Connected Reconfiguration of Lattice-Based Cellular Structures by Finite-Memory Robots

Sándor P. Fekete · Eike Niehs · Christian Scheffer · Arne Schmidt

Received: 30 July 2020 / Accepted: 6 June 2022 / Published online: 24 June 2022 © The Author(s) 2022, corrected publication 2022

Abstract
We provide algorithmic methods for connected reconfiguration of lattice-based cellular structures by finite-state robots, motivated by large-scale constructions in space. We present algorithms that are able to detect and reconfigure arbitrary polyominoes, while also preserving connectivity of a structure during reconfiguration; we also provide mathematical proofs and performance guarantees. Specific results include methods for determining a bounding box, scaling a given arrangement, and adapting more general algorithms for transforming polyominoes.

Keywords Finite automata · Reconfiguration · Polyominoes · Robots · Connectivity

1 Introduction
Building and modifying large-scale structures is a fundamental task in many practical settings, ranging from classic manufacturing and building construction to modern
aspects of digital production, engineering and even medicine. In many cases, the use of autonomous agents (such as robots) promises significant advantages, but also a number of additional challenges. Particularly intriguing challenges arise in space, with difficulties of expensive supply chains, scarcity of building materials, dramatic costs and consequences of even small errors, and the limitations of outside intervention in case of malfunctions. For many applications, an important aspect in this process is connectivity of the overall arrangement throughout the reconfiguration process, which is critical for overall coherence (in particular for assemblies in space, where disconnected pieces cannot regain connectivity), or for autonomous devices (such as catoms in claytronics [46]), which need connectivity for local motion, electric power and communication.

As described in parallel, practically oriented work [63], and visualized in our video [1], a number of significant advances have been made to facilitate overall breakthroughs; see Fig. 1 for an illustration. One important step has been the development of ultra-light and scalable composite lattice materials [48] that allow the construction of modular, reconfigurable, lattice-based structures [55]. A second step has been the design of simple autonomous robots [54, 56] that are able to move on the resulting lattice structures and move their elementary cell components, thereby allowing the reconfiguration of the overall edifice. Combining such materials and robots in space promises to greatly increase the dimensions of constructible facilities and spacecraft, as well as offering to extend mission capabilities with reconfiguration and re-use [47]. Because moving mass and volume into space remains a critical factor, it is conceivable that the involved robots (just like the building material) can be produced and assembled in space; because robots will still require more complex production steps than the passive material, their availability will remain a bottleneck, so limiting their number is highly desirable. Moreover, production in space is not possible for sophisticated components such as complex microchips, adding a second objective: enabling the use of very simple robots.

In this paper, we provide algorithmic foundations for enabling extremely simple robots, i.e., finite automata, to perform a complex spectrum of construction tasks for cellular structures in space, such as computing a bounding box for a cellular arrangement, scaling up a given seed construction, and a number of other design operations. At all stages in this algorithmic process, we maintain connectivity of the overall assembly.

1.1 Our Results

We present the following results.

1. We show how two finite-state robots suffice to construct a bounding box for a given connected planar arrangement of grid cells (a *polyomino P*) in a limited number of steps; see Fig. 1 for an illustration.

2. We provide an algorithmic method that enables two finite-state robots to construct (for a polyomino *P* with a surrounding bounding box) a scaled up copy of *P* in a limited number of steps, while preserving connectivity of intermediate arrangements; see Fig. 2.
Fig. 1  Demonstrating the connection between the simplified theoretical model and real-world robots: snapshots from building a bounding box for a Z-shaped polyomino using 2D simulator, 3D simulator, and staged hardware robots, synchronized so all are shown at steps {0, 24, 48, 72, 96, 120}

Fig. 2  Sequence from the video [1] illustrating the scaling procedure. (Note that in that video, the blue marker is not distinguished from the red robot; in the rest of this paper, we make that distinction by using a different shape for the marker.) We start with a polyomino $P$ (blue) in the top left and create a copy of $P$ that is scaled by factor of 3 (bottom right). See https://www.ibr.cs.tu-bs.de/users/fekete/Videos/SoCG/2020/Space_final.mp4, in particular, 4:35–5:38

3. We give a more efficient method for scaling a monotone polyomino $P$, while preserving connectivity of intermediate arrangements.
4. We describe how other arbitrary reconfigurations can be adapted to our connected scenario.
1.2 Related Work

The structures considered in this work are motivated by the possibilities of ultra-light material, as described by Cheung and Gershenfeld [11] and Gregg et al. [48]. Modular two-dimensional elements mechanically link in 3D to form reversibly assembled composite lattices. This process is not limited by scale, and it enables disassembly and reconfiguration. As shown by Cramer et al. [14] and Jenett et al. [53], large but lightweight structures can be built from these components. Jenett et al. have developed autonomous robots that move on the surface [52, 56] or within the cellular structure [54]. With the help of these robots, individual cells can be attached to an existing assembly, or moved to a different location. An approach for global optimization of a corresponding motion plan has been described by Costa et al. [13], while the design of hierarchical structures was addressed by Jenett et al. [57].

Assembly by simple robots has also been considered at the micro scale, where global control is used for supplying the necessary force for moving agents, e.g., see Becker et al. [6] for the corresponding problem of motion planning, Schmidt et al. [66] for using this model for assembling structures, and Balanza-Martinez et al. [5] for theoretical characterizations. Distributed self-assembly for modular robots with limited computing resources was studied by Tucci et al. [68]. Self-configuration of robots themselves has been considered by Naz et al. [62]. A basic model in which robots themselves are used as building material was introduced by Derakhshandeh et al. [23, 24]. This resembles Claytronics robots like Catoms, see Goldstein and Mowry [46]. A large spectrum of methods for tile-based assembly in biology has also been considered. Examples include DNA self-assembly, introduced by Winfree [69, 70] and extended to a variety of models [10, 20]. See Patitz [65] for a survey.

There has been a considerable amount of classic algorithmic work dealing with robots or agents on graphs. Blum and Kozen [8] showed that two finite automata can jointly search any unknown maze. Other work has focused on exploring general graphs (e.g., [35, 40, 64]), as a distributed or collaborative problem using multiple agents (e.g., [7, 9, 18, 28, 38]) or with space limitations (e.g., [3, 31, 39–41, 51]). Another problem related to robots on graphs is the rendezvous search problem [2], where two robots are searching for each other in a known or unknown environment. Anderson and Weber [4] presented an optimal strategy for discrete locations. The deterministic rendezvous problem as a variant of the previously described problem has also been considered [27, 61, 67]. For scenarios with more than two robots, we face the gathering problem, which was investigated on graphs [17, 30, 58] or, related to more realistic robotic scenarios, in the plane [12, 15, 37]. See also [36] for a comprehensive survey.

In the context of programmable matter, the spectrum of approaches includes self-folding matter [34, 49, 60] or self-disassembling magnetic robot pebbles [42]. From an algorithmic perspective, we are interested in different models representing programmable matter and further recent results. D’Angelo et al. [16] provide a simple model in which leader election can still be performed. Inspired by the single-celled amoeba, Derakhshandeh et al. [21] introduced the Amoebot model and later a generalized variant, the general Amoebot model [26]. The Amoebot model provides a framework based on a triangular lattice and active particles that can occupy a single
lattice vertex or a pair of adjacent vertices. With just a few possible movements, these particles can be formed into different shapes like lines, triangles or hexagons [23], and perform distributed tasks such as leader election [19, 26]. A universal shape formation algorithm within the Amoebot model was described by Di Luna et al. in [29]. An algorithm for solving the problem of coating an arbitrarily shaped object with a layer of self-organizing programmable matter was presented in [25] and analyzed in [22]. Kim et al. [59] generalized this model where each amoeba consists of two robots that simulate extend and retract by allowing at most two robots on a single cell. Other models with active particles were introduced in [71] as the Nubot model and in [50] with modular robots. Gmyr et al. [45] introduced a model with two types of particles: active robots acting like a deterministic finite automaton and passive particles. Furthermore, they presented algorithms for shape formation [44] and shape recognition [43]. Based on this model, Fekete et al. [32] presented algorithms to bound a given shape and for counting the number of particles or corners. They also introduced more complex geometric algorithms for copying, reflecting, rotating and scaling a given polyomino as well as an algorithm for constructing a bounding box surrounding a polyomino; however, this work did not account for connectivity of the particle configurations, which is critical for applications as the ones mentioned earlier.

2 Preliminaries

In the following, we introduce models, general definitions as well as a description of the underlying limitations.

2.1 Model

We consider an infinite square grid graph $G = (V, E)$, where $V = \mathbb{Z}^2$ is the set of pixels, and two pixels $p_1$ and $p_2$ are adjacent when $||p_1 - p_2||_1 = 1$. We call two pixels a diagonal pair when both their $x$-coordinates and their $y$-coordinates differ by one. We use the compass directions ($N, E, S, W$) for orientation when moving on the grid with a global orientation.

Every pixel of $G$ is either occupied by a passive particle (i.e., a tile) or empty. Passive particles cannot move or manipulate themselves. A maximal connected set of occupied pixels is called a polyomino. A tile $p$ of a polyomino $P$ is called convex if there is a $2 \times 2$ square solely containing $p$. Throughout this paper, we denote the number of convex tiles by $k$.

