T** simulates **P2T**
   c) {**AP2T**, **C2T**, **P2T**} simulates **NTL**
   d) The aforementioned gadgets simulate a variety of 2-toggle locks with 'round' and 'stacked' internal connections
   e) stacked antiparallel 2-toggle locks simulate **NWL**
   f) **NWL** simulates the *stacked tripwire-lock-tripwire*
   g) stacked tripwire-lock-tripwires simulate **XO**
2. Show that **NWL** , **CWL** , **NWT** , **NTL** , **CTL** , **P2T** and **C2T** each simulate **AP2T**, specifically by proving:
   a) **P2T** , **NTL** , **NWT** , **NWL** each simulate **AP2T**
   b) **C2T** simulates **P2T** (as shown above), and hence also **AP2T**
   c) **CTL** simulates **NTL** (and hence **AP2T**)
   d) **CWL** simulates **NWL** (and hence **AP2T**)

Remark. With little extra effort (namely by proving that **CWT** simulates **NWT**), we can close the circle and show that for the gadgets mentioned in the second part it is true that any one of them can simulate any of the others.

In this report, I will barely demonstrate any of the above proof steps, because they are mostly quite similar to each other. I will instead show the general pattern of a simulation proof, and then give a somewhat involved example. Showing that a system of gadgets simulates a specific gadget usually follows this scheme:

- Show that any sequence of traversals of the gadget can also be done in the system.
- Show that any sequence of traversals of the system can also be done in the gadget.

The above formulation precludes the possibility of leaving the system in a 'broken' state, meaning a state in which further traversals don't correspond to the gadget anymore. In the context of 2-state gadgets, this often boils down to showing that the system can be left in exactly two states (including the initial state) and then showing that any single traversal is in correspondence with the gadget.

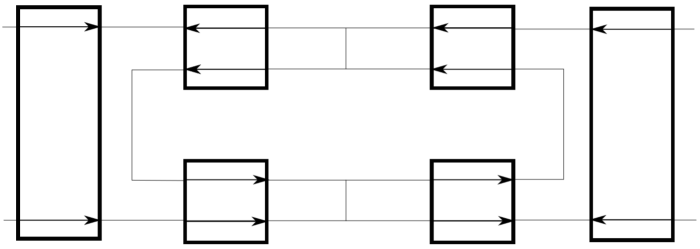

Figure 14: A system built from **P2T**s which is equivalent to an **AP2T** gadget, demonstrating that **P2T** simulates **AP2T**.

Lemma 3. ***P2T*** *simulates* ***AP2T****.*

Proof. Figure 14 gives a construction of an antiparallel-2-toggle out of parallel-2-toggles. There are two accessible states: As shown, and with the four inner **P2T** gadgets flipped. The former corresponds to the **AP2T** having a tunnel connecting the left two locations with its toggle oriented upward, and a tunnel connecting the right locations with its toggle oriented downward, while the latter corresponds to the two toggles flipped. First, let us examine the bottom right location in the state shown in the figure. After passing the rightmost **P2T**, the robot is blocked. No transitions or state changes are possible. This matches the desired behavior, because the right toggle in the **AP2T** being simulated is oriented down. Next, let us examine the top right location in the state shown in Figure 14. After passing the rightmost **P2T**, then the upper right **P2T**, the robot may now either proceed along the top tunnel, or down to the central loop. In the former case, the robot may pass through the upper left **P2T**, but then is blocked. In the later case, the robot may either proceed around the loop to the left or to the right. If the robot goes to the right, it can pass through the lower tunnel of the upper right **P2T**, but then is stuck. If the robot goes to the left, it can pass through the lower tunnel of the upper left **P2T**, then the upper tunnel of the lower left **P2T**. At this point, the robot may either continue around the loop, or exit the loop downward. If the robot continues around the loop, it can pass through the upper tunnel of the lower right **P2T**, but then is stuck. If it exits the loop, it can either go left or right on the bottom tunnel. If it goes left, it can pass through the lower tunnel of the lower left **P2T**, but then is stuck. If it goes right, it can pass through the lower tunnel of the lower right **P2T**, then the lower tunnel of the rightmost **P2T**, and exit the gadget. Overall, we observe that the robot can make exactly one transition, from top right to bottom right. The right toggle is traversed twice, and the inner toggles are all traversed once, leaving the gadget in the other accessible state. No other transition or state change is possible, from that entrance. Since the gadget is rotationally symmetric about its center, the possible transitions from the right mirror the possible transitions from the left. Since the other state is simply the state shown in the figure mirrored top-to-bottom, the transitions described mirror the transitions in the other state as well. □

