Forming tile shapes with simple robots

Forming Tile Shapes with Simple Robots

Robert Gmyr · Kristian Hinnenthal · Irina Kostitsyna · Fabian Kuhn · Dorian Rudolph · Christian Scheideler · Thim Strothmann

Abstract Motivated by the problem of manipulating nanoscale materials, we investigate the problem of reconfiguring a set of tiles into certain shapes by robots with limited computational capabilities. As a first step towards developing a general framework for these problems, we consider the problem of rearranging a connected set of hexagonal tiles by a single deterministic finite automaton. After investigating some limitations of a single-robot system, we show that a feasible approach to build a particular shape is to first rearrange the tiles into an intermediate structure by performing very simple tile movements. We introduce three types of such intermediate structures, each having certain advantages and disadvantages. Each of these structures can be built in asymptotically optimal $O(n^2)$ rounds, where n is the number of tiles. As a proof of concept, we give an algorithm for reconfiguring a set of tiles into an equilateral triangle through one of the intermediate structures. Finally, we experimentally show that the algorithm for building the simplest of the three intermediate structures can be modified to be executed by multiple robots in a distributed manner, achieving an almost linear speedup in the case where the number of robots is reasonably small, and explain how the algorithm can be used to construct a triangle distributedly.

Keywords Finite Automata · Reconfiguration · Tiles · Shape Formation

This work was begun at the Dagstuhl Seminar on Algorithmic Foundations of Programmable Matter, July 3–8, 2016. A preliminary version of this paper was presented at EuroCG 2017. This work is partly supported by DFG grant SCHE 1592/3-1. Fabian Kuhn is supported by ERC Grant 336495 (ACDC).

R. Gmyr
University of Houston, USA.
E-mail: rgmyr@uh.edu

K. Hinnenthal, D. Rudolph, C. Scheideler, T. Strothmann
Paderborn University, Germany.
E-mail: {krijan, dorian, scheidel, thim}@mail.upb.de

I. Kostitsyna
TU Eindhoven, the Netherlands.
E-mail: i.kostitsyna@tue.nl

F. Kuhn
University of Freiburg, Germany.
E-mail: kuhn@cs.uni-freiburg.de

1 Introduction

Various models and approaches for designing and manipulating nanoscale materials have already been proposed. A prominent approach in the DNA community has been to use DNA tiles [16]. In the most basic *abstract tile-assembly model* (aTAM), there are square tiles with a specific glue on each side [19]. Here, standard problems are to minimize the tile complexity (i.e., the number of different tile types) in order to form certain shapes, and to intrinsically perform computations guiding the assembly process. While in aTAM only individual tiles can be attached to an existing assembly, in more complex hierarchical assembly models, partial assemblies can also bind to each other (e.g., [6,7]). However, these approaches are based on strictly passive elements, so any changes to the structure have to be enforced externally (e.g., by changing the temperature or exposing the structure to certain kinds of radiation). A limited number of approaches has been proposed that are based on active elements instead [8, 11, 26]. However, since these elements are presumably more difficult to build, it might be far more costly to realize these approaches than the approaches based on DNA tiles.

In this paper, we investigate a *hybrid approach* for the *shape formation problem*, in which we are given a

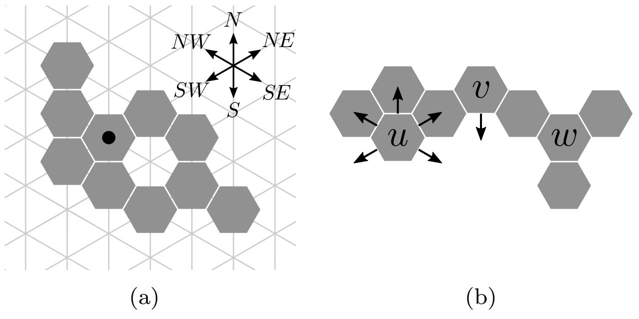

Fig. 1: (a) A connected set of tiles positioned on the triangular lattice. The black dots indicates the position of the robot. (b) Possible movements of tiles u, v, and w. Tile w cannot be moved anywhere without violating connectivity.

set of *passive* tiles, which are uniform and stateless, and (a limited number of) *active* robots. The robots, which only have the computational power of a finite automaton, can transport tiles from one position to another in order to form a desired shape. Compared to the DNA tile-based approach, this approach has the advantage that all tiles are of the same type and movements are exclusively performed by the robots. Furthermore, in contrast to the approaches based entirely on active elements, we believe that many problems can be solved in our hybrid model using only a few active elements. In this paper, we support this claim by showing that already a single robot is able to solve simple shape formation problems. Our ultimate goal is to investigate how multiple robots can cooperate to speed up the process of shape formation.

Although the complexity of our model is very restricted, actually realizing such a system, for example using complex DNA nanomachines, is currently still a challenging task. However, in recent years there has been significant progress in this direction. For example, nanomachines have been demonstrated to be able to act like the head of a finite automaton on an input tape [18], to walk on a one- or two-dimensional surface [12, 15, 25], and to transport cargo [20, 22, 24]. We therefore believe that, in principle, it should be feasible to build nanomachines with the capabilities assumed in this paper.

1.1 Model and Problem Statement

We assume that a single *active* agent (a *robot*) operates on a set of n *passive* hexagonal *tiles*. Each tile occupies exactly one node of the infinite triangular lattice $G = (V, E)$ (see Fig. 1a). A *configuration* (T, p) consists of a set $T \subset V$ of all nodes occupied by tiles, and the robot's position $p \in V$. We assume that the initial position of the robot is occupied by a tile. Note that every node $u \in V$ is adjacent to six neighbors, and, as indicated in the figure, we describe the relative positions of adjacent nodes by the six compass directions *N*, *NE*, *SE*, *S*, *SW* and *NW*.

Whereas tiles cannot perform any computation nor move on their own, the robot may change its position and carry a tile, thereby modify a configuration. The robot must stand on or be adjacent to a node occupied by a tile. Additionally, if the robot does not carry a tile, we require the subgraph of G induced by T to be connected; otherwise, the subgraph induced by $T \cup \{p\}$ must be connected. In a scenario where a tile structure swims in a liquid, for example, this restriction prevents the robot or parts of the tile structure from floating apart. Some examples of possible tile moving steps are shown in Fig. 1b.

The robot operates in rounds of `Look`-`Compute`-`Move` cycles. In the `Look` phase of a round the robot can observe its node p and the six neighbors of that node. For each of these nodes it can determine whether the node is occupied or not. In the `Compute` phase, the robot may change its internal state and determines its next move according to the observed information. In the `Move` phase, the robot can either (1) lift a tile from p, if $p \in T$, (2) place a tile it is carrying at p if $p \notin T$, or (3) move to an adjacent node while possibly carrying a tile with it. The robot can carry at most one tile.

Formally, we model the execution of an algorithm by the robot as transitions in a *deterministic finite automaton* $(Q, \Sigma, \delta, q_0, F)$. Q is a finite set that contains all of the robot's possible states. $\Sigma = \{0, 1\}^7$ represents the set of possible views of the robot: The first bit of an element in Σ indicates whether p is occupied, and the other bits indicate, in order, whether the robot's *N*, *NE*, *SE*, *S*, *SW*, and *NW* neighbor is occupied. In each round, the robot executes one transition of the transition function δ, which is defined as

$$\begin{aligned}\delta : \quad & Q \times \Sigma \times \{0, 1\} \\ & \rightarrow Q \times \{none, liftT, placeT, move_d\},\end{aligned}$$

The input of δ is the current state and view of the robot, as well as a bit that indicates whether the robot carries a tile. As an output, the transition function determines the robot's next state as well as one of the following actions: do nothing (*none*), lift a tile ($liftT$), place a tile ($placeT$), or move in some direction d of the six cardinal directions ($move_d$). $q_0 \in Q$ is the initial state of the robot, and $F \subseteq Q$ contains all final states. If in some round the robot is in a final state, it will not perform any further state transition; in this case we say the robot has *terminated*.

Note that we use the above definition to formally argue about the robot's capabilities and limitations. However, we will present our algorithms from a higher level by describing their behavior textually and through pseudocode. We remark that all our algorithms can easily be transformed into actual state machines. Further, note that even though we describe the algorithms as if the robot knew its global orientation, we do not actually require the robot to have a compass. For the algorithms presented in this paper, it is enough for the robot to be able to maintain its orientation with respect to its initial orientation.

We are interested in *Shape Formation* problems, where the goal is to transform any initial configuration into a configuration in which the tiles form a certain shape on the lattice. Particularly, the goal of the *Triangle Formation Problem* is to bring the set of all tiles into a triangular form.

