Hierarchical Shape Construction and Complexity for Slidable Polyominoes under Uniform External Forces*

Jose Balanza-Martinez† David Caballero† Angel A. Cantu† Mauricio Flores†
Timothy Gomez† Austin Luchsinger† Rene Reyes† Robert Schweller†
Tim Wylie†

Abstract

Advances in technology have given us the ability to create and manipulate robots for numerous applications at the molecular scale. At this size, fabrication tool limitations motivate the use of simple robots. The individual control of these simple objects can be infeasible. We investigate a model of robot motion planning, based on global external signals, known as the tilt model. Given a board and initial placement of polyominoes, the board may be tilted in any of the 4 cardinal directions, causing all slidable polyominoes to move maximally in the specified direction until blocked.

We propose a new hierarchy of shapes and design a single configuration that is *strongly universal* for any $w \times h$ bounded shape within this hierarchy (it can be reconfigured to construct any $w \times h$ bounded shape in the hierarchy). This class of shapes constitutes the most general set of buildable shapes in the literature, with most previous work consisting of just the first-level of our hierarchy. We accompany this result with a $O(n^4 \log n)$-time algorithm for deciding if a given hole-free shape is a member of the hierarchy. For our second result, we resolve a long-standing open problem within the field: We show that deciding if a given position may be covered by a tile for a given initial board configuration is PSPACE-complete, even when all movable pieces are 1×1 tiles with no glues. We achieve this result by a reduction from Non-deterministic Constraint Logic for a one-player unbounded game.

1 Introduction

Advances in technology have given us the ability to create and manipulate robots for numerous applications at the molecular scale. At this size, fabrication tool limitations often make precise construction of the objects infeasible and thus self-assembly methods have been employed. This allows the creation of simple primitives that come together algorithmically to form the desired objects [11]. Similarly, the individual control of these simple nano-scale objects can be infeasible due to the cost or the desired simplicity of the building blocks. Thus, a growing area of interest is robot motion planning based on global external signals, known as the tilt model [4, 5]. This simple model gives all robots the same movement signal, which means all robots can be identical, and only their location and the geometry of the board need to be considered for fabrication.

The tilt self-assembly model consists of a 2D grid board with "open" and "blocked" spaces, as well as a set of slidable polyominoes placed on the board. The model uses a global external force to give all movable particles the same movement instruction. This may be done through any external force such as a magnetic field or gravity. This model has a history of assuming gravity as the external global force, hence the term *tilt*. In this model, a board may be *tilted* (an application of the global external force) in any of the four cardinal directions, causing all slidable polyominoes to slide maximally in the respective direction until reaching an obstacle. Various puzzle games exist which utilize the mechanics of this model; the maximal movement of [1] and the global movement signals in [2].

1.1 Related Work.

In [8], the authors introduced a class of shapes constructed by "dropping" pixels on a fixed seed from any of the four cardinal directions. Along with this new class of "drop-shapes", they presented a polynomial-time algorithm to check if a given hole-free shape is in this drop class (i.e., it can be constructed via this pixel dropping method). Given a hole-free shape, they showed how to construct a tilt model micro-factory which can build the shape. In [10] the authors provide an algorithm to determine if any given shape is in the class of drop shapes with the run time being exponential in the number of holes.

*This research was supported in part by National Science Foundation Grant CCF-1817602.

†Department of Computer Science, University of Texas Rio Grande Valley, Edinburg, TX 78539-2999, USA

Shape Class	Result	Theorem
Hole-free Level-1 Drop Shapes ($\mathfrak{D}_1$)	$\mathcal{O}(n \log n)$	Thm. 5 in [8]
Level-1 Drop Shapes ($\mathfrak{D}_1$)	$\mathcal{O}(n^2 h!)$	Alg. 1 in [10]
Straight 2-Cuttable Hole-free Shape	$\mathcal{O}(n^3 \log n)$	Thm. 2 in [12]
Straight 2-Cuttable Shape with Convex Holes	$\mathcal{O}(n^4 \log n)$	Thm. 2 in [12]
Hole-free Drop Shapes ($\mathfrak{D}$)	$\boldsymbol{\mathcal{O}(n^4 \log n)}$	**Thm. 3.1**

Table 1: Known results for deciding membership in the drop-shape hierarchy of a size-n shape (We refer to size as the number of tiles.). h denotes the number of holes in the shape. Previous results only decided shapes in $\mathfrak{D}_1$. With holes, no polynomial time algorithm is known even for $\mathfrak{D}_1$.

Class	Universal	Tilts	Board Size	Theorem
Drop Shapes ($\mathfrak{D}_1$)	No	$\mathcal{O}(hw)$	$\mathcal{O}(h^3 w^3)$	Thm. 6 in [8]
	Yes	$\mathcal{O}(h^2 w)$	$\mathcal{O}(h^2 w)$	Thm. 4.1 in [3]
Drop Shape Hierarchy ($\mathfrak{D}$)	No	$\mathcal{O}(hw)$	$\mathcal{O}(h^2 w^2)$	Thm. 3 in [12]
	Yes	$\boldsymbol{\mathcal{O}(h^3 w^2)}$	$\boldsymbol{\mathcal{O}(h^2 w^2 \log^2(hw))}$	**Thm. 4.1**

Table 2: Construction results for shapes with a $h \times w$ bounding box.

The efficient construction of drop-shapes in the tilt model was explored in [12]. The authors present a construction which efficiently builds a particular set of shapes in sublinear time through the use of parallelization. They introduce a hierarchical process whereby multi-tile subassemblies may be successively combined in a drop fashion. They define two classes of shapes 2-cuttable and straight 2-cuttable as any polyomino that can be decomposed into monotone subpolyominoes using valid 2-cuts and valid straight 2-cuts respectively. They also give a polynomial time algorithm for determining if a given simple polyomino, or one with convex shaped holes, is straight 2-cuttable. They also have a polyomial time algorithm for finding if a valid 2-cut exists in the same two types of shapes. Given a shape in their buildable class, a tilt model configuration could be generated to assemble that shape through a sequence of tilts.

This drop-shape construction work was extended in [3]. Rather than focusing on designing configurations for specific drop-shapes, the authors create a general drop-shape constructor. They introduced the concept of *universal* constructors in the tilt model. These are constructors which, starting from one initial configuration, can construct any shape from a particular set. In this work, a configuration that is universal for the set of drop-shapes was created.

These construction results are usually paired with complexity results to decide what can be constructed. One of the most natural questions in the tilt model asks if a particular location on a board may ever be occupied by a particle from the starting configuration. This problem, known as the *occupancy problem*, was first introduced in [4]. The authors proved the problem was NP-hard, even when considering the restricted case of a configuration with only 1×1 tiles. The authors also showed that a dual-rail logic fanout was not possible with only 1×1 tiles, which seemed to serve as evidence that this problem was not PSPACE-complete. Following, in [3] the authors proved that the occupancy problem is PSPACE-complete if a single "large" polyomino is allowed (they used a single 2×2 polyomino along with 1×1 tiles).

1.2 Our Contributions. In this paper, we formalize the notion of hierarchical construction (first started in [12]) by presenting a hierarchy of shape classes. In Section 3, we formally define a drop-shape hierarchy. We also present a $O(n^4 \log n)$-time algorithm for determining if a given hole-free shape is in this hierarchy. To accompany this result, we present the design for a configuration that is *strongly universal* for any shape within the hierarchy in Section 4. This result constitutes the most general set of buildable shapes in the literature, with previous work consisting of just the first level of our hierarchy. The second main result of this paper is in Section 5, where we resolve an open problem from [4] by showing that the occupancy problem, contrary to expectation, is PSPACE-complete even when restricting all polyominoes to be 1×1 tiles.

Problem	Polyominoes	Result	Theorem
Occupancy/Relocation	1×1	NP-hard	Thm. 1 in [4]
	1×1, 2×2	PSPACE-Complete	Thm. 5.1 in [3]
	1×1	**PSPACE-Complete**	**Thms. 5.1, 5.1**
Reconfiguration Minimization	1×1	PSPACE-Complete	Thm. 4 in [6]
Reconfiguration	1×1, 2×2	PSPACE-Complete	Thm. 6.1 in [3]
	1×1	**PSPACE-Complete**	**Thm. 5.2**

Table 3: Known complexity results in the full tilt model. All current results for occupancy also imply hardness for relocation. Reconfiguration Minimization is finding the minimum length tilt sequence. Each of the results that use a 2×2 polyomino only uses a single one and multiple 1×1s.

2 Model Preliminaries

Board. A *board* (or *workspace*) is a rectangular region of the 2D square lattice in which specific locations are marked as *blocked*. Formally, an $m \times n$ board is a partition $B = (O, W)$ of $\{(x, y) | x \in \{1, 2, \ldots, m\}, y \in \{1, 2, \ldots, n\}\}$ where O denotes a set of *open* locations, and W denotes a set of *blocked* locations- referred to as "concrete" or "walls." We classify the different possible board geometries according to the following hierarchy:

- Connected:[1] A board is said to have *connected* geometry if the set of open spaces O for a board is a connected shape.
- Simple:[2] A connected board is said to be *simple* if O has genus-0.
- Monotone: A simple board is *monotone* if O is either horizontally monotone or vertically monotone.
- Convex: A monotone board is *convex* if O is both horizontally monotone and vertically monotone.
- Rectangular: A convex board is *rectangular* if O is a rectangle.

Our board definitions have changed since [3] in order to create the hierarchy shown above.

Tiles. A tile is a labeled unit square centered on a non-blocked point on a given board. Formally, a tile is an ordered pair (c, a) where c is a coordinate on the board, and a is an attachment label. Attachment labels specify which types of tiles will stick together when adjacent, and which have no affinity. For a given alphabet of labels Σ, and some *affinity* function $G : \Sigma \times \Sigma \to \{0, 1\}$ which specifies which pairs of labels attract ($G(a, b) = 1$) and which do not ($G(a, b) = 0$), we say two adjacent tiles with labels a and b are *bonded* if $G(a, b) = G(b, a) = 1$.