2.5 Motion Planning with Simpler Gadgets is in P

To finish our treatise of the proof, let's go back to our initial motivation of finding a minimal set of gadgets that guarantee hardness. Keeping in mind our original restrictions that gadgets (with the exception of **BH**) must be *deterministic reversible k-tunnels* with *less than two states* which contain *no trivial tunnels* and our earlier result that the problem is PSPACE-complete as soon as the puzzles contain any 2-combination of the three basic tunnel types except *tripwire-tripwire* and *lock-lock*, there are two interesting restrictions we can still add: we can consider 1-state gadgets and 1-tunnel gadgets:

Lemma 4. *Gadget puzzles with 1-state gadgets are in NL.*

Proof. Since the connectivity of the locations is static, we can lay out the gadget puzzle as a mixed graph. Path planning in mixed graphs is in NL [3]. □

Lemma 5. *Gadget puzzles with 1-toggle gadgets (i.e. deterministic reversible non-trivial 1-tunnels) are in NL.*

Proof. We reduce this problem to st-connectivity in mixed graphs. We draw the system of gadgets as a directed graph where the edges point in the direction of the toggles in the initial states. Let the robot start location be s and the goal location be t. We claim that there is a solution to the gadget puzzle if and only if there is a path from s to t in the graph. Obviously, if there is a path from s to t, there also exists a solution to the puzzle. All toggles on such paths are traversed exactly once. However, we can imagine a solution to the puzzle that cannot be found in the graph because some toggles were traversed multiple times. Therefore, assume that there is a solution which traverses some toggles more than once. Let t be the last toggle in the path that was traversed more than once, and let the final traversal through it be from u to v. Since t was traversed repeatedly, there was some previous point in the path where the robot was at v before it traversed t the second-to-last time. Because all toggles from v on out are traversed only once, we can simply delete all moves between the last and the second-to-last time the robot was at v from the solution. The new, shorter path contains two less traversals of t. We can continue to apply this mechanism until t was traversed at most one, and then move up the path until the same is true for all toggles on the path. In consequence, if there is a solution to the puzzle, it can be found in the graph as a direct path from s to t. □

Remark. NL stands for Nondeterministic Logarithmic-space. It is known that NL $\subseteq$ P.

3 Conclusion

The advances presented in the paper at hand fall neatly into a landscape of existing gadget puzzle hardness proofs. The challenges presented are not the first and certainly not the last to be overcome in this vast field. If we critically examine our restrictions laid out in the beginning, we can think of other, very simple and minimalistic gadget types (e.g. three-location non-tunnel gadgets, non-deterministic or non-reversible simple gadgets etc).

In terms of interesting applications, the authors analyze a type of puzzle encountered in the video game *Legend of Zelda: Oracle of Seasons* and conclude that generalized Legend of Zelda is indeed PSPACE-Complete too, which should give anyone who beat that game a boost of confidence, if nothing else.

References

[1] Erik D. Demaine, Isaac Grosof, Jayson Lynch and Mikhail Rudoy, *Computational Complexity of Motion Planning of a Robot through Simple Gadgets*, 9th International Conference on Fun with Algorithms (FUN 2018)

[2] Hubert de Fraysseix, János Pach and Richard Pollack, *How to draw a planar graph on a grid*, Combinatorica, 10(1):41–51, 1990

[3] Walter J.Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Comput. Syst. Sci., 4(2):177–192, 1970

[4] Erik D. Demaine, Isaac Grosof and Jayson Lynch, *Push-pull block puzzles are hard*, Algorithms and Complexity - 10th International Conference, CIAC 2017, pages 9-15