The boundary of a polyomino $P$ is denoted by $\partial P$ and includes all tiles of $P$ that are adjacent to an empty pixel or that form a diagonal pair with an empty pixel (see also Fig. 3 (a)). Polyominoes can have holes, i.e., finite maximal connected sets of empty pixels (see Fig. 3 (b)). Polyominoes without holes are called simple; otherwise, they are non-simple. The bounding box $bb(P)$ of a given polyomino $P$ is defined as the boundary of the smallest axis-aligned rectangle enclosing but not touching $P$; see Fig. 3 (d). Because the bounding box and polyomino are comprised of indistinguishable particles, we use a gap to differentiate the two.
We use a robot as active particle. This robot works like a finite deterministic automaton that can move around on the grid and manipulate the polyomino. Because we only consider one robot in this paper, the robot knows the global orientation. In particular, we may assume that the robot’s local orientation and the global orientation match in the beginning. As there are only four possible orientations (looking north, east, south, or west), the robot can always return to the initial orientation. The robot has the ability to move along the edges of the grid graph, to change the state of the current pixel by placing or removing a tile on it, and to place a special marker on top of a tile. These markers are indistinguishable passive particles that allow the robot to recognize a specific tile. The robot works in a series of Look-Compute-Manipulate-Move cycles.

Look Phase: The robot checks if it stands on a tile and if a marker lies on top.
Compute Phase: The robot computes the next motion step based on the current state and the information received from the look phase.
Manipulation Phase: The robot sets the state of the current pixel to empty or occupied, or places / removes a marker from a tile.
Move Phase: The robot performs the computed motion step w.r.t. its local orientation, which is one of the form \{up, right, down, left, ⊥\} with ⊥ denoting no motion.

Note that we can extend the look phase by checking any constant sized neighborhood. In particular, we may assume that the robot is able to observe the state of all eight surrounding pixels. For this we define the following look-up functions w.r.t. the robot’s local orientation.

\texttt{isOnTile}: Checks if the robot is on a tile.
\texttt{TOP, BOTTOM, LEFT, RIGHT}: Returns the state of the pixel to the top, bottom, left, or right of the robot’s current position.
\texttt{TOP-LEFT, TOP-RIGHT, BOTTOM-LEFT, BOTTOM-RIGHT}: Returns the state of the pixel in the respective corner of the robot’s current position.
\texttt{TOP-ROW, BOTTOM-ROW, LEFT-COLUMN, RIGHT-COLUMN}: Returns the states of all (diagonal) adjacent pixels that lie above, left, below, or right of the robot.

Further, many of the algorithms presented in this paper use a subroutine, where the robot moves into a given direction as long as there is still a tile in this direction. We denote this motion by appending \texttt{most} to the moving direction, e.g., \texttt{rightmost}.

In the following, we assume that connectivity is ensured if the union of all tiles and the robot is connected.
3 Constructing a Bounding Box

Fekete et al. [32] showed how to construct a bounding box around a polyomino. However, that algorithm does not guarantee connectivity. We describe an algorithm to construct the bounding box, maintaining connectivity after each step. To accomplish the required connectivity, we specify without loss of generality that the connection between $bb(P)$ and $P$ must be on the south side of the boundary. For easier distinction, we show the tiles of the original polyomino in blue, while those of the bounding box (which is composed of additional tiles that are added during the process) are shown in gray. Note that this is for illustrative purposes only; the robots cannot distinguish between those tiles. (We follow this convention in the illustrations throughout this paper.)

The construction can be split into three phases: (1) finding a start position, (2) constructing the bounding box, and (3) clean-up.

3.1 Phase 1: Find a Start Position

To find a suitable start position, the robot $R$ searches for a locally $y$-minimal pixel that is occupied by a tile; see Algorithm 1. This can be achieved by scanning the current row and moving down whenever possible. When there is no tile further down, $R$ places a tile $t$ with marker below the left most tile. Afterwards, $R$ checks if there is sufficient space to start the bounding box below $t$, i.e., if there is no adjacent tile from $P$. With sufficient space, $R$ can start constructing the bounding box and we proceed with phase (2). If not, we can restart phase (1) from the tile blocking construction.

```
1: function GETSTARTPOSITION
2:   FINDLOCALMINIMUM
3:   Move down
4:   if BOTTOM-ROW not empty then
5:       Move to an occupied position
6:       Go to Step 2
7:   Place tile with marker
8:   Move down
9:   if BOTTOM-ROW not empty then
10:      $x :=$ BOTTOM
11:      Move down and place tile
12:      Remove tile with marker and turn back
13:      Change current pixel to state $x$
14:      Move to one occupied pixel
15:      Go to Step 2
```

Algorithm 1: Algorithm to find a valid start position to build the bounding box (left)
and to find a local minimum (right).
3.2 Phase 2: Constructing the Bounding Box

The construction of the bounding box is performed clockwise around $P$, i.e., whenever possible, $R$ makes a right turn. For this, several subroutines will be performed.

CHECKTILE: Starting at a tile $t$ lying on a boundary, the robot follows this boundary until the marked tile is found. This function decides, if a boundary tile belongs to $P$ or the current construction. See also Algorithm 2.

SHIFTLINE: The last constructed line of the current construction gets shifted along the boundary of $P$. This routine returns true, if the line has been shifted. False is returned, if it is the first line of the construction. See also Algorithm 3.

CHECKTURN: Checks if a right turn is possible. If so, the robot performs the turn. See also Algorithm 4.

RESTART: When there is a tile $t$ of $P$ to the left of the starting position, the robot will not be able to construct $bb(P)$ completely. When $t$ is found, we call this routine to disassemble our construction to initiate a restart with phase (1) at $t$. See also Algorithm 5.

```
1: function CHECKTILE
2:   Turn left by $\pi/2$
3:   while No adjacent tile with a marker do $\triangleright$ Move along boundary.
4:     if LEFT $\neq$ empty then
5:       Turn left by $\pi/2$
6:       Move up
7:     else if TOP $\neq$ empty then
8:       Move up
9:     else
10:       Turn right by $\pi/2$.
11: $b = 0$
12:   if The marked tile lies south then
13:     $b = 1$
14:   Move two times down
15:   Set direction to East and move up $\triangleright$ Move back along the construction
16:   while TOP $\neq$ empty or RIGHT $\neq$ empty do
17:     if TOP $\neq$ empty then
18:       Move up
19:     else
20:       Move right
21:     Turn right
22: return $b$
```

Algorithm 2: An algorithm to check whether a tile belongs to $P$.

We will now show how the interplay of these subroutines constructs $bb(P)$, see Algorithm 6. As long as we can extend the current line, we place a new tile and check if we can perform a right turn. At some point during construction, $R$ finds a tile either belonging to $P$ or to the current bounding box construction. To decide whether a tile $t$ belongs to $P$ or the current bounding box, we start moving around the boundary of...
Algorithm 3: An algorithm to shift a line.

```plaintext
1: function SHIFTLINE
2:   Move downmost
3:   if Tile on the right contains marker then
4:     return false
5:   Move left
6:   if LEFT-COLUMN is empty then
7:     Place tile
8:     Move right
9:   if TOP ≠ empty then
10:    Move up
11:   Goto Step 5
12: else
13:   Move right
14:   GotoLine
15: return true
```

Algorithm 4: An algorithm to perform a right turn if possible.

```plaintext
1: function CHECKTURN
2:   Move right
3:   if RIGHT-COLUMN is empty then
4:     Place tile
5:     Turn right by \(\pi/2\)
6: else
7:   Move left
```

Algorithm 5: An algorithm initiate a restart.

```plaintext
1: function RESTART
2:   Set local orientation to global orientation
3:   Move leftmost
4:   Move up, take marker, move down
5:   Move leftmost
6:   while LEFT ≠ empty do
7:     Move left
8:     Place tile
9:     Place marker
10: REMOVEBRIDGE
```

the shape \(t\) belongs to. At some point, \(R\) reaches the marked tile. If \(R\) arrives from north then \(t\) is a tile of \(P\), otherwise \(t\) is a tile of the bounding box. To find \(t\) again, we move to the south of the marked tile and follow the construction until we cannot move any further. From there we can carry on building \(bb(P)\).

Now consider the two cases: If the tile does not belong to \(P\), we are done with phase (2) and can proceed with phase (3). If it is a tile belonging to \(P\), we need to shift the current line outwards and then continue the construction (see Fig. 4). If there are tiles blocking the shift process, then we deconstruct the current bounding box construction until we can shift again or until we can proceed adding tiles to the current line. If the line to shift is the first line of the constructed bounding box, we know that there exists a tile of \(P\) that has the same \(y\)-coordinate as the current starting position. Therefore,
Fig. 4 (a) The robot hits a tile belonging to $P$. (b) The triggered shifting process is finished.

Fig. 5 Traversing a gap by building a bridge.

we build a bridge to traverse this gap, as shown in Fig. 5. Afterwards, we can restart from phase (1).

Note that after a shifting process, the robot would perform a new check if it is still blocked. This would take $O(|\partial P|)$ time for each tile on $\partial P$. However, by keeping track of $\text{TOP}$ before and after the shifting process, the robot can identify a continuous line of $P$’s boundary. Therefore, we can shift a line until we find a convex corner of $P$. Thus, we can modify the procedure such that $\text{CHECK} \text{TILE}$ is called at most $k$ times, where $k$ is the number of convex tiles of $P$.