1.2 Related Work

There is a number of approaches to shape formation in the literature that use agents that fall somewhere in the spectrum between passive and active. For example, *tile-based self-assembly* [16] uses passive tiles that bond to each other to form shapes. Here, the way in which the tiles attach to each other is defined by different types of *glues* rather than deliberate movements from one position to another. A variant of *population protocols* proposed in [13] uses agents that are partly passive (i.e., they cannot control their movement) and partly active (i.e., upon meeting another, they can perform a computation and decide whether they want to form a bond). Finally, the *amoebot model* [8], the *nubot model* [26], and the modular robotic model proposed in [11] use agents that are completely active in that they can compute and control their movement. Shape formation has also been investigated in the field of *modular robotics* (see, e.g., [2, 14, 23]); here, the robots typically have much greater computational capabilities than in our model.

In contrast to the above models, our model is specified by exactly *two* types of agents, i.e., an active agent that acts on a set of uniform passive agents. We remark that some of the above models are more powerful than our model and could therefore easily simulate our algorithms. For example, in the amoebot model a set of n active agents could form the initial tile structure and simulate movements of the active agent by transferring its role from one agent to another. As every agent is able to move in that model, modifications of the tile structure could also be simulated. However, the simplicity of our model allows us to focus on the question whether already a *single* active agent with the power to manipulate the structure of passive agents suffices for complex, e.g., shape formation tasks.

When arguing about a robot traversing a tile structure without actually moving tiles, our model reduces to an instance of the ubiquitous *agents on graphs* model. The vast amount of research on this model covers many problems, including *Gathering* and *Rendezvous* (e.g., [17]), *Intruder Caption* and *Graph Searching* (e.g., [1, 9]), and *Graph Exploration* (e.g., [3]). Other approaches allow agents to move tiles (e.g., [5, 21]) but these focus on computational complexity issues or agents that are more powerful than finite automata.

Notably, [10] considers almost exactly the same hybrid model proposed in this paper with only a single robot, but instead of moving tiles the robot is only allowed to place *pebbles*. The authors study *shape recognition* problems and investigate the impact of equipping the robot with a set of pebbles; we briefly study a problem that has similar difficulties (i.e., finding a tile whose removal does not disconnect the tile structure) in Section 2.

1.3 Our Contribution

In this paper we mainly focus on the Triangle Formation Problem with a single robot. We begin with pointing out one of the limitations of our model: It is in general impossible for one robot to find a tile that can be removed without disconnecting the tile structure. We contrast this result by showing that having a single *pebble* already suffices to solve this problem.

We then show how to construct *intermediate structures* by using simple tile movements that allow for easy navigation and tile removal. More specifically, we present three intermediate structures. The simplest among them is a *line* structure; it can be constructed in $O(n^2)$ rounds. The second structure we introduce is a *block*. It has $O(D)$ diameter (D being the initial diameter of the tile set), and can often be constructed more efficiently than the line, namely in $O(nD)$ rounds. Finally, we describe a *tree* structure, which, in contrast to the previous structures, can be built completely inside the convex hull of the original tile set in $O(n^2)$ rounds. Using the block structure as an example, we argue that each of these intermediate structures can be transformed into a triangle by performing an additional $O(nD)$ rounds (D being the intermediate structure's diameter).

We finally discuss how the algorithm to construct a line can be transferred to the multi-robot case. We provide some first simulation results showing that a small number of robots can speed up line formation by a significant amount. As the number of robots becomes high, we observe the anticipated decline in speedup. We then describe how a triangle can be built from a line in a distributed manner.

2 Finding Safely Removable Tiles

In a naive approach to shape formation, the robot could iteratively search for a tile that can be fully removed from the structure without disconnecting the tile structure (a *safely removable tile*) and then move that tile to some position such that the shape under construction is extended. More formally, a safely removable tile is a tile that does not occupy a *cut vertex* v of the subgraph H of G induced by the nodes of T (i.e., a node whose removal from H does not increase the number of components in H). Since H is finite, not every node of H can be a cut vertex; therefore, there always is a safely removable tile. However, the following theorem shows that, in general, a single robot cannot *decide* whether a tile is safely removable, which makes this naive approach infeasible.

Theorem 1 *There does not exist a deterministic finite automaton A so that if the robot executes A on any configuration starting on an occupied node and without carrying a tile, it (1) never performs a tile lift, (2) terminates on a safely removable tile.*

Proof Suppose that there is such an automaton A, and let $s = |Q|$. We consider the configuration H_ℓ in which the tiles form a hollow hexagon of side length ℓ, and place the robot on the southernmost tile of the hexagon as depicted in Fig. 2a (we call this node the *southern vertex* of H_ℓ). We define the set of *border nodes* to be all vertices (i.e., the corners) of the hexagon, all empty nodes inside the hexagon that are adjacent to a vertex, and all empty nodes outside the hexagon whose only neighbor is a vertex (see Fig. 2a). Consider the finite sequence of *system states* $(S_1, S_2, \ldots, S_T)$ through which the robot progresses while executing A, where $S_i = (p_i, q_i)$ contains the robots position p_i and state q_i before executing round i. S_1 corresponds to the initial system state, and S_T is the first system state for which $q_T \in F$. We partition this sequence into *phases*, where we define a new phase to start whenever the robot visits a border node (i.e., for every phase $(S_i, \ldots, S_k)$, p_i is a border node, and for all j, $i < j \leq k$, p_j is not a border node).

Note that since there are at most 18 border nodes, there can be at most $18s$ phases: Otherwise, there must be two phases that begin with system states S_i, S_j, respectively, such that $S_i = S_j$ (w.l.o.g., let $i < j$). Since the tile structure is never altered by the robot, $S_{i+1} = S_{j+1}$, and, inductively, $S_{i+k} = S_{j+k}$ for all $k \geq 0$, which implies an infinite loop and contradicts the assumption that the sequence of system states is finite.

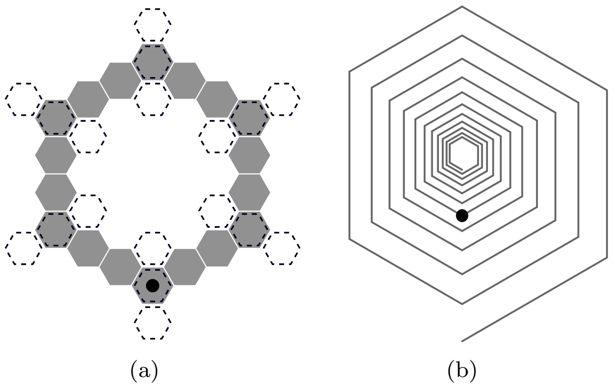
Fig. 2: (a) The hollow hexagon of side length $\ell = 4$. The border nodes of the hexagon are marked by dashed frames. (b) An example of the tile structure T. In both figures, the black dot indicates the initial position of the robot.

The way the robot traverses the hexagon depends on the side length ℓ. We define the *traversal sequence* associated with ℓ as the sequence $(S_1, S_2, \ldots, S_k)$ of all system states a phase begins with when A is executed on H_ℓ (i.e., S_i is the first system state of phase i, and k is the total number of phases). Note that a traversal sequence may be of length 1, i.e., if the robot never visits a border node except for its initial position. Since the algorithm takes at most $18s$ phases to choose the tile (for any choice of ℓ), there can only be at most $(18s)^{18s}$ distinct traversal sequences for different choices of ℓ. Hence, there is a finite number of traversal sequences and an infinite number of side lengths, which, according to the pigeonhole principle, implies that there must be an infinite set L of side lengths corresponding to the same traversal sequence.

Based on this observation, we now define a tile structure T for which the robot terminates on a tile that is not safely removable. This tile structure essentially consists of a spiral as depicted in Fig. 2b. We start at an arbitrary node of the triangular lattice and construct an outward spiral consisting of $72s$ line segments of tiles. The first line segment of the spiral goes NW and each subsequent line segment takes a $60°$ clockwise turn. The lengths of the line segments are chosen from L in such a way that the smallest side length ℓ_{min} is larger than $s + 2$, and such that the segments stay well-separated. This is possible since L is an infinite set and therefore we can always choose sufficiently large segment lengths. We initially place the robot at the last tile of the ($36s$)-th line segment, which is a tile with neighbors at *NW* and *NE*.

It remains to show that the algorithm fails to find a tile that can be safely removed when being executed on T. As above, we subdivide the execution of the algorithm into phases, where we define a new phase to start whenever the robot visits a border node of the spiral (which, analogous to the definition for the hexagon, we define as the three nodes at each turn of the spiral). Using induction on the phases, we show that the robot traverses T in a way that corresponds to the traversal sequence associated with the side lengths in L.

More specifically, we show that the i-th border node visited by the robot on T (1) has the exact same neighborhood as the i-th border node visited by the robot in a hexagon H_ℓ for all $\ell \in L$, and (2) is visited in the same state. This initially holds, as the robot is placed on a tile with only *NW* and *NE* neighbors in both structures, and starts in the initial state. For $1 < i \leq 18s$, w.l.o.g., assume that the i-th border node visited in T is occupied and has a tile at *NW* and *NE* (all the other cases are analogous), and the robot is in state q (note that the robot cannot have reached either end of the spiral after having visited fewer than $36s$ border nodes). Let ℓ_{NE} be the length of the line segment in direction *NE* from the robot's current position, and let ℓ_{NW} be the length of the line segment in direction *NW*. By the induction hypothesis, we have that the i-th border node v visited on H_ℓ is the the southern vertex of the hexagon for all $\ell \in L$, and the robot is in state q when visiting the i-th border node.

