Characterizing Universal Reconfigurability of Modular Pivoting Robots

Hugo A. Akitaya ORCID
University of Massachusetts Lowell, Lowell, MA, USA

Erik D. Demaine ORCID
Massachusetts Institute of Technology, Cambridge, MA, USA

Andrei Gonczi ORCID
Tufts University, Medford, MA, USA

Dylan H. Hendrickson
Massachusetts Institute of Technology, Cambridge, MA, USA

Adam Hesterberg
Harvard University, Cambridge, MA, USA

Matias Korman
Siemens Electronic Design Automation, Portland, OR, USA

Oliver Korten
Columbia University, New York, NY, USA

Jayson Lynch ORCID
University of Waterloo, Waterloo, ON, Canada

Irene Parada ORCID
TU Eindhoven, Eindhoven, The Netherlands

Vera Sacristán ORCID
Universitat Politècnica de Catalunya, Barcelona Spain

Abstract

We give both efficient algorithms and hardness results for reconfiguring between two connected configurations of modules in the hexagonal grid. The reconfiguration moves that we consider are "pivots", where a hexagonal module rotates around a vertex shared with another module. Following prior work on modular robots, we define two natural sets of hexagon pivoting moves of increasing power: restricted and monkey moves. When we allow both moves, we present the first universal reconfiguration algorithm, which transforms between any two connected configurations using $O(n^3)$ monkey moves. This result strongly contrasts the analogous problem for squares, where there are rigid examples that do not have a single pivoting move preserving connectivity. On the other hand, if we only allow restricted moves, we prove that the reconfiguration problem becomes PSPACE-complete. Moreover, we show that, in contrast to hexagons, the reconfiguration problem for pivoting squares is PSPACE-complete regardless of the set of pivoting moves allowed. In the process, we strengthen the reduction framework of Demaine *et al.* [FUN'18] that we consider of independent interest.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases reconfiguration, geometric algorithm, PSPACE-hardness, pivoting hexagons, pivoting squares, modular robots

Digital Object Identifier 10.4230/LIPIcs.SoCG.2021.56

Related Version Due to lack of space a large number of proofs are omitted. A full version of the paper is available at [2].

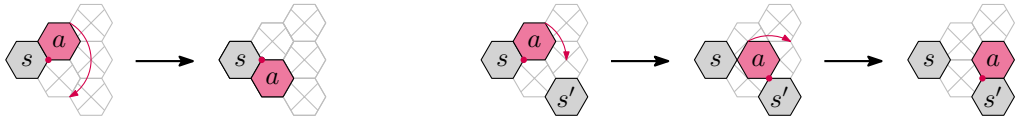
Funding *Jayson Lynch*: Supported by NSERC.
Vera Sacristán: Partially supported by MTM2015-63791-R (MINECO/FEDER) and Gen. Cat. DGR 2017SGR1640.

Acknowledgements This research started at the 34th Bellairs Winter Workshop on Computational Geometry in 2019. We want to thank all participants for the fruitful discussions and a stimulating environment. We would also like to thank a SoCG reviewer for their many contributions that helped improve the presentation of the paper.

1 Introduction

Reconfiguration problems encompass a large family of problems in which we need to provide a sequence of steps to transform one object into another. In this paper we consider the problem of reconfiguring a collection of modular robots (referred in this paper as **modules**) in a lattice using some prespecified set moves. Many variants of this problem have been studied both in the robotics and in the computational geometry communities. In this paper we study the reconfiguration problem for edge-connected configurations of hexagonal and of square modules. We follow the commonly used **single backbone condition** [11], that requires edge-connectivity to be maintained at all times. The moves allowed are pivots: a module can rotate around vertices shared with other modules and at the end of a move the pivoting module must lie in a lattice cell. The interior of two modules can never intersect.

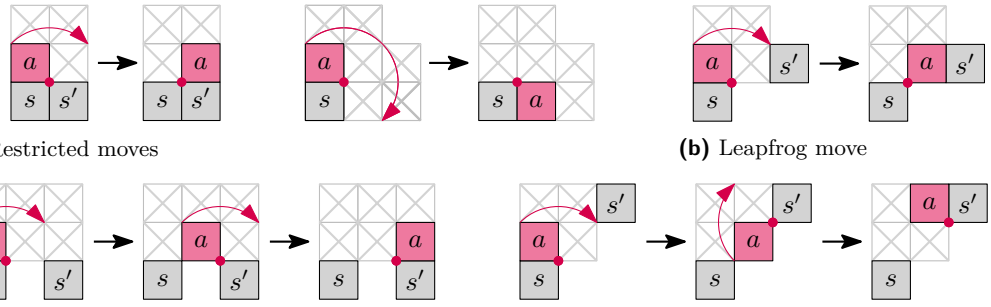
A hexagonal module can perform only two types of pivoting moves, illustrated in Figure 1. In a **restricted** move a module a adjacent to a module s pivots around a vertex v shared by a and s and ends the pivoting move in the other cell that has v on the boundary. The **restricted model** of pivoting only allows this move. In a **monkey** move a module a adjacent to a module s starts pivoting around a vertex v shared by a and s as in the restricted move, but halfway through the rotation another vertex w of a coincides with the vertex of a module s'. Then a continues the move pivoting around w in the same direction (clockwise or counterclockwise) as before until reaching a cell adjacent to s'. The **monkey model** of pivoting includes both the restricted and the monkey moves. Informally, the monkey move allows a module to keep pivoting in the same direction when a restricted move is not possible.



(a) Restricted move **(b)** Monkey move

Figure 1 Pivoting moves for hexagonal modules and their free-space requirements.

In the square grid two modules that share a vertex might not share an edge. Thus, for square modules there is a greater variety of pivoting moves. The three different sets of moves are illustrated in Figure 2. The **restricted model** includes only restricted moves, the **leapfrog model** includes both restricted and leapfrog moves, and the **monkey model** includes all moves.



(a) Restricted moves **(b)** Leapfrog move

(c) Monkey moves

Figure 2 Pivoting moves for square modules and their free-space requirements.

Related work and contribution.

One of the most natural questions for modular robots is whether universal reconfiguration is possible. That is, is there an algorithm to transform any (connected) configuration of n modules into another configuration with the same number of modules?

Efficient algorithms are known for universal reconfiguration of modular robots using moves that have significantly lighter free-space requirements [3, 10, 11, 12]. Relaxing the connectivity requirement has also lead to reconfigurability results [7].

The setting of this paper (pivoting robots) has proven to be more challenging. Instead, previous work has revolved around providing sufficient conditions for reconfiguration. Nguyen, Guibas and Kim [15] showed that reconfiguration of hexagonal modules using only restricted moves is always possible between configurations without the forbidden pattern illustrated in Figure 3 (left). Similarly, for pivoting squares, Sung et al. [16] presented an algorithm for reconfiguring between configurations without the patterns shown in Figure 3 (right). These algorithms do not provide reconfiguration guarantees as soon as the configuration contains a single copy of the forbidden pattern. In an attempt to remove global requirements, a recent result [1] introduced a different type of necessary condition: an efficient algorithm for reconfiguring between any two configurations that have 5 modules on the external boundary that can freely move (for pivoting squares in the monkey model). Other algorithms to reconfigure pivoting squares and hexagons are heuristics that do not provide termination guarantees [5, 14].

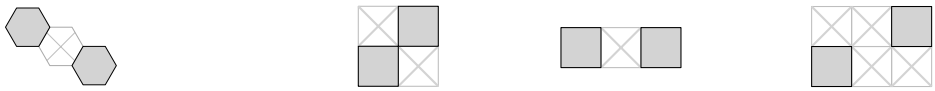


Figure 3 Forbidden patterns in previous algorithms for hexagonal and square pivoting modules

Despite many attempts, universal reconfiguration remains unsolved in the setting of edge-connected pivoting robots. In this work we answer this question for all five pivoting models for hexagons and squares. Specifically, we answer it positively for the hexagonal monkey model by giving a universal reconfiguration algorithm in Section 2. For all other models we show that it is PSPACE-hard to determine whether we can reconfigure one configuration to another. In the process, we prove a stronger PSPACE-hardness result about a restricted form of motion planning with reversible, deterministic gadgets from [9] (our reduction highly limits the direction in which each edge can be traversed, effectively reducing the number of cases to consider). This framework has already proven useful in other swarm robot motion planning models [6, 8] and we believe the improvements here will aid in future PSPACE-completeness proofs. The framework is described in Section 3.2 and is used afterwards for hexagonal restricted robots in Section 3.3, and for all square models in Sections 3.4 and 3.5. Table 1 summarizes our results.

Model	Restricted	Leapfrog	Monkey
Hexagons	PSPACE-hard (Thm. 14)	N/A	$O(n^3)$ universal (Thm. 13)
Squares	PSPACE-hard (Thm. 16)	PSPACE-hard (Thm. 17)	PSPACE-hard (Thm. 17) $O(n^2)$ if +5 modules [1]

Table 1 Summary of results. The leapfrog moves do not make sense for hexagonal modules.