1: function PHASETW0
2: Turn left by $\pi/2$
3: Move upmost
4: Move up
5: if TOP-ROW is empty then
6: Place tile
7: CHECKTURN
8: else
9: Let $t$ be the rightmost occupied tile in TOP-ROW.
10: if CHECKTILE then $\triangleright t$ is part of $P$
11: if not SHIFTLINE then
12: RESTART
13: GETSTARTPOSITION
14: Goto step 3
15: else $\triangleright$ Wrap around $P$ is complete
16: if $t$ lies top-right then
17: Place tile
18: Move up and turn right by $\pi/2$
19: else if $t$ lies top-left then
20: SHIFTLINE
21: Place Tile

Algorithm 6: An algorithm to perform phase 2.
3.3 Phase 3: Clean-up

We will now describe the final phase. A pseudocode for this phase can be found in Algorithm 7. Consider the case when $R$ reaches a tile of the bounding box. Firstly, we prepare the clean-up phase by shifting the last line outwards until the next corner is reached (see Fig. 6(a)). Let this position be $t$. Then we can search for another suitable connection between $P$ and $bb(P)$, place a tile there, and move back to $t$. From there, we move to the marked tile to remove all tiles of the bounding box between the marked tile and $t$ (see Fig. 6(b)-(d)). Note that during the preparation the robot may created a line of width two: When the last tile is placed underneath a corner that lies to the right, the robot continues construction until the next corner is hit. In order to remove the correct line in $\text{REMOVEAPPENDIX}$, we turn right whenever the next tile has a neighbor to the right and the top row contains at most two tiles (see Algorithm 7, $\text{REMOVEAPPENDIX}$, line 9).

Note that the new connection between $P$ and $bb(P)$ may not be on the south side of $P$. If desired, the robot can simply add a new connection to any side and remove the old connection by moving around the bounding box. This way, we can also ensure that the connection will always be on the south side of $P$.

**Theorem 1** Let $P$ be a polyomino of width $w$ and height $h$ with $k$ convex tiles. Constructing $bb(P)$ such that $bb(P)$ and $P$ are always connected can be performed by one robot with one marker in $O(\max(w, h) \cdot (w h + k \cdot |\partial P|))$ steps.

**Proof** Correctness:

It is straightforward to see that the robots will find a local $y$-minimum in phase (1): Consider a horizontal line $\ell$ the robot is on. If $\ell$ has tiles adjacent to the south, the robot eventually finds such tile by the scan from right to left. If $\ell$ has no tiles to the south, then $\ell$ is a local minimum and the robot will stop on the leftmost tile. By the simple check if there is sufficient space beneath this found local minimum, we can guarantee a suitable start position to construct $bb(P)$.

For phase (2), we first prove that the membership test of a tile $t$ is decided correctly by the robot. For that let $\mathcal{F}$ be the set of all maximally connected components of empty
Algorithm 7: An algorithm to clean up the construction to obtain $bb(P)$.

pixels. Assume that the tiles of the bounding box are placed in one of these connected components $F \subset \mathcal{F}$.

Then, $t$ is adjacent to a position of $F$, and also the marker is placed on a tile that is adjacent to a position of $F$. Because the robot traverses a boundary of $P$, it never steps on a tile that is not adjacent to a position of $F$, and thus the robot eventually reaches the marked tile. It is straightforward that $t$ must belong to $P$ if the marker is met from above, because the robot started the bounding box construction below the marker. In summary, the robot can decide whether $t$ belongs to $P$ or to the bounding box.

For the remaining correctness proof, we show that the construction encloses $P$ after phase (2). We show that (a) the tile of the construction placed last has distance two to the nearest tile of $P$ using the $L_\infty$-metric, and (b) we do not construct loops not containing any tile of $P$. It is straightforward to see that (a) and (b) are true after phase (1). Now assume both are true after placing tile $t$. We will show that (a) and (b) also hold after placing the next tile $t'$. Consider the operations from the algorithm.

Make a right turn: When we find a convex corner $t_{\text{corner}}$, we perform a right turn. Then $t$ and $t'$ are within distance two of $t_{\text{corner}}$.

Shift line: Because the next tile would connect a tile $t_{\text{shift}}$ of $P$ and $t$, a shift by one unit will not increase the distance between $t'$ (i.e., $t$ shifted by one unit) and $t_{\text{shift}}$.

Deconstruct line: This is similar to the previous case. Consider the lowest tile $t_{\text{block}}$ of $P$ that blocks further shifting. After deconstructing the current line two units below $t_{\text{block}}$, the remaining last tile $t'$ of the construction has distance two to $t_{\text{block}}$. 
Combining all three cases, the last placed tile of the construction has a tile of $P$ within distance two. Therefore, by induction, (a) holds for during the whole construction process.

Because we only perform a right turn after finding a convex corner of $P$, we must enclose at least one tile of $P$. Combining (a) and (b) it follows that we enclose $P$.

The easy structure of the construction after phase (2) make it straightforward to see that we can easily find a new connection between $bb(P)$ and $P$, and move to the marked tile to remove unwanted tiles in phase (3).

**Connectivity:** While assembling the bounding box or checking the membership of a tile, there is always the marked tile that holds together $P$ and the current construction. Whenever we have to build a bridge for traversing a gap, the constructed bridge between the previous starting position and the first intersection with $P$ will not be removed until one robot ensures connectivity at this intersection. Finally, there is at least one connection between $P$ and the bounding box when we are finishing the construction.

**Time:** Again, it is straightforward to see that phase (1) needs at most $O(N) \subseteq O(wh)$ steps to find a minimum. This includes also possible restarts from phase 2, because each maximal line of $P$ can be visited at most once.

For phase (2), we split the runtime analysis into two parts: check a tile for membership and the actual construction. Analogous to [32], we show that in the actual construction every position within the final bounding box is visited a constant number of times for a fixed starting position in phase (2). Consider a pixel $v$ within the final bounding box. At some point, $v$ will be visited for the first time while constructing $bb(P)$. Other visits are possible during shifting and removing the boundary tile placed on $v$. However, after a shift or removing the tile, which is part of the shifting process, we move away from $v$ and never visit $v$ again. Finally, there may be a constant number of additional visits while building a bridge for searching a new starting position. This results in $O(wh)$ steps.

However, during the construction, we may need to restart up to $\max(w, h)$ times. Figure 7 shows a worst-case example. There are possibly up to $\max(w, h)$ starting positions $s_i$ that have a pixel somewhere to the left blocking the construction at some point during construction. Then, a pixel $v$ could be visited after every restart of phase (2) implying that $v$ can be visited up to $\max(w, h)$ times. This yields $O(\max(w, h) \cdot wh)$ steps for the construction process.

For every tile that leads to a conflict during construction, the decision of whether that tile belongs to $P$, or the bounding box is completed by using the subroutine **checkTile** in Algorithm 2. This subroutine takes $O(|\partial P|)$ steps for tiles of $P$ and $O(|C|)$ steps for tiles on the current construction $C$ of $bb(P)$. It is also straightforward to see that $O(|C|) \subseteq O(|\partial P|)$: Every tile of $P$ can be charged with constant many tile of $C$. To this end, consider rays emanating from a tile $t \in C$. Then at least one ray hits a tile from $P$. Any tile of $\partial P$ can only hit by at most four of these rays. If no ray hits a tile of $P$, then $t$ must be a corner or adjacent to a corner of $C$. Because we only make turn at convex tiles, $C$ can have at most $k$ convex tiles. Therefore, these convex tiles and their neighbors can be charged to one convex corner of $\delta P$. In total, any tile of $\delta P$ is charged with $O(1)$ cost.
For every starting position $s_i$ there is some tile to the left of $s_i$ blocking the shifting process and forcing a restart. If $s_1$ is the first start position, we have $\max(w, h)$ restarts. Then the pixel $v$ is visited once after every restart.

As described above, we can shift a line until we reach a convex corner of $P$ without repeating a membership test. Thus, we only need to check for $O(k)$ tiles if they belong to $P$ or $bb(P)$. Therefore, we have a total of $O(\max(w, h) \cdot (wh + k \cdot |\partial P|))$ steps.

For phase (3) note that the robot moves at most twice over each tile of the current construction. Again, the number of these tiles can be bounded by $O(|\partial P|)$. Thus, the time complexity of phase (2) dominates the total time complexity.

If we know in advance that the given polyomino contains no holes, then we can build a non-simple bounding box. Because we can always look for a hole to distinguish tiles from $P$ and the bounding box, we require only a single robot without any markers. In particular, we adapt our algorithm as follows.