W.l.o.g, assume that the next border node visited by the robot on H_ℓ is not adjacent to the south-east vertex of H_ℓ (i.e., the robot does not follow the hexagon in direction *NE*). Then, in any H_ℓ, the robot will never move away from v in direction *NE* by more than $\ell_{min} - 2$ steps, as otherwise it would visit two nodes with the same neighborhood in the same state, which would cause a repetition that leads the robot to a border node at the south-east vertex. Thus, for every node visited by the robot on $H_{\ell_{NW}}$ before the next border node is visited, the robot visits a node with the exact same neighborhood on T. Therefore, the next border node visited by the robot on T has the same neighborhood than the next border node visited by the robot on H_ℓ for all ℓ, and is visited in the same state.

Therefore, the ($18s$)-th visited border node on T corresponds to the last border node visited by the robot on H_ℓ for all ℓ, from where the robot will not move farther away than by ℓ_{min} steps before terminating on a tile (otherwise, there would again be a repetition leading the robot to a border node). However, since the robot has only visited at most $18s$ border nodes, it cannot have reached either end of the spiral, and thus terminates on a tile that is not safely removable. This directly contradicts the assumption that the automaton works correctly and therefore shows that there is no such automaton. □

In contrast, the problem can be solved by equipping the robot with a single *pebble*, which we describe in the following. Here, we additionally assume that the robot is given a single pebble that it can pick up, carry, or place on a tile. Formally, we extend the set Σ that represents the possible view of the robot to be $\Sigma = \{0,1,2\}^7$, which for each of the seven nodes within the robot's vicinity indicates whether the node is empty (0), occupied (1), or occupied by a tile on which a pebble is placed (2). Further, we extend δ to

$$\begin{aligned}\delta: \quad & Q \times \Sigma \times \{0,1\}^2 \to Q \\ & \times \{none, liftT, placeT, move_d, pickP, placeP\},\end{aligned}$$

whose input now contains an additional bit indicating whether the robot currently carries a pebble, and whose set of actions contains one action for picking up a pebble, if the robot stands on a tile on which the pebble is placed (*pickP*), and one action for placing the pebble, if the robot carries its pebble and stands on a tile (*placeP*).

To that end, we first describe how the robot can use a single pebble to detect whether a given tile is safely removable. Let S be a maximal set of connected empty nodes. If S is finite, then it is a *hole*; otherwise, it is the infinite set of empty nodes around the structure. We refer to the subset of S adjacent to tiles as the *boundary* of S. Any tile t can be adjacent to at most three boundaries (see Fig. 3). We define the *outline* of a boundary as the set of its adjacent occupied nodes. For a tile t, consider the subgraph H of G induced by the empty nodes adjacent to the node of t; H has at most three components, which we call t's *regions*. For example, the two end tiles of a line have only one region, whereas all the tiles with two neighboring tiles have exactly two regions. Note that every region is contained in some boundary that we call its *corresponding boundary*; in the example of a line, all regions correspond to the same boundary.

Lemma 1 *A tile t is safely removable if and only if all of its empty regions correspond to different boundaries.*

Proof First, assume that t has at least two empty regions that belong to the same boundary (e.g., the outer boundary in Fig. 3). Consider a path P of empty nodes along the boundary that connects the two regions of t

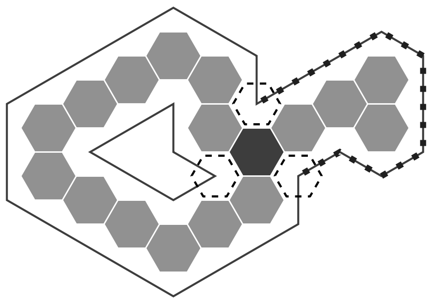

Fig. 3: A tile t (black) that is not safely removable with its three empty regions (dashed outlines). t is adjacent to two boundaries (black lines). The dotted path could be extended to a cycle if t was removed.

(the dotted path in Fig. 3). After the removal of t, we can extend P to a cycle C using the remaining empty node of t. C consists of empty nodes and surrounds a set of tiles A. However, as t had at least two empty regions, there must remain a tile t' adjacent to t that is not connected to any tile of A (as otherwise the two regions would belong to separate boundaries). Hence, the configuration is not connected anymore.

Now assume all empty regions of t belong to separate boundaries, and consider two different outlines containing t. The empty regions that are part of the corresponding boundaries are connected via tiles in t's neighborhood. Hence, any two outlines containing t are connected via tiles adjacent to t and thus remain connected after removing t. Thus, t is safely removable. □

Now we are ready to give our automaton to find a safely removable tile, which, for simplicity, we describe as an algorithm for the robot. To search for a safely removable tile, the robot first walks *N*, *NW*, and *SW* (in that precedence) until it reaches a locally north-westernmost tile t (i.e., a tile with no neighbors at *N*, *NW*, and *SW*). The empty node *NW* of t belongs to a boundary whose outline O contains t. As can easily be seen, O must contain a safely removable tile. To find it, the robot traverses O in clockwise order and checks each tile t' of O separately by the following procedure. First, it places the pebble on t'. Then, it traverses each boundary adjacent to t' and verifies whether it returns to t' within the same region, in which case all empty regions belong to separate boundaries. Together with Lemma 1, we conclude the following theorem.

Theorem 2 *There exists a deterministic finite automaton that the robot can execute to find a safely removable tile in $O(n^2)$ rounds with the help of a single pebble.*

3 Forming an Intermediate Structure

Although the robot cannot always find a safely removable tile (unless being equipped with a pebble), it can always perform local tile movements that preserve connectivity. In this section, we show how to construct *intermediate structures* by performing such movements. In the resulting structures the robot can easily navigate and move tiles without possibly violating connectivity. Therefore, it can easily disassemble such a structure and rearrange its tiles into the desired shape.

We aim to construct *simply connected* intermediate structures (i.e., structures without holes), as removability of a tile can easily be determined locally in such a structure: a tile is safely removable if and only if it only has one empty region. Such a tile can always easily be found in a simply connected structure. Note that although in the presented intermediate structures it is easy to determine a location where an arbitrarily sized shape can be built, a robot may not easily find such a location in an arbitrary simply connected structure.

We show how to construct three different intermediate structures. As a first simple example, we demonstrate how to construct a *line* in time $O(n^2)$. Clearly, the main drawback of this algorithm is that tiles might need to be moved by a distance linear in n. Our second algorithm avoids this pitfall by building a structure called a *block* in time $O(nD)$. Here, D is the *diameter* of the initial tile configuration, which is defined as the maximal length of a shortest node path between any two occupied nodes of the triangular lattice. The algorithm further ensures that no tile is moved farther than by a distance of D. The last and most complex algorithm builds a simply connected structure called a *tree* in time $O(n^2)$. The main advantage of this solution is that no tile is ever placed outside of the *convex hull* of the initial configuration. Here, we refer to the convex hull of the corresponding set of hexagonal tiles in the Euclidean plane.

3.1 Forming a Line

We present an algorithm for the robot to rearrange any connected tile configuration into a line (i.e., a sequence of connected tiles from north to south) in $O(n^2)$ rounds. The robot first moves S as far as possible, i.e., as long as there is a tile in direction S. Then, it alternates between a *tile searching phase*, in which it moves *N*, *NW*, and *SW* (in that precedence) until there is no longer a tile in any of these directions; and a *tile moving phase*, in which it lifts the tile, moves one step *SE*, moves S until it reaches an empty node, and then places the tile. The line is complete once the robot does not encounter

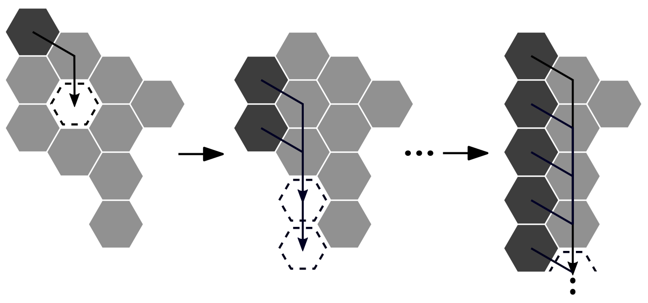
Fig. 4: First several steps of line formation. The black tiles are moved to the positions marked by dashed outlines.

any adjacent tiles to the east or west in the tile searching phase. Fig. 4 shows the first several steps of this algorithm.