2 Polynomial Algorithm for the Hexagonal Monkey Model

This section describes an algorithm that computes a sequence of $O(n^3)$ moves in the monkey model that transforms a given configuration with n modules into another. Our approach uses a **canonical configuration** defined as the configuration with n modules whose contact graph is a path and each module is only adjacent to modules above and/or below it. Since each move is reversible, an algorithm that takes a configuration and transforms it into the canonical configuration within $O(n^3)$ moves can be used to compute $O(n^3)$ moves between any pair of configurations. The main strategy is to increase the connectivity of the contact graph[1]. Note that if the contact graph is 2-connected, every convex corner of the configuration is movable, including the modules that are extremal in a grid direction. Then, there is a module that can move to become the new topmost module by attaching itself to a previous topmost module. We proceed in this manner inductively building the canonical configuration.

Definitions and Preliminaries. The contact graph G is the adjacency graph of the modules in a configuration. Since connectivity is important for the problem, we use the **block tree** $\mathcal{B}$ of the contact graph G. A graph is **2-connected** if it contains no **cut vertices**. A **block** (also 2-connected component) of G is a maximal subgraph of G that is 2-connected. We call a block containing a single edge a **trivial block**. We define $\mathcal{B}$ to be a bipartite tree whose nodes are the cut vertices of G in one partite set, and its blocks in the other partite set. There is an edge between two nodes if the corresponding cut vertex is contained in the corresponding block. The deletion of a cut vertex v of a connected graph G splits it into two or more components. A subgraph induced by such a component union with $\{v\}$ is called a **split component** of v. Similarly, a **2-cut** is a pair of vertices $\{v_1, v_2\}$ whose deletion increases the number of components of G. Its **2-split components** are the subgraphs induced by $\{v_1, v_2\}$ united with each of the components obtained by the deletion of $\{v_1, v_2\}$.

We now give some more specific definitions used in the algorithm. Note that a module corresponds to a vertex in G. We refer to them interchangeably. We label the topmost rightmost module of G the **root**. We root $\mathcal{B}$ at the node containing the root module. A cut vertex (2-cut) defines one **parent** (2-)split component, containing the root module, and one or more **child** (2-)split components. Such a cut vertex (2-cut) is called the **parent** of its children (2-)split components. A 2-split component ℓ is **trivial** if $|V(\ell)|$ is 3 or 4. The parent of such a component is also called trivial. Note that because G is a subset of the triangular grid, one of its faces is either the external face (whose edges form the **boundary**), a triangle, or encloses an empty position of the grid, which we call a **pocket**. In a 2-connected block ℓ, if v is a vertex in the boundary of ℓ and it is not incident to any pocket, deleting v can cause *only adjacent* vertices to become cut vertices. We call a 2-cut $\{v_1, v_2\}$ **adjacent** if v_1 and v_2 are adjacent. Note that when $\{v_1, v_2\}$ is an adjacent trivial 2-cut, the faces of the trivial 2-split component are triangles. Our algorithm uses the following fact about adjacent nontrivial 2-cuts.

▶ **Observation 1.** *If $\{v_1, v_2\}$ is an adjacent nontrivial 2-cut, then $\{v_1, v_2\}$ has only two 2-split components. Furthermore, if v_1 is movable, $\{v_1, v_2\}$ is the only 2-cut containing v_1.*

The previous observation comes from the maximum degree of the triangular grid. For an adjacent 2-cut to have three 2-split components, two of them must be trivial. The fact that v_1 needs 3 adjacent empty positions around it to be movable implies that v_1 must be adjacent

[1] Increasing connectivity of the contact graph of the configuration is a concept that has recently proven useful in the different setting of reconfiguring **sliding squares** [13].

to 2 modules other than v_2. Any cycle through v_1 connecting the (2-)split components of $\{v_1, v_2\}$ must go through the two modules adjacent to v_1 that are not v_2.

The main technical part of our algorithm is a procedure called Merge that increases the 2-connectivity of G, i.e., decreases the number of nodes in $\mathcal{B}$. For that, we want to move modules in order to create new paths between blocks of G without destroying previously existing blocks. We define a **2-free** module to be a movable module whose movement preserves 2-connectivity in the block containing it (a module that is not in a 2-cut). A **crew** $c = (m_1, \ldots, m_k)$ is a sequence of modules that induce a connected component of G such that m_1 is 2-free, and m_i, $i \in \{2, \ldots, k\}$ is 2-free after the deletion of all m_j, $j \in \{1, \ldots, i-1\}$. For a given 2-connected subgraph ℓ of G, let $\overline{\ell}$ be the induced subgraph of G given by $V(G) \setminus V(\ell)$. A **bridge** from ℓ is a set of modules that were previously a crew that moved to create a path between ℓ and $\overline{\ell}$, thus potentially not being 2-free anymore. We say a set of modules **bridges from** ℓ if they move to create a new path between ℓ and $\overline{\ell}$. One of the goals of the algorithm is to get a crew of size three in a group of grid positions called a **flower** that is otherwise empty. That allows us to maneuver the modules in the crew to create a bridge while not creating new blocks. Let a **flower** be a set of grid positions defined by a **center** cell and the six adjacent positions. A flower is **adjacent** to a grid position if the flower does not contain it but contains a grid position that is adjacent to it. A flower is **valid** for a 2-connected configuration ℓ and a disjoint crew c if it contains *exactly* the modules in c (all modules in c and no other modules), and is adjacent to a module in ℓ.

The following are definitions that help us describe positions in the configuration. We might reflect and/or rotate the configuration in order to fit our description w.l.o.g., and the following definition always refer to the current frame of reference. A **row** containing a position p is the set of all positions $p + (-\frac{\sqrt{3}}{2}, \frac{1}{2})i$ for some integer i. An **ascending** (**descending**) path in a row ρ is a path $(m_1, \ldots, m_k)$ induced by modules in ρ such that m_{i+1} is the top-left (bottom-right) neighbor of m_i. An **extreme path** is a path induced by modules that are on the convex hull. Due to the geometry of the grid, extreme paths can only have six possible directions. A **SW extreme path** of a configuration ℓ is an ascending or descending path in the lower hull of ℓ. Given a position p in the grid, we use a sequence of arrow superscripts on p to describe positions nearby. For example, $p^{\uparrow\nearrow}$ refers to the position to the top-right of the position above p, i.e., $p + (\frac{\sqrt{3}}{2}, \frac{3}{2})$. We overload this notation to refer to the current positions of modules, replacing p by a module.

Main algorithm. We split the contact graph into two parts: the canonical path P which is a canonical configuration, and the remainder of the graph G. We initialize G to be the entire contact graph and P to be empty. Let $\mathcal{B}$ be the block tree of G rooted at the block containing the topmost rightmost module. We divide our algorithm into three phases. **Phase 1** is a prepossessing procedure that eliminates all trivial leaves of $\mathcal{B}$. Then, assume that every leaf of $\mathcal{B}$ contains at least three modules and no further procedure will change that. **Phase 2** transforms G into a 2-connected graph. While $\mathcal{B}$ is not a single node, let ℓ be a leaf of $\mathcal{B}$. We will apply Merge(ℓ), outlined in Algorithm 1, that will cause ℓ to merge with other nodes of $\mathcal{B}$ until G becomes 2-connected. **Phase 3** builds P. We decrease the size of G while adding modules to P by moving a crew on its boundary so that each of its members in turn move to become the new topmost module in the contact graph. We use a slightly modified version of Merge to produce such a crew without breaking the 2-connectivity of G.

2.1 Phase 1: Removing Trivial Leaves

Phase 1 reconfigures a connected configuration into one without vertices of degree 1 (which are in trivial leaves) in G. There are configurations in which it is not enough to just pivot the degree-1 modules, i.e., this task requires coordination with other modules. See Figure 4.

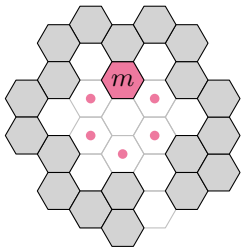


Figure 4 Configuration with one trivial leaf (m) that cannot be removed by pivoting it.

