Algorithmica (2021) 83:144–176
https://doi.org/10.1007/s00453-020-00752-0

Building a Nest by an Automaton

Jurek Czyzowicz[1] · **Dariusz Dereniowski**[2] · **Andrzej Pelc**[1]

Received: 19 September 2019 / Accepted: 15 July 2020 / Published online: 25 July 2020

Abstract
A robot modeled as a deterministic finite automaton has to build a structure from material available to it. The robot navigates in the infinite oriented grid $\mathbb{Z} \times \mathbb{Z}$. Some cells of the grid are full (contain a brick) and others are empty. The subgraph of the grid induced by full cells, called the *shape*, is initially connected. The (Manhattan) distance between the furthest cells of the shape is called its *span*. The robot starts at a full cell. It can carry at most one brick at a time. At each step it can pick a brick from a full cell, move to an adjacent cell and drop a brick at an empty cell. The aim of the robot is to construct the most compact possible structure composed of all bricks, i.e., a *nest*. That is, the robot has to move all bricks in such a way that the span of the resulting shape be the smallest. Our main result is the design of a deterministic finite automaton that accomplishes this task and subsequently stops, for every initially connected shape, in time $O(sn)$, where s is the span of the initial shape and n is the number of bricks. We show that this complexity is optimal.

Keywords Finite automaton · Plane · Grid · Construction task · Brick · Mobile agent · Robot

A preliminary version of this paper appeared in the Proc. 27th Annual European Symposium on Algorithms (ESA 2019).
J. Czyzowicz: Supported in part by NSERC discovery grant.
D. Dereniowski: Partially supported by National Science Centre, Poland, Grant Number 2015/17/B/ST6/01887.
A. Pelc: Supported in part by NSERC discovery Grant 2018-03899 and by the Research Chair in Distributed Computing of the Université du Québec en Outaouais.

✉ Dariusz Dereniowski
deren@eti.pg.edu.pl

Jurek Czyzowicz
jurek@uqo.ca

Andrzej Pelc
pelc@uqo.ca

[1] Département d'informatique, Université du Québec en Outaouais, Gatineau, Canada

[2] Faculty of Electronics, Telecommunications and Informatics, Gdansk University of Technology, Gdańsk, Poland

1 Introduction

1.1 The Problem

A mobile agent (robot) modeled as a deterministic finite automaton has to build a structure from material available to it. The robot navigates in the infinite oriented grid $\mathbb{Z} \times \mathbb{Z}$ represented as the set of unit square cells in the two-dimensional plane, with all cell sides vertical or horizontal. The robot has a compass enabling it to move from a currently occupied cell to one of the four cells (to the North, East, South, West) adjacent to it. Some cells of the grid contain a brick, i.e., are *full*, other cells are *empty*. The subgraph of the grid induced by full cells, called the *shape*, is initially connected. We point out that the robot may traverse both full and empty cells. The (Manhattan) distance between the furthest cells of the shape is called its *span*. We use the term *span* as opposed to *diameter* since the latter is understood as the longest distance in the graph induced by full cells. Thus, the distance is an upper bound on the span and for some shapes the two can be by a multiplicative factor of $\Omega(\sqrt{n})$ apart. Moreover, this diameter may be sometimes undefined, if the shape becomes disconnected. The robot starts at a full cell. It can carry at most one brick at a time. At each step, the robot can pick up a brick from the currently occupied full cell (if it does not carry any brick at this time), moves to an adjacent cell, and can drop a brick at the currently occupied empty cell (if it carries a brick). The robot has no a priori knowledge of the initial shape, of its span or of the number of bricks.

The aim of the robot is to construct the most compact possible structure composed of all bricks, i.e., a *nest*. That is, the robot has to move all bricks in such a way that the span of the resulting shape be the smallest. The above task has many real applications. In the natural world, animals use material scattered in a territory (pieces of wood, small branches, leaves) to build a nest, and minimizing the span is important to better protect it. A mobile robot may be used to clean a territory littered by hazardous material, in which case minimizing the span of the resulting placement of contaminated pieces facilitates subsequent decontamination. A more mundane example is the everyday task of sweeping the floor, whose aim is to gather all trash in a small space and then get rid of it. From the theoretical point of view, this work falls into the line of research aiming at understanding the limits of computational capabilities in mobile agent computing. More precisely, the underlying general theoretical question is which tasks can be performed by simple low-memory agents?

1.2 Our Results

Our main result is the design of a deterministic finite automaton that accomplishes the task of building a nest and subsequently stops, for every initially connected shape, in time $O(sn)$, where s is the span of the initial shape and n is the number of bricks. The time is defined as the number of moves of the robot. We show that this complexity is optimal. While simpler solutions can be designed to solve this task for shapes without "holes", the novelty of our building algorithm is in the fact that it works for all possible connected shapes with optimal complexity.

The essence of our nest building algorithm is to instruct the robot to make a series of trips to get consecutive bricks, one at a time, and carry them to some designated compact area. In order to achieve an optimal complexity, we overcome two difficulties to carry out this plan. The first one is that each trip should be short, i.e., of length $O(s)$. The second is that the span of the initial shape may be much larger than the memory of the robot, and hence the robot that already put several bricks in a compact area and goes for the next brick cannot remember the way back to the area where it started building. Thus we need to prepare the way, so that the robot can recognize the backtrack path locally at each decision point. We strongly highlight that the backtracking mechanism is the main contribution and the main algorithmic idea that we have developed. We do this by potentially disconnecting the shape during the execution of the algorithm, but a special care has to be taken so that the connected components of intermediary shapes be close to each other, to prevent the robot from getting lost in large empty spaces. We also point out that some trips to obtain a new brick indeed have to be of length $\Omega(s)$: as an example suppose that the designated building area is connected by a path of length $\Theta(s)$ to, say, a square. In this example, removing bricks from the path would place the two parts of the shape far away from each other and eventually the robot would not be able to navigate between them.

It is interesting to compare the task of constructing structures from available material using an automaton to that of exploration of mazes by automata, that is a classic topic with over 50 years of history (see Sect. 1.3). The main result of Budach[9] (translated to our terminology) states that an automaton that can only navigate in the shape and cannot move bricks cannot explore all connected shapes, i.e., it cannot visit all bricks. Budach's impossibility result holds even if we do not require that the automaton stops, i.e., any automaton moving indefinitely will not be able explore some shapes. By contrast, it follows from our result that the ability of moving bricks enables the automaton not only to see all bricks but to build a potentially useful structure using all of them and stop, and to accomplish all of that with optimal complexity.

1.3 Related Work

Problems concerning exploration and navigation performed by mobile agents or robots in an unknown environment have been studied for many years (cf.[32, 38]). The relevant literature can be divided into two parts, according to the environment where the robots operate: it can be either a geometric terrain, possibly with obstacles, or a network modeled as a graph in which the robot moves along edges.

In the geometric context, a closely related problem is that of pattern formation[1, 12, 13, 19, 21, 40]. Robots, usually modeled as points freely moving in the plane have to arrange themselves to form a pattern given as input. This task has been mostly studied in the context of asynchronous oblivious robots having full visibility of other robots positions.

The graph setting can be further specified in two different ways. In[2, 4, 5, 17, 28] the robot explores strongly connected directed graphs and it can move only in

the tail-to-head direction of an edge, not vice-versa. In[3, 6, 9, 22–24, 37, 39] the explored graph is undirected and the robot can traverse edges in both directions. Graph exploration scenarios can be also classified in another important way. It is either assumed that nodes of the graph have unique labels which the robot can recognize (as in, e.g.,[17, 24, 37]), or it is assumed that nodes are anonymous (as in, e.g.,[4, 5, 9, 10, 39]). In our case, we work with the infinite anonymous grid, hence it is an undirected anonymous graph scenario. The efficiency measure adopted in papers dealing with graph exploration is either the completion time of this task, measured by the number of edge traversals, (cf., e.g.,[37]), or the memory size of the robot, measured either in bits or by the number of states of the finite automaton modeling the robot (cf., e.g.,[22, 28, 29]). We are not concerned with minimizing the memory size but we assume that this memory is bounded, i.e., it is constant as a function of the input grid size. However we want to minimize the time of our construction task.

The capability of a robot to explore anonymous undirected graphs has been studied in, e.g.,[7, 9, 22, 29, 34, 39]. In particular, it was shown in[39] that no finite automaton can explore all cubic planar graphs (in fact no finite set of finite automata can cooperatively perform this task). Budach[9] proved that a single automaton cannot explore all mazes (that we call connected shapes in this paper). Hoffmann[33] proved that one pebble does not help to do it. By contrast, Blum and Kozen[7] showed that this task can be accomplished by two cooperating automata or by a single automaton with two pebbles. The size of port-labeled graphs which cannot be explored by a given robot was investigated in[29].

Recently a lot of attention has been devoted to the problem of searching for a target hidden in the infinite anonymous oriented grid by cooperating agents modeled as either deterministic or probabilistic automata. Such agents are sometimes called ants. It was shown in[25] that 3 randomized or 4 deterministic automata can accomplish this task. Then matching lower bounds were proved: the authors of[11] showed that 2 randomized automata are not enough for target searching in the grid, and the authors of[8] proved that 3 deterministic automata are not enough for this task. Searching for a target in the infinite grid with obstacles was considered in[35].

Our present paper adopts the same model of environment as the above papers, i.e., the infinite anonymous oriented grid. However the task we study is different: instead of searching for a target, the robot has to build some structure from the available material. We note that a task of constructing a boundary box in this setting has been considered in[26], where a bounding-box algorithm is proposed as a tool for more complex tasks like counting and transformations (the tiles themselves, and the automaton, are used to simulate a Turing machine that helps with these operations). In[27] similar problems have been considered for two cooperating finite automata.