- The first bounding box element, which is placed right beneath a possible starting position has two vertically consecutive unoccupied pixels. (see Fig. 8 (a)).
- Instead of creating a boundary of width one as described before, we construct it as a strip of width three, consisting of an outer, middle and inner lane. On the middle lane, every second pixel is unoccupied. Because there are no holes within simple polyominoes, the robot can easily distinguish between $P$ and the bounding box (see Fig. 8 (b)).
Fig. 9 Bridge construction in case of simple polyominoes. (a) The bridge cannot be completed. (b) Using the additional bridge element

Fig. 10 Finishing process in case of simple polyominoes. (a) Searching for a new connection. (b) Using the additional connection element. (c) Finished construction after retracing to initial start and removing unnecessary boundary elements

- When the robot recognizes that there is a tile $t$ of $P$ to the left of the starting position, we build the bridge as shown in Fig. 9: We add the gadget of Fig. 8 (c) to $t$ and afterwards deconstruct the bounding box beginning with the starting position. Now we can restart with phase (1).
- The minimum distance between $P$ and $bb(P)$ can either be two or three and may differ for every side of the bounding box. Because it may be necessary to search for a new connection between $bb(P)$ and $P$ while finishing the construction, and the minimal distance between the southern bounding box and $P$ could be three, we need a second connection element, shown in Fig. 8 (d), with three vertically consecutive unoccupied pixels. For an example, see Fig. 10.

This yields the following corollary.

**Corollary 1** Given a simple polyominino $P$ of width $w$ and height $h$, building a bounding box surrounding $P$ such that the boundary and $P$ are always connected, can be done with one robot in $O(\max(w, h) \cdot wh)$ steps.
4 Scaling Polyominoes

Now we consider scaling a given shape by a positive integer factor $c$. We start by increasing the scale of the shape (upscaling, i.e., $c > 1$) in Sect. 4.1, before considering how to decrease the size (downscaling, i.e., $\frac{1}{c} < 1$) in Sect. 4.2.

4.1 Upscaling

In the following we assume that the robot has already built the bounding box and is positioned below the marked tile. See Figs. 2 and 11.

The scaling process can be divided into two phases: (1) the preparation phase, and (2) the scaling phase. In phase (1) we proceed as follows (see Algorithm 8).

- We fill up the last, i.e., rightmost column within $bb(P)$ except for the topmost pixel,
- add a tile in the second last column above the south side of $bb(P)$,
- remove the lowest tile (called column marker) and third lowest tile (called row marker) on the east side of $bb(P)$ (see Fig. 11(a)).

This yields three columns within the bounding box (including $bb(P)$ itself). The first (from west to east) is the current column of $P$ to scale. The second column, is used to ensure connectivity and to decide if we finished scaling the current column. The third column marks the current overall progress, i.e., we can find the pixel in the correct current column and row that we want to scale next.

After this preparation, we can remove the marked tile that was used for connectivity in the beginning. Note that in special cases the marked tile had distance two to the right side of $bb(P)$. In this case, we move the marker one unit to the left during the preparation. See Algorithm 8 line 4.

```
1: function SCALEPREPARATION
2:   Move rightmost
3:   Move left twice and up
4:   if is on marked tile then
5:     Remove marker and move left
6:     Place tile with marker and move right
7:   Place tile and move right
8:   while Top = empty do
9:     Place tile and move up
10:    Move right
11:   Move downmost
12:   Remove tile and move up twice  \(\triangleright\) Mark current column
13:   Remove tile and move down  \(\triangleright\) Mark current row
14:  while not on marked tile do
15:    Move left
16:  Remove marked tile and move down
```

Algorithm 8: An algorithm to initialize scaling $P$.

In phase (2) we consider four main operations using three variables. $b_{\text{COL}}$ denotes whether we scaled the last pixel of the current column, $b_{\text{PIXEL}}$ denotes the state of the
pixel that we want to scale, and $b_{\text{new}}$ denotes whether we started scaling up a new column. We initialize phase (2) with $b_{\text{col}} := 0$, and $b_{\text{new}} := 1$.

**Initialize next scaling:** The robot moves along the bottom side of $bb(P)$ until it finds an empty pixel. This is the column marker. Then we check if we need to prepare the next column. See Algorithm 9 left. Afterwards, we scale the next pixel.

**Prepare next column:** If $b_{\text{col}} = 1$, then we prepare the next column. We move the column marker one pixel left and remove the tile two pixels above the column marker, i.e., we place the row marker. In addition, we place a tile one pixel above and two pixel to the left of the column marker. If this pixel was already occupied, then we completely scaled $P$ and we can move on to phase (3). See Algorithm 9 right.

**Scaling the next pixel:** We start from the column marker and move up until we reach the row marker. We place a tile, move one pixel up and set its state to empty. If there is no tile to the left, then we are about to scale the last pixel in the current column. Let $b_{\text{col}}$ denote whether the pixel was empty or not. Then the robot moves to the pixel that is scaled next, i.e., the robot moves one pixel down and two pixels left. The robot stores the state of the pixel in $b_{\text{pixel}}$, and places a tile if $b_{\text{pixel}} = \text{empty}$. See also Algorithm 10 left. Afterwards, we perform the scaling operation.

**Perform the scaling operation:** First we move to the bottom side of the bounding box. Then we move left until the next pixel is empty. If $b_{\text{new}} = 1$, we build the $c \times c$-square to the left (with a middle tile missing if $b_{\text{pixel}} = \text{empty}$) and we set $b_{\text{new}} := 0$. Else, if $b_{\text{new}} = 0$, we move up until we step on an empty pixel and build the $c \times c$-square. We leave out the middle tile of the square if $b_{\text{pixel}} = \text{empty}$. In case of $c$ being even, this corresponds to the top-right pixel of the middle square. See also Algorithm 10 right.

```plaintext
1: function INITNEXTSCALE(b_{col})
2: Move downmost
3: Move rightmost
4: Move right
5: if b_{col} then
6: return PREPARENEXTCOLUMN
7: return 0

1: function PREPARENEXTCOLUMN
2: Place tile and move up twice
3: Move left and remove tile
4: Move down
5: Move left twice
6: b_{FIN} = isOnTile
7: Set pixel to occupied
8: Move down and right
9: Remove tile
10: return b_{FIN}
```

**Algorithm 9:** An algorithm to find the column marker and to prepare scaling the next column.

As seen in Algorithm 11, we iterate through these operations until we scale a column that only contains empty pixels. This is recognized when trying to place a tile during the preparation of the next column. See Algorithm 9 right, line 6. Afterwards, we start the clean-up phase. We remove all tiles from the old bounding box area, proceeding
Algorithm 10: An algorithm to scale the next pixel. *: While constructing the square, we leave out the middle pixel if \( b_{\text{POS}} \) is false.

Algorithm 11: An algorithm to perform phase 2 of the scaling procedure using Algorithms 9 and 10.

column-wise from right to left. Due to the simple structure, we know that we finished removing this area when the removed column has no empty pixel.

If necessary, all scaled empty tiles can also be removed. For each scaled column (e.g. from left to right), we remove empty scaled pixels from top to bottom until we find a scaled occupied pixel \( t \). Then we remove scaled empty pixels from bottom to top until we reach \( t \) again. This way, the whole arrangement is always connected.

**Theorem 2** After building \( bb(P) \), scaling a polyomino \( P \) of width \( w \) and height \( h \) by a constant scaling factor \( c \) without losing connectivity can be performed with a single robot in \( O(wh \cdot (c^2 + cw + ch)) \) steps.

**Proof** Correctness: We scan through the whole bounding box of \( P \) and scale every pixel. This implies that we scale every tile of \( P \). Because we scale column-wise, we ensure that every scaled tile is built at the correct position. Connectivity is guaranteed because we never remove a tile that is necessary to preserve connectivity.

**Time:** Each of the \( w \cdot h \) pixels within the bounding box of \( P \) is scaled. To this end, the robot has to move \( O(c(w + h)) \) steps to reach the position at which the scaled pixel needs to be constructed. A further \( O(c^2) \) steps are needed to construct the tile. Finding the next pixel to scale takes \( O(c(w + h)) \) steps, including going back, finding the correct column and row, and moving the row and column marker. In total we have \( O(wh(c^2 + cw + ch)) \) steps. \( \square \)
Fig. 11 Intermediate steps of scaling by factor $c = 3$. Dashed and dotted lines denote the robot’s paths. (a) Configuration after the preparation phase showing the column marker (dotted blue circle), row marker (dashed red circle) and the next tile to scale (teal circle). (b)-(d) Cases that appear during the scaling: (b) Scaling an occupied pixel; (c) scaling an empty pixel; (d) preparing the next column. See also Fig. 2

4.2 Downscaling

In this section, we show that a shape $P$ that is already scaled up by a factor $c > 1$, can be downscaled to its original size, i.e., we scale the shape by the factor $\frac{1}{c}$. We assume that the scaled area, i.e., the $cw \times ch$ area containing $P$, is filled with scaled empty pixels (i.e., $c \times c$ squares with their middle tiles missing) and the scaled tiles of $P$. If this is not the case, we can simply fill up this area until we obtain a rectangular box. See Fig. 12 for an arrangement that satisfies these conditions and Fig. 13 for an overview of the construction.

In the following, we describe a strategy that positions the downscaled version of $P$ at the bottom right corner of the scaled area. Similar to our other strategies, we proceed in three phases: (1) the preparation phase, (2) the downscaling phase, and (3) the clean-up phase.