▶ **Lemma 2.** *A connected configuration of $n > 2$ hexagons can be transformed in $O(n^2)$ moves into a configuration without trivial leaves in the contact graph without breaking connectivity.*

Proof sketch. Let m be a degree-1 module. If it is possible to move m to a place where it is adjacent to more than one modules, then we do so. Else, we move m so that its shortest path to the root module is maximized. The full proof uses a detailed case analysis to show that, because of the specific position chosen for m, there is a nearby movable module with which m can coordinate to locally reduce the total number of new trivial leaves. ◀

2.2 Phase 2: Merging Leaves

The goal of Phase 2 is to take a connected configuration with no degree 1 vertices, and transform it into a 2-connected configuration in $O(n^3)$ moves. The main technical tool of this phase is the Merge procedure, outlined in Algorithm 1, which allows us to reduce the number of 2-connected components by merging them. Its input is a child (2-)split component of a cut vertex v (adjacent 2-cut $\{v_1, v_2\}$). We first apply the necessary rotations so that v ($\{v_1, v_2\}$) is farthest from the row ρ_0 containing the extreme SW path of ℓ. We then assume that ρ_0 does not include v ($\{v_1, v_2\}$) and neither does the row above it except for the base case when $|V(\ell)| = 3$ and ℓ is a split component, or when $|V(\ell)| = 5$ and ℓ is a 2-split component. The output of the algorithm is a set of modules that, after $O(|V(\ell)|^2)$ moves, bridges from ℓ.

Refer to Algorithm 1. Merge uses several other sub-procedures which we outline here. We call m the **ascending** module, which by its definition in line 2 is movable. It is either 2-free, in which case we will try to move it by cw pivots to its highest possible position in ρ_0 before it leaves ℓ; or it is part of a 2-cut, in which case we make it 2-free using sub-procedures. The end goal is to either bridge using m while it ascends in ρ_0 if it gets blocked by a vertex $m^* \notin \ell$, or accumulate 2-free modules at the top of the configuration where a valid flower will form. Then, the Bridge sub-procedure moves the valid flower around ℓ until it hits $\overline{\ell}$ where we create a bridge with the crew. There are three main Cases given by lines 4, 16 and 21.

Assume we are in Case 1. If m is in a trivial 2-cut, it will try to move up as explained before. Let m' be the module at $m^{\nearrow}$. By Observation 1, $\{m, m'\}$ is the only 2-cut containing m. If m succeeds in moving up, at least one unit, that leaves m' a cut vertex. Then, in line 7, we move the (up two) modules that are in the 2-split component of $\{m, m'\}$, restoring 2-connectivity. During m's ascension in ρ_0, we identify whether a valid flower gets formed. In the positive case, Bridge will accomplish our goal. During its ascension, m might be blocked

by a module $m^* \in \overline{\ell}$. If certain conditions are satisfied, the Local-Bridge sub-procedure uses m to create a bridge to m^*. Else, the Incorporate sub-procedure moves m to the row ρ_1 above it, or out of ℓ, and we can find a new ascending module.

Now assume that m is part of a nontrivial 2-cut (Case 2). Then, either m is part of an adjacent 2-cut or it is incident to a pocket. In the case m or an adjacent module is part of an adjacent 2-cut we recurse in the child 2-split component, which makes m 2-free. Else (Case 3), we either use Deflate, which decreases the number of empty positions enclosed by the pocket, or Bubble-Up, which moves one of such empty positions up. In some situations, Deflate produces a 2-free module in ρ_0 that will be the next ascending module.

Algorithm 1 Merge(ℓ)

```
1  while True do
2  |  Let m be the topmost module in a SW extreme path of ℓ;
3  |  Let ρ_{-1}, ρ_0 and ρ_1 be the rows below, of, and above m respectively;
4  |  if m is 2-free or part of an adjacent trivial 2-cut then
5  |  |  Pivot m cw to the highest position in ρ_0 before it leaves ℓ;
6  |  |  if m was part of a trivial 2-cut then
7  |  |  |  Pivot cw once the other modules in the trivial child;
8  |  |  if m is ever in a crew c of size 3 in a valid flower F during its ascension then
9  |  |  |  Return Bridge(F, ℓ − c);
10 |  |  else if m bridges from ℓ then
11 |  |  |  Return m;
12 |  |  else if the requirements of Local-Bridge(m) are met then
13 |  |  |  Return Local-Bridge(m);
14 |  |  else
15 |  |  |  Incorporate(m);
16 |  else if {m, m^↗} or {m^↘, m^{↘↗}} is a nontrivial 2-cut then
17 |  |  Let ℓ' be the child 2-split component of the highest such 2-cut;
18 |  |  c' := Merge(ℓ');
19 |  |  if c' bridges between ℓ and ℓ̄ then
20 |  |  |  Return c';                          ▷ c' already merges ℓ into another block.
21 |  else
22 |  |  Deflate(m^↗) or Bubble-Up(m^↗);                          ▷ m^↗ is empty
23 |  end
24 end
```

Bridge(ℓ, F)**.** The operation takes a 2-connected ℓ and a valid flower F containing a crew $c = (m_1, m_2, m_3)$ where m_1 was an ascending module. It returns c after a sequence of moves that transforms c into a bridge from ℓ. Compute a maximal sequence of flowers $(F_1 = F, \ldots, F_k)$, where each subsequent flower is adjacent to ℓ, containing no modules except for c, and obtained by moving the previous flower by one grid unit around the boundary of ℓ. We choose to move cw or ccw around ℓ based on the following condition. If ℓ has a parent cut vertex, then choose arbitrarily. Else, if ℓ has a parent adjacent 2-cut $\{v_1, v_2\}$ where v_1 is movable, we chose the direction towards v_2 so that F_k is not adjacent to v_1. Since G is connected and planar, and there are vertices in $\overline{\ell}$, F_k is adjacent to a module m^* in $\overline{\ell}$. We show how to compute the sequence of moves to bring the the crew with the sequence of flowers $(F_1, \ldots, F_k)$ and finally bridge between ℓ and m^* in F_k.

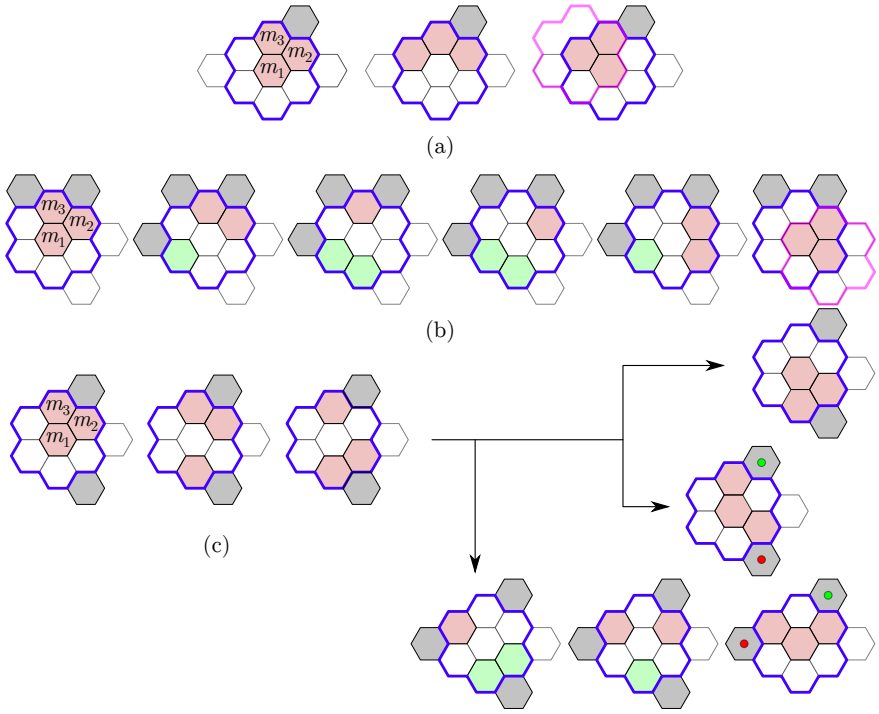


Figure 5 Maneuvers used to rotate around m_1 a crew that induces a cycle. A possible next flower is shown in pink.

If the modules in c induce a connected graph, this graph is either a triangle, a straight path or a "bent" path. A configuration of c in a valid flower is **useful** if a module of c is adjacent to ℓ and c induces a triangle or a "bent" path with m_1 in the center of the flower (i.e., both endpoints are adjacent to modules outside the flower). We show how to reach every useful configuration of c in a valid flower F_i. This is enough to accomplish our objective since: (i) by definition, $F_i \cap F_{i+1}$ is adjacent to ℓ and there is a useful configuration contained in the intersection of F_i and F_{i+1} (Figure 5 (a)–(b)); and (ii) if $m^- \in \ell$ is the only module adjacent to F_k and m^* is the only module of $\overline{\ell}$ adjacent to F_k and they are across from the center of F_k, e.g. m^- (m^*) is at the topmost (bottommost) position adjacent to F_k, then we can move F_k one more unit along the boundary of ℓ, contradicting the maximality of the sequence. By (i) we can transition between flowers F_i and F_{i+1} through a useful configuration, making both valid. By (ii) there is a useful configuration at F_k that bridges between m^- and m^*.

In the full version [2], we present four **maneuvers** shown in Figure 5 along with omitted proofs. Note that, by the fact that c is a crew, we have a guarantee that some positions adjacent to the flower are empty. We then use them to show the following lemma.