Theorem 3 *There exists a deterministic finite automaton that the robot can execute to transform any connected tile configuration into a line and terminate after $O(n^2)$ rounds.*

Proof We define a *column* to be a maximal sequence of connected tiles from *N* to *S*. The correctness of the algorithm follows from the following observations: (1) the robot always finds a locally northwesternmost tile in the tile searching phase, (2) if there is more than one column in the tile configuration, the tile searching phase does not terminate in the northernmost tile of an easternmost column, (3) the tile moving phase does not disconnect the tile configuration and (4) the algorithm terminates when a line is formed. Together with the fact that every tile is moved exactly one step in *SE* direction, it follows from (1) - (3) that the line will eventually be built, in which case the robot terminates by observation (4).

The first observation is obvious by the definition of the first phase of the algorithm. The second observation follows from the fact that preference is given to the *NW* and *SW* directions when searching: If the target tile configuration has not yet been achieved, and the robot stops at some locally northwesternmost tile, there must be tiles east of that position.

For the third observation, suppose that the tile moving phase disconnects the tile configuration. Let t be the locally northwesternmost tile being moved. The tile configuration can get disconnected after removing t only if there are adjacent tiles *NE* and *S* of t, but no adjacent tile *SE*, since otherwise the adjacent tiles will still be locally connected after removing t. But in that case the tile t will be placed in the empty position at the *SE* neighbor and reconnect the adjacent *NE* and *S* tiles. Therefore, during the second phase of the algorithm the tile configuration does not get disconnected.

Finally, for the fourth observation consider the following two cases: (1) if the structure is initially a line, then the robot completely traverses the line from south to north in the first tile searching phase and finds no tile to the east or west; (2) otherwise, the structure eventually becomes a line after the robot has placed the line's southernmost tile, in which case the robot terminates after the next tile searching phase.

Finally, we show that the algorithm takes $O(n^2)$ rounds. The first steps of moving south take $O(n)$ rounds. For the two phases, we first bound the number of times the robot moves a tile by one step in the tile moving phase. By the above observations, the tiles of an easternmost column in the initial tile configuration are never moved. Since furthermore the initial tile configuration is connected, and tiles are exclusively moved *SE* and *S*, each tile is moved at most $2n$ steps. Therefore, $s_{\text{move}} = O(n^2)$ move steps are performed in total.

Now, consider the tile searching phase. We assign coordinates to each node, where the x-coordinate grows from west to east and the y-coordinate grows from north to south (e.g., moving *S* increases y by 1 while moving *SW* decreases x by 1 and y by $\frac{1}{2}$). Whereas the sum of the coordinates of the robot increases at every step in the tile moving phase, and decreasing at every step in the tile searching phase. More specifically, at each step of the tile moving phase, the sum of the coordinates of the robot increases by at most $\frac{3}{2}$, and at every step of the tile searching phase, the sum of the coordinates decreases by at least $\frac{1}{2}$. Thus, the total number of steps in the tile searching phase can be bounded by $s_{\text{search}} < 3 \cdot s_{\text{move}} + (x_0 + y_0) - \min_i(x_i + y_i)$, where (x_0, y_0) denotes the robot's initial coordinates, and the value $\min_i(x_i + y_i)$ is taken over all possible placements of all tiles. As the initial tile configuration is connected, $(x_0 + y_0) - \min_i(x_i + y_i) = O(n)$, and $s_{\text{search}} = O(n^2)$. Therefore, the total number of search steps is $O(n^2)$. Since each move and search step takes $O(1)$ rounds, the total number of rounds is $O(n^2)$. □

Note that it is not hard to see that $\Omega(n^2)$ rounds are necessary to rearrange an arbitrary initial tile configuration into a line by a single robot. If starting from an initial configuration with diameter $O(\sqrt{n})$, a constant fraction of the tiles has to be moved by a distance linear in n and thus, in total, $\Omega(n^2)$ move steps are necessary.

3.2 Forming a Block

Although a line can be constructed efficiently, its linear diameter might make it an undesirable intermediate

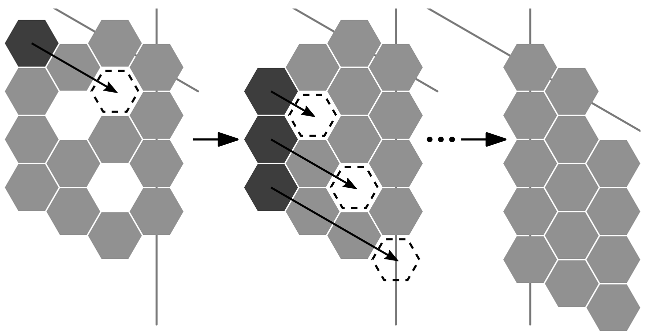

Fig. 5: Transformation of an initial structure into a block. The gray lines indicate some fixed x- and y-coordinates for reference.

structure. In fact, if both the initial diameter and the diameter of the desired shape are small, moving tiles by a linear distance seems to be an excessive effort. Therefore, we introduce another intermediate structure, which is called a *block*: In a block, all tiles except those farthest to the west have a neighbor to the northwest. Therefore, a block has only one westernmost column, and every row (i.e., a maximal sequence of connected tiles from *NW* to *SE*) begins with a tile from that column (see right picture in Fig. 5). Clearly, a line is a special case of a block that only consists of one column. Our algorithm builds a block in $O(nD)$ time and does not move any tile farther than by a distance D (recall that D is the diameter of the initial structure). An example of a transformation of an initial structure into a block is shown in Fig. 5.

We present the algorithm in two steps. First, we describe a non-halting algorithm by giving simple tile moving rules similar to the rules of the line construction algorithm. Eventually this algorithm will build a block structure. We then extend the algorithm with additional checks to detect whether a block structure has been built.

As in the line algorithm, the robot alternates between a searching and a moving phase: It first searches for a locally northwesternmost tile by repeatedly moving *NW*, *SW*, or *N* (in that precedence). The robot then picks up the tile, moves *SE* until it reaches an empty node, and places the tile there.

We first show the correctness of this simple algorithm in the following sequence of lemmas. In the following, we assign coordinates to each node, where the x-coordinate grows from west to east and the y-coordinate grows from north to south (e.g., moving N increases y by 1 while moving SW decreases x by 1 and y by $\frac{1}{2}$). For the following three lemmas, let 0 be the maximum x-coordinate of all tiles in the initial tile configuration, i.e., the x-coordinates of the easternmost tiles are 0 and all others have negative x-coordinates.

Lemma 2 *During the algorithm's execution, any two tiles with x-coordinate 0 are connected via a simple path of tiles whose x-coordinates are at most 0.*

Proof The claim initially holds. Let P be the simple path connecting two tiles u and v with x-coordinate equal to 0. We show that after the robot has moved any other tile, there remains a path between u and v. Note that the robot does not violate the claim by picking a tile with x-coordinate greater than 0. If the robot picks up a tile t with x-coordinate equal to 0, then t cannot lie on P, as t does not have adjacent tiles at *N*, *NW*, and *SW*. Thus, moving t does not affect the path. Now, assume the robot picks up a tile t with x-coordinate smaller than 0. If t lies on P, then, since P is simple, t must have two adjacent tiles t' and t'' at *NE*, *SE*, or *S* that are part of P. If the *SE* position is not empty, t' and t'' remain connected after the removal of t. Otherwise, t will be placed there. In both cases a path between u and v is maintained. □

Lemma 3 *If there is a tile with x-coordinate 0, and the robot picks up a tile at some node v with x-coordinate x_v, then $x_v \leq 0$.*

Proof The node of the first tile the robot picks up has x-coordinate at most 0. Now assume that afterwards the robot picks up a tile at some node v of the triangular lattice with x-coordinate x_v. If there is a tile at the S neighbor of v, then the next tile the robot will lift has x-coordinate at most x_v. If there is no tile at the S neighbor of v, the next tile the robot will lift is at the *SE* neighbor of v.

This implies that in order for the robot to first lift a tile t_2 with x-coordinate greater than 0, it has to have previously lifted a tile t_1 at 0 with no neighbor at S. Therefore, t_1 could not have been connected to any other tile at 0 via a path of tiles with x-coordinate at most 0. Thus, by Lemma 2, t_1 was the only tile at 0 when it was lifted. Therefore, there is no tile with x-coordinate 0 when t_2 is lifted. □

Note that in the next lemma we do not yet assume that the algorithm will terminate when a block structure has been built, but only show that a block will eventually be built.

Lemma 4 *Let the maximum x-coordinate of the tiles in the initial tile structure be 0. Then the algorithm rearranges the tiles into a block, in which the westernmost column of tiles has x-coordinate 1, in $O(nD)$ rounds.*

Proof We first show the correctness of the algorithm. First, note that the robot always finds a tile to move. By Lemma 3, the robot will repeatedly pick tiles with x-coordinate at most 0 until there is no such tile anymore. At this point, every tile with x-coordinate at least 2 has a neighbor at NW. This is due to the fact that each such tile must have had a NW neighbor at the time of its placement, and by Lemma 3 none of these tiles have been moved yet. Therefore, the tiles are arranged as a block in which the westernmost tiles have x-coordinate at most 1.