In phase (1), we start building a vertical line $\ell_v$ on the right side of the scaled area starting to the right of the top right corner. We construct the line downwards until the last tile $t_s$ of $\ell_v$ has no tile to its left. The line $\ell_v$ will guarantee connectivity throughout the downscaling process, and we can use $t_s$ to recognize the start of the scaled area. Then, from the tile above $t_s$, we construct a horizontal line $\ell_h$ of length three and place a further tile above the rightmost tile of $\ell_h$. We use $\ell_h$ for indicating where we want to construct the downscaled version of $P$. In particular, the last tile of $\ell_h$ marks the current column of $P$ we are processing. See Fig. 12 for the configuration after the preparation phase. See also Algorithm 12 for pseudocode of phase (1).

In phase (2), we downscale $P$ from left to right and from bottom to top. We use the Boolean variable $b_{\text{NEW}}$ to decide whether to start a new column in the next iteration or not. We initialize phase (2) with $b_{\text{NEW}} = 1$. With $b_{\text{PIXEL}}$, we store the state of the scaled...
**Algorithm 12:** An algorithm to initialize downscaling.

```
1: function INITSCALEDOWN
2:     Move to the left side of an $c \times c$ square.
3:     Move upmost
4:     Move rightmost
5:     Move right
6:     while LEFT $\neq$ empty do $\triangleright$ Construct $\ell_v$
7:         Place tile and move down $\triangleright$ Place $t_s$
8:         Place tile and move up $\triangleright$ Construct $\ell_h$
9:         Repeat three times: Move right and place tile
10:     Move up and place tile
11:     Move down
```

**Fig. 12** (a) An arrangement that satisfies the described prerequisites. The $cw \times ch$ area contains $P$ (blue) and scaled empty pixels. (b) Starting arrangement for downscaling where the target area lies to the right of $P$ pixel that we want to downscale. We split phase (2) into three main operations: move to next pixel, get state of next pixel, and downscale pixel. See also Algorithm 13 for pseudocode of phase (2).

**Move to next pixel:** Assume the robot starts on $\ell_h$. Then the robot moves leftmost. Then, if $b_{\text{NEW}} = 0$, it moves up as long as there is a tile above our position and no tile to the left. Three cases occur:

1. $b_{\text{NEW}} = 0$ and there is a tile to the left (see Fig. 13 (a), (c), and (e) right). The robot moves leftmost.
2. $b_{\text{NEW}} = 1$ and there is no tile below the robot’s position (see Fig. 13 (b), and (d) right). In this case $b_{\text{NEW}} = 1$, i.e., we start a new column.
3. $b_{\text{NEW}} = 1$ and $t_s$ is below the robot (see Fig. 13 (f) right). In this case we know that the downscaling procedure is completed and we can proceed with phase (3).

The robot now stands in the bottom-left corner of the next scaled pixel.

**Get state of next pixel:** The robot moves to the center of the scaled pixel and stores the state of this scaled pixel, i.e. $b_{\text{PIXEL}} := \text{occupied}$, if the $c \times c$ square is completely filled with tiles, and $b_{\text{PIXEL}} := \text{empty}$, otherwise. Then we proceed with downscaling the pixel.

**Downscaling pixel:** We remove the scaled pixel currently hosting the robot, such that it sits on a pixel to the right after removal. While removing the top row of the $c \times c$ square, the robot checks if there are further tiles above the top row. If that is the case,
we set $b_{\text{NEW}} := 0$ (or $b_{\text{NEW}} := 1$, otherwise). For both cases see also Figure 13 left. After removing the $c \times c$ square, the robot moves down until there are no more tile below its position. In case there are no tiles to the right (i.e., it moved to $t_s$), it moves one step up. Then it moves right until no further tile lie to the right (i.e., the robot reached the rightmost tile of $\ell_h$). From there, it moves up until it steps on the first empty pixel. It places a tile, if $b_{\text{NEW}} = 0$, and moves one step down and left. This is the position where we want to downscale our current pixel. The robot changes the state of the pixel to empty if $b_{\text{PIXEL}} := \text{empty}$, or to occupied if $b_{\text{PIXEL}} := \text{occupied}$. Afterwards it moves one step right and moves down to $\ell_h$. If $b_{\text{NEW}} = 1$, then it extends $\ell_h$ by one tile on the right side plus a tile above it. Then it gets the state of the next pixel.

Algorithm 13: An algorithm to scale a shape down.

Figure 13 shows an example for phase (2). In the clean-up phase (3), we remove the auxiliary structures, first $t_2$, then $\ell_v$ from top to bottom. Afterwards, we can remove the rightmost tile of $\ell_h$, the tile above it and the last column from top to bottom, which is not part of the polyomino. To remove the rest of $\ell_h$, we start to remove the line from the right to left until there is a tile $t$ of the polyomino above $\ell_h$. Then we remove the remaining part of $\ell_h$ from left to right until we find $t$ again. Only the downscaled version of our original shape remains.

**Theorem 3** Given a polyomino $P$ scaled by $c$ of width $cw$, height $ch$. Downscaling $P$ by $c$ without losing connectivity can be performed without markers in $O(wh \cdot (c^2 + cw + ch))$ steps.
Fig. 13 Intermediate steps of downscaling a given polyomino $P$ (blue). The resulting polyomino is shown in dark grey. Left: Path of the robot for *scale down pixel*. Right: Path of the robot for *move to next pixel*.
Proof Correctness: We prove that (a) we scan the columns completely from bottom to top, and (b) we downscale each scaled pixel at the correct position. Assume that we leave out one scaled pixel \( p \) in a column. Then we will not consider the scaled pixels above \( p \) and we start a new column. This, however, requires that the robot has not seen any tile above the previous scaled pixel. Thus it must have been the last scaled pixel in the column and \( p \) does not exist.

Consider a scaled pixel \( p \) that we downscale next. Because the column above the rightmost tile of \( \ell_h \) shows how many pixels are already downscaled in the current column, we will find the correct position to downscale \( p \).

Due to the simple structure, the clean-up phase is straightforward.

Connectivity: Similar to the upscaling operation, the column above the rightmost tile of \( \ell_h \) guarantees connectivity of the downscaled area. For this, consider any connected shape \( P \), we can cut through \( P \) with a vertical line and add tiles along this line. Clearly, this structure is still connected. Because all tiles have a connection to this vertical line, we can remove everything to the right of the line without losing connectivity.

Similarly, \( \ell_v \) preserves connectivity of the upscaled area with the downscaled area, especially when processing the last column.

Time: Phase (1) takes at most \( O(ch) \) time to construct \( \ell_v \) and \( O(1) \) to construct \( \ell_h \). In phase (2), for each scaled pixel we get its state in time \( O(cw + ch) \). Removing this scaled pixel needs \( O(c^2) \) time, scaling it down takes another \( O(cw + ch) \) steps. Because there are \( wh \) many scaled pixels, we need \( O(wh \cdot (c^2 + cw + ch)) \) steps for phase (2).

Removing the auxiliary structures in phase (3) needs only \( O(ch) \) steps to remove \( \ell_v \) and \( O(w) \) to remove \( \ell_h \). For all three phases, we get a total runtime of \( O(wh \cdot (c^2 + cw + ch)) \).

\( \square \)

5 Scaling Monotone Polyominoes

When the polyomino \( P \) is \( x \)- or \( y \)-monotone, we can scale \( P \) using two robots without constructing a bounding box. \( P \) is called \( x \)-monotone (\( y \)-monotone) if the intersection with any vertical (horizontal) line with \( P \) is a connected set. See Fig. 14 for a polyomino and its scaled version. In the following we consider an \( x \)-monotone polygon \( P \). We proceed in two phases: (I) scaling the columns of \( P \) vertically, (II) scaling the columns of \( P \) horizontally.
5.1 Phase I

During the first phase, we process the polyomino column-wise from left to right to scale the columns of \( P \) vertically by the factor \( c \). The marker is placed on the uppermost tile of the current column. For every tile in that column, the robot moves the marker one unit down. If there is no further tile, the robot keeps the marker. The robot extends the current column on the upper side by \( c - 1 \) tiles and moves down until the marker or the end of the column is reached. In the latter case, the current column is finished. See Fig. 15 for an example.

To scale the next column, we consider three cases. The lowest tile of the next column may have an equal, a higher or a lower \( y \)-coordinate than the lowest tile of the current column. Let \( t_n \) (\( t_c \), resp.) be the lowest tile of the next (current, resp.) column and \( y_{tn} \) (\( y_{tc} \), resp.) the \( y \)-coordinate of that tile. The three cases are handled as follows; see also Algorithm 14.