▶ **Lemma 3.** *Every useful configuration of a crew c in a valid flower can be reached from any useful configuration.*

▶ **Lemma 4.** *Bridge(ℓ, F) performs $O(|V(\ell)|)$ moves and bridges from ℓ while not breaking connectivity. After its execution, ℓ is still 2-connected. If $\{v_1, v_2\}$ is the parent 2-cut of ℓ and v_1 is movable, then v_1 remains movable.*

Deflate(p) and Bubble-Up(p). These operations take an empty position p to the top-right of a module m which is a corner of a 2-connected subgraph ℓ of the contact graph. We assume

that m is a corner of ℓ in its SW extreme path, i.e., $m^{\nwarrow}$, $m^{\swarrow}$, and $m^{\downarrow}$ are empty. Then, p is enclosed by ℓ by 2-connectivity. Refer to Figure 6. Deflate requires that positions surrounded by a red line in Figure 6 (a) or (c) are as shown. In particular, if $m^{\searrow\nearrow}$ is full, then $m^{\uparrow\nwarrow}$ is empty. Then, the operation fills p with a module adjacent to m and preserves 2-connectivity of ℓ, effectively reducing its area. Bubble-Up requires that positions surrounded by a red line in Figure 6 (b) are as shown. In particular, if $m^{\searrow\nearrow}$ is full, then $m^{\uparrow\nwarrow}$ is full. We additionally require that $\{m^{\searrow}, m^{\searrow\nearrow}\}$ is not a nontrivial 2-cut. Then, the operation moves the empty position and m to their top-left position while preserving 2-connectivity.

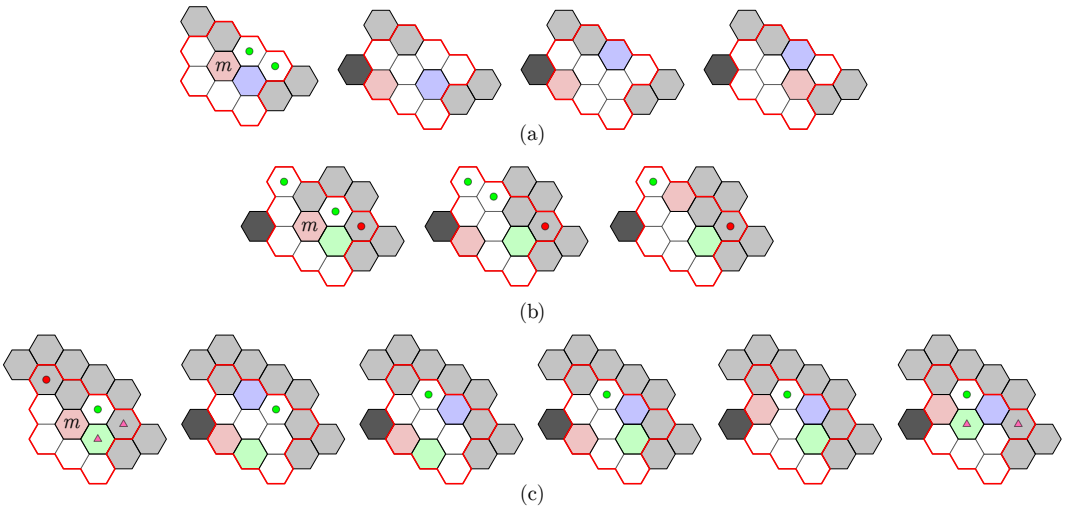


Figure 6 Operations used in Deflate(m_1).

Figure 6 shows the operations assuming that m performs a monkey move by pivoting cw.

▶ **Lemma 5.** *Deflate*(p) *and Bubble-Up*(p) *perform* $O(1)$ *moves, do not break connectivity, and the resulting* ℓ *is 2-connected.*

Shift(M)**.** Although not used directly in Merge, this operation is used in following subprocedures. The input is an ascending or descending path of modules $M = (m_1, \ldots, m_t)$ in the same row ρ and in the boundary of G. We require that none of the modules are cut vertices, m_1 is movable, and $m_i^{\downarrow}$ is empty $\forall i \in \{1, \ldots, t-1\}$. We describe the operation for descending M (Figure 7). There are two cases. In the first case, no module $m_i \in M$ is such that $m_i^{\nearrow}$ is empty and $m_i^{\nearrow\searrow}$ is either empty or contains a module only adjacent to M. Then, move each m_i cw from $i = 1$ to t (Figure 7 (d) to (f)). In the second case, let $m_i \in M$ be the first module such that $m_i^{\nearrow}$ is empty and $m_i^{\nearrow\searrow}$ is either empty or contains a module m' only adjacent to M. Move all m_j from $j = 1$ to $i - 1$ and m' (if it exists) by pivoting cw. Then, apply Deflate(m_i) and move back all the m_j and m' to their original positions by a ccw pivot. This vacates m_i's original position (Figure 7 (a) to (d)). If $i \neq t$, apply Shift recursively on $(m_{i+1}, \ldots, m_t)$.

In the full version [2] we prove the following statement.

▶ **Lemma 6.** *Shift*(M) *performs* $O(|M|)$ *moves, does not break the connectivity and, after it terminates, all pockets of* ℓ *remain intact except for possibly one that has* m_t *in its boundary.*

Inflate(m)**.** This operation uses Shift to make a concave corner convex, possibly creating a new empty space enclosed by ℓ. This will be used in Local-Bridge. The input m is

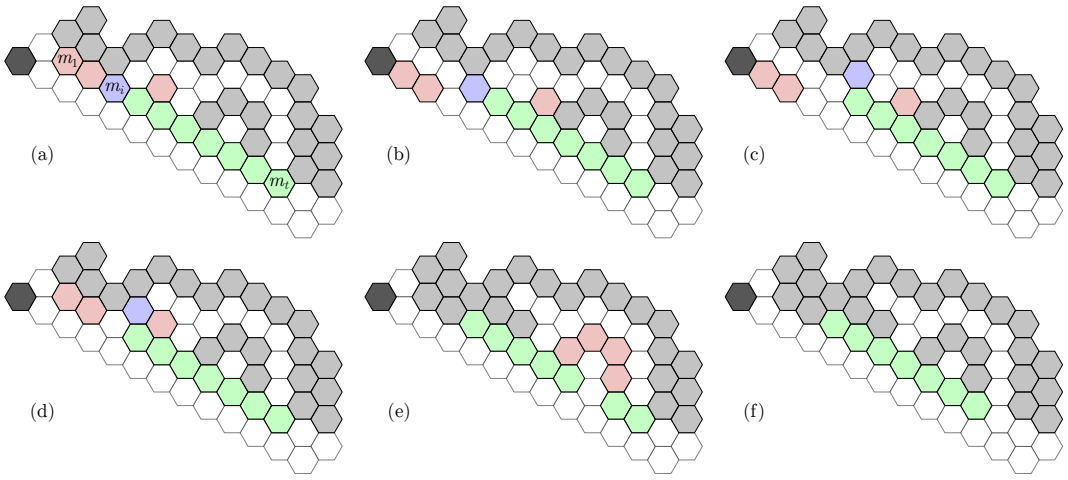


Figure 7 Illustration of Shift(M) where M is descending.

an ascending module in a 2-connected ℓ. We require that $m^{\uparrow}$ and $m^{\nearrow}$ are full, neither $\{m^{\uparrow}, m^{\uparrow\nearrow}\}$ nor $\{m^{\uparrow\uparrow}, m^{\uparrow\uparrow\nwarrow}\}$ are in adjacent nontrivial 2-cuts, and that at least one position in $\{m^{\uparrow\nwarrow}, m^{\uparrow\nearrow}, m^{\uparrow\uparrow\nwarrow}\}$ is full. Inflate moves the module at $m^{\uparrow}$ to $m^{\nwarrow}$ via a series of operations, returning such module.

Refer to Figure 8 (a)–(c) for examples (or [2] for a full proof). In short, we use Shift to move away modules adjacent to the blue module, so that we can move it out. Then, we reverse the Shift operation to put the moved models, except for the blue one, into their original place.