Slidable Polyominoes. A *slidable polyomino* is a finite set of tiles $P = \{t_1, \ldots t_k\}$ that is 1) connected with respect to the coordinates of the tiles in the slidable polyomino and 2) *bonded* in that the graph of tiles in P with edges connecting bonded tiles is connected. When the context is clear, we often refer to these simply as polyominoes. A slidable polyomino that consists of a single tile is informally referred to as a "tile".

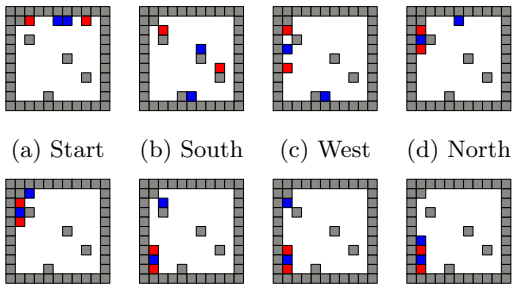
Configurations. A configuration is an arrangement of polyominoes on a board such that there are no overlaps among polyominoes, or with blocked board spaces. Formally, a configuration $C = (B, P = \{P_1 \ldots P_k\})$ consists of a board B, along with a set of non-overlapping polyominoes P that each do not overlap with the blocked locations of board B.

Step. A *step* is a way to turn one configuration into another by way of a global signal that moves all polyominoes in a configuration one unit in a direction $d \in \{N, E, S, W\}$ when possible without causing an overlap with a blocked location, or another polyomino. Formally, for a configuration $C = (B, P)$, consider the translation of all polyominoes in P by 1 unit in direction d. If no overlap with blocked board spaces occurs, then the new configuration is derived by first performing this translation, and then merging each pair of polyominoes that each contain one tile from a now (adjacently) bonded pair of tiles. If an overlap does occur, for each polyomino for which the translation causes an overlap with a blocked space, temporarily add these polyominoes to the set of blocked spaces and repeat. Once the translation induces no overlap with blocked spaces, execute the translation and merge polyominoes based on newly

[1] In [3], this hierarchy level was known as *complex*. We modify this class to allow for a proper hierarchy of shape classes.

[2] In [3], they define simple geometry based on the connectivity of W. We also modify the concept for hierarchical purposes.

bonded tiles to arrive at the new configuration. If all polyominoes are marked as blocked spaces, then the step transition does not change the initial configuration. If a configuration does not change under a step transition for direction d, we say the configuration is *d-terminal*. In the special case that a step causes a polyomino (or subpolyomino) to leave the board, we remove the polyomino from the configuration.



(a) Start (b) South (c) West (d) North

(e) West (f) South (g) West (h) South

Figure 1: Tilt Example

Tilt. A *tilt* in direction $d \in \{N, E, S, W\}$ for a configuration is executed by repeatedly applying a step in direction $d \in \{N, E, S, W\}$ until a d-terminal configuration is reached. We say that a configuration C can be *reconfigured in one move* into configuration C' (denoted $C \rightarrow_1 C'$) if applying one tilt in some direction d to C results in C'. We define the relation $\rightarrow_*$ to be the transitive closure of $\rightarrow_1$. Therefore, $C \rightarrow_* C'$ means that C can be reconfigured into C' through a sequence of tilts.

Tilt Sequence. A *tilt sequence* is a series of tilts which can be inferred from a series of directions $D = \langle d_1, d_2, \ldots, d_k \rangle$; each $d_i \in D$ implies a tilt in that direction. For simplicity, when discussing a tilt sequence, we just refer to the series of directions from which that sequence was derived. Given a starting configuration, a tilt sequence corresponds to a sequence of configurations based on the tilt transformation. An example tilt sequence $\langle S, W, N, W, S, W, S \rangle$ and the corresponding sequence of configurations can be seen in Figure 1.

Universal Configuration. A configuration C' is universal to a set of configurations $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ if and only if $C' \rightarrow_* C_i \ \forall \ C_i \in \mathcal{C}$.

Configuration Representation. A configuration may be interpreted as having constructed a "shape" in a natural way. Define a shape to be a connected subset $S \subset \mathbb{Z}^2$. A configuration *strongly* represents S if the configuration consists of a single polyomino whose tile coordinates are exactly the points of some translation of S. A weaker version allows for some "helper" polyominoes to exist in the configuration and not count towards the represented shape. In this case, we say a configuration *weakly* represents S.

Universal Shape Builder. Given this representation, we say a configuration C' is *universal* for a set of shapes U if and only if there exists a set of distinct configurations $\mathcal{C}$ such that 1) each $u \in U$ is represented by some $C \in \mathcal{C}$ and 2) C' is universal for $\mathcal{C}$. If each $u \in U$ is strongly represented by some $C \in \mathcal{C}$, we say C' is *strongly universal* for U. Alternately, if each $u \in U$ is weakly represented by some $C \in \mathcal{C}$, we say C' is *weakly universal* for U. In a similar way, a configuration can be universal for a set of patterns.

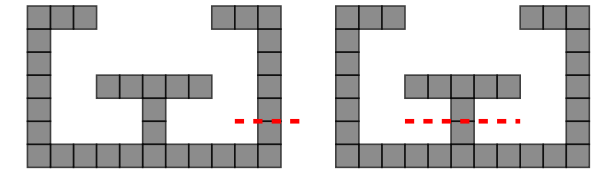
3 Drop-Shape Hierarchy

We utilize the same definitions for polyominoes and cuts as used in [12].

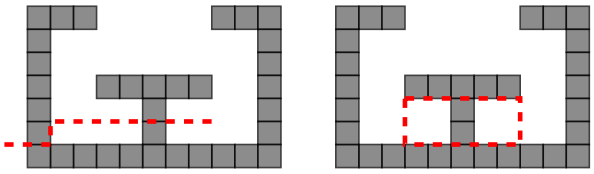
Polyomino. For a set $P \subset \mathbb{Z}^2$ of grid points in the plane, let the graph G_P be the induced grid graph in which two vertices $p_1, p_2 \in P$ are connected if they are unit distance apart. For any set P with connected grid graph G_P, a *polyomino* may be induced by replacing each point $p \in P$ with a unit square centered at p. A polyomino is said to be *hole-free* or *simple* if the induced graph $\mathbb{Z}^2 \setminus P$ is connected. We say that polyominoes which are equal up to translation have the same *shape*. We often use these terms interchangeably.

Cuts. A *cut* is an orthogonal curve moving between points of $\mathbb{Z}^2$. A p-cut is a cut that splits a polyomino P into p subpolyominoes. We say that a cut is *valid* if all of the induced subpolyominoes may be partitioned into two nonempty groups which may be pulled apart in opposite directions without blocking each other. A polyomino A is *blocked* by another polyomino B in direction d if A cannot be pulled in direction d without colliding with B (note that A sliding adjacent to tiles in B is also considered as a collision). See Figure 2 for examples of valid and invalid cuts.

Tangled Polyomino. A *tangled* polyomino is one which does not have a valid 2-cut. That is, any cut either splits the polyomino into two subpolyominoes which cannot be pulled apart (invalid), or it splits the polyomino into more than two subpolyominoes (p-cut where $p > 2$). Figure 3a provides an



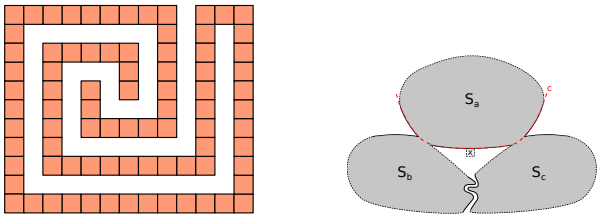
(a) Valid 2-cut (b) Invalid 2-cut



(c) Valid 3-cut (d) Invalid 3-cut

Figure 2: Examples of valid and invalid cuts for a given polyomino. It is important to note that along with direct collision, we consider a cut to be invalid if the resulting polyominoes must slide past adjacent squares.

example of a tangled polyomino.



(a) (b)

Figure 3: (a) A tangled polyomino for which no 2-cut exists. (b) An abstract representation of tangled polyomino S from Lemma 3.2 which depicts the required properties as stated in the proof.

Drop Construction Step. A drop construction step is described by a direction $\{N, E, S, W\}$ and a latitude or longitude l representing the column/row that a shape is placed. The shape arrives from (l, ∞) for north, $(l, -\infty)$ for south, (∞, l) for east, and $(-\infty, l)$ for west. The shape moves along the row/column until it reaches the first grid position adjacent to to an existing tile. We will refer to this as dropping a tile or shape.

Drop-Shape Hierarchy. Here we define the hierarchy of drop-shapes (denoted $\mathfrak{D}$). We use notation $\mathfrak{D}_h$ to denote level-h of the drop-shape hierarchy. Beginning with level-1, the levels of our hierarchy are defined recursively, containing the specified elements and nothing else:

- Level-0 ($\mathfrak{D}_0$). The single tile.
- Level-1 ($\mathfrak{D}_1$). The set of shapes in $\mathfrak{D}_1$ is defined recursively:
  - (Base) $\mathfrak{D}_0 \subset \mathfrak{D}_1$
  - (Drop Combinations) For any $A \in \mathfrak{D}_1$, any polyomino C that is produced by dropping a single tile onto A is also in $\mathfrak{D}_1$.
- Level-h ($\mathfrak{D}_h$). In general, the set of shapes in $\mathfrak{D}_h$ is defined recursively:
  - (Base) $\mathfrak{D}_{h-1} \subset \mathfrak{D}_h$
  - (Drop Combinations) For any $A \in \mathfrak{D}_{h-1}$ and $B \in \mathfrak{D}_h$, any polyomino C that is produced by dropping A onto B is also in $\mathfrak{D}_h$.
- Strict Level-h ($\widehat{\mathfrak{D}}_h$): The set of shapes that are in $\mathfrak{D}_h$, but not in $\mathfrak{D}_{h-1}$. Formally, $\widehat{\mathfrak{D}}_h = \mathfrak{D}_h \setminus \mathfrak{D}_{h-1}$.
- The Hierarchy ($\mathfrak{D}$): The hierarchy is defined as the union of all levels. Formally, $\mathfrak{D} = \bigcup_{i=0}^{\infty} \mathfrak{D}_i$

Since tangled polyominoes cannot be decomposed, no tangled polyomino is in the hierarchy.