There exists a number of works on moving hexagonal tiles by a robot in order to form certain selected shapes (there, the connectivity of the shape is enforced in the model and the robot needs to stay next to full cells). Authors of[31] propose a way of constructing shapes by using simple tile movements performed by a finite automaton: first some intermediate structures are obtained, and the complexity of obtaining them is always $O(n^2)$. Some shape detection problems by a finite automaton have been considered in this model in[30].

We also mention a related field of programmable matter. However, as noted in[16], even different computational models of programmable matter themselves allow the particles to have different capabilities that, in general, prohibit any direct translation of results between those models. This difference of assumptions is even more significant between various programmable matter models that are distributed by nature, and our single automaton case. This difference is reflected in often sublinear time of distributed shape formations[14, 19] which should be contrasted to our results. The *amoebot* model introduced in[18] shows similarities to ours, in the sense that it involves shape formation by moving automata that have sense of direction. An extension of this model that uses the leader particle[15] may potentially allow for simulations of more centralized algorithms that emerge from a single automaton. We also point out, that feasibility questions different from ours arise in the programmable matter models, including those of a possibility of formations of certain patterns[36]. In[20], shape formation algorithms have been proposed in a distributed multi-particle model, where the complexity measured as the number of rounds is $O(n^2)$. The latter is due to intermediate line formations but also to leader election protocol used by the algorithm.

2 The Model

We consider the infinite oriented grid $\mathbb{Z} \times \mathbb{Z}$ represented as the set of unit square cells tiling the two-dimensional plane, with all cell sides vertical or horizontal. Each cell has 4 adjacent cells, North, East, South and West of it. Some cells of the grid contain a brick, i.e., are *full*, other cells are *empty*. The subgraph of the grid induced by full cells is initially connected. At each step of the algorithm this subgraph can change, due to the actions of the robot, described below. At each step, the subgraph induced by the full cells is called the current *shape*. Any maximal connected subgraph of the current shape is called a *component*. Throughout the paper, the *distance* between two cells (x, y) and (x', y') of the grid is the Manhattan distance between them, i.e., $|x - x'| + |y - y'|$. The number of cells of a shape is called its *size*, and the distance between two furthest cells of a shape is called its *span*. A nest of size n is a shape that has the minimum span among all shapes of size n.

We are given a mobile entity (robot) starting in some cell of the initial shape and traveling in the grid. The robot has a priori no knowledge of the shape, of its size or of its span. The robot is formalized as a finite deterministic Mealy automaton $\mathcal{R} = (X, Y, \S, \delta, \lambda, S_0, S_f)$. $X = \{e,f\} \times \{l,h\}$ is the input alphabet, $Y = \{N, E, S, W\} \times \{e,f\} \times \{l,h\}$ is the output alphabet, $\S$ is a finite set of states with two special states: S_0 called initial and S_f called final, $\delta : \S \times X \to \S$ is the state transition function, and $\lambda : \S \times X \to Y$ is the output function.

The meaning of the input and output symbols is the following. At each step of its functioning, the robot is at some cell of the grid and has some weight: it is either light, denoted by l (does not carry a brick) or heavy, denoted by h (carries a brick). Moreover, the current cell is either empty, denoted by e or full, denoted by f. The input $x \in \{e,f\} \times \{l,h\}$ gives the automaton information about these facts. The robot is in some state $\tilde{S} \in \S$. Given this state and the input x, the robot outputs

the symbol $\lambda(\tilde{S}, x) \in \{N, E, S, W\} \times \{e, f\} \times \{l, h\}$ with the following meaning. The first term indicates the adjacent cell to which the robot moves: North, East, South or West of the current cell. The second term determines whether the robot leaves the current cell empty or full, and the third term indicates whether the robot transits as heavy or as light to the adjacent cell. Since the robot can only either leave the current cell intact and not change its own weight, or pick a brick from a full cell leaving it empty (in the case when the robot was previously light), or drop a brick on an empty cell leaving it full (in the case when the robot was previously heavy), we have the following restrictions on the possible values of the output function λ:

$\lambda(\tilde{S}, e, l)$ must be $(\cdot, e, l)$
$\lambda(\tilde{S}, e, h)$ must be either $(\cdot, e, h)$ or $(\cdot, f, l)$
$\lambda(\tilde{S}, f, l)$ must be either $(\cdot, f, l)$ or $(\cdot, e, h)$
$\lambda(\tilde{S}, f, h)$ must be $(\cdot, f, h)$

Seeing the input symbol x and being in a current state $\tilde{S}$, the robot makes the changes indicated by the output function (it goes to the indicated adjacent cell, possibly changes the filling of the current cell as indicated and possibly changes its own weight as indicated), and transits to state $\delta(\tilde{S}, x)$. The robot starts light in a full cell in the initial state S_0 (hence its initial input symbol is (f, l)) and terminates its action in the final state S_f.

3 Terminology and Preliminaries

In the description and analysis of our algorithm we will categorize full cells. A full cell is said to be a *border cell* if it is adjacent to an empty cell. A full cell that has only one full adjacent cell is called a *leaf*. A full cell c is called *special*, if it is either a leaf, or there exist at least two full cells c_1, c_2 adjacent to c such that c_1 and c_2 share a common corner.

A finite deterministic automaton may remember a constant number of bits by encoding them in its states. We will use this fact to define several simple procedures and simplifications that we use in the sequel. The first simplification is as follows. We formulate the actions of the robot based on the configuration of bricks in its neighborhood. More precisely, at any step, the robot knows whether each cell at distance at most $r = 8$ from its current cell is full or empty. This can be achieved by performing a bounded local exploration with return, after each move of the robot.

We will use the notion of current *orientation* of the robot. At the beginning of its navigation, the robot goes in one of the four cardinal directions. Then its orientation is determined in one of the two ways: either by its last step (North, East, South or West) or by a *turn*: we say that the robot *turns left* (respectively *right*) meaning that it changes its orientation in the appropriate way while remaining at the same cell. The robot can remember its orientation, using its states. We use

the cardinal directions to refer to the cells adjacent to the current cell of the robot depending on its orientation. Thus, e.g., if the robot is oriented East then we say that its adjacent North (resp. East, South or West) cell is *left* (resp. *in front*, *right*, *back*) of it.

Whenever we say that the robot located at a cell c and not carrying a brick *brings* a brick from a full cell c' to c we mean that the robot moves from c to c', picks the brick from c', moves back to c and restores its original orientation. Whenever we say that the robot located at a cell c and carrying a brick *places* it at an empty cell c' we mean that the robot moves from c to c', drops the brick at c', moves back to c and restores its original orientation.

Whenever the robot selects a cell according to some condition that is fulfilled by more than one cell, the robot selects the cell that is minimal with respect to the following total order $\preceq$ on the set of all cells. For cells $c = (x, y)$ and $c' = (x', y')$, $c \preceq c'$ holds if and only if either $y < y'$, or $y = y'$ and $x \leq x'$. We denote by $|S|$ the number of cells in a sequence or a set S of cells.

We define a *disc* of radius $r \geq 0$ with center c to be the set of all cells at distance at most r from c, see Fig. 1. A disc of radius r contains $n_r = 2(1 + 3 + 5 + \cdots + (2r - 1)) + (2r + 1) = 2r^2 + 2r + 1$ cells and has span $2r$. A *rough disc* of size n, where $n_r \leq n < n_{r+1}$ is defined as follows. If $n = n_r$, then the rough disc is the disc of radius r. Otherwise, let F be the set of cells not belonging to the disc D of radius r but adjacent to some cell of it. Add to D exactly $n - n_r$ cells belonging to F, starting from the North neighbor of the East-most cell of D and going counterclockwise. If $n_r < n \leq n_r + 2r + 2$ then the rough disc of size n has span $2r + 1$ and if $n_r + 2r + 2 < n < n_{r+1}$ then the rough disc of size n has span $2r + 2$, the same as the disc of radius $r + 1$ that has size n_{r+1}.Note that, when defining a rough disc, one may allow other placements of the $n - n_r$ bricks at the cells from F as long as the resulting shape is minimizing the span. However, we use this somewhat technical definition with such a precise choice of the cells in F to simplify the description of some actions of the robot.

Proposition 3.1 *Any rough disc is a nest.*

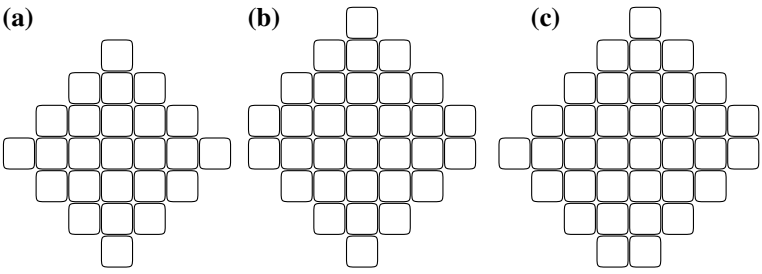

Fig. 1 **a** Disc of size n_r for $r = 3$; **b** a rough disc of size $n_r + 7$ and span $2r + 1$, $r = 3$; **c** a rough disc of size $n_r + 11$ and span $2r + 2$, $r = 3$

Proof We write $d(a, b)$ to denote the distance between two cells a and b. For a given span s, consider a shape C_s that has span s and a maximum number of cells among shapes of this span. Consider three consecutive full cells f_1,f_2 and f_3 of C_s in a row or in a column. Let c_1, c_2 and c_3 be three consecutive cells such that c_i is adjacent to f_i, $i \in \{1, 2, 3\}$. We prove that c_2 is full. We proceed by contradiction, i.e., suppose that c_2 is empty. Consider the shape C' obtained from C_s by adding a brick to the cell c_2. By definition of C_s, the span of C' is larger than s. Thus, there exists a shortest path $\mathcal{W} = (w_1, \ldots, w_{s+2})$ in the grid, that connects two full cells w_1 and w_{s+2} of C'. If c_2 is different from w_1 and w_{s+2} then C_s has span larger than s, which is a contradiction. Thus, without loss of generality, we can assume $c_2 = w_{s+2}$. We have $d(w_1, f_2) \geq s$ because the length of $\mathcal{W}$ is $s + 1$ and f_2 and c_2 are neighbors. Thus, since f_2 is a neighbor of both f_1 and f_3, we have either $d(f_1, w_{s+2}) \geq s + 1$ or $d(f_3, w_{s+2}) \geq s + 1$. Since f_1,f_3 and w_{s+2} are in C_s, this gives a contradiction with the span of C_s being at most s. Hence c_2 is full.