We now turn to the runtime of the algorithm. It is easy to see that each tile is moved for at most $2D$ steps until the block is established, which implies that at most $O(nD)$ move steps are performed in total. Note that each time a tile is moved the sum of the robot's coordinates increases by $\frac{3}{2}$. On the other hand, each search step decreases this sum by at least $\frac{1}{2}$. Thus, the total number of search steps is bounded by $3m+O(n)$, where $m = O(nD)$ is the total number of move steps. Therefore, the total number of search steps is also bounded by $O(nD)$. Since each step is performed within a constant number of rounds, the number of steps the algorithm takes until it builds a block structure, with x-coordinate of a westernmost tile equal to 1, is $O(nD)$. □

Next, we show how the robot can detect when a block has been successfully built by performing a series of tests alongside the algorithm's execution. Consider a block as a stack of *rows*, i.e., sequences of consecutive tiles from *NW* to *SE*. Note that according to the above algorithm the robot will move each tile of the westernmost column of a finished block, starting with the northernmost tile, placing each at the first empty position *SE* of it. Thereby, the robot can detect that a block has been built by verifying the following conditions: (1) after placing a tile, the robot performs at most one SW movement before it lifts the next tile, (2) while moving a tile t, the robot does not traverse a node (except for t's previous position) that has a neighbor at *NE*, but not at *N*, or a neighbor at *S*, but not at SW, (3) the robot never places a tile at a node that has a neighbor at *SE*. A test verifying the above conditions is initiated whenever the robot picks a tile that does not have a *NE* neighbor. If thereafter any of the above conditions gets violated, the test is aborted. If otherwise the robot places the southernmost tile without encountering any violation, the algorithm terminates.

Theorem 4 *There exists a deterministic finite automaton that the robot can execute to transform any connected tile configuration into a block and terminate after $O(nD)$ rounds.*

Proof By Lemma 4, the robot builds a block within $O(nD)$ rounds. Assume the robot lifts a tile t that does not have a *NE* neighbor and initiates the test sequence. If the structure is a block already, the robot will move its westernmost tiles as described above and, after moving the last tile, the test series finishes without any violation.

Now assume the structure is not a block at the time t is lifted by the robot. We show that in this case the test series fails. If a tile s of the column below t has a neighbor at SW, the test series will fail at the latest when the robot moves the northern neighbor of s and afterwards takes at least two steps SW, thereby violating Condition 1. If otherwise no such tile exists, i.e., no tile in the column below t has a neighbor at SW, then there must be a tile s farther east than t that has no neighbor at NW and that is adjacent to a tile of a row r of t's column; if no such tile existed, the structure would be a block already. We distinguish the following two cases: (1) s is a southern neighbor of r, in which case the test series fails at the latest when the robot traverses r's row by Condition 2. (2) s lies *NE* to a tile of r. If the robot places a tile *NW* of s (by moving a tile in the row above r), the test series will fail by Condition 3. Otherwise, the NW neighbor of s will still be empty when r is traversed by the robot, in which case the test series fails by Condition 2. □

Note that since tiles are exclusively moved *SE*, the resulting block has at most D rows consisting of at most D tiles each, and therefore diameter $O(D)$. Similar to the construction of a line, it can be easily seen that the runtime to construct a block is asymptotically optimal: Consider a line of tiles from SW to *NW*. In order to transform the initial structure into a block, a constant fraction of tiles needs to be moved by a distance linear in D.

3.3 Forming a Tree

So far we have been mainly focusing on how to *quickly* construct suitable intermediate structures. However, regarding potential practical applications, it may also be desirable to minimize the required work space. Whereas the previous structures are in many cases built almost completely outside of the initial configuration's convex hull, in this section we present an algorithm that builds a simply connected structure by exclusively moving tiles inside the structure's convex hull.

First we introduce some additional notation. An *overhang* is a set of vertically adjacent empty nodes such that (1) the northernmost node has a tile at *N*, (2) the southernmost node has a tile at *S*, and (3) all

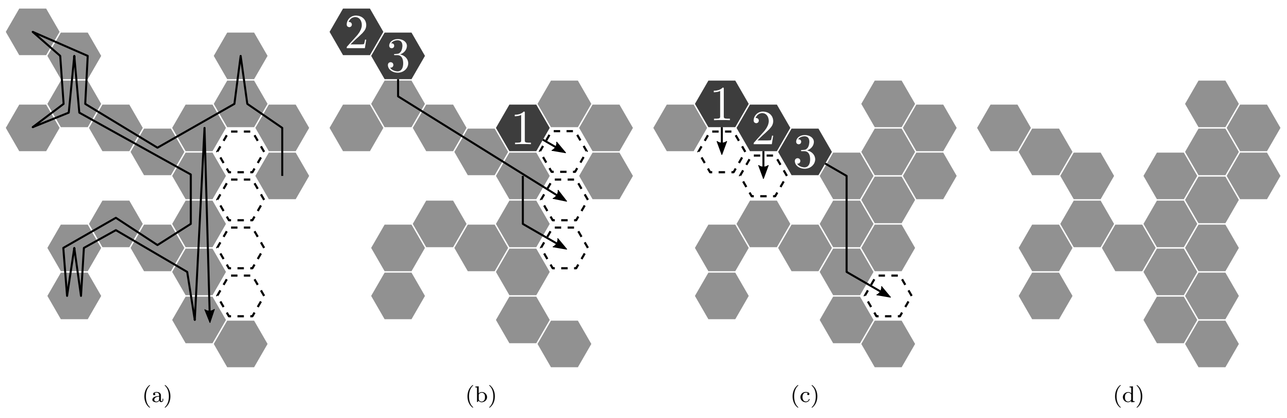

Fig. 6: (a) The traversal of the robot in an arbitrary connected structure, starting from an easternmost column, until detecting an overhang (dashed outlines). (b) The first three tiles are placed into the overhang. (c) Tiles (1) and (2) are moved south before (3) is brought into the overhang. (d) The resulting tree.

nodes have adjacent tiles at NW and SW. A *tree* is a connected tile configuration without an overhang. Examples of an overhang and a tree are shown in Fig. 6a and 6d, respectively. Since the westernmost nodes of a hole are part of an overhang, a tree is simply connected. The *branches* of a tree's column are defined as its western adjacent columns, where two columns are called adjacent if at least two of their tiles are adjacent. Finally, a *local tree* is a column whose connected component, obtained by removing all of its eastern neighbors, is a tree.

We present an algorithm that transforms any initial tile configuration into a tree in $O(n^2)$ rounds and without ever placing a tile outside the initial structure's convex hull. The pseudocode of the algorithm is given in Algorithm 1. From a high-level, the algorithm works as follows. The robot first traverses the tile structure in a recursive fashion until it encounters an overhang. It then fills the overhang with tiles and afterwards restarts the algorithm. Once the whole structure can be traversed without encountering any overhang, the tiles are arranged in a tree and the robot terminates.

More precisely, the robot does the following. In the `initialize` phase, it first successively moves to eastern columns until it reaches a locally easternmost column. Then the robot starts moving west. Upon entering a column, it moves N and then enters the northernmost branch. If the column has no branches, a locally westernmost column has been reached. In this case the robot checks whether the current column has an adjacent eastern overhang by traversing the column from N to S. If so, it fills the overhang as described in the next paragraph and afterwards restarts the algorithm. Otherwise, the robot searches for an adjacent eastern column (of which there can be at most one). If there is none, then the algorithm terminates. Otherwise, the robot either continues its traversal in the first branch south to the branch from which it entered the current column, or, if no such branch exists, verifies whether the current column has an adjacent eastern overhang and proceeds as described above. An illustration of a traversal of the robot, in which the robot eventually detects an overhang, is given in Fig. 6a.

We now describe how the robot fills an overhang. First, to find a tile to place into the overhang, the robot moves in a way that assures that it will find its way back. The robot alternates between moving N as long as there is a tile at N (`get_tile_N` phase), and moving NW as long as there is a tile at NW and no tile at SW or N (`get_tile_NW` phase). The robot's path either ends (1) in the `get_tile_N` phase at a tile that does not have a neighbor at NW or SW, in which case the tile is lifted (e.g., Tile 1 in Fig. 6b), (2) in the `get_tile_NW` phase at a locally northwesternmost tile, which would also be lifted (e.g., Tile 2 in Fig. 6b), or (3) in the `get_tile_NW` phase at a tile t that has a SW neighbor (e.g., Tile 1 in Fig. 6c).

In the third case, t is moved one step S. If thereafter t has a neighbor at S or SE, the robot lifts t's NE neighbor t'. Otherwise, it moves onto t' and continues its search. Both situations are illustrated in Fig. 6c: First, the tile labeled 1 is moved south. As this tile does not have a neighbor at S or SE, the robot continues at the tile labeled 2, which, after having been moved south as well, has a southern neighbor; therefore, the robot lifts tile 3.