Decomposition Tree. A decomposition tree for a given polyomino P is a rooted binary tree of polyominoes with root node being P, each leaf being either a single tile or a tangled polyomino, and all non-leaf nodes having two children consisting of two polyominoes that make a valid 2-cut for the polyomino at the given node. A decomposition tree is said to be *atomic* if all leaves are single tiles.

Strict Decomposition Tree. A decompostion tree is said to be a Strict Decomposition Tree if for every node p,either $p \in \mathfrak{D}_0$, or for some $h > 0$, $p \in \widehat{\mathfrak{D}}_h$ and has children $p_i \in \mathfrak{D}_h$ and $p_j \in \mathfrak{D}_{h-1}$.

Singly-Connected Tile. A tile t in polyomino P is said to be singly-connected w.r.t. P if and only if it is connected to $P - t$ on only one of its four edges.

3.1 Membership in the Drop-Shape Hierarchy.

A key contribution of this paper is a board configuration that is universal for all shapes in the drop-shape hierarchy of up to a given size n. This leads to the natural question of how efficiently can we decide if a given polyomino is a member of the drop-shape hierarchy $\mathfrak{D}$. $\mathfrak{D}$ is equivalent to the set of 2-cuttable

shapes defined in [12]. There, the authors present an algorithm for determining if a shape is in the set of *straight* 2-cuttable shapes, which is known to be a subset of $\mathfrak{D}$. It is currently unknown if the set of straight 2-cuttable shapes is equivalent to $\mathfrak{D}$. In this section we develop an efficient algorithm for deciding membership in $\mathfrak{D}$ for the case of genus-0 polyominoes. We first formally define the drop-shape hierarchy membership problem. We then prove two technical lemmas (Lemmas 3.1, 3.2) that will be used to establish the correctness of an efficient algorithm for deciding membership, provided in Theorem 3.1.

Drop-Shape Hierarchy Membership Problem. The drop-shape hierarchy membership problem asks: Given a polyomino P, is $P \in \mathfrak{D}$?

LEMMA 3.1. *Given a hole-free polyomino P, $P \in \mathfrak{D}$ if and only if there exists an atomic decomposition tree for P.*

Proof. This proof builds off of Theorem 2 from [7], where they show that decomposition is the reverse of construction. Clearly, P is in the drop-shape hierarchy if such a decomposition tree exists, as the reverse drop construction step sequence can be performed to yield P. If no such decomposition tree exists, this means that all decomposition trees contain at least one leaf which is a tangled polyomino. Since a tangled polyomino cannot be in the hierarchy (as there are no valid 2-cuts), there must always exist a drop construction step which uses a polyomino that is not in the hierarchy. □

LEMMA 3.2. *If there exists a decomposition tree for hole-free polyomino P which has a tangled polyomino S as a leaf, S must be a leaf in every decomposition tree of P.*

Proof. Given that there exists a decomposition tree T_A which has tangled shape S as a leaf, assume that some other decomposition tree T_B exists which does not have S as a leaf. This implies that T_B uses some decomposition sequence which involves a polyomino P' such that $S \subset P' \subseteq P$ and there exists a 2-cut for P' which also goes through subpolyomino S.

Let c be a valid 2-cut for P' which also cuts through S. Since we know that S by itself is tangled, one of the following must be true: 1) c was an invalid 2-cut through S which is now valid because of the additional tiles introduced by $P' - S$. 2) c was a valid p-cut (with $p > 2$) through S which has now become a valid 2-cut because of the additional tiles introduced by $P' - S$. Since the addition of more tiles can never unblock a cut, we see that the first case cannot be true. The second case is not so straightforward, and requires more investigation.

First, consider the stand-alone polyomino S. We know c was a valid p-cut through S (with $p > 2$) which has now become a valid 2-cut through P'. For now, let it suffice to assume that c was a valid 3-cut through S. We will generalize to p later. This means that c cuts S into 3 subpolyominoes $\{S_a, S_b, S_c\}$ that can be partitioned into two sets which may be pulled apart in opposite directions without blocking each other. W.l.o.g., let that partition be $\{\{S_a\}, \{S_b, S_c\}\}$. Figure 3b shows a sketch that highlights the essential properties S must have. Since c is a 3-cut through S, we know that there must exist a path of empty spaces which separates S_b from S_c and begins at an empty space x along c which is adjacent to S_a. Furthermore, we know that subpolyominoes S_b and S_c must both block each other in the direction that $\{S_b, S_c\}$ is pulled apart from $\{S_a\}$; otherwise, a valid 2-cut would have existed in S (by pulling S_b or S_c off of S by itself).

Now, consider the polyomino P' with subpolyomino S. If location x contained a tile in polyomino P', we know that said tile would be blocked in each of the four directions by one of S's three subpolyominoes (there would be no way to remove the tile without also removing some portion of S). This, along with the fact that S is a node in the decomposition tree T_a, shows that location x must be empty in polyomino P'. In order for c to be a 3-cut through S and a 2-cut through P', the induced subpolyominoes S_b and S_c must be connected via a path of tiles in $P' - S$. Clearly, this path cannot exist on S_a's side of c, as it would make c an invalid 2-cut for P'. So, this path must connect S_b and S_c on their side of c. Since location x cannot contain a tile in P' we see that the connection between S_b and S_c must cut off x's connection to the outside plane. This cannot be the case, however, because P' is a hole-free polyomino.

So, c could not have simultaneously been a valid valid 2-cut through P' and valid 3-cut through S. It is easy to observe that any valid p-cut through S (which is also a valid 2-cut through P') must have at least one pair of induced polyominoes with the same properties as S_b and S_c, and all others may be considered as S_a for the purposes of this proof. Thus, no valid 2-cut through P' may cut through S. This implies that decomposition tree T_B must not exist, as was originally assumed, and every decomposition

tree must contain S as a leaf. □

THEOREM 3.1. *A hole-free size-n polyomino P can be checked for membership in the drop-shape hierarchy in $O(n^4 \log n)$ time.*

Proof. This algorithm is straightforward. Given hole-free polyomino P, perform the following steps:

1. **Base Case:** If P is a single tile, return *yes*. If P is tangled, return *no*. Otherwise, continue to step 2.
2. Select a valid 2-cut which cuts P into subpolyominoes A and B. Continue to step 3.
3. Recurse on A and B by performing to step 1 for each of them. If both return *yes*, return *yes*; otherwise, return *no*.

We see that this algorithm creates a decomposition tree for polyomino P. The proof of correctness follows from the previous lemmas. If all leaves in the decomposition tree are single tiles, Lemma 3.1 tells us that P is in the hierarchy. If the decomposition tree contains a leaf which is a tangled polyomino, Lemmas 3.1 and 3.2 together show that P is not in the hierarchy.

The runtime is achieved as follows: Since a decomposition tree for a size-n hole-free polyomino P has at most n leaves, we know the tree has at most $2n-1$ nodes. Therefore, we can say that our algorithm is called at most $2n-1$ times. Step 1 of the algorithm checks if the polyomino is tangled (i.e., it checks if P has a valid 2-cut). Theorem 5 from [12] shows that a valid 2-cut can be found for a hole-free size-n polyomino in $O(n^3 \log n)$ time. So, our algorithm is called at most $2n-1$ times, with each call spending at most $O(n^3 \log n)$ time to find a valid 2-cut. Thus, we achieve a runtime of $O(n^4 \log n)$ to determine if P is in the drop-shape hierarchy. □

3.2 Hierarchy Characteristics. In this section we derive some key properties of the drop-shape hierarchy. In particular, in Theorem 3.2 we show that any member of the hierarchy of size-n must be a member of level $\mathfrak{D}_{\lfloor \log |P| \rfloor}$. This result allows us to efficiently bound the size of the board needed to assembly size-n polyominoes in Section 4. Next, in Theorem 3.3 we establish that the drop-shape hierarchy is a true hierarchy, i.e. that each level h of the hierarchy contains shapes that are not members of any level lower than h.

LEMMA 3.3. *For any polyomino $P \in \widehat{\mathfrak{D}}_H$ there exists a Strict decomposition Tree, for any $H \geq 0$.*

Proof. We will prove this by induction on the size of the shape. For our base case we know any single tile has a strict decomposition tree.

Now for our inductive step assume that there exists a strict decomposition tree for any polyomino p where $1 \leq |p| \leq n$ for some n. Consider a polyomino $P \in \widehat{\mathfrak{D}}_H$ where $|P| = (n+1)$. We know that since $P \in \widehat{\mathfrak{D}}_H$ there exists two polyominoes $P_a \in \mathfrak{D}_H$ and $P_b \in \mathfrak{D}_{H-1}$ that can be dropped onto each other to build P. We know that since both P_a and P_b are subpolyominoes of P, $n \geq |P_a|$ and $n \geq |P_b|$. From our inductive step we can assume that both P_a and P_b have strict decomposition trees and the only new node we add is P which has the properties required by a strict decomposition tree.