This implies that C_s has the following property: each border cell of C_s is adjacent to two or three empty cells. Define an *extremity* to be an East-most, West-most, North-most or South-most cell of a shape. Thus, either there exists a single East-most (respectively West-most, North-most or South-most) extremity, or there are two adjacent such extremities. Moreover, each border cell of C_s that is not an extremity does not have an adjacent border cell (intuitively, the border is composed of "staircases"). Denote by $\mathcal{F}_s$ the class of shapes with the above two properties. Note that C_s which has maximum size among shapes of span s belongs to $\mathcal{F}_s$.

We will compute the number of cells in any shape C in the class $\mathcal{F}_s$. Partition the cells of C into rows $R_1, \ldots, R_t$ counted from North to South. We have proved that $|R_1| \in \{1, 2\}$ and $|R_l| \in \{1, 2\}$ because these sets consist of extremities only. We first consider the case when there is a total of four extremities in C. Thus, in particular, $|R_1| = |R_t| = 1$. (Note that, in the considered case, s must be even). Let j be the maximal index such that $|R_j| < |R_{j+1}|$ and let j' be the minimal index such that $|R_{j'-1}| \leq |R_{j'}|$. Since for each $i \in \{2, \ldots, j\}$, $|R_i| = |R_{i-1}| + 2$, we obtain $\sum_{i=1}^{j} |R_i| = j^2$. By symmetry, $\sum_{i=j'}^{t} |R_i| = j^2$. Denote $m = j' - j - 1$. The span of C equals $s = t + m - 2 = 2j + 2m - 2$, and hence $m = s/2 - j + 1$. We have $|R_i| = 2j + 1$ for each $i \in \{j + 1, \ldots, j' - 1\}$. This implies $\sum_{i=j+1}^{j'-1} |R_i| = m(2j + 1) = j(s + 1) - 2j^2 + s/2 + 1$. Thus, the size of C is $n(j) = \sum_{i=1}^{t} |R_i| = j(s + 1) + s/2 + 1$. It follows that for a shape of maximum size in $\mathcal{F}_s$ that has a total of four extremities, the index j must be as large as possible, i.e., $j = s/2$. Hence such a shape is a disc of span s, i.e., of radius $r = s/2$.

Now consider all other shapes in the class $\mathcal{F}_s$, i.e., those in which some of the extremities are pairs of adjacent cells. Every such shape F' can be obtained from a shape F in $\mathcal{F}_{s-1} \cup \mathcal{F}_{s-2}$ with exactly four extremities, by adding bricks in some cells adjacent to the border of F. The number of cells in each such shape F' equals $|F| + c_1 x + c_2 y$ for some nonnegative constants c_1 and c_2, where x is the span of F and y is the index j in the preceding paragraph, for the shape F. Consider a shape F' of maximum size among all shapes in $\mathcal{F}_s$. Hence the corresponding shape F has maximum size among all shapes in $\mathcal{F}_{s-1}$ or in $\mathcal{F}_{s-2}$ that have exactly four extremities. As proved above, such a shape F must be a disc. Denote by $h(s)$ the maximum size

of a shape in $\mathcal{F}_s$. In particular we have that $h(s) = \max\{n_{s/2}, n_{(s-2)/2} + 2s - 1\} = n_{s/2}$ for even s, and $h(s) = n_{(s-1)/2} + s$ for odd s.

Let n be such that $n_r \leq n < n_{r+1}$ and consider a nest N of size n. Denote its span by s_n. We have three cases depending on the possible values of n.

First suppose that $n = n_r$. We have $s_n \geq 2r$ because otherwise, for a maximum-size shape M in $\mathcal{F}_{2r-1}$, we would have $|M| \geq |N|$, and $|M| = h(2r-1) = n_{r-1} + 2r = 2r^2 + 1 < n_r$, which would imply $|N| < n$, contradicting the definition of N. Thus, the disc of size $n = n_r$ has span at most s_n and thus it is a nest.

Next suppose that $n_r < n \leq n_r + 2r + 2$. We have $s_n \geq 2r + 1$ because otherwise, for a maximum-size shape M in $\mathcal{F}_{2r}$, we would have $|M| \geq |N|$, and $|M| = h(2r) = n_r < n$, which would imply $|N| < n$, contradicting the definition of N. Hence, the rough disc of size n has span at most s_n and thus it is a nest.

Finally suppose that $n_r + 2r + 2 < n < n_{r+1}$. We have $s_n \geq 2r + 2$ because otherwise, for a maximum-size shape M in $\mathcal{F}_{2r+1}$, we would have $|M| \geq |N|$, and $|M| \leq h(2r+1) = n_r + 2r + 1 < n$, which would imply $|N| < n$, contradicting the definition of N. This proves that the rough disc of size n has span at most s_n and thus it is a nest. □

The nests built by our automaton will be discs or rough discs, depending on the number of available bricks. The following proposition shows that the complexity of our nest-building algorithm is optimal, regardless of the relation between the size of the initial shape and its span (recall that, by definition, the span must be smaller than the size n and it must be in $\Omega(\sqrt{n})$). Our lower bound on complexity follows from geometric properties of the grid, and hence it holds regardless of the machine that builds the nest, i.e., even if the robot is a Turing machine knowing a priori the initial shape.

Proposition 3.2 *Let $s < n$ be any positive integers such that $s \in \Omega(\sqrt{n})$. There exists an initial shape of size n and span $\Theta(s)$, such that any algorithm that builds a nest starting from this shape must use time $\Omega(sn)$.*

Proof The proposition is proved by considering an appropriate initial shape that requires large time to be transformed to a nest. We define a *rough rectangle* of size n and sides $a \leq b$, such that $n - ab < b$, to be the following set of cells: there is a grid $a \times b$ of cells (with a rows and b columns), and the remaining $n - ab$ cells are attached to the North-most cells of the $n - ab$ West-most columns.

Case 1 $s \geq 10\sqrt{n}$.

Take the rough rectangle R with sides $b = s$, $a = \lfloor n/b \rfloor$ as the initial shape. The span of R is $\Theta(s)$. There exist subsets A and B of R of size at least $n/3$ at distance at least $s/4$: A is composed of the $\lceil n/(3a) \rceil$ West-most columns of the grid and B is composed of the $\lceil n/(3a) \rceil$ East-most columns of the grid. Since the span of a nest of size n is smaller than $2\sqrt{n} \leq s/5$, in order to transform R into a nest, at least

$n/3$ bricks have to be moved at distance at least $s/40$. Hence the time of building a nest from the initial shape R is at least $sn/120 \in \Omega(sn)$.

Case 2 $s < 10\sqrt{n}$.

In this case we have $s \in \Theta(\sqrt{n})$. Take the rough rectangle R with sides $b = \lceil 10\sqrt{n} \rceil$, $a = \lfloor n/b \rfloor$ as the initial shape. The span of R is $\Theta(\sqrt{n}) = \Theta(s)$. There exist subsets A and B of R of size at least $n/3$ at distance at least $s/4$. Since the span of a nest of size n is smaller than $2\sqrt{n} \leq s/5$, in order to transform R into a nest, at least $n/3$ bricks have to be moved at distance at least $s/40$. Hence the time of building a nest from the initial shape R is at least $sn/120 \in \Omega(sn)$.

□

4 The Algorithm

The robot will move bricks from the original shape and build two special components. One of them will be a rough disc that will be gradually extended. The second one will be a one-cell component whose only cell is called the *marker*. The robot will periodically get at large distances from the rough disc being built, and the role of the marker will be to indicate to the robot that it got back in the vicinity of the rough disc. Any component that is different from the rough disc and from the marker will be called a *free component*.

During the execution of the entire algorithm, the robot will not ensure that the full cells outside of the rough disc and of the marker form one component—they may form several components—but after adding a new brick to the rough disc these components will be always at a bounded distance, i.e., at distance $O(1)$, from the rough disc that the robot is constructing.

We are now ready to sketch the high-level idea of the algorithm, whose pseudocode is presented at the end of this section as algorithm Nest. First the robot performs some preliminary actions by establishing the marker and the initial rough disc and by calling procedure Sweep, which together result in constructing the first rough disc D (of size one), placing the marker next to it and ensuring that no full cells other than the marker are at distance at most 7 from D. Then each iteration of the *main loop* of algorithm Nest performs three actions. First, it executes procedure FindNextBrick that allows the robot to find a brick in a free component $\mathcal{C}$. This brick must be carefully chosen. For example, greedily picking the closest available brick would soon result in creating large empty spaces between components of the shape, in which the robot could get lost. This brick will be later used to extend the rough disc. However, this procedure may lead the robot far from the rough disc and may also disconnect $\mathcal{C}$ into many components. Disconnecting $\mathcal{C}$ is one of the main tools in our construction. It is done by the robot on purpose to allow it to find its way back to the marker and so that it is possible to recover the connectivity of $\mathcal{C}$ on the way back. Such a walk back to the marker is the second action performed in the main loop and described as procedure ReturnToMarker. The third action is done once the robot is back at the marker, and it is given as procedure ExtendRoughDisc. This procedure extends the rough disc, ensuring the property that there are no full cells at

a prescribed bounded distance from the rough disc, except the marker. While restoring this property, the robot may again disconnect some components but all of them are at a bounded distance from the rough disc and thus the robot will be able to find them easily. Additionally, it may happen that the cell brought to the rough disc was the last cell of $\mathcal{C}$. In such a case, as the last part of procedure ExtendRoughDisc, the robot places the marker near another component close to the rough disc, if one exists. This will be the new free component $\mathcal{C}$ in which the robot will find the next brick in the next iteration of the main loop. If no such $\mathcal{C}$ exists, then the robot adds the brick from the marker to the rough disc, thus completing the construction of the nest.