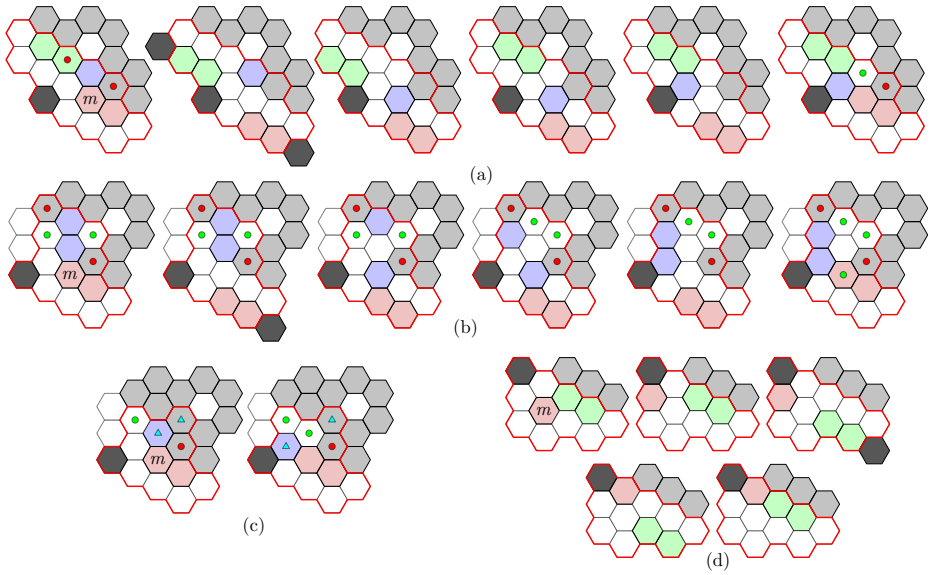


Figure 8 Operations used in Local-Bridge.

▶ **Lemma 7.** *Inflate*(m) *performs* $O(|V(\ell|)$ *moves, does not break connectivity and preserves 2-connectivity of* ℓ.

Local-Bridge(m)**.** This operation is used when there is an opportunity to create a bridge

when m either gets blocked by m^* on its way to the top of ρ_0 or it reaches the top and it would jump to $\overline{\ell}$. We require that, if a cw pivot brings m to ρ_{-1}, then $m^{\nearrow}$ is full and at least one position in $\{m^{\uparrow\nwarrow}, m^{\uparrow\nearrow}, m^{\uparrow\uparrow\nwarrow}\}$ is full. We recurse on a child component, calling Merge if there is an adjacent nontrivial 2-cut forbidden by Inflate. That guarantees that we can apply Inflate which would create a bridge from ℓ. If a cw pivot maintains m in ρ_0, then it would land on a module $m^* \in \overline{\ell}$. We require that the maximal ascending path M ending in $m^{\uparrow}$ can be shifted down by Shift(M), i.e., ρ_0 must contain only m below M; see Figure 8 (d). Then, we "squeeze" m in the space between m^* and M creating a bridge. We do that by moving m out of ℓ, Shift M down, moving m back and Shift M back.

▶ **Lemma 8.** *Local-Bridge(m) bridges from ℓ, does not break connectivity, and preserves 2-connectivity of ℓ. It uses $O(|V(\ell)|) + T_m(|V(\ell')|)$ moves where $T_m(|V(\ell')|)$ is the number of operations performed by Merge in ℓ'.*

Incorporate(m). Whenever a local bridge was not possible, this operation either incorporates m into ρ_1 or leaves m attached to a module in $\overline{\ell}$ with the promise that some module will ascend in ρ_0 and bridge (Figure 9 (d)). There are four cases. In case 1, we check if we can call Deflate at position $m^{\uparrow\nearrow}$. In the positive case, we move m and a possible neighbor m' in ρ_0 out of the way, call Deflate, and move m and possibly m' back (Figure 9 (a)–(b)). In case 2, $m^{\nearrow}$ is empty and $m^{\uparrow\nearrow}$ is full. Then, we "squeeze" m into $m^{\nearrow}$ by using Shift operations, similar to Bridge (Figure 9 (c)). In case 3, if we pivot m cw, that brings m to ρ_1 and makes its degree 1. Then, we apply some local movements in order to incorporate m into ρ_1 while maintaining 2-connectivity (Figures 9 (e) and (g)). In case 4, we are not in the previous cases and we simply pivot m cw. Note that m might leave ℓ (Figure 9 (d)). We explore this case from now. As shown in the proof of Lemma 9, there is a guarantee that a subsequent module s in ρ_0 will ascend. There are three possible cases, either (i) s creates a bridge using m (as in Figure 9 (d)), in which case nothing needs to be done; (ii) s calls Local-Bridge or Bridge. Then, pivot m twice counterclockwise before Bridge or after Local-Bridge; or (iii) s calls Incorporate. Then, there is either another module in ρ_0 or we can move s back to ℓ and apply Local-Bridge.

▶ **Lemma 9.** *Incorporate(m) uses $O(|V(\ell)|)$ moves and brings m to ρ_1 in every situation that m would go to ρ_{-1} by pivoting cw to which Local-Bridge does not apply. It does not break connectivity and maintains 2-connectivity of ℓ. Any created degree-1 module outside ℓ can be reincorporated in ℓ, thus, no new block is created.*

Analysis.

▶ **Lemma 10.** *If $\ell \neq G$ is a leaf block of $\mathcal{B}$, Merge(ℓ) performs $O(|V(\ell)|^2)$ moves merging ℓ and a subset of nodes of $\mathcal{B}$ into a single block while not creating any other new blocks.*

Proof sketch. A key observation is that each section of the perimeter can only be traversed by at most three ascending modules until either a local bridge or a valid flower is formed. Every time we use Incorporate to hide a module in ρ_1 we have the guarantee that, if either the next or the next two ascending modules reaches m, then Local-Bridge or Bridge will be called and the method terminates. We can then charge the moves of a module to the perimeter. Hence, each level of recursion of Merge makes a linear number of moves. Another key observation is that there are only a constant number of recursive calls. Since we always recurse on a smaller problem, the upper bound on the number of moves is $O(|V(\ell)|^2)$. ◀

▶ **Corollary 11.** *G can be made 2-connected in $O(n^3)$ moves.*

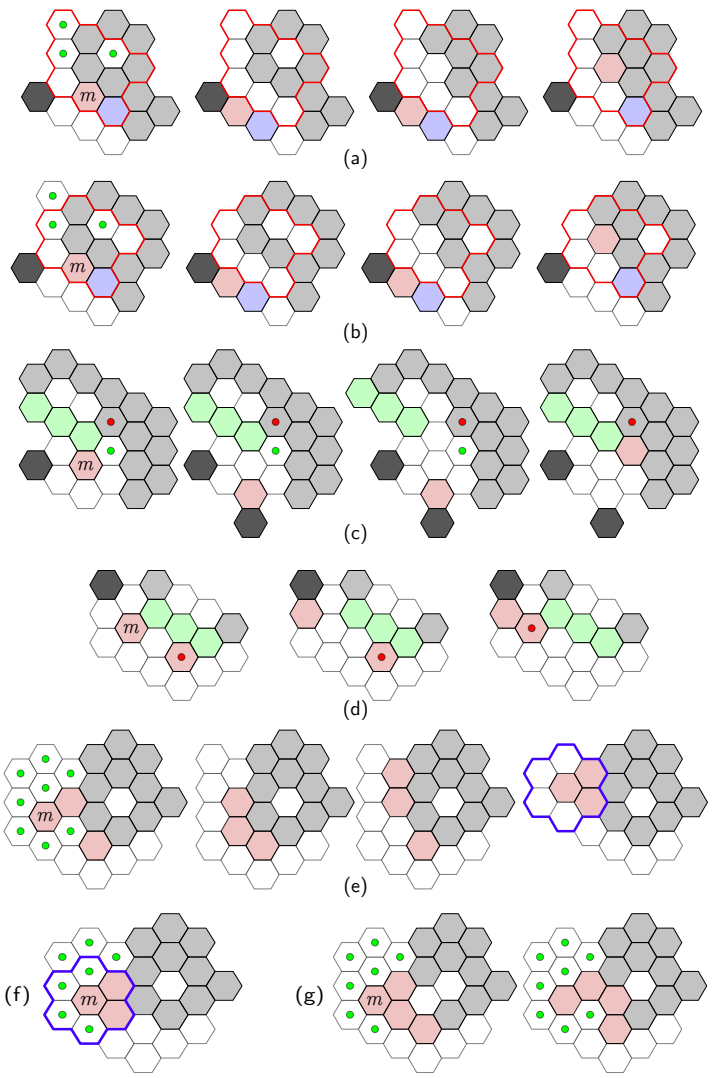


Figure 9 Operations used in Incorporate.

2.3 Phase 3: Building the Canonical Path

In the final phase, we will show that once the configuration is 2-connected, we can start moving modules onto the end of our path P at a cost of $O(n^2)$ moves per module.

▶ **Lemma 12.** *If G is 2-connected, in $O(n^2)$ moves we can produce a 2-free module on an extreme path of G while maintaining the 2-connectivity of G.*

Proof. We apply a subset of operation Merge to G. Then, this proof becomes a special case of Lemma 10, where $\ell = G$. In Merge, our goal is to bridge between ℓ and $\overline{\ell}$ maintaining ℓ 2-connected. Here, $\overline{\ell}$ is empty and Local-Bridge will never be called since there are no obstacles for ascending modules. Then, we are always able to produce a crew in an extremal position. Moving the crew to P does not affect 2-connectivity of G by definition. ◀

Our main theorem is a direct consequence of Lemma 2, Corollary 11 and Lemma 12.