1. \( y_{tn} = y_{tc} \): We can simply continue with the next column, see Algorithm 15.
2. \( y_{tn} > y_{tc} \): Before moving to the next column we have to scale empty pixels. For each empty pixel \( t_e \) below \( t_n \) with \( y \)-coordinate \( y_{te} \geq y_{tc} \), we move every column to the right of the current column \( c - 1 \) units up. This can be done by marking the next empty pixel and by letting the robot shift the columns up. The shifting process can be done by adding \( c - 1 \) tiles above each column and afterwards removing \( c - 1 \) tiles from the bottom. See also Algorithm 16 for Pseudocode, and Fig. 16 for an example.
3. \( y_{tn} < y_{tc} \): This case is more complex because it is not immediately clear in which direction all remaining columns have to be shifted in order to scale the next column correctly. We perform a scaling of the next column and a shifting of remaining columns at the same time. We proceed as follows. Let \( t \) be the tile with \( y \)-coordinate \( y_t = y_{tc} \). We scale the column from top to \( t \) as described above, i.e., we run \textsc{MonotoneCaseI} until we remove the marker from a pixel for which \textsc{Left} is occupied and \textsc{Bottom-Left} is empty. For each tile below \( t \) starting from the bottom, we add \( c - 1 \) to the bottom of the column and shift all columns to the right \( c - 1 \) steps down, which can be performed analogously to the shifting process from \textsc{MonotoneCaseII}. Note that this process is completed when the robot returns to the marker and when the marker is to the right of \( t_c \), i.e., \textsc{Bottom-Left} is empty and \textsc{Left} is occu-
Fig. 16 Example for $y_{t_n} > y_{t_c}$. Scaled tiles are marked dark gray. (a) Recognizing the case. (b) Two tiles (gray) placed above every following column. (c) Two tiles removed from the bottom end of every column. (d) Scaling the column

Fig. 17 Recognizing $y_{t_n} < y_{t_c}$. Scaled tiles are marked gray. (a) Recognizing the case. (b) All pixels having a $y$-coordinate of at least $y_{t_c}$ are scaled upwards. (c) All pixels having a $y$-coordinate of strictly less than $y_{t_c}$ are scaled downwards

Algorithm 14: An algorithm to scale a monotone shape. Algorithms for MONOTONECASEI and MONOTONECASEII can be found in Algorithm 15 and 16, respectively. We leave out the Pseudocode for MONOTONECASEIII as it is analogously to a combination of the previous two.

```plaintext
1: function MONOTONEPHASEI
2:   MONOTONECASEI
3:   while RIGHT ≠ empty do
4:     Move right
5:     if BOTTOM-LEFT = empty and BOTTOM = empty then
6:       MONOTONECASEI
7:     else if BOTTOM-LEFT = empty and BOTTOM ≠ empty then
8:       MONOTONECASEII
9:     else if BOTTOM-LEFT ≠ empty and BOTTOM = empty then
10:    MONOTONECASEIII
```

5.2 Phase II

When all columns have been scaled vertically, we proceed with phase II to scale the columns horizontally. The idea of phase II is as follows. Proceed column-wise; for a column $C$ shift all remaining columns $c-1$ steps to the right to make space for
Algorithm 15: An algorithm to scale the current column if $y_{tn} = y_{tc}$.

```plaintext
1: function MONOTONECASEI
2: Move upmost
3: Place marker
4: while ISON TILE do
5: Take marker, move down
6: if ISON TILE then Place marker
7: Move upmost
8: loop $c - 1$ times
9: Move up and place tile
10: while BOTTOM $\neq$ empty and marker not reached do
11: Move down
12: if Tile contains marker then
13: Remove marker
14: Move down
15: Move up
16: while RIGHT = empty and TOP $\neq$ empty do
17: Move up
```

Algorithm 16: An algorithm to scale the current column if $y_{tn} > y_{tc}$. *Note that we can recognize the stop, if there is a marker somewhere to the right.

```plaintext
1: function MONOTONECASEII
2: while LEFT $\neq$ empty do
3: Move down
4: while not ISON TILE do
5: Place tile with marker and move up
6: while not ISON TILE do
7: Move up
8: Move left
9: repeat
10: Move right
11: Move upmost
12: Add $c - 1$ tiles above the column
13: while RIGHT $= $ empty and BOTTOM $\neq$ empty do
14: Move down
15: until RIGHT $\neq$ empty
16: Move downmost
17: repeat
18: loop $c - 1$ times
19: Remove tile and move up
20: while LEFT = empty do
21: Move up
22: Move left
23: while DOWN $\neq$ empty do
24: Move down
25: if Marked tile lies to the right then
26: Move right
27: until On marked tile
28: Remove marked tile and move up
29: MONOTONECASEI
```

$\triangleright$ Scale tiles in current column
building \( c - 1 \) copies of \( C \) (see Fig. 18). To identify \( C \) in a later step, we place a marker on the bottom most tile of \( C \). Clearly, we cannot simply shift a column \( c - 1 \) steps to the right, as this would disconnect the shape. Instead, we process the columns from right to left until we reach \( C \) and shift the columns in the following way.

For a column \( C' \), copy it \( c - 1 \) times to the right. Then, starting from the bottommost tile of \( C' \), we remove \( c - 1 \) tiles to the right leaving the tile of the rightmost copy of \( C' \). We repeat this process with the rows above until we find a row with a tile \( t \) of \( C' \) that has an adjacent tile to the left. Note that there is no tile below \( t \). Thus, we can simply remove rows of \( C' \) and \( c - 2 \) copies from the top as long there is a tile below the left most tile.

After this clean-up, the rightmost copy of \( C' \) still exists and is connected with its previous column by a bridge. We can now proceed with the column to the left of \( t \). If this column is \( C \), then we perform the same steps without the clean-up phase. See also Algorithm 17 for Pseudocode.

Algorithm 17: An algorithm for the second phase to scale monotone shapes.
**Theorem 4** Scaling an x-monotone polyomino $P$ with $N$ tiles, width $w$, and height $h$ by a constant scaling factor $c$ without losing connectivity can be done with two robots in $O(wc^2N)$ steps.

**Proof**

**Correctness phase I:** Scaling the first column is correct, because for each tile we add $c - 1$ tiles to the top of the column, which results in scaling the column by factor $c$. Furthermore, the lowest tile $t_c$ of the first column still has the same neighbor $t$ (either an empty pixel or a tile) with the same number of empty and occupied pixels above and below $t$. We use this fact as an invariant to prove that the remaining columns are also scaled correctly. Now assume we correctly scaled the first $k$ columns. In the first case, when $y_{t_c} = y_{t_n}$, we scale the next column similar to the first column. Therefore, the lowest tile still has the same neighbor after scaling. Now consider the $i$-th tile $t_i$ in the column from below before scaling. Then, $t_i$ moves $c(i - 1)$ units upwards after scaling the first $i - 1$ tiles. This results in $t_i$ having its correct neighbor to the left, namely the $i$-th pixel from the previous column (note that we can consider all empty pixels above any column already scaled). Thus, after scaling all tiles, the neighbors to their left are all correct. In conclusion, the invariant still holds for the first $k + 1$ columns for the first case.

For the second case ($y_{t_c} < y_{t_n}$), we shift all remaining columns by $c - 1$ steps up, such that $t_n$ has the correct neighbor to the left. Note that $t_n$ maintains its right neighbor during the shifting process. Thus, after the shifting process, the invariant holds and we can scale the column.

In the remaining case, we can scale all tiles in the next column with $y$-coordinate at least $y_{t_c}$, which does not change the invariant for the first $k + 1$ columns. When scaling the lower tiles, we shift all remaining columns down, such that $t_n$ keeps its neighbor to the right. Thus, after scaling this complete column, the invariant holds for the first $k + 1$ columns.

By induction, we correctly scale the columns.

**Correctness phase II:** To prove the correctness of phase II, we show that the shift of columns is performed without losing connectivity. The scaling is done within the shifting process by ignoring the clean-up for the last column. Thus, a repeated shifting phase scales the polyomino.

Consider the rightmost column. After copying this column $c$ times, the whole polyomino is still connected. In the clean-up phase, we keep a bridge between the shifted column and the next column, maintaining connectivity. This bridge can then be used to copy the next column. By induction, we can shift $n$ columns without losing connectivity.

Combining phase I and II, we scale the polyomino by the factor $c$.

**Time phase I:** Let $n_i$ denote the number of tiles in the $i$th column in $P$. Before scaling the $i$th column, we may need to shift all remaining columns by at most $O(cn_{i-1})$ unit steps in case 2, or $O(cn_i)$ unit steps in case 1. Because there are at most $w$ columns, this lasts $O(wc(n_{i-1} + n_i))$ steps. Scaling each tile in the column takes at most $O(cn_i)$ steps. Summing this up over all columns, we have at most $O(cn_1) + \sum_{i=2}^{w} O(wc(n_{i-1} + n_i)) + O(cn_i) = O(wcN)$.

**Time phase II:** Consider the $i$th column of $P$ containing $cn_i$ tiles. Shifting all remaining columns $c - 1$ unit steps to the right needs $O(c^2N)$ steps. Scaling
the $i$th column needs another $O(c^2n_i)$ steps. Over all columns, phase (2) needs $O\left(\sum_{i=1}^{w} c^2 N + c^2 n_i\right) = O(wc^2 N)$ steps in total. \hfill \Box

6 Adapting Algorithms

As shown in [32], there are algorithms that may not preserve connectivity. An immediate consequence of being able to scale a given shape is that we can simulate any algorithm $\mathcal{A}$ within our model while guaranteeing connectivity: We first scale the polyomino by three and then execute $\mathcal{A}$ by always performing three steps into one direction if $\mathcal{A}$ does one step. If at some point the robot needs to move through empty pixels, then we place a $3 \times 3$-square with the middle pixel empty (if a clean-up is desired at the end of $\mathcal{A}$, i.e., removing all scaled empty pixels, we fill up the complete row/column with these squares). This guarantees connectivity during the execution and we obtain the following theorem.