Many of the difficulties described above resulted from our goal to keep the complexity of the algorithm optimal, i.e., $O(sn)$. If complexity were not an issue, a simpler algorithm would be possible. The construction of a nest could be done in two steps: first the robot builds a horizontal line, which is technically much easier and can be done keeping the shape connected at all times. When the shape is transformed into a line, the robot will recognize this and transform it into a nest by iteratively picking bricks from one endpoint of the line and using them to build consecutive rough discs at the other endpoint of the line. However, even the second step alone requires time $\Theta(n^2)$ due to all traversals of the line, which is suboptimal for small spans s. Thus we proceed with the detailed description of the optimal algorithm Nest whose high-level idea was described before.

4.1 Moving Bricks Out of the Way

One of the challenges in constructing the rough disc is to have enough room so that, while expanding, it does not merge with the remainder of the shape. This is one of the goals of procedure Sweep. Its high-level idea is the following. It ensures an invariant that has to hold whenever the agent goes to retrieve the next brick in order to extend the rough disc. This invariant is that there are no full cells, other than the cell M that is the marker, within a given bounded distance from the current rough disc D. This procedure is called when the robot is at the marker, in two situations. The first one is right before the main loop of algorithm Nest: in this case the marker, the rough disc (of size one) and its corresponding neighborhood occupy a constant number of cells and hence in this case the robot is able to decide whether a given cell is in $D \cup \{M\}$. The second situation occurs after each extension of the rough disc. In this case, the size of D may be unbounded but a walk around its border (note that, thanks to the marker, the robot is able to make a single full traversal of the border of D starting and ending at the marker) allows to determine if there is a full cell c within a bounded distance from D, that does not belong to the rough disc itself. Whenever such a full cell c is found, the robot picks the brick from c and searches for an empty cell at distance at least 7 from the rough disc. This searching walk is done in such a way as to ensure the return to the rough disc. At the end of the procedure, the robot places the marker next to some free component. Below is the pseudo-code of procedure Sweep. In this pseudo-code, we use the notion of the robot going *in direction away* from the rough disc D. This is the direction which

strictly increases the distance between the robot and the rough disc. In the case where there are two such directions, we use the priority North, East, South, West.

Procedure Sweep: sweeping away bricks that are close to the rough disc

```
1  M := the marker
2  go to the closest cell of the rough disc D
3  perform a full counterclockwise traversal of the border of D, executing the following
     actions after each step:
4      for each full cell c ∉ D ∪ {M} at distance at most 7 from the robot do
5          c' := the cell currently occupied by the robot
6          go to c and pick the brick
7          move in direction away from D and stop at the first empty cell at distance at
             least 7 from D
8          drop the brick and return to c'
9  if there exists a free component C then
10     pick the brick from marker and place it at distance 3 from the rough disc and at
         distance 4 from C, creating a new marker
11 go to the marker
```

4.2 Finding the Next Brick

The high-level idea of procedure FindNextBrick is the following. In may happen that the cells that belong to a free component $\mathcal{C}$ and are close to the marker cannot be rearranged in such a way that the robot be able to obtain a brick that it can then use to extend the rough disc and at the same time keep the connectivity of $\mathcal{C}$ and ensure that $\mathcal{C}$ remains close to the marker (for the simplest example of such situation consider $\mathcal{C}$ to be a line). Thus, the robot has to retrieve the needed brick by following a potentially long walk; we will call it a *search walk* and formally define it in Sect. 4.2.1. The search walk needs to be carefully chosen to ensure that it ends at a location where it is possible to find the desired brick and so that the robot be able to return to the marker. Moreover, this walk has to be sufficiently short to guarantee the time $O(sn)$ of the algorithm, i.e., the length of each walk must be $O(s)$. We ensure the latter as follows: each search walk $\mathcal{W}$ leads alternatingly in two non-opposite directions, e.g., North and West. The return is guaranteed by repeatedly performing an action called *switch* while walking along $\mathcal{W}$, which we formally describe in Sect. 4.2.2. Intuitively speaking, the switch eliminates the cells adjacent to $\mathcal{W}$ at which the robot may incorrectly turn on its way back to the marker. The switch potentially disconnects the component but the robot is able to recover the connectivity while backtracking along $\mathcal{W}$. Finally, in Sect. 4.2.3, we describe how the desired brick can be found at the end of $\mathcal{W}$.

4.2.1 Search Walks

Suppose that the robot is currently at a full cell and there is a full cell in front of the robot. A *left-free* (respectively *right-free*) segment consists of all full cells

that will be visited by the robot that moves without changing its direction until one of the following conditions holds:

(S1) the robot arrives at a full cell that has an empty cell in front of it and has an empty cell to the left (respectively right) of it; such a segment is called *terminal*,
(S2) the robot arrives at the first special cell such that the cell to the left (respectively right) of the robot is full.

Whenever the orientation is not important or clear from the context we will refer to a left-free or right-free segment by saying *segment*. Note that not every special cell terminates the above sequence of moves.

We now define a *search walk* $\mathcal{W}$ of the robot in an arbitrary component $\mathcal{C}$ (cf. the example in Fig. 2). A search walk depends on the initial position of the robot in $\mathcal{C}$ and on its orientation. We make two assumptions in the definition: the robot is initially located at a full cell of $\mathcal{C}$ and, if $|\mathcal{C}| > 1$, then there is a full cell in front of the robot. The search walk $\mathcal{W}$ is a concatenation of segments. The first segment is both left-free and right-free. If the cell c at the end of this segment is a leaf, then the construction of $\mathcal{W}$ is completed. Otherwise, note that there is a full cell to the left of the robot located at c or to the right of it. In the former case, the search walk is called *left-oriented* and in the latter it is *right-oriented*. Intuitively, a left-oriented search walk prescribes going straight until it is possible to go left, then going straight until it is possible to go right, and so on, alternating directions, until a stop condition is satisfied. A similar intuition concerns right-oriented search walks.

More formally, if $|\mathcal{C}| > 1$, then the search walk $\mathcal{W}$ consists of a single cell. Otherwise, in a right-oriented (respectively left-oriented) search walk, the segments are sequentially added to $\mathcal{W}$, cyclically alternating the following construction steps.

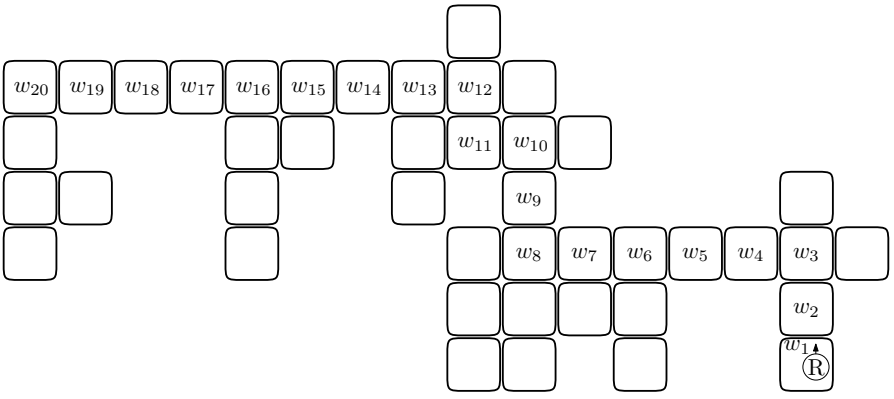

Fig. 2 An example of a search walk $\mathcal{W} = (w_1, \ldots, w_{20})$ that is constructed by the robot initially located at w_1 and facing North. This search walk is left-oriented, and has three left-free segments $S_1 = (w_1, w_2, w_3)$, $S_3 = (w_8, w_9, w_{10})$, $S_5 = (w_{11}, w_{12})$ and three right-free segments $S_2 = (w_3, \ldots, w_8)$, $S_4 = (w_{10}, w_{11})$, $S_6 = (w_{12}, \ldots, w_{20})$

(W1) The robot traverses a right-free (respectively left-free) segment, adding it to $\mathcal{W}$. This segment becomes the last segment in $\mathcal{W}$ if it is a terminal. If the segment is not the last one, then the robot turns right (respectively left).
(W2) The robot traverses a left-free (respectively right-free) segment, adding it to $\mathcal{W}$. This segment becomes the last segment in $\mathcal{W}$ if it is a terminal. If the segment is not the last one, then the robot turns left (respectively right).