▶ **Theorem 13.** *Any connected configuration of n hexagonal modular robots can be reconfigured to any other with $O(n^3)$ pivoting moves in the Monkey model, while maintaining connectivity.*

3 PSPACE-hardness Reductions

In this section we show PSPACE-hardness for all other models. Our reduction follows the framework introduced in [9]. We reduce from a reachability problem: given an agent that moves along a graphlike structure whose traversability changes in response to the agent's actions, is there a series of moves which takes the agent from a start to a target location.

▶ **Theorem 14.** *Given two configurations of n hexagonal modules, it is PSPACE-hard to determine if we can reconfigure from one to the other using only restricted moves.*

In Section 3.1 we describe the reachability problem introduced in [9] and the pieces we need to simulate to create the reduction. We introduce a few modifications to this problem and show it remains PSPACE-hard in Section 3.2. In Section 3.3 we discuss how to simulate each of the gadgets with hexagonal modules. Reductions for other models are in Section 3.4.

3.1 Preliminaries

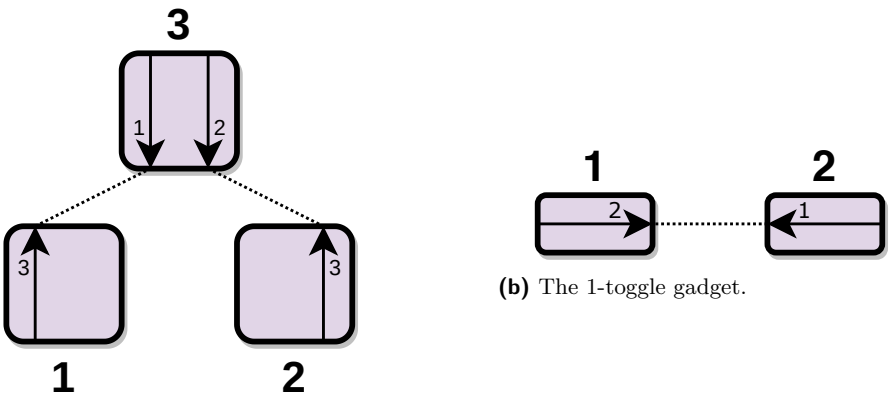
We reduce from a variation of 1-player motion planning with the locking 2-toggle (L2T) [9]. This restricted variant is called *1-toggle-protected motion planning with the locking 2-toggle* and described in Section 3.2. In the **1-player motion planning** problem we want to decide whether an agent has a series of moves that will take it to a target location. The constructs we use in this problem are **gadgets** which have **locations** (entrances and exits), **states**, and **transitions**. The agent is always at some unique location. Transitions are an ordered pair of state and location pairs. If an agent is at some specific gadget location and the gadget is in a state matching the first pair, then the agent can move to the location in the second pair which changes the state of the gadget to the state in the second pair (see Figure 10). A **system of gadgets** is a set of gadgets and **connections** between locations in those gadgets. The agent can freely move between locations that have connections. Some gadget transitions form a matching - we call these matched pairs **tunnels**.

In order for us to reduce from this problem, we need to use modules to represent the agent, the gadgets (specifically a locking 2-toggle and a branching hallway gadget), connections between locations, and a goal location. In order to reduce from 1-toggle-protected motion planning with the locking 2-toggle we need to create the following constructions:

- **Wires** which allow the modules to travel between parts of the configuration. These simulate the connection graph edges that allow the agent to travel between locations.
- **Branching hallways** which connect three wires together and allow the modules to travel down any of them.
- **Locking 2-toggle** which is a 3 state, 4 location gadget shown in Figure 10a. The gadget has two tunnels which are both traversable in state 3. After taking either transition, the only option is returning back and restoring the gadget to its prior state.
- **Win gadget** which can only be reconfigured if two additional modules reach it, simulating the goal location in the motion planning problem.

3.2 1-toggle-protected Motion Planning

In this section we strengthen the result from [9] to show that motion planning with reversible, deterministic gadgets with interacting tunnels is PSPACE-complete even when connections



(a) The locking 2-toggle gadget (L2T).

(b) The 1-toggle gadget.

Figure 10 Examples of reversible, deterministic gadgets. Purple boxes are states of the gadget, labeled with a number outside the box. Arrows represent transitions from one location to another. The small number close to an arrow indicates the state obtained by the transition. Dotted lines help visualize which states are connected by transitions in the gadget.

can only be traversed as though they are 1-toggles. We will consider only branchless systems of gadgets, but we will allow the branching hallway gadget. In a **branchless** system of gadgets, the connections between locations form a matching [4]. The **branching hallway gadget** is a 1-state, 3-location gadget with traversals among all three pairs of locations.

An instance of **1-toggle-protected motion planning** with a set of gadgets G is an instance of branchless 1-player motion planning with G as well as the branching hallway gadget and the 1-toggle, where one end of every connection is a location on a 1-toggle. Intuitively, this requires that every edge in the connection graph acts as a 1-toggle.

▶ **Theorem 15.** *1-toggle-protected planar 1-player motion planning problem with a reversible, deterministic, on-tunnels gadget with interacting tunnels is PSPACE-complete.*

3.3 Reduction for Hexagonal Modules

We now focus on describing how to simulate each of the pieces with hexagonal modules. The agent is represented by two modules and while these could go different ways, our instance contains several obstacles that can only be crossed by two modules working together. For simplicity we refer to the two modules that form the agent as *the agent modules* (shown in orange in the figures).

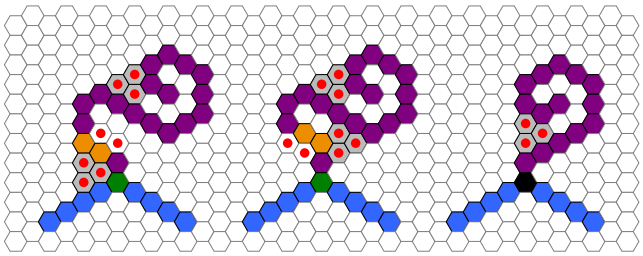


Figure 11 Each corner contains a spiral that prevents it from moving. In the protected case, the agent can go to a specific location allowing two other modules on the other side of the spiral to move (left and center), while in the blocked case, the spiral cannot be crossed at all (right).

We simulate wires with sequences of modules in line segments. We also need to be able to turn without letting corner modules move. We simulate these turns using two types of corners: **protected** and **blocked** (Figure 11).

The branching hallway, shown in Figure 12, allows an agent that arrives on any of the three wires to leave on any of the other two. This construction acts as a branching hallway with 1-toggles on two of its wires. We can implement the toggle on the third wire by adding two protected corners.

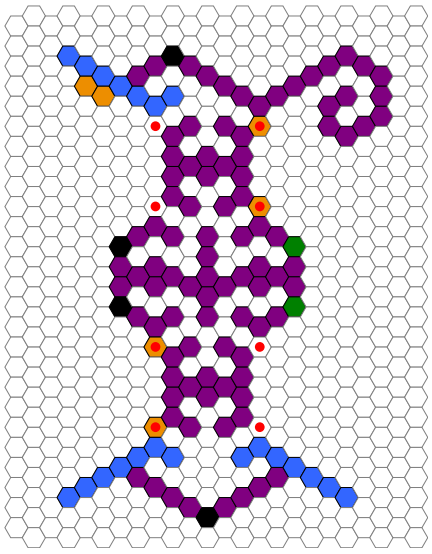


Figure 12 The branching hallway gadget simulated with hexagonal modules. Wires are shown in blue, and the two modules simulating the agent are shown in orange. Protected corners are drawn as green modules, while blocked corners are drawn as black. Both types of corners contain additional robots (omitted for clarity).

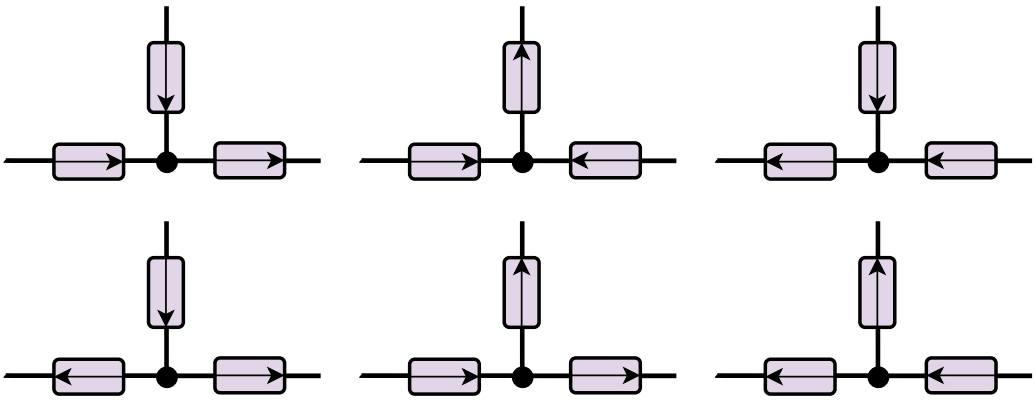


Figure 13 The six configurations of a branching hallway with endpoints connected to 1-toggles.