Once the robot has lifted up a tile, it returns to its originating column c by moving S and SE (in this order of precedence). As the robot has never stepped on a tile with a southern neighbor outside of c, and has

Algorithm 1 Algorithm to form a tree.

```
 1: phase initialize:
 2:    Move to locally easternmost column
 3:    Move $N$ as far as possible
 4:    goto search_next_branch
 5: phase search_next_branch:
       Branches farther north are local trees
 6:    Move $S$ until
 7:       case tile at $NW$ or $SW$ then
 8:          Move $NW$ (or $SW$)
 9:          Move $N$ as far as possible
10:       case reached column's end then
11:          goto check_overhangs
12: phase check_overhangs:
       Current column is a local tree
13:    Move $N$ until end
14:    Move $S$ until
15:       case found eastern overhang then
16:          goto get_tile_N
17:       case reached column's end then
18:          goto move_E
19: phase move_E:
       Column is a local tree and has no overhangs
20:    Move $N$ until
21:       case tile at $SE$ or $NE$ then
22:          Move $SE$ (or $NE$)
23:          if tile at $S$ then
24:             Move $S$
25:             goto search_next_branch
26:          else
27:             goto check_overhangs
28:       case reached column's end then
29:          terminate
30: phase get_tile_N:
31:    Move $N$ as far as possible
32:    if no tile at $NW$ or tile at $SW$ then
33:       goto bring_tile
34:    else
35:       goto get_tile_NW
36: phase get_tile_NW:
37:    if tile at $NW$ and no tile at $SW$ and $N$ then
38:       Move $NW$
39:    else if tile at $N$ then
40:       goto get_tile_N
41:    else if tile at $SW$ then
          There cannot be a tile at S
42:       Lift tile and move it $S$
43:       if tile at $S$ or at $SE$ then
44:          Move $NE$
45:          goto bring_tile
46:       else
47:          Move $NE$
48:    else
          Reached locally northwesternmost tile
49:       goto bring_tile
50: phase bring_tile:
51:    Lift tile and move $S$, $SE$ (in that precedence)
       until there is tile at $NE$
52:    Move $S$ until there is no tile at $SE$
53:    Move $SE$ and drop tile
54:    if tile at $S$ then
          Overhang was filled
55:       goto initialize
56:    else
57:       Move $SW$
58:       goto get_tile_N
```

never performed a *SW* movement, it thereby precisely retraces its search path, and the first tile it encounters that has a *NE* neighbor must lie in c. The robot continues to bring tiles as described until the overhang is filled, in which case it again turns to the `initialize` phase.

Lemma 5 *If the algorithm is executed by the robot starting on a tree, the robot traverses the tree completely and terminates within $O(n)$ rounds. Otherwise, the robot finds an adjacent eastern overhang of a local tree within $O(n)$ rounds.*

Proof We first show the first part of the lemma. Assume a configuration is a tree. Then, every column has at most one adjacent eastern column. Furthermore, there is exactly one column that does not have an eastern neighbor. Following the `initialize` phase, the robot first moves to the northernmost tile of that column and then turns to the `search_next_branch` phase. We show that the robot traverses the local tree of each column completely in a recursive fashion. Since the local tree of the easternmost column is the whole tree, this implies the claim.

We claim that upon entering a column c, (1) the robot first moves to the northernmost tile of c, (2) completely traverses its branches from north to south, (3) verifies that c does not have any overhang, and then (4) enters c's adjacent eastern column c' via the southernmost tile of c' that is adjacent to c. If there is no such tile, then the robot has traversed the whole tree and terminates in Line 29.

We prove the claim by induction on the depth of the local tree of c. First, note that (1) holds due to Line 3 if c is the initial column, or Line 9 if c is reached via an eastern column c'. Now assume c does not have any branches. Then the robot traverses the column from north to south (Line 6) and immediately turns to the `check_overhangs` phase (Line 11). It then traverses the column once more to verify that is has no overhangs. Afterwards, by following the `move_east` phase, the robot traverses c from south to north until it reaches the southernmost tile of c' adjacent to c.

Now assume c has branches. When first entering c, the robot moves to the column's northernmost tile, and then enters the northernmost branch of c following the `search_next_branch` phase. By the induction hypothesis, it eventually reaches c again via its southernmost tile t that is adjacent to that branch. If there are branches further south, t must have a southern neighbor and the robot continues to search for the next branch (Line 23). Following the above procedure, the robot traverses all branches of c until it eventually reaches its southernmost tile and turns to the `check_overhangs` phase (Line 11 or 27). As there are no overhangs, it enters its adjacent eastern branch c' via the southernmost tile of c' that is adjacent to c. We conclude that if the configuration is a tree, then the robot moves through the whole structure and eventually terminates in its easternmost column.

If otherwise the configuration is not a tree, the robot will traverse the structure as described above until it eventually detects an eastern overhang in some column c during the `check_overhangs` phase in a column c. Since the robot must have traversed all branches of c, c is a local tree.

It is easy to see that in any case each tile is visited no more than 6 times. Therefore, the robot halts within $O(n)$ rounds. □

Lemma 6 *After detecting the northernmost eastern overhang of a column c in the* `check_overhangs` *phase, the robot will fill it and then turn to the* initialize *phase. At all times, the structure remains connected and c remains a local tree.*

Proof First, after encountering the overhang, the robot turns to the `get_tile_N` phase in Line 16 and moves to the northernmost tile r of c. If r does not have a neighbor at NW or does have a neighbor at SW, then the robot lifts r and, by following the `bring_tile` phase, places it at the northernmost node of the overhang. As c has an overhang, and thus consists of at least two tiles, the robot can only disconnect the tile structure by removing r if there is a tile NE but not SE of r's previous position, in which case r is directly placed SE, reconnecting both parts (Tile 1 in Fig. 6b).

If otherwise r has a neighbor at NW but not at SW, the robot initiates a search for a safely removable tile by turning to the `get_tile_NW` phase in Line 35. First, the robot moves NW as long as there is a tile at NW, no tile at SW, and no tile at N. If it reaches a tile that has a neighbor at N (Line 39), it again turns to the `get_tile_N` phase and continues as above. Since the robot only moves N and NW, and the tile set is finite, the robot eventually faces one of three situations. We show that in all three situations the robot identifies a tile that can be taken away without disconnecting the structure.

In the first situation, the robot reaches a locally northwesternmost tile during the `get_tile_NW` phase (Line 48, Tile 2 in Fig. 6b after Tile 1 has been moved). Since this tile must have been reached via its SE neighbor, it can be safely removed.

In the second situation, the robot encounters a northernmost tile t that has no tile at NW or a tile at SW during the `get_tile_N` phase (Line 32, Tile 3 in Fig. 6b after 1 and 2 have been moved). As t lies in a branch of c, in which there cannot be any overhangs by the discussion of Lemma 5, t cannot have a neighbor at NE and thus can be safely removed.

In the third situation, the robot reaches a tile t that has no neighbor at N, but at SW, during the `get_tile_NW` phase (Line 41, Tile 1 in Fig. 6c). t must have been reached via its SE neighbor t', which, consequently, cannot have a neighbor at N nor SW. First, the robot moves t one step south. If t does not have a neighbor at S nor SE, then the robot moves onto t' and continues with the *get_tile_NW* phase (Tile 1 in Fig. 6c after having been moved south). Note that the same situation might happen again for t', and may even repeat for every tile of the corresponding row, which will be moved south one after the other.

However, the robot must eventually move a tile t south that faces a neighbor at S or SE. In this case, its NE neighbor t' is lifted (Line 45). We have to show that the structure remains connected and c remains a local tree. We distinguish two cases. First, if t only has a neighbor at S, then $t' \neq r$ (Tile 2 in Fig. 6c, after 1 and 2 have been moved south). As there are no overhangs in the local tree of c, t' therefore cannot have a neighbor at NE; furthermore, the tile south of t must be a tile of a different branch than the branch of r. Therefore, connectivity of the structure is preserved after lifting t'. Furthermore, the new connection established by moving t cannot introduce any overhangs into c's local tree. Second, if t has a neighbor at SE, then $t' = r$. By the argument above, bringing r to the overhang cannot violate connectivity.

We finally argue that the lifted tile is correctly placed into the overhang. If the robot lifts a tile in the first or second situation above, then no tile traversed by the robot can have a NE neighbor (except for r), and the robot has never moved SW. If the robot lifts a tile in the third situation, then the same holds for the search path up to the position of that tile. Therefore, moving S and SE (in that precedence) in the `bring_tile` phase (Line 51) precisely retraces the robot's search path, and brings the robot back into column c. As c has an eastern overhang, there the robot must encounter a tile that has a neighbor at *NE*. The first empty *SE* neighbor south of that position must be a node the northernmost eastern overhang of *c*. If after placing the tile at that node there is a tile at *S* (Line 54), the overhang has been filled and the `initialize` phase is entered. Otherwise, the robot continues until all nodes of the overhang are filled. □

Lemma 7 *Following the algorithm, the robot never places a tile outside of the convex hull of the initial configuration.*

Proof The robot does not leave the convex hull by placing a tile t into an overhang. The only other movement of t occurs in Line 42 in the situation depicted in Fig. 6c. The robot has reached t by moving a sequence of *NW* and *N* steps starting at the northernmost tile of a column c with an adjacent eastern overhang. Since c has at least 2 tiles, the robot will not move t outside the configuration's convex hull. □

Using the previous lemmas, we are now ready to prove the following theorem.