4.2.2 Ensuring the Return from a Search Walk

We start with a high-level idea of the mechanism that will ensure the return from a search walk. Whenever the robot follows a search walk $\mathcal{W}$, it may a priori not be able to return to the origin of $\mathcal{W}$. This is due to the fact that, e.g., if $\mathcal{W}$ is left-oriented, then any segment that is right-free may have an unbounded number of special cells such that each of them is adjacent to a cell that does not belong to $\mathcal{W}$. Thus, the returning robot is not able to remember, using its bounded memory, which of such cells do not belong to the search walk and should be skipped. To overcome this difficulty, the robot will make small changes in the shape close to the search walk while traversing it for the first time. These changes may disconnect the component in which the robot is walking, and this may result in creating many new components. While doing so, we will ensure two properties. First, thanks to the modifications in the shape performed while traversing $\mathcal{W}$, the robot is able to return to the first cell of this search walk. Second, while backtracking on $\mathcal{W}$, the robot is able to undo earlier changes and recover the connectivity of the component.

Each cell c at which the robot stops to perform the above-mentioned modification will be called a *break point* and is defined as follows. First, we require that c be an internal cell of a segment S, i.e., neither the first nor the last cell of S. Second, if S is left-free (respectively right-free), then when the robot traversing S is at c, there is a full cell f to the right (respectively left) of it. Clearly, the cell e to the left (respectively right) of the robot is empty. The following couple of actions performed by the robot located at such a cell c are called a *switch*: if the robot is not carrying a brick, then the robot brings the brick from f and then places it at e, and if the robot is carrying a brick, then the robot places it at e and then brings the brick from f. Note that the switch may disconnect the component in which the robot is located thus creating two new components. One new component is the one in which the robot is located and this is the component that contains the search walk. The second new component is the one that contains the cell f' adjacent to f and at distance two from c, if f' is full. If the cell f' is full and belongs to a separate component, then this second component containing f' will be called a *switch-component*. Whenever the robot traversing a segment performs the switch at each internal special cell of the segment, we say that this is a *switch-traversal* (cf. the example in Fig. 3).

4.2.3 Obtaining a Brick at the End of a Search Walk

Informally, the purpose of traversing the entire search walk by a robot is to arrive at a location in the current component $\mathcal{C}$ of the robot, where the robot can start a procedure aimed at obtaining a brick whose removal will not disconnect $\mathcal{C}$. We will say

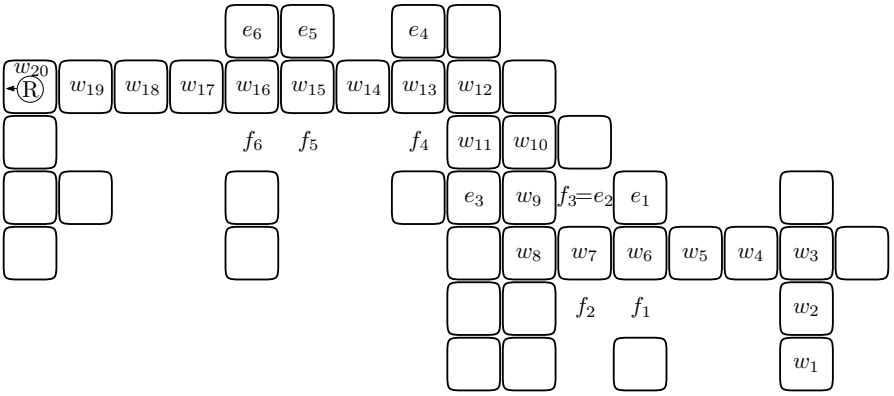

Fig. 3 The shape from Fig. 2 after switch-traversal of the search walk from Fig. 2. The cells $w_6, w_7, w_9, w_{13}, w_{15}$ and w_{16} are the break points at which the robot moves a brick from a cell f_i to e_i for $i \in \{1, \ldots, 6\}$. Note that a brick is moved from f_2 to e_2 while traversing the second segment and then the same brick is moved to e_3 while traversing the third segment

that such a brick is *free*. In our algorithm, we check the condition (S1) to learn if the last segment is terminal. According to the condition, the terminal segment may end with a leaf, and in such a case the robot is at a cell with a free brick. If the terminal segment does not end with a leaf, then there need not exist a free brick located in a close neighborhood of the robot. However, we will prove that it is possible to perform a series of changes to the shape that results in creating a configuration of bricks that does contain a free brick.

We now define the behavior of the robot that ended the switch-traversal of the last segment S of a search walk $\mathcal{W}$ and arrived at a cell that is not a leaf. The following series of moves is called *shifting* (cf. Fig. 4). Suppose that the cell to the right (respectively left) of the robot is full. Note that this implies that S is left-free

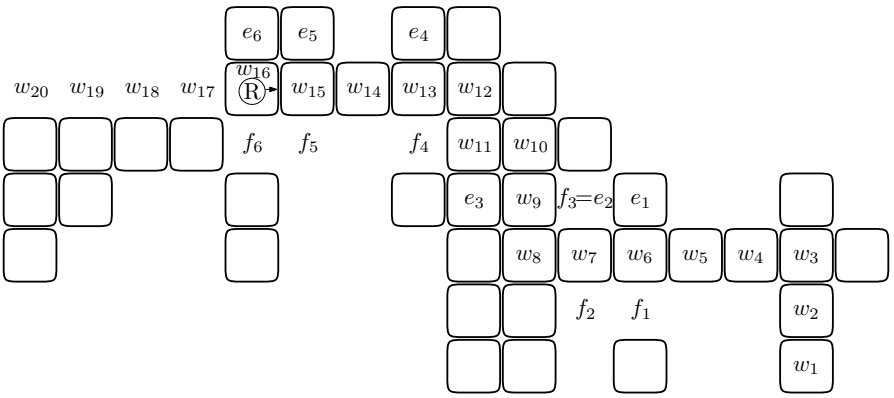

Fig. 4 The shape from Fig. 3 at the end of shifting. The shifting ends with a right-free segment, at the cell w_{16} because it has a full neighbor, the cell e_6. There is one fewer brick than in Fig. 3 and this is the free brick obtained and carried by the robot

(respectively right-free). First the robot changes its direction so that a cell of S is in front of it (i.e., the robot turns back). The following three actions are performed until the stop condition specified in the third action occurs. First, the robot picks the brick from the currently occupied cell. Second, the robot moves one step forward—thus backtracking along S. Third, when the robot is at a special cell, then the shifting is completed, and otherwise the robot places the brick at the cell to the left (respectively right) of it. Note that when the robot arrives at a special cell, it is carrying a brick and this is the desired free brick.

We now give the pseudo-code of procedure FindNextBrick that obtains this brick.

Procedure FindNextBrick: finding a free brick

```
1 W := the search walk that starts at the cell where the robot is
    located
2 go to the nearest cell belonging to a free component
3 perform a switch-traversal of W
4 if the robot is at a leaf then
5     pick the brick
6 else
7     perform shifting
```

4.3 Back to the Marker

Before presenting the high-level idea of procedure ReturnToMarker that takes the robot carrying a brick back to the marker, we need the following definitions. If S is a segment, then the *reversal* of S, denoted by $\varphi(S)$, is the segment composed of the same cells as S but in the reversed order. For a search walk $\mathcal{W}$ that is a concatenation of segments $S_1, \ldots, S_l$, define the *reversal* of $\mathcal{W}$, denoted by $\varphi(\mathcal{W})$, to be the walk that is the concatenation of segments $\varphi(S_l), \varphi(S_{l-1}), \ldots, \varphi(S_1)$, in this order. Thus, following $\varphi(\mathcal{W})$ means backtracking along $\mathcal{W}$, and in this section we give a procedure performing it, that reconnects the previous free component on the way. We also define the orientation of $\varphi(\mathcal{W})$ as follows. If the last segment of $\mathcal{W}$ is left-free, then $\varphi(\mathcal{W})$ is left-oriented, and otherwise $\varphi(\mathcal{W})$ is right-oriented. Thus, if $\mathcal{W}$ ended with a left-free (respectively right-free) segment, then $\varphi(\mathcal{W})$ also starts with a left-free (respectively right-free) segment.

At a high level, the robot will perform a switch-traversal along $\varphi(\mathcal{W})$, stopping at each break point to reconnect the corresponding switch-component with the component in which the robot is walking. However, we need to ensure that, at the end, the robot stops at the right point, i.e., at the marker. In order to ensure this, we define the following condition:

(S1′) the robot arrives at a cell at distance at most 4 from the marker.

We define a *return switch-traversal* of $\varphi(\mathcal{W})$ to be a switch-traversal of $\varphi(\mathcal{W})$ in which each verification of condition (S1) is replaced by the verification of condition (S1′). Recall that the condition (S1) is checked in the definition of a switch-traversal to determine the termination of a segment and consequently the termination of the entire search walk. Intuitively, by replacing condition (S1) with (S1′) we change the behavior of the robot so that it is looking for the marker while backtracking along $\mathcal{W}$, i.e., going along $\varphi(\mathcal{W})$.

A high-level sketch of procedure ReturnToMarker is the following. As indicated earlier, the robot essentially follows $\varphi(\mathcal{W})$ and, as the return switch-traversal dictates, reconnects the switch components. However, there is one special case in which the robot should not perform a switch while being at a cell c' of $\varphi(\mathcal{W})$, although c' satisfies the definition of a break point. This case occurs if the shifting moved all internal cells of the last segment of $\mathcal{W}$. In this case, it is enough for the robot to move to the next cell after c' and start the return switch-traversal of $\varphi(\mathcal{W})$ from there. This is feasible because the cell c and its neighbors are at a bounded distance from the robot when the shifting is completed. Below is the pseudo-code of procedure ReturnToMarker.