**Theorem 5** If there is an algorithm $\mathcal{A}$ for some problem $\Pi$ with runtime $T(\mathcal{A})$, such that the robot moves within a $w' \times h'$ rectangle, then there is an algorithm $\mathcal{A}'$ for $\Pi$ with runtime $O(wh \cdot (w + h) + \max((w' - w)h', (h' - h)w') + T(\mathcal{A}))$ guaranteeing connectivity during the execution.

**Proof** Approach:

The first step is to scale the polyomino $P$ of width $w$ and height $h$ by a factor $c = 3$ with the strategy above. We keep any $3 \times 3$ square in the clean-up phase of the scaling algorithms. This gives us a $3w \times 3h$ rectangle containing the scaled polyomino and $3 \times 3$ segments representing empty pixels from the $w \times h$ area of the initial polyomino.

We adapt the algorithm $\mathcal{A}$ as follows. When the robot places a tile in $\mathcal{A}$, then we place a tile in the middle of the $3 \times 3$ square in $\mathcal{A}'$. A removal in $\mathcal{A}$ is represented by removing the middle tile of the $3 \times 3$ square in $\mathcal{A}'$.

For each step the robot moves into direction $d$ in $\mathcal{A}$, we move 3 steps into direction $d$ in $\mathcal{A}'$, i.e., the robot effectively moves on a grid scaled up by a factor of three (see Fig. 19(a)). If the robot reaches a pixel at which no $3 \times 3$ square is present (see Fig. 19(b)), then the robot adds such a square representing an empty node and leaves the middle bottom tile missing (see Fig. 19(c)). Then the robot fills up the complete row/column with $3 \times 3$ squares and returns to the pixel at which the middle bottom tile is missing (see Fig. 19(d)). From there we can continue the execution of $\mathcal{A}$.

**Analysis:** The scaling procedure requires $O(wh \cdot (w + h))$ steps. Due to the described movement adjustment of doing three steps in $\mathcal{A}'$ instead of one step in $\mathcal{A}$, the runtime of $\mathcal{A}$ increases by the constant factor three. Finally, we consider the fill-in procedure to be used in case of stepping on an empty pixel. Depending on the robot's movement area for $\mathcal{A}$, we get $O(\max((w' - w)h', (h' - h)w'))$ steps for filling in. This results in an overall runtime of $O(wh \cdot (w + h) + \max((w' - w)h', (h' - h)w') + T(\mathcal{A}))$ steps. \hfill \Box

When the algorithm terminates and there is only one polyomino left in the constructed rectangle area, we can either remove empty cells or downscale the polyomino to its original size.
Fig. 19  (a) A polyomino scaled by a factor of $c = 3$. The robot moves along the red lines when an algorithm is executed. (b) The robot enters an empty pixel when it tries to move south. (c) Marked the first $c \times c$ segment, which the robot has entered, by removing the bottom-middle tile. (d) The robot has filled the row with $c \times c$ segments and moves back to the previously marked segment

7 Conclusion

We have demonstrated how geometric algorithms for finite automata can be used to enable very simple robots to perform a number of fundamental construction tasks while preserving connectivity. At this point, the main focus of our paper has been a feasibility study of the considered problems rather than a tight analysis of the proposed solutions; as a consequence, non-trivial lower bounds have yet to be established.

There is also a whole range of possible extensions. Is it possible to scale general polyominoes without the preceding bounding box construction? A possible approach is to cut the polyomino into a subset of monotone polyominoes, which could be handled separately. Expanding the existing repertoire of operations to three-dimensional configurations and operations is another logical step. An equally relevant challenge is to develop distributed algorithms with multiple robots that are capable of solving a range of problems with the requirement of connectivity, without having to rely on the preceding scaling procedure that we used in our work. Other questions arise from additional requirements of real-world applications, such as the construction and reconfiguration of space habitats.

Funding  Open Access funding enabled and organized by Projekt DEAL.