From here we see that every polyomino in a strict level $H \geq 0$ has a strict decomposition tree. □

LEMMA 3.4. *For any polyomino $P \in \widehat{\mathfrak{D}}_H$ such that $H > 0$, any strict decomposition tree of P must have a node P' with two children P_l and P_r such that $P' \in \widehat{\mathfrak{D}}_H$ and both $P_l \in \widehat{\mathfrak{D}}_{H-1}$ and $P_r \in \widehat{\mathfrak{D}}_{H-1}$.*

Proof. First we will prove there must exist a node P' with two children P_l and P_r such that $P' \in \widehat{\mathfrak{D}}_H$ and both $P_l \in \mathfrak{D}_{H-1}$ and $P_r \in \mathfrak{D}_{H-1}$, then we will show why they both must be strict.

Let us first look at the root P. If both children of P are in $\mathfrak{D}_{H-1}$ than P is the node we are describing. Now consider the case where p is a child of P and $p \notin \mathfrak{D}_{H-1}$. Since we are looking at a strict decomposition tree we know that $p \in \widehat{\mathfrak{D}}_H$. Since the tree rooted at p is also a strict decomposition tree we have the same two cases. However since a strict tree is also an atomic tree all the leaves of the tree must be single tiles so eventually the first case must be true.

Now we will show $P_l \notin \mathfrak{D}_{H-2}$ and $P_r \notin \mathfrak{D}_{H-2}$. Without loss of generality assume $P_l \in \mathfrak{D}_{H-2}$, this would mean that P' can be built by dropping a shape in $\mathfrak{D}_{H-2}$ onto a shape in $\mathfrak{D}_{H-1}$ which by definition would mean $P' \in \mathfrak{D}_{H-1}$ which we know is not true since $P' \in \widehat{\mathfrak{D}}_H$.

Since $P_l \in \mathfrak{D}_{H-1}$ and $P_r \in \mathfrak{D}_{H-1}$, and $P_l \notin \mathfrak{D}_{H-2}$ and $P_r \notin \mathfrak{D}_{H-2}$ we can see that $P_l \in \widehat{\mathfrak{D}}_{H-1}$ and $P_r \in \widehat{\mathfrak{D}}_{H-1}$. Therefore there must exist a node $P' \in \widehat{\mathfrak{D}}_H$ with both children in $\widehat{\mathfrak{D}}_{H-1}$. □

LEMMA 3.5. *For any polyomino $P \in \widehat{\mathfrak{D}}_H$, $|P| \geq 2^H$*

Proof. We prove this by induction on H. For the base case we can see that the only strict level-0 shape is the single tile. We can see that $1 \geq 2^0$ is true.

For the inductive step assume for any $p \in \widehat{\mathfrak{D}}_H$, $|p| \geq 2^H$. Consider a polyomino $P \in \widehat{\mathfrak{D}}_{H+1}$. We know from lemma 3.3 there must exist a strict decomposition tree and from lemma 3.4 that there must exist a node in a strict decomposition tree P' that has two children $P_l, P_r \in \widehat{\mathfrak{D}}_H$. We can also see that since P_l and P_r are children of P',

$$|P'| = |P_r| + |P_l|$$

From our assumption since both $P_l, P_r \in \widehat{\mathfrak{D}}_H$, $|P_r| \geq 2^H$ and $|P_l| \geq 2^H$

$$|P'| \geq 2^H + 2^H$$

$$|P'| \geq 2^{H+1}$$

Finally, since P' is a subpolyomino of P, $|P| \geq |P'|$.

$$|P| \geq 2^{H+1}$$

□

THEOREM 3.2. *For any polyomino $P \in \mathfrak{D}$, $P \in \mathfrak{D}_{\lfloor \log |P| \rfloor}$*

Proof. We can see from lemma 3.5 for any $P \in \widehat{\mathfrak{D}}_h$, $\log |P| \geq h$. We also know that for any $H \geq h, P \in \mathfrak{D}_H$. Let $H = \lfloor \log |P| \rfloor$, we can see that $P \in \mathfrak{D}_{\lfloor \log |P| \rfloor}$. □

LEMMA 3.6. *For all positive integers h, given a hole-free polyomino $P \in \mathfrak{D}_h$, for any singly-connected, non-blocked tile $t \in P$, $P - t \in \mathfrak{D}_h$.*

Proof. We will prove this is true by induction on h. Lemma 3 of [8] showed that given a hole free polyomino $P_1 \in \mathfrak{D}_1$, for any tile $t_1 \in P_1$ that is non-cut ($P_1 - t_1$ is connected), non-blocked, and convex (there exists a 2×2 square that solely contains t_1), $P_1 - t_1 \in \mathfrak{D}_1$. For simplicity we will refer to a singly-connected, non-blocked tile as a *candidate* tile. A candidate tile is convex and non-cut. In general, we will say that $C(\mathfrak{D}_h)$ is true if and only if for all polyominoes $P_h \in \mathfrak{D}_h$, for any candidate tile $t \in P_h$, $P - t \in \mathfrak{D}_h$.

The base case $C(\mathfrak{D}_1)$ was shown to be true by Becker et al. in [8]. Assume $C(\mathfrak{D}_h)$ holds. We will prove by contrapositive that $C(\mathfrak{D}_h) \implies C(\mathfrak{D}_{h+1})$.

Assume $\neg C(\mathfrak{D}_{h+1})$. Consider a polyomino $P \in \mathfrak{D}_{h+1}$, a candidate tile $t \in P$, and a polyomino $P - t = P' \notin \mathfrak{D}_{h+1}$. Let T be a strict decomposition tree of P. The first cut in T is not solely removing t, since that would result in $P' \notin \mathfrak{D}_{h+1}$ (violating the strict decomposition tree), so we will also consider T', a decomposition tree for P' that initially makes the same cuts as T. Since T is a strict decomposition tree, for the children of its root, p_l and p_r, one is in $\mathfrak{D}_{h+1}$ and the other is in $\mathfrak{D}_h$. W.L.O.G., let t exist in the child node p_l. Now, consider the nodes derived from making equivalent cuts in T', p'_l and p'_r. Since t only existed in p_l, $p_r = p'_r$. There are two cases. First, if $p_l \in \mathfrak{D}_h$. We know $p'_l \notin \mathfrak{D}_h$ because $p'_r = p_r$ and $p_r \in \mathfrak{D}_{h+1}$, but $P' \notin \mathfrak{D}_{h+1}$. This means that $p_l \in \mathfrak{D}_h$ and $t \in p_l$ was a candidate tile, and $p_l - t = p'_l \notin \mathfrak{D}_h$, and therefore $\neg C(\mathfrak{D}_h)$. The second case is $p_l \notin \mathfrak{D}_h$, but since it is a strict decomposition tree it must then be in $\widehat{\mathfrak{D}}_{h+1}$. If this is the case, then consider p_l as the root polyomino. Repeat the same process to p_l and p'_l that we did to P and P'. Since all leaves in a strict decomposition tree are single tiles, we eventually arrive at a child node p in $\mathfrak{D}_h$, where $t \in p$ is a candidate tile, and a $p - t = p' \notin \mathfrak{D}_h$. Therefore, $\neg C(\mathfrak{D}_{h+1}) \implies \neg C(\mathfrak{D}_h)$, and by contrapositive: $C(\mathfrak{D}_h) \implies C(\mathfrak{D}_{h+1})$. □

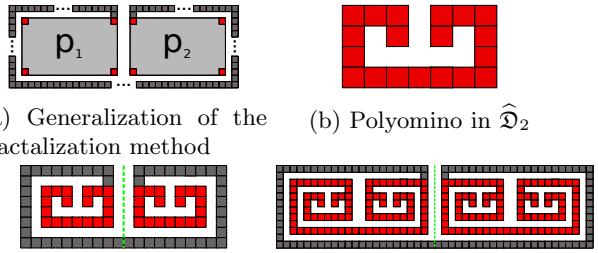
THEOREM 3.3. *For all positive integers h, there exists a polyomino in $\widehat{\mathfrak{D}}_h$.*

Proof. Polyominoes in $\widehat{\mathfrak{D}}_1$ have previously been labeled as drop shapes. This is the set of shapes that can be built by dropping only single tiles, excluding the singleton tile itself. A polyomino in $\widehat{\mathfrak{D}}_2$ can be seen in Figure 4b. We will call this polyomino P_2. We can use a method we will refer to as *fractalization* to generate a polyomino in $\widehat{\mathfrak{D}}_3$ from P_2. See Figure 4c.

To show existence of a polyomino for every strict level of the hierarchy, we will show how a polyomino $p \in \widehat{\mathfrak{D}}_h$ that was generated through repeated fractalization to P_2 can be used to create a polyomino $p' \in \widehat{\mathfrak{D}}_{h+1}$. This method takes 2 copies of p, which we will label p_1 and p_2, and places them horizontally adjacent to each other 2 unit spaces apart (without loss of generality, assume p_1 is to the left of p_2). Then, two tiles are added; one above the north-most right-most tile of p_1 and the other above the north-most left-most tile of p_2. A string of single tiles is then wrapped around both polyominoes such that there is 1 unit space between the border and the bounding boxes of the inner polyominoes. This border polyomino connects the two tiles that were added from above. The border polyomino blocks both p_1 and p_2 in all directions, creating polyomino

p', See Figure 4a.