Procedure ReturnToMarker: going back to the marker

```
1 W := the search walk traversed in the last call to FindNextBrick
2 let S_1, ..., S_l be the segments in W
3 if the robot is at the first cell of S_l then
4     turn towards the penultimate cell of S_{l-1}
5     move min{2, |S_{l-1}| − 1} cells forward
6     if |S_{l-1}| = 2 and l > 2, then make a turn towards the penultimate cell of S_{l-2}
7 starting at the current location, perform a return switch-traversal of φ(W)
8 go to the marker
```

4.4 Extending the Rough Disc

The aim of procedure ExtendRoughDisc is double: it adds a new brick to the current rough disc in a specific place, and it calls procedure Sweep to extend, if necessary, the empty space around the rough disc and to ensure that the marker is close to some free component. Whenever procedure ExtendRoughDisc is called, the following conditions will be satisfied: the robot is at the marker and it is carrying a brick. Below is the pseudo-code of procedure ExtendRoughDisc.

Procedure ExtendRoughDisc: adding one brick to the rough disc D

```
1 place the brick at the unique cell e such that D ∪ {e} is a rough disc
2 call procedure Sweep
```

Now the pseudo-code of the entire algorithm can be succinctly formulated as follows.

Algorithm Nest: building a nest from any connected shape

```
1  if the span of the shape is at most 2 then
2      exit ▷ the shape is already a nest
3  the cell occupied by the robot becomes the marker
4  a full cell at distance 2 from the marker becomes the initial rough disc
5  call procedure Sweep
6  while there exists a free component C do
7      call procedure FindNextBrick
8      call procedure ReturnToMarker
9      call procedure ExtendRoughDisc ▷ moves the marker if necessary
10 pick the brick (the marker) and place it at the unique cell e of the rough disc
     D such that D ∪ {e} is a rough disc
```

The following is the main result of this paper.

Theorem 4.1 *Algorithm Nest builds a nest starting from any connected shape of size n and span s in time O(sn). This time is worst-case optimal.*

5 Analysis of the Algorithm

This section is devoted to the proof of Theorem 4.1.

5.1 Search Walks

The first three observations provide simple properties of search walks that will be used in the sequel.

Observation 5.1 *Suppose that $\mathcal{W}$ is the search walk from procedure FindNextBrick. After performing the switch-traversal in line 3 of the procedure, the robot is at the end of $\mathcal{W}$.*

Observation 5.2 *A left-free (respectively right-free) segment becomes right-free (respectively left-free) as a result of a switch-traversal.*

Observation 5.3 *Let $\mathcal{W}$ be a search walk whose last segment S is left-free (respectively right-free). Upon completion of a switch-traversal of $\mathcal{W}$, if the robot is not at a leaf, then the cell to the right (respectively left) of the robot is full and the cells in front of it and to the left (respectively right) of it are empty.*

We will need some additional notation. Let $\mathcal{W}$ be a search walk in a component $\mathcal{C}$ and b be the last break point of $\mathcal{W}$. All cells of $\mathcal{W}$ starting at b are called the *prefix of* $\varphi(\mathcal{W})$. Note that, at each point of the execution of procedure FindNextBrick, some parts of $\mathcal{C}$ become the switch-components. The other component, namely the one that overlaps with $\mathcal{W}$ will be called the $\mathcal{W}$*-component*. (Note that we specify that it overlaps with $\mathcal{W}$ and does not contain $\mathcal{W}$ since, after the shifting, some cells of $\mathcal{W}$ that are in the prefix of $\varphi(\mathcal{W})$ are empty and thus are not a part of the $\mathcal{W}$-component.) Upon completion of shifting, each component that contains a brick placed during shifting will be called a *shift-component*.

Lemma 5.1 *Upon completion of shifting performed in procedure FindNextBrick, we have the following properties.*

(i) *The shift-component is unique.*
(ii) *Let c' be the cell adjacent to the cell occupied by the robot and also adjacent to the cell on which the last brick has been placed during shifting. If c' is full, then it belongs to the shift-component and it belongs to $\mathcal{W}$.* If *c' is empty, then it is adjacent to a cell of the shift-component.*
(iii) *The robot is located at the penultimate special cell of $\mathcal{W}$ and is carrying a brick.*

Proof Consider a shifting that occurs after a switch-traversal of $\mathcal{W}$. Let S be the last segment of $\mathcal{W}$. The segment S is either left-free or right-free, by the definition of a search walk. We consider the former case (cf. the example in Fig. 5); the latter one is analogous and will be skipped. Intuitively, property (i) follows from the observation that shifting moves a series of bricks by one (one cell to the North in Fig. 5). The uniqueness of the shift component comes then from its connectivity, which is ensured because these cells originally had no neighbors outside of the search walk.

By Observation 5.1 and the formulation of procedure FindNextBrick, when the shifting starts the robot is at the last cell of S. Denote this cell by c_1. Let $c_2, \ldots, c_l$ be the cells of S such that c_i is adjacent to c_{i-1} for each $i \in \{2, \ldots, l\}$, c_l is a special cell in $\mathcal{W}$ and $c_2, \ldots, c_{l-1}$ are not special cells in $\mathcal{W}$. Note that the latter sequence may be empty if two last cells of S are special cells. Denote by c'_i the cell that is to the left of the robot when it is at c_i during shifting.

According to the definition of shifting, the following occurs: for each $i \in \{1, \ldots, l-1\}$, the brick from c_i is moved to the cell c'_{i+1}. We argue that shifting is feasible, i.e., that the cell c_i, $i \in \{1, \ldots, l-1\}$, is full when the robot picks the brick from it and that the cell c'_{i+1}, $i \in \{1, \ldots, l-2\}$, is empty when the robot then drops the brick at it, for $i \in \{1, \ldots, l-1\}$. The former is due to the fact that c_i belongs to a segment which by definition is a sequence of full cells. The latter holds because $c_2, \ldots, c_{l-1}$ are not special cells, i.e., their neighbors outside of S, in particular the cells $c'_2, \ldots, c'_{l-1}$, are empty.

We now analyze what types of components are created during shifting from the component $\mathcal{C}$ in which it started. Consider the components $\mathcal{C}_1, \ldots, \mathcal{C}_t$ obtained from $\mathcal{C}$ by making the cells $c_1, \ldots, c_{l-1}$ empty. These are all cells from which the robot picked the bricks during shifting. First observe that there are exactly two

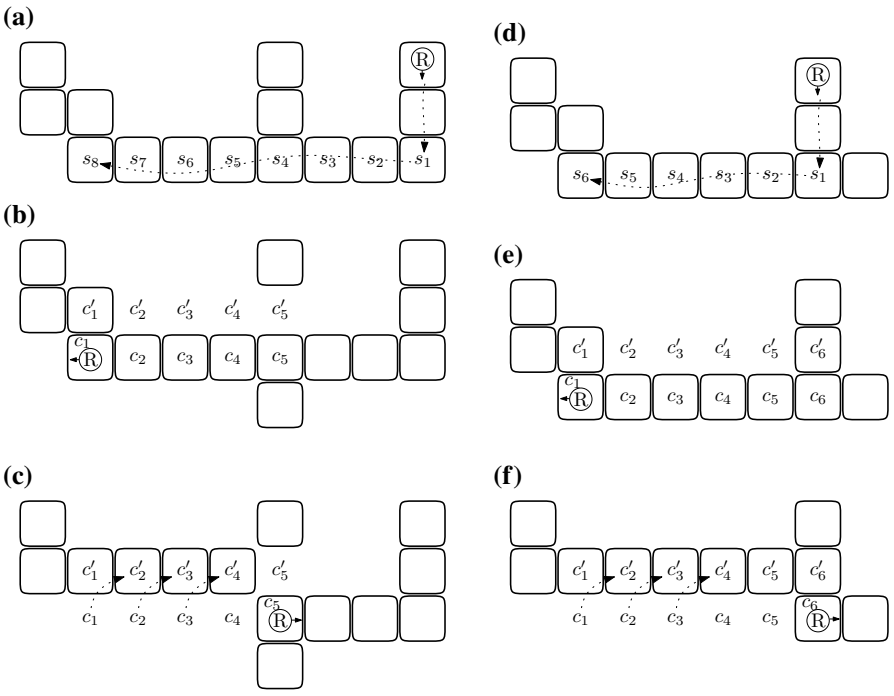

Fig. 5 **a**–**c** illustrates the case when the cell c' in Lemma 5.1(ii) is empty: **a** shows the robot traversing the penultimate segment after which, in (**b**), a switch-traversal of the last segment $(s_1, \ldots, s_8)$ is completed, and **c** depicts the shape at the end of shifting. Note that in this case $c' = c_l = c'_5$. **d**–**f** illustrates the case when c' is full: **d**, **e** again depict the switch-traversal of the last two segments, and **f** shows the shape at the end of shifting. In this case $c' = c_l = c'_6$ and this cell belongs to the penultimate segment of the search walk. During the traversal of the last segment no switch occurred. As we will prove, the two latter properties are always satisfied together

components as above, i.e., $t = 2$. The first one, say $\mathcal{C}_1$, contains the cell c'_1 because, according to the 'if' statement in line 4 of procedure FindNextBrick, c_1 is not a leaf and hence, due to Observation 5.3, the cell c'_1 is full. The second component, let it be $\mathcal{C}_2$, contains c_l, which is full because it belongs to $\mathcal{W}$ and has not been emptied during shifting. To finish the argument that $t = 2$, note that by definition of shifting, the cells $c_2, \ldots, c_{l-1}$ are not special cells and thus have no full neighbors outside of S, and by Observation 5.3, the cell c_1 has no other full neighbors in $\mathcal{C}$ except c_2 and c'_1. Observe that making the cells $c'_2, \ldots, c'_{l-1}$ full during shifting glues these cells to $\mathcal{C}_1$, resulting in a single component. This is the unique shift-component. Thus property (i) is proved.

Properties (ii) and (iii) are basically due to the fact that the robot performing the shifting is backtracking along the last segment of the search walk, and the shifting either ends in the middle of the segment (in which case this is its last break point) or at its beginning.