Open Access  This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abdel-Rahman, A., Becker, A.T., Biediger, D.E., Cheung, K.C., Fekete, S.P., Gershenfeld, N.A., Hugo, S., Jenett, B., Keldenich, P., Niehs, E., Rieck, C., Schmidt, A., Scheffer, C., Yannuzzi, M.: Space ants: Constructing and reconfiguring large-scale structures with finite automata. In: Symposium on Computational Geometry (SoCG), pages 73:1–73:7, (2020). Video available at https://www.ibr.cs.tu-bs.de/users/fekete/Videos/SoCG/2020/Space_final.mp4
2. Alpern, S.: The rendezvous search problem. SIAM J. Control. Optim. 33(3), 673–683 (1995)
3. Ambühl, C., Gasieniec, L., Pelc, A., Radzík, T., Zhang, X.: Tree exploration with logarithmic memory. ACM Trans. Algorithms (TALG) 7(2), 1–21 (2011)
4. Anderson, E.J., Weber, R.R.: The rendezvous problem on discrete locations. J. Appl. Probab. 27(4), 839–851 (1990)
5. Balanza-Martínez, J., Luchsinger, A., Caballero, D., Reyes, R., Cantu, A.A., Schweller, R., García, L.A., Wylie, T.: Full tilt: universal constructors for general shapes with uniform external forces. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 2689–2708 (2019)
6. Becker, A.T., Fekete, S.P., Keldenich, P., Krupke, D., Rieck, C., Scheffer, C., Schmidt, A.: Tilt assembly: algorithms for micro-factories that build objects with uniform external forces. Algorithmica, 82(2), 1–23 (2017)
7. Bender, M.A., Slonim, D.K.: The power of team exploration: two robots can learn unlabeled directed graphs. In: Symposium on Foundations of Computer Science (FOCS), pages 75–85 (1994)
8. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: Symposium on Foundations of Computer Science (FOCS), pages 132–142 (1978)
9. Brass, P., Cabrera-Mora, F., Gasparri, A., Xiao, J.: Multirobot tree and graph exploration. IEEE Trans. Rob. 27(4), 707–717 (2011)
10. Chalk, C., Martinez, E., Schweller, R., Vega, L., Winslow, A., Wylie, T.: Optimal staged self-assembly of general shapes. Algorithmica 80(4), 1383–1409 (2018)
11. Cheung, K.C., Gershenfeld, N.: Reversibly assembled cellular composite materials. Sci. 341(6151), 1219–1221 (2013)
12. Cieliebak, M., Flocchini, P., Prencipe, G., Santoro, N.: Distributed computing by mobile robots: Gathering. SIAM J. Comput. 41(4), 829–879 (2012)
13. Costa, A., Abdel-Rahman, A., Jenett, B., Gershenfeld, N., Kostitsyna, I., Cheung, K.: Algorithmic approaches to reconfigurable assembly systems. In: IEEE Aerospace Conference, pages 1–8 (2019)
14. Cramer, N.B., Cellucci, D.W., Formoso, O.B., Gess, C.E., Jenett, B.E., Kim, J.H., Lendraitis, M., Swei, S.S., Trinh, G.T., Trinh, K.V., et al.: Elastic shape morphing of ultralight structures by programmable assembly. Smart Mater. Struct. 28(5), 055006 (2019)
15. Czyzowicz, J., Gasieniec, L., Pelc, A.: Gathering few fat mobile robots in the plane. Theoret. Comput. Sci. 410(6), 481–499 (2009)
16. D’Angelo, G., D’Emidio, M., Das, S., Navarra, A., Prencipe, G.: Leader election and compaction for asynchronous silent programmable matter. In: Proceedings of the 19th International Conference on Autonomous Agents and MultiAgent Systems, pages 276–284 (2020)
17. D’Angelo, G., Stefano, G.D., Navarra, A.: Gathering asynchronous and oblivious robots on basic graph topologies under the look-compute-move model. In: Alpern, S., Fokkink, R., Gasieniec, L., Lindelauf, R., Subrahmanian, V. (eds.) Search Theory, pp. 197–222. Springer, New York (2013)
18. Das, S., Flochini, P., Kuten, S., Nayak, A., Santoro, N.: Map construction of unknown graphs by multiple agents. Theoret. Comput. Sci. 385(1), 34–48 (2007)
19. Daymude, J.J., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Improved leader election for self-organizing programmable matter. In: Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics (ALGOSENSORS), pages 127–140 (2017)
20. Demaine, E.D., Demaine, M.L., Fekete, S.P., Ishaque, M., Rafalin, E., Schweller, R.T., Souvaine, D.L.: Staged self-assembly: nanomanufacture of arbitrary shapes with o (1) glues. Nat. Comput. 7(3), 347–370 (2008)
21. Derakhshande, Z., Dolev, S., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Brief announcement: Amoebot - a new model for programmable matter. In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 220–222 (2014)
22. Derakhshande, Z., Gmyr, R., Porter, A., Richa, A.W., Scheideler, C., Strothmann, T.: On the runtime of universal coating for programmable matter. In: International Conference on DNA Computing and Molecular Programming (DNA), pages 148–164 (2016)
23. Derakhshande, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: An algorithmic framework for shape formation problems in self-organizing particle systems. In: International Conference on Nanoscale Computing and Communication (NANOCOM), pages 21:1–21:2 (2015)
24. Derakhshande, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), pages 289–299 (2016)
25. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal coating for programmable matter. Theoret. Comput. Sci. 671, 56–68 (2017)
26. Derakhshandeh, Z., Gmyr, R., Strothmann, T., Bazzi, R., Richa, A.W., Scheideler, C.: Leader election and shape formation with self-organizing programmable matter. In: 21st International Conference on DNA Computing and Molecular Programming (DNA), pages 117–132 (2015)
27. Dessmark, A., Fraigniaud, P., Kowalski, D.R., Pelc, A.: Deterministic rendezvous in graphs. Algorithmica 46(1), 69–96 (2006)
28. Devisnes, S., Laman, A., Petit, F., Tixeuil, S.: Optimal torus exploration by oblivious robots. Comput. 101(9), 1241–1264 (2019)
29. Di Luna, G.A., Floccini, P., Santoro, N., Viglietta, G., Yamauchi, Y.: Shape formation by programmable particles. Distrib. Comput. 33(1), 69–101 (2020)
30. Di Stefano, G., Navarra, A.: Optimal gathering of oblivious robots in anonymous graphs and its application on trees and rings. Distrib. Comput. 30(2), 75–86 (2017)
31. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. J. Algorithms 51(1), 38–63 (2004)
32. Fekete, S.P., Gmyr, R., Hugo, S., Keldenich, P., Scheffer, C., Schmidt, A.: Cadbots: algorithmic aspects of manipulating programmable matter with finite automata. Algorithmica, 83(1), 1–26 (2020)
33. Fekete, S.P., Niehs, E., Scheffer, C., Schmidt, A.: Connected reconfiguration of lattice-based cellular structures by finite-memory robots. In: International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics, pages 60–75. Springer (2020)
34. Felton, S., Tolley, M., Demaine, E., Rus, D., Wood, R.: A method for building self-folding machines. Sci. 345(6197), 644–646 (2014)
35. Fleischer, R., Trippen, G.: Exploring an unknown graph efficiently. In: European Symposium on Algorithms (ESA), pages 11–22 (2005)
36. Floccini, P., Prencipe, G., Santoro, N.: Distributed computing by mobile entities. Current Research in Moving and Computing, vol. 11340. Springer LNCS (2019)
37. Floccini, P., Prencipe, G., Santoro, N., Widmayer, P.: Gathering of asynchronous robots with limited visibility. Theoret. Comput. Sci. 337(1), 147–168 (2005)
38. Fraigniaud, P., Gasieniec, L., Kowalski, D.R., Pelc, A.: Collective tree exploration. Networks 48(3), 166–177 (2006)
39. Fraigniaud, P., Ilcinkas, D.: Digraphs exploration with little memory. In: Symposium on Theoretical Aspects of Computer Science (STACS), pages 246–257 (2004)
40. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph Exploration by a Finite Automaton. Theoret. Comput. Sci. 345(2–3), 331–344 (2005)
41. Gasieniec, L., Radzik, T.: Memory efficient anonymous graph exploration. In: Workshop on Graph-Theoretic Concepts in Computer Science (WG), pages 14–29 (2008)
42. Gilpin, K., Knaian, A., Rus, D.: Robot pebbles: One centimeter modules for programmable matter through self-disassembly. In: IEEE International Conference on Robotics and Automation (ICRA), pages 2485–2492 (May 2010)
43. Gmyr, R., Hinmenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C.: Shape Recognition by a Finite Automaton Robot. In: International Symposium on Mathematical Foundations of Computer Science (MFCS), pages 52:1–52:15 (2018)
44. Gmyr, R., Hinmenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C., Strothmann, T.: Forming tile shapes with simple robots. In: International Conference on DNA Computing and Molecular Programming (DNA), pages 122–138 (2018)
45. Gmyr, R., Kostitsyna, I., Kuhn, F., Scheideler, C., Strothmann, T.: Forming tile shapes with a single robot. In: European Workshop on Computational Geometry (EuroCG), pages 9–12 (2017)
46. Goldstein, S.C., Mowry, T.: Claytronics: A scalable basis for future robots. Robosphere, 1–6 (2004)
47. Gregg, C.E., Jenett, B., Cheung, K.C.: Assembled, modular hardware architectures - what price reconfigurability? In: IEEE Aerospace Conference, pages 1–10 (2019)
48. Gregg, C.E., Kim, J.H., Cheung, K.C.: Ultra-light and scalable composite lattice materials. Adv. Eng. Mater. 20(9), 1800213 (2018)
49. Hawkes, E., An, B., Benbernou, N.M., Tanaka, H., Kim, S., Demaine, E.D., Rus, D., Wood, R.J.: Programmable matter by folding. Proc. Natl. Acad. Sci. 107(28), 12441–12445 (2010)
50. Hurtao, F., Molina, E., Ramaswami, S., Sacristán, V.: Distributed reconfiguration of 2D lattice-based modular robotic systems. Auton. Robot. 38(4), 383–413 (2015)
51. Ilcinkas, D.: Setting port numbers for fast graph exploration. Theoret. Comput. Sci. 401(1–3), 236–242 (2008)
52. Jenett, B., Abdel-Rahman, A., Cheung, K.C., Gershenfeld, N.: Material-robot system for assembly of discrete cellular structures. IEEE Robot. Autom. Lett. 4(4), 4019–4026 (2019)
53. Jenett, B., Calisch, S., Cellucci, D., Cramer, N., Gershenfeld, N., Swei, S., Cheung, K.C.: Digital morphing wing: active wing shaping concept using composite lattice-based cellular structures. Soft Rob. 4(1), 33–48 (2017)
54. Jenett, B., Cellucci, D.: A mobile robot for locomotion through a 3d periodic lattice environment. In: IEEE International Conference on Robotics and Automation (ICRA), pages 5474–5479 (2017)
55. Jenett, B., Cellucci, D., Gregg, C., Cheung, K.: Meso-scale digital materials: modular, reconfigurable, lattice-based structures. In: ASME International Manufacturing Science and Engineering Conference (MSEC) (2016)
56. Jenett, B., Cheung, K.: Bill-e: Robotic platform for locomotion and manipulation of lightweight space structures. In: AIAA/AHS Adaptive Structures Conference, page 1876 (2017)
57. Jenett, B., Gregg, C., Cellucci, D., Cheung, K.: Design of multifunctional hierarchical space structures. In: IEEE Aerospace Conference, pages 1–10 (2017)
58. Kamei, S., Lamani, A., Ooshita, F., Tixeuil, S.: Asynchronous mobile robot gathering from symmetric configurations without global multiplicity detection. In Kosowski, A., Yamashita, M. (Eds.), International Colloquium on Structural Information and Communication Complexity (SIROCCO), pp. 150–161. Springer LNCS
59. Kim, Y., Katayama, Y., Wada, K.: Pairbot: A novel model for autonomous mobile robot systems consisting of paired robots. arXiv preprint arXiv:2009.14426, (2020)
60. Knaian, A.N., Cheung, K.C., Lobovsky, M.B., Oines, A.J., Schmidt-Neilsen, P., Gershenfeld, N.A.: The milli-motein: A self-folding chain of programmable matter with a one centimeter module pitch. In: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 1447–1453 (2012)
61. Marco, G.D., Gargano, L., Kranakis, E., Krizanc, D., Pelc, A., Vaccaro, U.: Asynchronous deterministic rendezvous in graphs. Theoret. Comput. Sci. 355(3), 315–326 (2006)
62. Naz, A., Piranda, B., Bourgeois, J., Goldstein, S.C.: A distributed self-reconfiguration algorithm for cylindrical lattice-based modular robots. In: IEEE International Symposium on Network Computing and Applications (NCA), pages 254–263 (2016)
63. Niehs, E., Schmidt, A., Scheffer, C., Biediger, D.E., Yannuzzi, M., Jenett, B., Abdel-Rahman, A., Cheung, K.C., Becker, A.T., Fekete, S.P.: Recognition and reconfiguration of lattice-based cellular structures by simple robots. In: 2020 IEEE International Conference on Robotics and Automation (ICRA), pages 8252–8259. IEEE (2020)
64. Panaite, P., Pelc, A.: Exploring unknown undirected graphs. J. Algorithms 33(2), 281–295 (1999)
65. Patitz, M.J.: An introduction to tile-based self-assembly and a survey of recent results. Nat. Comput. 13(2), 195–224 (2014)
66. Schmidt, A., Manzoor, S., Huang, L., Becker, A.T., Fekete, S.P.: Efficient parallel self-assembly under uniform control inputs. IEEE Robot. Autom. Lett. 3(4), 3521–3528 (2018)
67. Ta-Shma, A., Zwick, U.: Deterministic rendezvous, treasure hunts, and strongly universal exploration sequences. ACM Trans. Algorithms 10(3), 12:1-12:15 (2014)
68. Tucci, T., Piranda, B., Bourgeois, J.: A distributed self-assembly planning algorithm for modular robots. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS), pages 550–558 (2018)
69. Winfree, E.: Algorithmic self-assembly of DNA. PhD thesis, California Institute of Technology (1998)
70. Winfree, E., Liu, F., Wenzler, L.A., Seeman, N.C.: Design and self-assembly of two-dimensional DNA crystals. Nature 394(6693), 539 (1998)
71. Woods, D., Chen, H.-L., Goodfriend, S., Dabby, N., Winfree, E., Yin, P.: Active self-assembly of algorithmic shapes and patterns in polylogarithmic time. In: 4th Conference on Innovations in Theoretical Computer Science (ITCS), pages 353–354 (2013)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.