We know that $p' \in \mathfrak{D}_{h+1}$, as it can clearly be 2-cut across the border resulting in two polyominoes in $\mathfrak{D}_h$ that can be built by dropping single tiles onto p. This cut is shown in Figure 4. We must now show that p' is not in $\mathfrak{D}_h$. If p' was in $\mathfrak{D}_h$, there would exist a valid 2-cut in which at least one of the resulting polyominoes derived from that cut was in $\mathfrak{D}_{h-1}$. This cut can not occur solely across the border connecting the two polyominoes, as the two resulting polyominoes derived from that cut are polyominoes that can be turned into p by repeatedly removing non-blocked, singly-connected tiles. Therefore by Lemma 3.6, since $p \in \widehat{\mathfrak{D}}_h$ we can see that these polyominoes can not be in $\mathfrak{D}_g$ for $g < h$. It is also clear that this cut can not occur solely within p_1 and/or p_2, as they are are now both blocked in all directions. It is also not possible for the cut to cross both the border and p_1 or p_2. First observe that due to the way these polyominoes are connected, there will never be 2×2 square fully occupied by tiles. Since there are also no holes, there is never a choice of which path to take to get from one tile to another. This means there is a single path of connectivity between any two tiles in p', and therefore a single path of connectivity between any tile in p_1 and any tile in p_2. This path will always include the border that connects the two. If the cut occurs within the border, then this path is removed, meaning there are now 2 disconnected polyominoes. W.l.o.g., assume this cut also occurs within p_1, splitting it into p_l and p_r. Since there is only one path between any two tiles, there is now no path between p_l and p_r. There is also no path between p_2 and p_l or p_r. It follows that there are now 3 disconnected polyominoes as a result of this cut, leaving this cut an n-cut, where $n = 3$. If the cut continued through p_1 or into p_2, n could increase, but never decrease. This shows that the 2-cut in question does not exist, meaning $p' \notin \mathfrak{D}_h$. Since $p' \in \mathfrak{D}_{h+1}$ and $p' \notin \mathfrak{D}_h$, $p' \in \widehat{\mathfrak{D}}_{h+1}$. This shows that the fractalization method can be repeated to generate a polyomino in $\widehat{\mathfrak{D}}_h$ for all positive integers h. Figure 4 shows the fractalization method being used to create a polyomino in $\widehat{\mathfrak{D}}_3$ and a polyomino in $\widehat{\mathfrak{D}}_4$. □



(a) Generalization of the fractalization method

(b) Polyomino in $\widehat{\mathfrak{D}}_2$

(c) Polyomino in $\widehat{\mathfrak{D}}_3$

(d) Polyomino in $\widehat{\mathfrak{D}}_4$

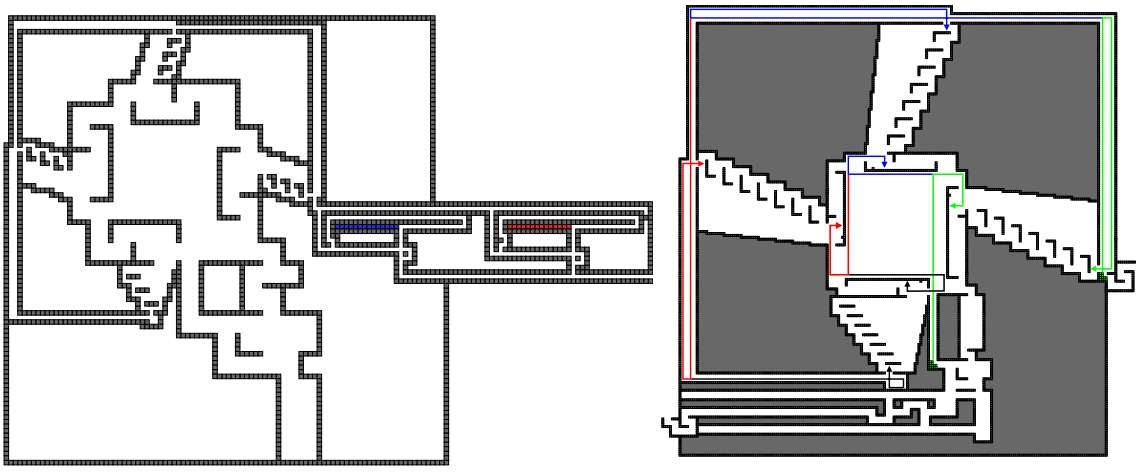
Figure 4: Fractalization method used to reach higher strict levels of the hierarchy. The gray tiles represent the single tile border created, while the red tiles represent the polyominoes in $\widehat{\mathfrak{D}}_h$ used. The green dashed line represents the 2-cut that shows this shape is in $\mathfrak{D}_{h+1}$.

4 Drop-Shape Hierarchy Constructor

This section presents a construction that builds any shape in the drop-shape hierarchy. The construction is a direct extension of [3]. In that work, a universal constructor was given for drop-shapes ($\mathfrak{D}_1$ of our hierarchy) which fit in a $w \times h$ bounding box. At a high-level, we can extend the construction with larger chambers that are functionally identical to the $\mathfrak{D}_1$ constructor, but are scaled to allow the dropping of large polyominoes.

$\mathfrak{D}_1$ Constructor. This construction (shown in Figure 5a) was introduced in [3] and is capable of building any polyomino $P \in \mathfrak{D}_1$, provided P fits within a given $w \times h$ bounding box. The high level idea for this constructor is that tiles can be extracted from the fuel chambers and dropped onto any row or column of a shape in the center construction area. For the reader, we have included a descriptions of the tile selection, direction selection, and column selection process from [3] in Figures 6 and 5b, as well as the tilt sequences for these commands in Table 4. This constructor uses a modified tilt selection gadget which allows for the disposal of fuel tiles once the desired polyomino has been built.

$\mathfrak{D}_i$ Constructor Gadget. This gadget (shown in Figure 5b) is an extension to the $\mathfrak{D}_1$ constructor. Overall, this gadget is similar to the $\mathfrak{D}_1$ constructor, but scaled to allow the dropping of $w \times h$ polyominoes rather than single tiles. Because we are dropping two multi-tile polyominoes together, we need to double the dropping area to account for any drop that might be blocked by the adjacent wall in the dropping area. However, the dimensions of the polyomino built after the drop cannot exceed the $w \times h$ bounding box. The same tilt sequences used in the $\mathfrak{D}_1$ constructor are used in this constructor. This allows for a large polyomino to be dropped onto another in the same



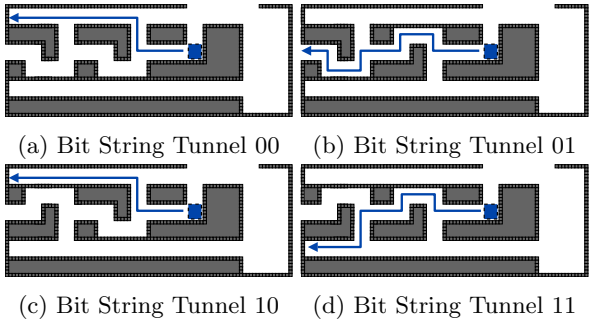
(a) $\mathfrak{D}_1$ Constructor (b) $\mathfrak{D}_i$ Constructor Gadget

Figure 5: (a) The universal drop-shape constructor from [3] which builds any shape in $\mathfrak{D}_1$. (b) The $\mathfrak{D}_i$ constructor gadget which allows the dropping of large polyominoes, it also depicts the sequence for selecting a direction to drop from. The paths from $\mathfrak{D}_i$ constructor gadget look the same in the $\mathfrak{D}_1$.

fashion as a tile is dropped in the other constructor. By attaching a series of these constructors onto a $\mathfrak{D}_1$ constructor, we can build polyominoes which are in higher levels of the hierarchy.

Bit String Tunnels. To connect the constructor chambers, we use bit string tunnels (Figure 8) which require a unique move sequence to move a polyomino from a constructor $\mathfrak{D}_i$ to a constructor $\mathfrak{D}_{i+1}$. These tunnels require a sequence of up/down tilts, which can be thought of as 0 or 1 bits. It's easy to see that a construction which builds level h shapes requires $\log h$ many up/down selectors. At the end of every bit string tunnel there is a *reset gadget* as shown in Figure 7b section 3 which enforces all subpolyominoes to be in their launch configurations once a subpolyomino has traversed from constructor to constructor.

$\mathfrak{D}_4$ Constructor. Figure 9 shows a complete constructor which can build any polyomino P s.t. $P \in \mathfrak{D}_4$ and P fits in a 4×4 bounding box. Since P can have at most 16 tiles, Theorem 3.2 tells us that $P \in \mathfrak{D}_4$, i.e., the largest hierarchy level needed is four. Thus, the bit string tunnel only needs two up/down choice to encode all of the possible tunnel transitions.



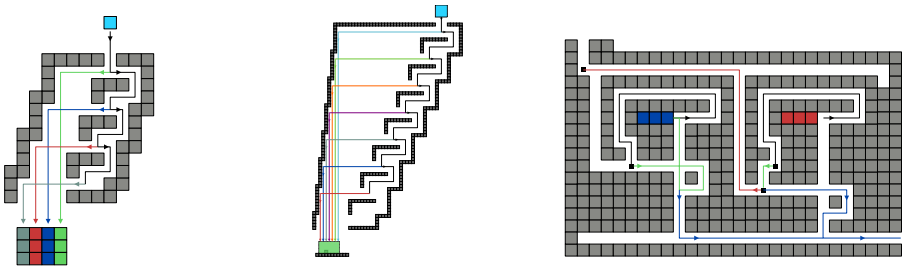
(a) Bit String Tunnel 00 (b) Bit String Tunnel 01

(c) Bit String Tunnel 10 (d) Bit String Tunnel 11

Figure 8: This Figure demonstrates a 2-bit string tunnel. Notice that the tilt sequence $\langle W, N, W, S, W, S, W, N, W \rangle$ would only allow a shape to completely traverse bit string tunnel 01.