The cell that is to the right of the robot upon completion of shifting is c'_l. Note that $c' = c'_l$ and this cell is adjacent to c'_{l-1}. Hence, if c'_l is empty, then (ii) holds. Suppose

that c'_l is full. The shift-component $\mathcal{C}_1$ and the $\mathcal{W}$-component $\mathcal{C}_2$ are the same: c'_l is adjacent to the full cell c'_{l-1} of $\mathcal{C}_1$ and is also adjacent to the full cell c_l that belongs to $\mathcal{C}_2$. Since S is left-free, by the definition of switch-traversal we obtain that the brick located at c'_l has not been dropped at c'_l as a result of a switch. Moreover, it cannot be a neighbor of an internal cell of S. Since this is not the last cell of S because $l > 1$, we have that c_l is the first cell of S. Hence c'_l is the penultimate segment of $\mathcal{W}$. This proves property (ii).

Finally, the brick picked from c_{l-1} is carried by the robot at the end of shifting. At the end of shifting, after picking the brick, the robot moves to c_l which is the penultimate special cell of $\mathcal{W}$ because the cells $c_2, \ldots, c_{l-1}$ are not special cells by definition. This proves property (iii) and completes the proof of the lemma. □

We say that the rough disc has a *gap of width k* if each free component is at distance $k + 1$ from the rough disc. We say that a shape is *structured* if it satisfies the following conditions:

(F1) the marker is at distance 3 from the rough disc and at distance 4 from some free component,
(F2) the rough disc has a gap of width 7.

The next lemma characterizes the situation upon completion of procedure FindNextBrick. Its claims (i) and (ii) essentially follow from the previous lemma. Claim (iii) is due to an observation that all possible modifications to the shape (shifting, a switch) are 'local' and thus can decrease the gap by a constant which turns out to be at most 2.

Lemma 5.2 *Suppose that the shape is structured at the beginning of an execution of procedure FindNextBrick. Let $\mathcal{W}$ be the search walk traversed by the robot in a free component $\mathcal{C}$ during this execution. At the end of this execution we have the following properties*:

(i) *the robot is located in the prefix $\varphi(\mathcal{W})$ and carries a brick.*
(ii) *each brick that belongs to $\mathcal{C}$ at the beginning of the execution is either in a switch-component, in the shift-component or in the $\mathcal{W}$-component*,
(iii) *if $\mathcal{W}$ starts with a cell at distance 7 from the rough disc, then the rough disc has a gap of width at most 7 and at least 5.*

Proof Consider the robot at the end of the execution of procedure FindNextBrick. We consider two cases depending on the execution of the 'if' instruction in line 4 of procedure FindNextBrick. In the first case, suppose that the instruction in line 5 is executed. The robot is at the last cell c of $\mathcal{W}$ and carries a brick that it picked from c. This proves (i) in this case. Since c was a leaf upon arrival of the robot at c, picking c did not disconnect the $\mathcal{W}$-component, which implies that the shift-component equals the $\mathcal{W}$-component and consequently proves (ii) of the lemma.

In the second case, suppose that instruction in line 7 of procedure FindNextBrick is executed, i.e., the robot performed shifting. Then, statement (i) follows

from Lemma 5.1(iii), since, by definition, the prefix $\varphi(\mathcal{W})$ contains all cells between the last special cell and the penultimate special cell of $\mathcal{W}$, inclusively. Again by Lemma 5.1(i), the shift-component is unique which implies statement (ii) in this case.

We now prove (iii). Since the shape is structured at the beginning of the execution of procedure FindNextBrick, it has a gap of width 7. The first cell of $\mathcal{W}$ does not change its position and since its distance from the rough disc is 7, we have that the gap is not larger than 7 at the end of the execution. Now we analyze how much the gap can decrease. During the execution of procedure FindNextBrick, the robot performs two types of changes in the shape. The first one is a switch, which places each brick next to a full cell of the component. Thus, this can decrease the gap by at most one. The second one is the shifting, which moves each brick to a cell at distance 2 from its origin, which may decrease the gap by at most 2. The bricks moved during a switch do not belong to $\mathcal{W}$ and the bricks moved during shifting belong to it. This proves (iii) and concludes the proof of the lemma. □

5.2 Reversals of Search Walks

If a component is at distance larger than 7 from the rough disc, then it is called a *lost component*. Recall the definition of a switch, in which the robot located at a cell c brings a brick from a full cell f and places it at an empty cell e. The cell e will be called a *switch-cell*. If, during a return switch-traversal, the robot is backtracking on a segment S, which was added to the walk using condition (W1) or (W2), and it checks, upon arrival at a special cell c of S, whether the cell to the left (respectively right) of c is full, then we say that the robot is *looking for left* (respectively *right*) *turn* during this traversal of S. Suppose that the robot is located at a cell c of a segment S during backtracking. We say that the robot is *oriented* on $\varphi(S)$ if it is directed towards the beginning of S. When the robot backtracks on S, then all cells that belong to S and are located between c (inclusively) and the beginning of S (exclusively) are called the *leftover of* S. In other words, these cells are the ones to be yet visited by the robot backtracking on S.

In the three following lemmas we prove that procedure ReturnToMarker can be correctly executed by the robot. The proof of Lemma 5.3 carries out technical observations that the mentioned actions can be done locally by the robot because they require remembering constants related to l and $|S_l|$, and the direction of the search walk.

Lemma 5.3 *The robot can correctly execute lines 1–6 of procedure ReturnToMarker.*

Proof Note that lines 1 and 2 result in no actions of the robot and only introduce notation for the description of the remaining steps of the procedure. The condition in the 'if' statement in line 3 can be checked using Lemma 5.1(ii): the robot tests whether the cell c' in the lemma is full or not and it is full if and only if the robot

is at the first cell of S_l. Now we analyze the instructions within the 'if' statement. In line 4, the penultimate cell of S_{l-1} is the above cell c' and hence the robot can perform the correct turn. The value of $|S_{l-1}|$, may be large and thus unknown to the robot. However, for calculating $\min\{2, |S_{l-1}| - 1\}$ it is enough to know whether $|S_{l-1}| = 2$ or $|S_{l-1}| > 2$. This can be remembered by the robot while performing the traversal of a search walk in procedure FindNextBrick (although the robot does not know whether the current segment is penultimate, it may always remember the information about the two recently traversed segments). To execute the instruction in line 6, the robot can remember whether $l > 2$, and it turns in the opposite direction to the turn just made in line 4, which follows from the definition of a search walk. □

The proof of the next lemma reveals how each switch performed by the robot along a segment S of a search walk allows it to backtrack. Informally, the fact that the robot made a switch-traversal, and not just a traversal, along a left-free segment S precisely guarantees that on the way back, i.e., when the orientation of the robot is 'reversed', this segment (which is then $\varphi(S)$) is also left-free. This allows the robot to traverse the entire $\varphi(S)$, once it started moving along $\varphi(S)$. Finally, the robot turns correctly towards the next segment because the next segment provides a left turn with respect to the robot returning along $\varphi(S)$.

Lemma 5.4 *Suppose that the robot is oriented on $\varphi(S)$, where S is a left-free (respectively right-free) segment of some search walk. If*:

(i) *the robot previously performed a switch-traversal along S,*
(ii) *the robot is looking for a left (respectively right) turn while traversing $\varphi(S)$,*

then, during the return switch-traversal, the robot arrives at the end of $\varphi(S)$ and turns left (respectively right).

Proof Assume that the robot is looking for a left turn. The other case is analogous and we omit it. Thus, S is left-free prior to the switch-traversal performed in (i). By Observation 5.2, $\varphi(S)$ is also left-free after the switch-traversal. Let S' be the maximal sequence of cells that the robot traversed during the return switch-traversal while looking for a left turn, before it turned left. Since $\varphi(S)$ is left-free, this implies that the robot reaches the end of $\varphi(S)$, i.e., S' contains the endpoint of $\varphi(S)$. Let c' be the cell at which the robot finishes the traversal of S'. Let c be the first cell of S. We need to show that $c = c'$, i.e., that the robot does make a turn when located at c. By the definition of a search walk, the first cell of S is a special cell. By the definition of a switch-traversal, the robot does not perform a switch at the end nor at the beginning of a segment. Thus, in particular, no switch occurred when the robot was located at c during the traversal of S while performing the search walk $\mathcal{W}$. This means that a neighbor of c that is not in S is full during the traversal of S in $\mathcal{W}$ if and only if this neighbor is full during the traversal of S'. Thus, c is a special cell when the robot arrives at c while traversing S'. Moreover, this special cell ends the segment $p(S)$ that precedes S in $\mathcal{W}$. Since $p(S)$ is not a terminal segment, we have that it ended by satisfying condition (S2). Since S is left-free, the segment $p(S)$ is right-free

by the definition of a search walk. This implies that when the robot arrived at c during the search walk $\mathcal{W}$, it had a full cell to its right. This full cell belongs to S and therefore, when the robot arrives at c while traversing S', it has a full cell to its left. Since the robot is looking for a left turn, c becomes the last cell of S' proving that $c = c'$, and moreover, the robot turns left at c, as required by the lemma. □

The following lemma shows that, while executing procedure ReturnToMarker, the robot does not lose its way during the backtrack on a previously traversed search walk. Its proof is inductive and its high-level idea is the following. The base case of the induction asks for the analysis of the behavior of the robot in the last segment of the search walk $\mathcal{W}$. This boils down to using Lemma 5.2 that analyzes what happens at the end of $\mathcal{W}$, i.e., during shifting, and then using Lemmas 5.3 and 5.4 , which ensure proper behavior of the robot in the rest of the last segment of $\mathcal{W}$, i.e., after shifting till getting to the beginning of the penultimate segment of $\mathcal{W}$. The inductive argument then directly follows from Lemma 5.4, except that some special technical case needs to be analyzed that depends on the length of the penultimate segment S_{l-1}, which is related to lines 5 and 6 of procedure ReturnToMarker.