Theorem 5 *There exists a deterministic finite automaton that the robot can execute to transform any connected tile configuration into a tree in $O(n^2)$ rounds and without placing a tile outside the initial configuration's convex hull.*

Proof If the configuration is a tree, then by Lemma 5 the robot terminates within $O(n)$ rounds. Therefore, assume the configuration is not a tree. Lemma 5 states that the robot will find an overhang, which, by Lemma 6, will subsequently be filled. Afterwards, the algorithm will be restarted (Line 55). We define a *possible overhang* as a maximal but finite column of vertically adjacent unoccupied nodes. Nodes that are not part of an overhang initially can only ever become part of one if they are inside a possible overhang. Since the robot does not create any new possible overhangs, it can only fill finitely many overhangs before the tile configuration is arranged as a tree. After performing a last traversal through the tree, the robot terminates.

We now turn to the runtime of the algorithm. Clearly, there are only $O(n)$ possible overhangs in an initial configuration. Since traversing the structure before finding an overhang takes $O(n)$ rounds by Lemma 5, and the algorithm is restarted at most $O(n)$ times, the total number of rounds needed for traversing the structure is $O(n^2)$. Since tiles are only moved *S* or *SE* and, by Lemma 7, tiles are never moved outside the initial configuration's convex hull, each tile is moved at most $O(n)$ steps. Furthermore, for every search step in direction *N* and *NW* in the `get_tile_N` phase and `get_tile_NW` phase, the robot either moves a tile by one step in the opposite direction, or moves a tile *S* and takes a single step *NE*. Therefore, $O(n^2)$ rounds are needed for searching and moving tiles, which implies that the robot terminates within $O(n^2)$ rounds. □

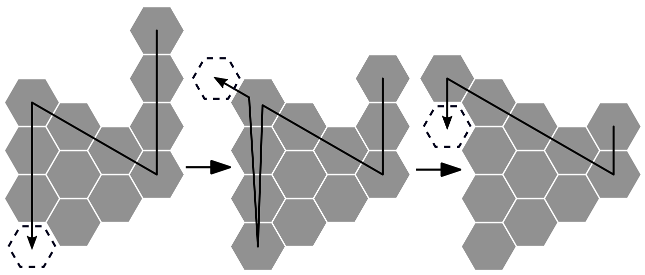
Fig. 7: Snapshots of triangle formation. If the number of tiles is not triangular, the final layer will not be completely filled.

4 Forming a Triangle

We will now describe how the robot can transform an intermediate structure into a *triangle*. More precisely, a triangle consists of columns whose northernmost tiles form a row, and each column consists of exactly one tile more than its eastern adjacent column (except for the westernmost column, which is only partially filled if n is not a triangular number). In the following, we assume that a block has already been built. It can be easily seen that a line and a tree can be transformed in a similar way. The triangle is built by repeatedly taking the easternmost tile of the block's northernmost row, carrying it south to the *vertex* of the forming triangle, and adding it to the westernmost *layer* of the triangle (see Fig. 7).

First, the robot creates the vertex (i.e., the easternmost column consisting of one tile) of the triangle by placing a tile on the node v below the westernmost column of the block. A second tile is then placed *NW* of v. Every other tile of the triangle is then placed as follows. The robot brings a tile to the triangle's vertex, and then walks *NW* and *S* (in that precedence) there is not tile in any of these directions. If there is a tile at *SE*, the robot moves one step *S* and places the tile. Otherwise, the robot moves *N* to the top of the layer, takes one step *NW*, and places the tile. In this manner, the robot continues to extend the triangle tile by tile until the block only consists of the triangle's vertex. By Theorem 4, and since each tile can be brought and placed within $O(D)$ rounds, we conclude the following theorem.

Theorem 6 *There exists a deterministic finite automaton that the robot can execute to transform any connected tile configuration into a triangle and terminate after $O(nD)$ rounds.*

It is not hard to see, using similar arguments as in the previous sections, that the runtime is asymptotically optimal. In the case that an initial configuration's diameter is low, i.e., $D = O(n^{1/2})$, we conclude that it can be rearranged into a triangle in $O(n^{3/2})$ rounds.

5 Towards Multiple Robots

As a first step towards extending our algorithms to the multi-robot case, we show that multiple robots can cooperatively construct a triangle using a line as an intermediate structure. We believe that some of our ideas may also be useful to solve more difficult problems. First, we present and discuss the underlying model assumptions. Then, we briefly describe how the line formation algorithm can be adapted for multiple robots. We experimentally show that the construction of a line can be sped up significantly by using multiple robots. Finally, we describe a simple algorithm to transform a line into a triangle using multiple robots.

Model Discussion. We consider the following extension of our model to incorporate multiple robots. For brevity, we leave out a formal definition as in Section 1.1. Each node is occupied by at most one robot at any time. We adapt our notion of connectivity and require all robots to be adjacent to occupied nodes, and the subgraph of G induced by all occupied nodes T and the positions P_c of all robots carrying tiles to be connected. In the look phase, for each adjacent node a robot can additionally observe whether the node is occupied by another robot, and determine the state of that robot. It then uses this information to determine its next state and move in the compute phase, and may change the state of each adjacent robot. In the move phase, the robot is further allowed to pass a carried tile to an adjacent robot that does not yet carry a tile.

We assume an asynchronous model in which robots are *activated* in an arbitrary sequence of *activations*, where a robot performs exactly one look-compute-move cycle before the next robot is activated (see, e.g., [4]). Correspondingly, a round is over whenever each robot has been activated at least once. For simplicity, we not only assume that all robots have the same chirality, but share a common compass. In fact, lifting this restriction imposes difficult challenges outside the scope of this paper, since symmetry breaking is very hard in our deterministic model. We leave this issue as a future research question.

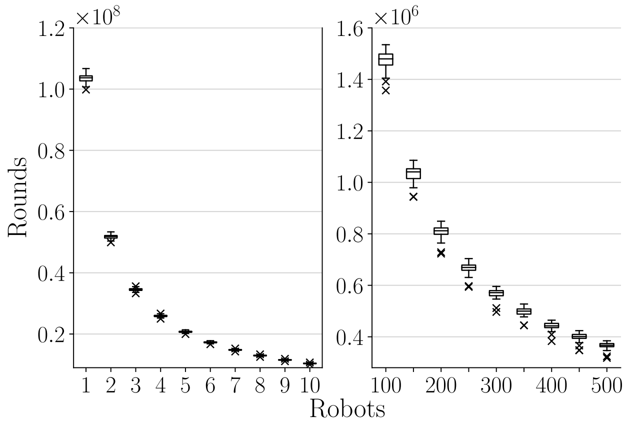

Fig. 8: Simulation results for 10000 tiles.

Distributed Line Formation. In order to extend the line algorithm to work with multiple robots, we propose three main modifications to the line formation algorithm. The pseudocode of the algorithm can be found in Algorithm 2. First, a robot r that carries a tile and is blocked in S or SE direction by a robot that searches for a tile can pass its tile and state to the blocking robot. Additionally, if r stands on a tile, it turns to the search phase. Otherwise, r has left the tile structure (we say it is *hanging*) and subsequently traverses its boundary in clockwise order, maintaining its connectivity to the outline of the tile structure (i.e., the outermost tiles of the tile structure), until it reaches an empty tile to step on. We make sure that no hanging robot is disconnected from the tile structure by a robot picking up a tile by performing additional checks.

Second, we ensure that no hanging robot ever ends up in a deadlock whilst traversing the boundary by avoiding to walk into *bottlenecks*, i.e., empty nodes with tiles on two non-adjacent sides. A traversal that avoids bottlenecks is depicted in Fig. 9. Finally, in order to eventually let each robot detect that the line has been built, we slightly modify the way tiles are moved. More specifically, we do not immediately move a lifted tile SE and place it at the first empty position in the column as in the single-robot algorithm. Instead, after lifting a tile, a robot first walks S until it actually encounters a column to its east, and only then moves SE to place its tile there; if there is no such column, the tile is simply placed at the bottom of the current column. This way, termination of line formation can easily be determined by all robots.