THEOREM 4.1. *Given positive integers $w, h \in \mathbb{Z}^+$, there exists a configuration which is* strongly *universal for the set of shapes $U = \{u \mid u \in \mathfrak{D}, u$ fits in a $w \times h$ bounding box$\}$. This configuration has size $\mathcal{O}(h^2 w^2 \log^2(hw))$ and uses $\mathcal{O}(h^3 w^2)$ tilts to move into a configuration which strongly represents any shape $u \in U$.*

Proof. This is a proof by construction. We begin with a configuration C where the chambers of all $\mathfrak{D}_i$ gadgets are empty except for the fuel chambers in $\mathfrak{D}_1$. We know from [3] that the $\mathfrak{D}_1$ gadget can build any level-1 drop shape. The building process follows the flowchart in Figure 7a using the tilt sequences in Table 4. This allows building any i-level shape bounded by h and w where $i \leq \log hw$ (by Theorem 3.2). While building the desired polyomino, all subpolyominoes follow the same sequence and thus reach



(a) $\mathfrak{D}_1$ Column Selection (b) $\mathfrak{D}_i$ Column Selection (c) $\mathfrak{D}_i$ Tile Selection Gadget

Figure 6: (a) The column selection gadget for $\mathfrak{D}_1$ shapes. Assuming the shape to build is at a fixed location, this gadget allows any column to be selected to drop the new tile onto. The number of columns to drop from in this gadget determines the size of the shape we can build. Thus, this is for a drop shape within a 4×4 bounding box. This gadget is repeated on each of the four sides of the drop-shape constructor (with a slightly modified one on the south side in order to allow a non-conflicting move sequence.). (b) The column selection gadget for a 4×4 $\mathfrak{D}_i$ constructor. Notice that the tunnels are large enough to accommodate shapes that fit in a 4×4 bounding box, and the attachment area is twice as big as the $\mathfrak{D}_1$ version. This is to allow large polyominoes to be dropped onto any row/column of another large polyomino. (c) The fuel selection gadget for $\mathfrak{D}_1$. Each tile is pulled out with the sequence $\langle E, N, W, S, E, S\rangle$, and stops at the first square. Then the left tile type (blue) is either pulled out of the gadget or put back in the storage area. This shows it being added back to the storage with $\langle E, S, W, N, W, S\rangle$. This sequence puts the next tile type (red) in a decision location. The red tile is selected with the sequence $\langle W, N, W, S, W, S\rangle$. Once the desired shape has been built, one can remove the remaining fuel tiles off the board with $\langle E, N, W, S, E, S, W, S, E, S, W, S, E, W\rangle$. This sequence extracts both tile types off the storage area, and then removes them off the board.

the same position in their respective constructors simultaneously. Once a subpolyomino needs to move from a $\mathfrak{D}_i$ to a $\mathfrak{D}_{i+1}$ constructor, all subpolyominoes are sent to their corresponding bit string tunnel. The uniqueness of each tunnel guarantees only one subpolyomino successfully traverses towards the next constructor while the others are held back in the bit selector tunnel. The reset gadget at the end of each bit string tunnel enforces sequence R in Table 4 which sets every polyomino to its launch position.

Once the desired polyomino has been built and is located in the $\mathfrak{D}_i$ constructor, we can perform sequence D until there is no more fuel pieces in the fuel chamber. We then have configuration C' from C by performing a series of tilts. Consequentially, $C \rightarrow_* C'$. The overall process of transitions from the starting configuration to the configuration that represents shape $u \in U$, is shown in Figure 7a. □

5 Occupancy, Relocation, and Reconfiguration Complexity

Here, we prove that the occupancy, relocation, and reconfiguration problems are PSPACE-complete when limited to only 1×1 tiles by a polynomial time reduction from Non-Deterministic Constraint Logic. The occupancy problem asks whether a given position within a given board configuration can be occupied by a polyomino for some tilt sequence. Relocation asks whether a given position may be occupied by a specific given polyomino. And Reconfiguration asks if a given initial board configuration may be converted into a second given board configuration. The occupancy problem was originally defined by Becker et al. using only 1×1 tiles. They showed it is NP-hard and the minimum move sequence for reconfiguration is PSPACE-complete [4, 6]. The authors also showed the impossibility of a fan out with dual rail logic which could be viewed as evidence against the problem being PSPACE-complete. However, recent work showed that with even a single additional 2×2 polyomino, the relocation and reconfiguration problems are PSPACE-complete [3]. In this section we definitively answer the question with only 1×1 tiles and show all three problems to be PSPACE-complete with only 1×1 tiles. To achieve this result we provide a polynomial time reduction from Non-Deterministic Constraint Logic [9]. The formal problem definitions are as follows.

Occupancy Problem. The occupancy problem asks whether or not a given location can be occupied

Command	Tilt Sequence
1. Extract blue tile (E_{Blue})	$\langle E, N, W, S, E, S, W, N, W, S, E, S, W, N, W \rangle$
2. Extract red tile (E_{Red})	$\langle E, N, W, S, E, S, E, S, W, N, E, S, W, S, W, N, W \rangle$
3. Add from east (A_E)	$\langle N, E, S, W, S \rangle + \langle S, W, N, W \rangle^j + \langle N, W, S, E, S, E, S \rangle$
4. Add from north (A_N)	$\langle N, W, N, E, S \rangle + \langle E, S, W, S \rangle^j + \langle W, S, E, N, E, S, E, S, W, \ S, E, S, E, S \rangle$
5. Add from west (A_W)	$\langle N, W, S, W, N, E \rangle + \langle N, E, S, E \rangle^j + \langle S, E, N, W, N, E, S, E, S, E, S, E, S, W, S, E, S, E, S, W, N \rangle$
6. Add from south (A_S)	$\langle N, W, S, E, S, W, N \rangle + \langle W, N \rangle^j + \langle E, N, W, S, W, N, E, S, E, S \rangle$
7. Send to BST (S_{BST})	$\langle N, W, S, E, S, E, S, W, S, E, S, W, S \rangle$
8. Traverse 1-bit (T_1)	$\langle W, N, W, S, W \rangle$
9. Traverse 0-bit (T_0)	$\langle W, S, W, N, W \rangle$
10. Reset to launch (R)	$\langle W, S, E, S, W, N, W \rangle$
11. Remove from board (D)	$\langle E, N, W, S, E, S, W, S, E, S, W, S, E, W \rangle$

Table 4: Commands E_{Blue}, E_{Red} move either a red or blue tile into launch configuration. Commands A_E, A_N, A_W, and A_S add a subpolyomino in the launch configuration into the shape being built. These said commands have been slightly modified from [3] to avoid conflicting with the remaining commands. Commands S_{BST}, T_0, and T_1 send the subpolyominoes to their respective bit string tunnels (BST's) and allow one of them to traverse through the tunnels. Command R returns back any polyominoes to their corresponding launch configuration. Finally, command D removes one red and blue tile from the board.

by any tile on the board. Formally, given a configuration $C = (B, P)$ and a coordinate $e \in B$, does there exist a tilt sequence such that $C \rightarrow_* C'$ where $C' = (B, P')$ and $\exists p \in P'$ that contains a tile with coordinate e?

Relocation. The relocation problem asks whether a specified polyomino can be relocated to a particular position. That is, given a configuration, a polyomino within that configuration, and a translation of that polyomino, does there exist a sequence of tilts which moves the original polyomino to its translation?

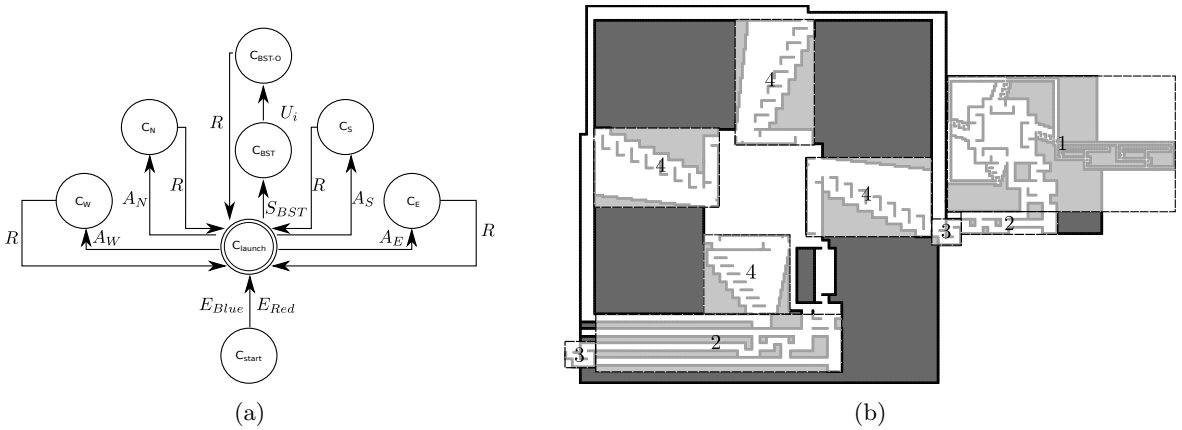
Reconfiguration. The reconfiguration problem asks whether a configuration can be reconfigured into another. Formally, given two configurations $C = (B, P)$ and $C' = (B, P')$, does there exist a tilt sequence such that $C \rightarrow_* C'$?

5.1 Non-Deterministic Constraint Logic. A constraint logic graph is a weighted directed graph with a constraint on each of the vertices [9]. The constraint specifies the minimum weight required from the edges directed in (the sum of the inflow) to any vertex. An example of two vertices can be seen in Figure 11b. When given a graph, the usual problem studied is whether a particular edge can be "flipped"- the direction of the edge changed, i.e., is there a sequence of edge flips that maintain the constraints on all vertices, and allows the target edge to be flipped? This is a one-player unbounded game. The problem is still PSPACE-complete when the edge weights are all strength 1 or 2, and vertices have max degree 3. We address the following equivalent problem.

Configuration-to-Configuration Problem. Given two states of a constraint graph G and G', does there exist a sequence of edge flips starting with G that results in G' [9].

5.1.1 Vertex Gadget. We will assume a max degree of three for all vertices, which means there are 8 possible arrangements of in/out edges. Define the vertex *state* as a label from 0 to 7 determined by the directions of its incident edges. We label each edge of a vertex $\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c$. The state is then the decimal value of a binary string of length three with each bit representing an edge ($\mathcal{E}_a\mathcal{E}_b\mathcal{E}_c$) where an edge directed inward is a 0, and an edge directed outward is a 1. Thus, the state values go from 000 to 111. We say a vertex is in a legal state if the weight of all edges pointed inward is greater than or equal to the constraint of the graph.