The other main gadget needed for the reduction is the locking 2-toggle shown in Figure 14.

3.3.1 Finishing steps

Proof. (of Theorem 14) Our reduction follows the framework in [9]. Given a problem instance for 1-toggle-protected motion planning with the locking 2-toggle, we embed in a way that

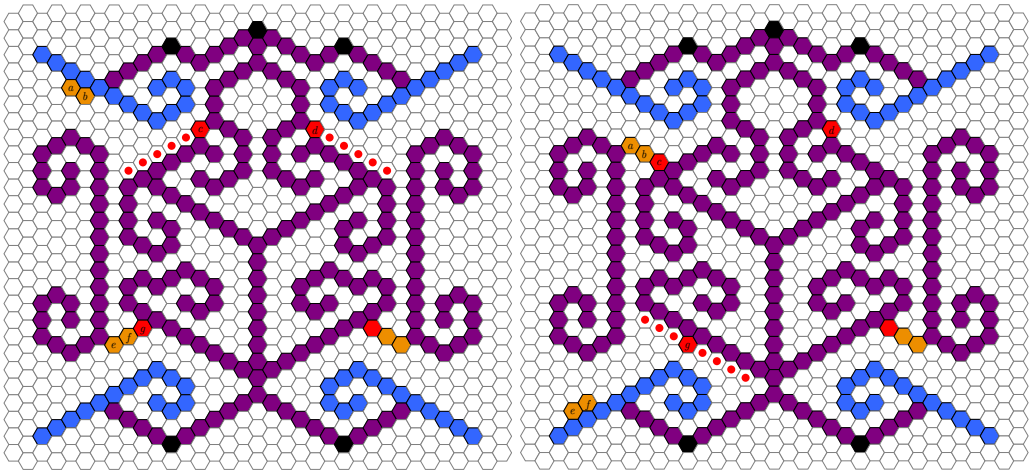


Figure 14 (left) The locking 2-toggle simulated with hexagonal modules (state 3 in Figure 10a). Other than the agent (a and b), the modules that can move are c and d (but not at the same time). (right) Once a, b and c form a bridge, they create a cycle allowing e, f and g to move. Two of the three modules can exit the gadget along the bottom wire. This changes the state of the 2-toggle (state 1 in Figure 10a).

all edges are drawn with polylines that are multiples of 60°, replacing gadgets with the corresponding module configurations (adding side switch and wire cut gadgets as needed, as well as 1-gaps to all wire segments). Finally, we place two additional modules at the initial position to define the agent. Since each gadget takes constant space, the problem instance will have polynomial size. Our goal configuration is the same configuration with only one change (the state of the win gadget).

If the problem instance is solvable, there is a way for the agent to reach the win gadget, change its state, and then return back to the initial position in the exact reverse path. By doing so we reset every gadget except the win gadget back to its original state. If the problem instance is not solvable, the agent cannot reach the win gadget and thus the reconfiguration problem will also be infeasible. ◀

3.4 Square Modules with the Restricted Move Model

▶ **Theorem 16.** *Given two configurations of n square modules, it is PSPACE-hard to determine if we can reconfigure from one to the other using only restricted moves.*

Our reduction is analogous to the hexagonal reduction. We quickly list the pieces and a small description for each, but for brevity the proof of correctness of each single gadget is removed. The arguments are analogous to the hexagonal counterpart and we present a full list of our gadgets in the full version [2].

The branching hallway gadget is show in Figure 15 and works like the hexagonal version.

The L2T gadget in the open state can be seen in Figure 16. Again, this gadget has exactly the same functionality as its hexagonal counterpart. The reduction works similarly and the proof for Theorem 16 follows a similar format as Theorem 14.

3.5 Hardness for the Square Model for Monkey and Leapfrog Models

Our final reduction applies to both remaining models for square modules.

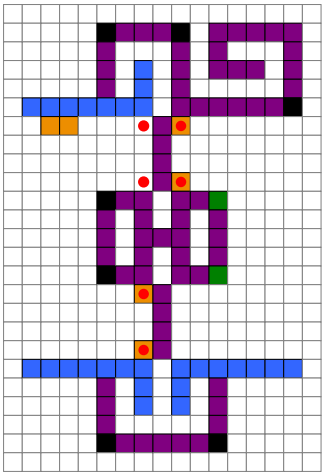


Figure 15 The branching hallway gadget for squares under the restricted model.

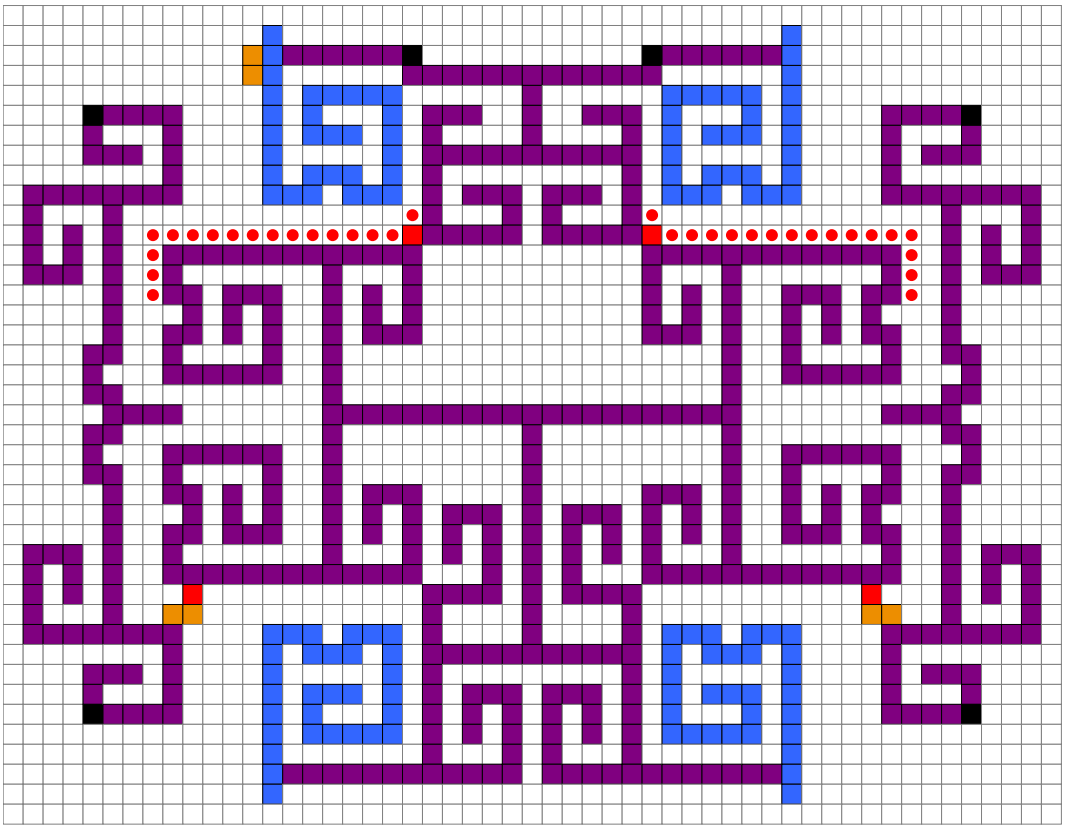


Figure 16 L2T in the open state. At this point the gadget can come from either of the top wires.

▶ **Theorem 17.** *Given two configurations of n square modules, it is PSPACE-hard to determine whether one can be reconfigured into the other in both the monkey and the leapfrog models.*

As before, the reduction is from 1-toggle-protected motion planning with the locking 2-toggle, but simpler. The main differences are as follows:

- A leapfrog move can pass through obstacles or bends without creating global cycles. All the cycles created by the agent module are local, with size at most 8, which allows us to have purely local arguments.
- Because of this change, we can now represent the agent with a single module. This eliminates the need to prove that multiple modules have to work together (and all other intricacies related to the case of a 2-module agent).
- Another interesting advantage is that we can represent a wire with two parallel sequences of modules (5 units apart). The agent will move between the two lines, which reduces the need of worrying about which side the agent is on.
- Finally, the reduction works for the leapfrog model, but even if we allow monkey moves the result holds. Thus, a single reduction will work for both models.

The gadgets we use are shown in Figure 17 and Figure 18. Due to space constraints, we defer the description and proof of correctness to [2].