Simulation Data. Although correctness can be proven for this multi-robot approach, it is difficult to make any runtime guarantees. This is due to the fact that, when there are many robots compared to the number of tiles, many robots are blocked by others and must wait to

Algorithm 2 Algorithm to form a line using multiple robots.

```
Remark: Whenever the movement of a robot $r$ that carries a tile is blocked by a robot $r'$ that does not carry a tile, $r$ passes
its tile and state to $r'$ and goes to phase search, if $r$ stands on a tile, and to phase hanging, otherwise. In the latter case
$r$, sets next_dir to N. If $r$ is blocked by a robot also carrying a tile, it waits.
 1: phase search:
 2:     find a locally northwesternmost tile by moving $NW$, $SW$, and $N$
 3:     wait whenever the next tile is already occupied by a robot
 4:     if removing the tile does not locally disconnect a hanging robot then
 5:         lift the tile
 6:         is_line ← TRUE
 7:         goto carry_orig_col
 8: phase carry_orig_col:                                    ▷ Move tile $S$ until an eastern column is reached
 9:     if there is western or eastern (carried) tile then                              ▷ Not a line yet
10:         is_line ← FALSE
11:     if (carried) tile at $NE$ or $SE$ then                                ▷ Reached eastern column
12:         move $SE$ and goto carry_next_col
13:     else if (carried) tile at $N$ and not standing on tile then                    ▷ End of column
14:         drop tile
15:         if is_line then goto done else goto search
16:     else if robot in phase done at $S$ then                   ▷ Line has been built, pass tile down
17:         pass tile and state to robot at $S$ and goto done
18:     else
19:         move $S$
20: phase carry_next_col:                                  ▷ Move $S$ as far as possible and drop tile
21:     if standing on tile then
22:         move $S$
23:     else
24:         drop tile and goto search
25: phase hanging:                ▷ Traverse the boundary of the tile structure until an empty node is reached
26:     if adjacent to empty tile then
27:         move there and goto search
28:     for $i$ in $(-2, -1, 0, 1, 2)$ do         ▷ Choose next direction according to clockwise traversal; the or-
                                              der ensures that the robot does not walk through a bottleneck
29:         let $d$ ($d'$) be the cardinal direction obtained by turning $i$ ($i + 1$) steps in clockwise order starting at next_dir
30:         if no tile at $d$ and tile at $d'$ then
31:             next_dir ← $d$
32:             if no robot at next_dir then
33:                 move next_dir
34:             break
```

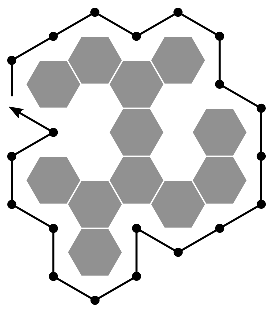
Fig. 9: A boundary traversal that does not pass through bottlenecks.

make progress. However, as a first step, we experimentally evaluated the number of rounds until all robots halt. The results for $n = 10000$ and a varying number k of robots can be found in Fig. 8. We conducted 50 simulations for each k, each initialized with a randomly generated tile configuration on which the robots were randomly placed. The robots were activated in a random order, each exactly once in every round. Each tile configuration was generated by the following procedure: First, randomly choose $10000 \cdot 16^2/2.02$ nodes of an equilateral parallelogram with side length $\sqrt{10000} \cdot 16$ to be occupied by a tile. Then, repeat the experiment until the largest connected component of the generated tile set contains at most 10500 tiles. The final configuration is obtained by repeatedly removing random tiles from the component whose removal does not disconnect the structure until 10000 tiles remain.

The simulations show that using a reasonably small number of robots significantly reduces the required number of rounds compared to using a single robot. The curve on the left part of Fig. 8 first decreases almost linearly (e.g., going from one to two robots essentially halves the runtime). However, for a large number of robots the benefit gained from employing more robots is almost negligible (right part of Fig. 8). This phenomenon can likely be explained by the fact that the likeliness of robots waiting on each other increases with the number of robots. Nevertheless, these preliminary results suggest that the model indeed allows multi-robot algorithms whose runtime drastically decreases if the number of robots is reasonably small.

Distributed Triangle Formation. If the structure is arranged as a line, a triangle can easily be built in a distributed manner: In order to retrieve tiles, the robots traverse the line from south to north. Once a robot reaches the northernmost tile, it waits until there is no robot north of it anymore, then lifts the tile and carries it to the triangle by following the boundary of the structure in clockwise order. The next position to place a tile into the triangle can easily be found, and the robot returns to the line by moving to the triangle's vertex. Whenever a robot's move is hindered by another robot, it simply waits.

Arguably, this algorithm makes use of multiple robots more effectively than the distributed line algorithm: no robot is ever forced to leave the tile structure, or to wait in order to preserve connectivity. This higher degree of coordination between the robots is facilitated using additional knowledge of the tile structure; coming up with a similar strategy for arbitrary structures seems to be rather difficult.

References

1. Bonato, A., Nowakowski, R.J.: The Game of Cops and Robbers on Graphs. AMS (2011)
2. Chirikjian, G., Pamecha, A., Ebert-Uphoff, I.: Evaluating efficiency of self-reconfiguration in a class of modular robots. Journal of Robotic Systems **13**(5), 317–338 (1996)
3. Das, S.: Mobile agents in distributed computing: Network exploration. Bulletin of the European Association for Theoretical Computer Science **109**, 54–69 (2013)
4. Daymude, J.J., Hinnenthal, K., Richa, A.W., Scheideler, C.: Computing by programmable particles. In: Distributed Computing by Mobile Entities: Current Research in Moving and Computing, pp. 615–681. Springer, Cham (2019)
5. Demaine, E., Demaine, M., Hoffmann, M., O'Rourke, J.: Pushing blocks is hard. Computational Geometry **26**(1), 21–36 (2003)
6. Demaine, E., Tachi, T.: Origamizer: A practical algorithm for folding any polyhedron. In: Proc. 33rd International Symposium on Computational Geometry (SoCG), pp. 34:1–34:16 (2017)
7. Demaine, E.D., Fekete, S.P., Scheffer, C., Schmidt, A.: New geometric algorithms for fully connected staged self-assembly. Theoretical Computer Science **671**, 4–18 (2017)
8. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: Proc. 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pp. 289–299 (2016)
9. Fomin, F.V., Thilikos, D.M.: An annotated bibliography on guaranteed graph searching. Theoretical Computer Science **399**(3), 236–245 (2008)
10. Gmyr, R., Hinnenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C.: Shape recognition by a finite automaton robot. In: 43rd International Symposium on Mathematical Foundations of Computer Science (MFCS, pp. 52:1–52:15 (2018)
11. Hurtado, F., Molina, E., Ramaswami, S., Sacristán, V.: Distributed reconfiguraiton of 2D lattice-based modular robotic systems. Autonomous Robots **38**(4), 383–413 (2015)
12. Lund, K., Manzo, A., Dabby, N., Michelotti, N., Johnson-Buck, A., Nangreave, J., Taylor, S., Pei, R., Stojanovic, M., Walter, N., Winfree, E.: Molecular robots guided by prescriptive landscapes. Nature **465**(7295), 206–210 (2010)
13. Michail, O., Spirakis, P.G.: Simple and efficient local codes for distributed stable network construction. Distributed Computing **29**(3), 207–237 (2016)
14. Murata, S., Kurokawa, H., Kokaji, S.: Self-assembling machine. In: Proc. IEEE Int. Conf. on Robotics and Automation (ICRA), pp. 441–448 (1994)
15. Omabegho, T., Sha, R., Seeman, N.: A bipedal DNA brownian motor with coordinated legs. Science **324**(5923), 67–71 (2009)
16. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Natural Computing **13**(2), 195–224 (2014)
17. Pelc, A.: Deterministic rendezvous in networks: A comprehensive survey. Networks **59**(3), 331–347 (2012)
18. Reif, J.H., Sahu, S.: Autonomous programmable DNA nanorobotic devices using dnazymes. Theoretical Computer Science **410**, 1428–1439 (2009)
19. Rothemund, P., Winfree, E.: The program-size complexity of self-assembled squares. In: Proc. 32nd Annual ACM Symposium on Theory of Computing (STOC), pp. 459–468 (2000)
20. Shin, J., Pierce, N.: A synthetic DNA walker for molecular transport. Journal of the American Chemical Society **126**, 4903–4911 (2004)
21. Terada, Y., Murata, S.: Automatic modular assembly system and its distributed control. International Journal of Robotics Research **27**(3–4), 445–462 (2008)
22. Thubagere, A., Li, W., Johnson, R., Chen, Z., Doroudi, S., Lee, Y., Izatt, G., Wittman, S., Srinivas, N., Woods, D., Winfree, E., Qian, L.: A cargo-sorting DNA robot. Science **357**(6356) (2017)
23. Tomita, K., Murata, S., Kurokawa, H., Yoshida, E., Kokaji, S.: Self-assembly and self-repair method for a distributed mechanical system. IEEE Transactions on Robotics and Automation **15**(6), 1035–1045 (1999)
24. Wang, Z., Elbaz, J., Willner, I.: A dynamically programmed DNA transporter. Angewandte Chemie International Edition **51**(48), 4322–4326 (2012)
25. Wickham, S., Bath, J., Katsuda, Y., Endo, M., Hidaka, K., Sugiyama, H., Turberfield, A.: A DNA-based molecular motor that can navigate a network of tracks. Nature Nanotechnology **7**(3), 169–173 (2012)
26. Woods, D., Chen, H., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In: Proc. 4th Conference of Innovations in Theoretical Computer Science (ITCS), pp. 353–354 (2013)