A vertex gadget contains a single 1×1 tile referred to as the *state tile*, a *transition area*, and a number of *state gadgets* equal to the number of legal states of that vertex. Since there are eight states, there are eight basic paths in the gadget that the state tile could be in representing the vertex's state. Figure 11a gives an example vertex gadget (a CL AND vertex) and the possible paths, and also shows the numbering of the states and the corresponding orientations of the original edges for that state. Table 5 gives the only move sequences needed for the system.



(a) (b)

Figure 7: (a) A flowchart where each state represents a set of configurations and the symbols represent sequences that can move from one state to another. The sequences for each of the symbols is shown in Table 4. The sequence marked as U_i is a unique combination of T_0 and T_1 tilt sequences for each ith constructor. Note that after performing the S_{BST}, U_i, R sequences one has successfully relocated the shape located in $\mathfrak{D}_i$ to $\mathfrak{D}_{i+1}$ and can add both shapes located in constructor $\mathfrak{D}_{i+1}$ with either of the A_N, A_E, A_S, A_W sequences. (b) This is an overview of the different sections of the hierarchy constructor. Section 1 is the $\mathfrak{D}_1$ shape constructor as shown in [3], section 2 is the bit string tunnels, section 3 is the reset gadget and section 4 is the column selection chambers for the $\mathfrak{D}_i$ constructor.

Flipping an edge is represented by a move sequence performed while in a valid state that moves the state tile from one state path to another, which happens simultaneously in two vertex gadgets since an edge connects two vertices. This edge flip happens in all vertex gadgets, but if the edge is not incident to that vertex, there is no effect on the path of the state tile.

5.1.2 State Transition Gadget. The state transition gadget is the transition area and the concrete that encode legal transitions from a given state. This will be unique to each vertex gadget. These are outlined in Figure 11a as states $0-4$. Figure 11c shows one of these areas with an explanation of the different paths. There are $|\mathcal{E}|$ levels on the right, where each level represents an edge in the graph. When a $\langle W \rangle$ command moves the state tile left, the tile stops at the blocked spot on that level. All edges not incident to the vertex have no effect on the path of the state tile (the dotted lines in Figure 11a and white rows in Figure 11c). The three edges that are incident will change the path of the state tile when a $\langle S \rangle$ command follows. If it is not a valid edge flip (due to the constraints), the state tile will be permanently stuck in a path representing an invalid state. If the new state is valid, the state tile will be in that state path.

Flip edge $e_k \in \mathcal{E}$	$\langle\langle E, S\rangle^k, \langle W, N, E\rangle\rangle$
Extraction	$\langle\langle E, S\rangle^{\lvert\mathcal{E}\rvert+1}, \langle W, S\rangle\rangle$

Table 5: Move sequences for the reduction. $\langle\cdot\rangle^K$ means repeat the sequence K times. $|\mathcal{E}|$ is the number of edges in the original constraint graph, as opposed to $\langle E \rangle$ which is the 'east' command.

Note this will happen for both vertex gadgets representing a vertex incident to the edge chosen. Figure 11d shows an example of two state transition areas in two vertex gadgets representing two vertices that share an edge.

5.1.3 Goal Area. An overview of the reduction layout is in Figure 11e where the goal area is shown at the bottom of all the vertex gadgets. Once all the tiles are in positions that represent the target configuration, the tiles can be extracted into the goal area. An extraction is made the final level in a state gadget. After extraction the tiles enter the goal area. The goal area consists of two rows. The valid row and the invalid row. The invalid row (top row) traps any tiles that enter when a vertex was not in the specified (in the target configuration) state. The valid row (bottom row) contains the goal position (g in Figure

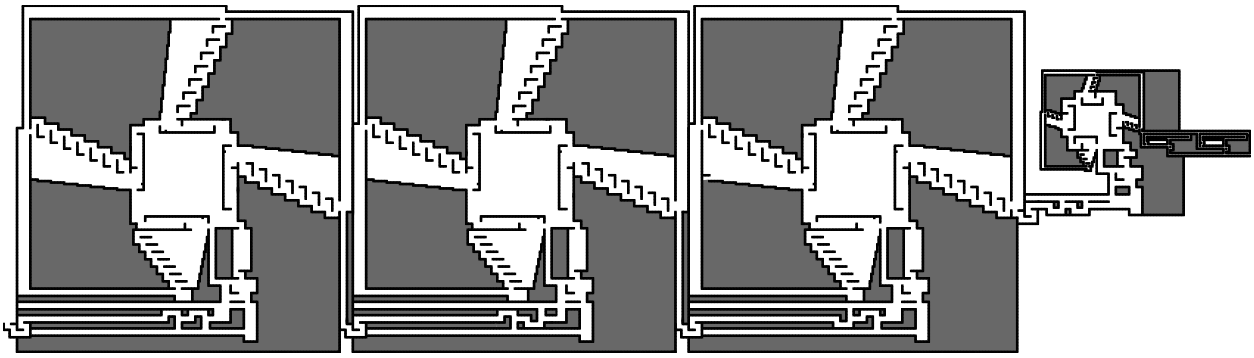


Figure 9: $\mathfrak{D}_4$ Constructor. This constructor is capable of building any 4×4-bounded polyomino in $\mathfrak{D}_4$.

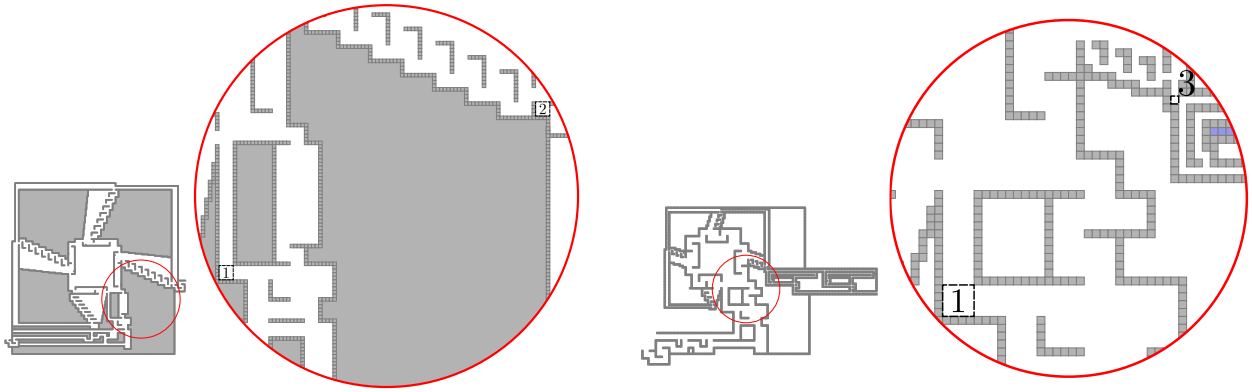
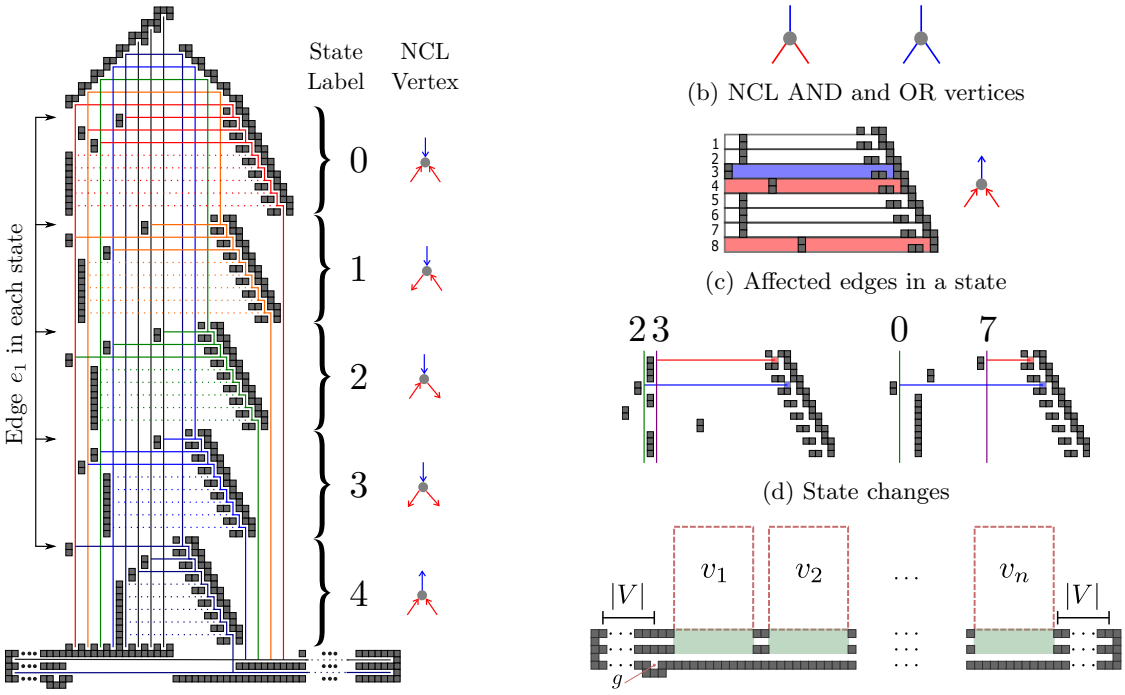


Figure 10: (a) $\mathfrak{D}_i$ Constructor in C_{launch} configuration. Section 1 is where shapes in all $\mathfrak{D}_i$ constructors will be located when in C_{launch} configuration. Section 2 indicates where a shape outputted from a reset gadget will be located when in C_{launch} configuration. (b) $\mathfrak{D}_1$ Constructor in C_{launch} configuration. Section 3 indicates where a single tile will be located after it is extracted from it's corresponding storage chamber. Similarly, once a tile is extracted from its storage chamber all shapes throughout the constructors will be located in Section 1.