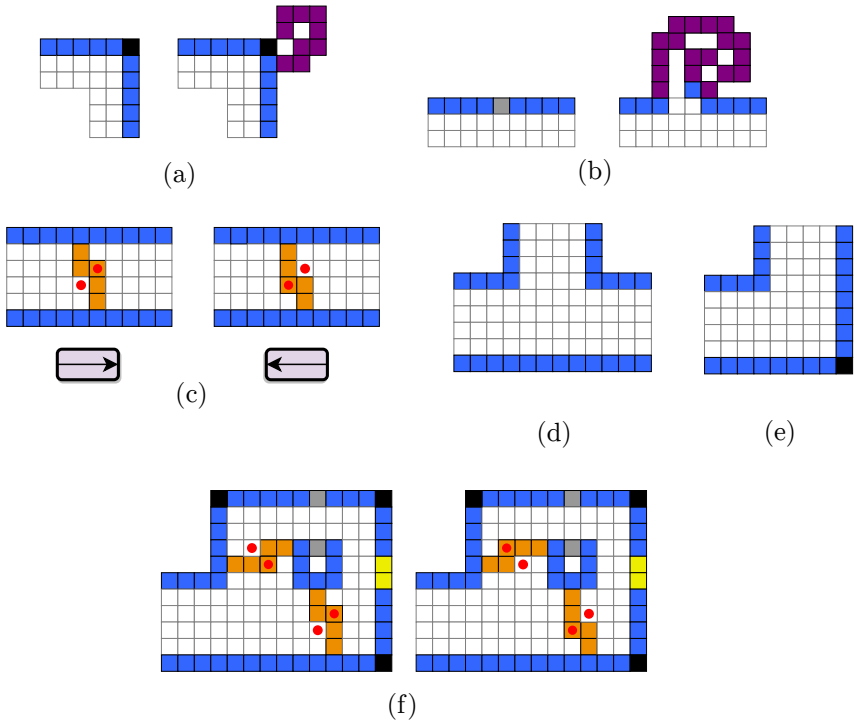


Figure 17 Gadgets used in PSPACE reduction (for leapfrog and monkey models).

4 Conclusions

This paper answers fundamental question, but also opens up further line of research. First, for hexagonal modules under the monkey model (where universal reconfiguration is possible), there is a gap between the upper bound of our algorithm (Theorem 13) and the naive $\Omega(n^2)$ lower bound (number of moves needed to transform a horizontal strip into a compact hexagon). Even if the gap is closed, then the interest would be to design a distributed algorithm and/or to consider a strategy that does many moves in parallel.

For models in which universal reconfiguration is not possible it would be nice to find a local property that would allow reconfiguration between many configurations. For example,

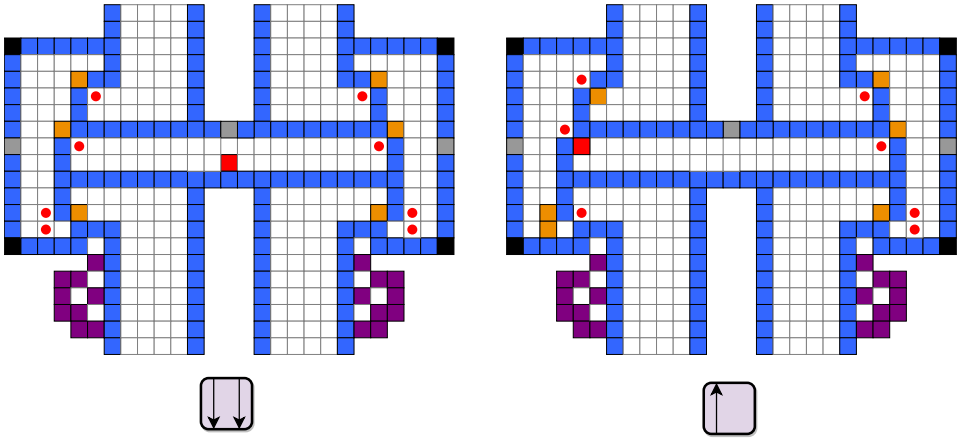


Figure 18 L2T gadget with square modules for the monkey model. In the figure two of the three possible states are shown (third one is symmetric).

with square modules and the monkey operation, reconfiguration is possible as long as both configurations have five modules on the outer shell that can move (these modules are called *musketeers* [1]).

References

1 Hugo A. Akitaya, Esther M. Arkin, Mirela Damian, Erik D. Demaine, Vida Dujmovićc, Robin Flatland, Matias Korman, Belén Palop, Irene Parada, André van Renssen, and Vera Sacristán. Universal reconfiguration of facet-connected modular robots by pivots: The $O(1)$ musketeers. In *Proc. 27th Annual European Symposium on Algorithms (ESA)*, volume 144, pages 3:1–3:14, 2019. doi:10.4230/LIPIcs.ESA.2019.3.

2 Hugo A. Akitaya, Erik D. Demaine, Andrei Gonczi, Dylan H. Hendrickson, Adam Hesterberg, Matias Korman, Oliver Korten, Jayson Lynch, Irene Parada, and Vera Sacristán. Characterizing universal reconfigurability of modular pivoting robots. *CoRR*, abs/2012.07556, 2020. arXiv:2012.07556.

3 Greg Aloupis, Sébastien Collette, Mirela Damian, Erik D. Demaine, Robin Flatland, Stefan Langerman, Joseph O'Rourke, Val Pinciu, Suneeta Ramaswami, Vera Sacristán, and Stefanie Wuhrer. Efficient constant-velocity reconfiguration of crystalline robots. *Robotica*, 29(1):59–71, 2011. doi:10.1017/S026357471000072X.

4 Joshua Ani, Erik D. Demaine, Dylan H. Hendrickson, and Jayson Lynch. Trains, games, and complexity: 0/1/2-player motion planning through input/output gadgets. *CoRR*, abs/2005.03192, 2020. arXiv:2005.03192.

5 Nora Ayanian, Paul J. White, Ádám Hálász, Mark Yim, and Vijay Kumar. Stochastic control for self-assembly of XBots. In *Proc. ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference (IDETC-CIE)*, pages 1169–1176, 2008. doi:10.1115/DETC2008-49535.

6 Jose Balanza-Martinez, Austin Luchsinger, David Caballero, Rene Reyes, Angel A Cantu, Robert Schweller, Luis Angel Garcia, and Tim Wylie. Full tilt: Universal constructors for general shapes with uniform external forces. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2689–2708, 2019.

7 Nadia M. Benbernou. *Geometric Algorithms for Reconfigurable Structures*. PhD thesis, Massachusetts Institute of Technology, 2011.

8 David Caballero, Angel A. Cantu, Timothy Gomez, Austin Luchsinger, Robert Schweller, and Tim Wylie. Relocating units in robot swarms with uniform control signals is PSPACE-complete. In *Proc. 32th Canadian Conference on Computational Geometry*, 2020.

9 Erik D. Demaine, Dylan H. Hendrickson, and Jayson Lynch. Toward a general complexity theory of motion planning: Characterizing which gadgets make games hard. In *Proc. 11th Innovations in Theoretical Computer Science Conference (ITCS)*, volume 151, pages 62:1–62:42, 2020. doi:10.4230/LIPIcs.ITCS.2020.62.

10 Adrian Dumitrescu and János Pach. Pushing squares around. *Graphs and Combinatorics*, 22(1):37–50, 2006. doi:10.1007/s00373-005-0640-1.

11 Adrian Dumitrescu, Ichiro Suzuki, and Masafumi Yamashita. Motion planning for metamorphic systems: feasibility, decidability, and distributed reconfiguration. *IEEE Transactions on Robotics*, 20(3):409–418, 2004. doi:10.1109/TRA.2004.824936.

12 Robert Fitch, Zack Butler, and Daniela Rus. Reconfiguration planning for heterogeneous self-reconfiguring robots. In *Proc. 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 3, pages 2460–2467, 2003. doi:10.1109/IROS.2003.1249239.

13 Irina Kostitsyna, Irene Parada, Willem Sonke, Bettina Speckmann, and Jules Wulms. Compacting squares. Manuscript, 2020.

14 Tom Larkworthy and Subramanian Ramamoorthy. A characterization of the reconfiguration space of self-reconfiguring robotic systems. *Robotica*, 29(1):73–85, 2011. doi:10.1017/S0263574710000718.

15 An Nguyen, Leonidas J. Guibas, and Mark Yim. Controlled module density helps reconfiguration planning. In *Algorithmic and Computational Robotics: New Dimensions (2000 WAFR)*, pages 23–36. A. K. Peters, 2001.

16 Cynthia Sung, James Bern, John Romanishin, and Daniela Rus. Reconfiguration planning for pivoting cube modular robots. In *Proc. 2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1933–1940, 2015. doi:10.1109/ICRA.2015.7139451.