11e). The goal position is $|V|$ positions to the right of the left wall. Thus, $|V|$ tiles must be in this row in order to have the last tile be in this location. In order to have enough tiles to reach the goal position, each vertex must be in the correct state to output the tile to the goal area. This ensures all vertices, and thus the entire graph, is in the specified target configuration.

LEMMA 5.1. *After performing a move sequence to flip an edge, only the two vertex gadgets representing vertices incident to that edge will have their state tile change state paths. All other vertex gadgets will have their state tile stay in the same state path.*

Proof. Since we enumerate all the edges and make each state gadget have an exit point for each edge, a move sequence to select and flip an edge moves all tiles out of their state gadget to the transition area. We create the transition area for each vertex gadget based on the edges that are connected to that vertex. If flipping an edge causes a vertex to change states we place a concrete block to stop the tile in the state column of the new state. If flipping an edge does not affect a vertex then when that tile leaves the state gadget and goes to the transition area there will be a block of concrete to stop the tile in the state column of the state it was previously in. Since all tiles start at the top of the state gadget, they all move out of the same level, so only two vertex gadgets will have their state tile change state gadgets. We can see this in Figure 11d. Flipping the first edge will change the state of the right vertex but not the left. Flipping the third edge edge will change the state of both. ∎



(a) An AND vertex gadget

(e) Goal gadget overview

Figure 11: (a) A vertex gadget with the path/transition areas labeled with the corresponding state numbers and the representative edge orientations in a constraint graph. Each state is a different color. Paths that do not change the state of a tile are dotted. The grey lines are invalid states. (b) The necessary vertices for a one-player unbounded game are reversible AND and OR vertices in a constraint graph with constraint 2 [9]. The AND vertex has two red edges (weight 1) and a blue edge (weight 2). Directing the blue edge outward requires both red edges to be directed inward. The OR vertex has three blue edges. Only one edge needs to be directed inward. (c) An example of a state transition area the vertex it represents. White rows represent edges that do not change the state of the vertex (they are not incident). Red and blue rows represent the incident edges and the weights of those edges. (d) The state gadgets for two different vertices V_l and V_r. Both vertices share edge 3. V_l represents state 4 of an *OR* vertex. V_r represents state 3 of an *AND* vertex. If edge 1 is selected (the red tile and line), V_l will remain in state 3 while V_r will go to state 7. Selecting edge 3 changes the state of both gadgets. (e) An overview of the layout of the different components for the reduction. The dotted red lines represent the vertex gadgets (not to scale), the green boxes below denote the geometry specific to each vertex to force the state tile into the top row (if in the wrong state) or the bottom row (if in the correct state). The bottom row requires all $|V|$ state tiles in order for a tile to get into the goal location g.

LEMMA 5.2. *If a vertex enters an illegal state, the representative vertex gadget's state tile will be trapped in an 'illegal' state path and cannot be extracted.*

Proof. If an edge flip would cause a vertex to enter an illegal state the tile will still be sent out of the state gadget into that states column. Since when we create the vertex gadget we block off the right of the top of any illegal state columns once a tile goes to the top or bottom of a state column it can only travel between the both of these. The only way this tile can be removed from this column is if a second tile enters this column. However, there is no way for another tile to enter the vertex gadget so there is no way to extract the tile once it becomes trapped in the column. □

THEOREM 5.1. *Occupancy is PSPACE-complete with only* 1×1 *tiles in the full tilt model.*

Proof. We show Occupancy is PSPACE-hard with only 1×1 tiles by a reduction from Non-Deterministic Constraint Logic. Given an instance of a constraint graph G (with initial configuration C_i) and a goal configuration C_g of that graph, enumerate the edges and vertices of the configuration. For each vertex in the graph create a Vertex Gadget. We create the configuration in the tilt model, S_i, as described above with the goal location g and vertex gadgets

laid out side by side above the goal area as shown in Figure 11e. The vertex gadgets will have a state transition gadget for each legal state of that vertex. The transition area is built based on the edges of the vertex. Then there exists a sequence of edge flips to transition C_i to C_g if and only if there exists a sequence of tilt commands that transition board S_i to a board configuration with some tile at location g.

Given a sequence of edge flips to transition C_i to C_g in $G = (V, \mathcal{E})$, $F = \langle f_1, \dots f_l \rangle$ where $f_i \in \mathcal{E}$, we can directly translate this into the move sequence necessary based on Table 5. Each vertex has its state tile start in the starting state of the vertex. Lemma 5.1 shows we can select any specific edge and perform a move sequence to select and flip that edge to change the state of two vertices and leave all other state tiles in the same state path. Since there exist move sequences to flip edges and change the states of vertex gadgets, a series of edge flips that are a solution to an NCL configuration-to-configuration problem can be turned into a move sequence that changes all vertex gadgets to their goal states which can then be extracted to solve the occupancy problem.

If given S_i and a sequence of tilts $T = \langle t_1, \dots t_z \rangle$, where $t_i \in \{N, E, S, W\}$, that solved the occupancy problem, the edges to flip could be found from the sequences of Table 5. We know by Lemma 5.2 that any tilt sequence not corresponding to a legal edge flip would trap a state tile, and thus occupancy could not be solved. Thus, if our sequence solves the problem, only legal edge flips were made. Further, to solve occupancy, we need all $|V|$ state tiles in the goal area, meaning all vertex gadgets were in the correct state, as specified in C_g. □

Corollary 5.1. *Relocation is PSPACE-complete with only* 1×1 *tiles in the full tilt model.*

Proof. If we ask whether we can relocate the state tile in the vertex gadget for v_n to the goal location g, we have an equivalence of the Occupancy problem. □

Corollary 5.2. *Reconfiguration is PSPACE-complete with only* 1×1 *tiles in the full tilt model.*

Proof. Since state tiles remain in their vertex gadget, they remain in the same order when extracted. The goal configuration is all tiles extracted from the vertex gadgets in the valid row ordered by vertex number. In order to extract all the tiles into the valid row, all gadgets must be in the correct state when extracted. □

6 Conclusion

In this paper we presented a hierarchy of shapes that are buildable within the full tilt model. We proved several characteristics about the drop-shape class, then gave an algorithm to decide membership in the class for hole-free shapes. We then provided a universal constructor that strongly builds this class of shapes. We then answered an open question by proving that the Occupancy problem in full tilt is PSPACE-complete even with only 1×1 tiles.

We leave a number of open problems. When considering drop-shape membership, our algorithm does not consider shapes with holes. Does there exist an efficient algorithm to determine membership in $\mathfrak{D}$ for all shapes? Also in defining membership in levels of our hierarchy we only consider one tile type that sticks to itself in determining if a cut is valid. What shapes can be built in lower levels of the hierarchy if more tile types are allowed? Does there exist a tile type hierarchy and how does it relate to the drop shape hierarchy? In regards to Theorem 3.2, does there exist a tighter bound on the level of the hierarchy a shape must be in? Lastly, for complexity of the occupancy, relocation, and reconfiguration problems, we use a connected board to show PSPACE-completeness. Is the problem easier when limiting the board type to simple or monotone, or does it remain PSPACE-complete?

References

[1] *Atomix*, Thalion Software, 1990.

[2] *Mega maze*, Phillips Media, 1995.

[3] Jose Balanza-Martinez, David Caballero, Angel Cantu, Luis Garcia, Austin Luchsinger, Rene Reyes, Robert Schweller, and Tim Wylie, *Full tilt: Universal constructors for general shapes with uniform external forces*, 30^{th} Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'19, 2019.

[4] Aaron T. Becker, Erik D. Demaine, Sándor P. Fekete, Golnaz Habibi, and James McLurkin, *Reconfiguring massive particle swarms with limited, global control*, Algorithms for Sensor Systems (Berlin, Heidelberg), Springer Berlin Heidelberg, 2014, pp. 51–66.

[5] Aaron T. Becker, Erik D. Demaine, Sándor P. Fekete, Jarrett Lonsford, and Rose Morris-Wright, *Particle computation: complexity, algorithms, and logic*, Natural Computing (2019).

[6] Aaron T. Becker, Erik D. Demaine, Sándor P. Fekete, and James McLurkin, *Particle computation: Designing worlds to control robot swarms with only*

global signals, IEEE International Conference on Robotics and Automation, ICRA'14, May 2014, pp. 6751–6756.

[7] Aaron T. Becker, Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, Christian Rieck, Christian Scheffer, and Arne Schmidt, *Tilt Assembly: Algorithms for Micro-Factories that Build Objects with Uniform External Forces*, 28th International Symposium on Algorithms and Computation (ISAAC 2017), Leibniz International Proceedings in Informatics (LIPIcs), vol. 92, 2017, pp. 11:1–11:13.

[8] Aaron T. Becker, Sándor P. Fekete, Phillip Keldenich, Dominik Krupke, Christian Rieck, Christian Scheffer, and Arne Schmidt, *Tilt assembly: Algorithms for micro-factories that build objects with uniform external forces*, Algorithmica (2018).

[9] Robert A. Hearn and Erik D. Demaine, *The nondeterministic constraint logic model of computation: Reductions and applications*, Proceedings of the 29th International Colloquium on Automata, Languages and Programming (London, UK, UK), ICALP '02, Springer-Verlag, 2002, pp. 401–413.

[10] Sheryl Manzoor, Samuel Sheckman, Jarrett Lonsford, Hoyeon Kim, Min Jun Kim, and Aaron T. Becker, *Parallel self-assembly of polyominoes under uniform control inputs*, IEEE Robotics and Automation Letters **2** (2017), no. 4, 2040–2047.

[11] Matthew J. Patitz, *An introduction to tile-based self-assembly and a survey of recent results*, Natural Computing **13** (2014), no. 2, 195–224.

[12] Arne Schmidt, Sheryl Manzoor, Li Huang, Aaron T. Becker, and Sándor Fekete, *Efficient parallel self-assembly under uniform control inputs*, IEEE Robotics and Automation Letters (2018), 1–1.