Lemma 5.5 *Let $\mathcal{W}$ be a search walk. Suppose that the robot traversed $\mathcal{W}$ by executing procedure FindNextBrick and then executed procedure ReturnToMarker. These two executions ensure that the robot traverses $\varphi(\mathcal{W})$ after the traversal of $\mathcal{W}$.*

Proof Denote by $S_1, \ldots, S_l$ the segments whose concatenation, in this order, gives the search walk $\mathcal{W}$. By definition, a return switch-traversal is a walk that traverses a sequence of segments denoted by $S'_1, \ldots, S'_{l'}$. We argue by induction on $i \in \{1, \ldots, l'\}$ that:

(a) $S'_i = \varphi(S_{l-i+1})$, and
(b) upon arrival at the end of S'_i, the robot becomes oriented on S_{l-i}.

Before giving the inductive proof we note that property (i) implies that the return switch traversal is a concatenation of segments $\varphi(S_l), \ldots, \varphi(S_1)$ and thus it is the walk $\varphi(\mathcal{W})$ as required by the lemma. We note that property (b) is used for technical reasons to conduct the inductive argument.

For the base case of $i = 1$ we need to argue that the robot traverses the segment $\varphi(S_l)$ and makes the turn after which the traversal of $\varphi(S_{l-1})$ will start. By Lemma 5.2(i), the robot is at a cell c' in the segment $\varphi(S_l)$ at the end of execution of procedure FindNextBrick. Thus, the robot is in $\varphi(S_l)$ when procedure ReturnToMarker starts. The segment S_l has been appended to $\mathcal{W}$ in construction step (W1) or (W2) as a left-free or a right-free segment. We assume that S_l is left-free. The proof for a right-free segment S_l is analogous and will be omitted. If shifting did not occur during the execution of procedure FindNextBrick then, according to line 5, the robot is at the end of the search walk $\mathcal{W}$, i.e., at the end of S_l. If shifting did occur in line 7 of procedure FindNextBrick, then c' is not the last cell of S_l.

If c' is the first cell of S_l (which is the last cell of $\varphi(S_l)$), then by Lemma 5.3, the robot can correctly verify this in line 3 of procedure ReturnToMarker. Moreover, this implies that (a) holds for $i = 1$. Then, the robot performs the turn in line 4 which proves (b).

Now suppose that c' is not the first cell of S_l. The assumption (i) of Lemma 5.4 is satisfied, where we take $S = S_l$ in the lemma. By the definition of return switch-traversal, the fact that $\mathcal{W}$ ended with a left-free segment (recall that S_l is left-free) implies that, when the robot starts its walk in line 7 of procedure ReturnToMarker, it follows a left-oriented walk. The traversal of S'_1 is thus determined by the first construction step in (W1) for this left-oriented walk. Thus, the robot traverses a left-free segment as the first segment S'_1, i.e., it is looking for a left turn. This implies that the assumption (ii) of Lemma 5.4 is satisfied. Then, (a) and (b) follow from Lemma 5.4. This completes the proof of the base case.

For the proof of the inductive step suppose that assertions (a) and (b) hold for some $i \in \{1, \ldots, l-1\}$ and we prove them for $i+1$. By the inductive assumption, the robot is at the end of $\varphi(S_{l-i+1})$ when procedure ReturnToMarker starts. We consider three cases.

Case 1. $i + 1 = 2$, S_{l-1} is of length 2, and the robot is at the beginning of S_l upon completion of shifting performed in line 7 of procedure FindNextBrick.

By Lemma 5.3, the condition in the 'if' statement in line 3 of procedure ReturnToMarker is correctly verified by the robot, and by inductive assumption, it is oriented on $\varphi(S_{l-1})$. Since the length of S_{l-1} is 2, the robot makes one step forward (see line 5) and makes a turn (see line 6). This ensures claims (a) and (b) in this case.

Case 2. $i + 1 = 2$, S_{l-1} is of length larger than 2, and the robot is at the beginning of S_l upon completion of shifting performed in line 7 of procedure FindNextBrick.

By the inductive assumption, the robot is oriented on S_{l-1}. By Lemma 5.3, the robot makes two steps along $\varphi(S_{l-1})$ by executing instruction in line 5 of procedure ReturnToMarker and thus it is at the third cell of $\varphi(S_{l-1})$ when the return switch-traversal in line 7 starts. Thus, since the robot is at a cell of S_{l-1}, condition (i) of Lemma 5.4 is satisfied. By arguments analogous to the ones in the base case of induction, condition (ii) holds as well. Thus by Lemma 5.4, claims (a) and (b) hold for $i + 1$.

Case 3. $i + 1 > 2$ or ($i + 1 = 2$ and the robot is not at the beginning of S_l upon completion of shifting performed in line 7 of procedure FindNextBrick).

The robot is at the beginning of S_{l-1} when the return switch-traversal starts. As before, conditions (i) and (ii) of Lemma 5.4 are satisfied. Thus by Lemma 5.4, claims (a) and (b) hold for $i + 1$. □

We say that the shape is *strongly structured* if the following conditions are satisfied:

(T1) the shape is structured,
(T2) there are no lost components, and
(T3) the robot is at the marker.

We will show that the shape is strongly structured at the beginning of any iteration of the 'while' loop in Algorithm Nest.

The following lemma shows that while backtracking on a search walk, the robot reconnects all switch components that were disconnected during procedure FindNextBrick. Moreover, this backtracking stops when the robot is at the marker. An outline of the proof is as follows. An assumption that the shape is strongly structured allows the robot to find a free component by walking around the current rough disc. The argument then is composed of using earlier lemmas in the following way. Lemma 5.5 allows us to say that the robot goes along a search walk $\mathcal{W}$ and gets back to the marker by following $\varphi(\mathcal{W})$. The fact that no lost components are left behind as a result of this is due to Lemmas 5.1 and 5.2 .

Lemma 5.6 *Suppose that at the beginning of an iteration of the "while" loop in algorithm Nest the shape is strongly structured. Then there are no lost components in the shape and the robot is at the marker after the execution of procedures FindNextBrick and ReturnToMarker.*

Proof Since, by assumption, the robot is at the marker at the beginning of the iteration of the 'while' loop, the robot is at the marker when procedure FindNextBrick starts. Thus, in line 2 of this procedure, the robot goes to a free component $\mathcal{C}$. A free component exists due to the condition of the loop. The fact that such a component $\mathcal{C}$ is at distance 4 from the marker follows from the assumption that the shape is structured. Let $\mathcal{W}$ be the search walk traversed in a component $\mathcal{C}$ during the execution of procedure FindNextBrick. By Observation 5.1, the robot arrives at the last cell of $\mathcal{W}$ as a result of the switch-traversal in line 3 of procedure FindNextBrick. By Lemma 5.2(ii), each component obtained from $\mathcal{C}$ is either a $\mathcal{W}$-component, a switch-component or a shift-component. By Lemma 5.1(i), the shift-component is unique. The shift-component equals the $\mathcal{W}$-component if shifting did not occur during the execution of procedure FindNextBrick, i.e., when line 5 of this procedure has been executed. We now consider the case when the shifting did occur. We have two possibilities, depending on whether the cell c' from Lemma 5.1(ii) is full or empty upon completion of shifting, i.e., at the beginning of the execution of procedure ReturnToMarker. By Lemma 5.1(ii), if c' is full, then it belongs to the shift component and to $\mathcal{W}$. Thus, the shift-component is identical to the $\mathcal{W}$-component. By Lemma 5.1(ii), if c' is empty, then c' is adjacent to a cell c'' of the shift component. Since c is a special cell, there is a full cell f to the left or to the right of the robot. Since c' is to the left or to the right of the robot and c' is empty, we have that $f \neq c'$. Moreover, by Lemma 5.1(ii), c is not the first cell of the last segment in $\mathcal{W}$. Thus, the condition in the 'if' statement in line 3 is false and hence the robot starts the return switch-traversal in line 7 in procedure ReturnToMarker at the cell c. In this switch-traversal, the robot moves the brick from f to c' thus reconnecting the shift component (whose full cell c'' is adjacent to c') with the $\mathcal{W}$-component. At this point, the shift component and the $\mathcal{W}$-component are equal and therefore the initial shape $\mathcal{C}$ is transformed into a shape with the $\mathcal{W}$-component and possibly many switch-components.

By Lemma 5.5, the robot backtracks on $\mathcal{W}$ by executing procedure ReturnToMarker. In other words, the robot traverses $\varphi(\mathcal{W})$. Since the robot traverses the entire search walk $\varphi(\mathcal{W})$, by the definition of return switch-traversal, each switch-component is reconnected with the $\mathcal{W}$-component. It remains to argue that during the return switch-traversal no lost component is created. Such a component could be created, if there existed a full cell c' adjacent to a cell c of $\varphi(\mathcal{W})$ such that the robot performs a switch at c. This switch would remove the brick from c' thus creating a lost component. We now show that this is impossible. The cell c' has been made full during the switch-traversal in line 3 of procedure FindNextBrick. Thus, this happened during shifting in line 7. Since during shifting, after moving each brick, the robot makes one step forward along $\varphi(\mathcal{W})$, c does not belong to the first segment of $\varphi(\mathcal{W})$. By the definition of a search walk, c belongs to the second segment S of $\varphi(\mathcal{W})$, and c is the second cell in S. By Lemma 5.3, the robot correctly executed the 'if' instruction in line 3 of procedure ReturnToMarker. If S is of length 2, then the robot arrives at c in line 5 and makes a turn in line 4, and consequently the switch does not occur at c. If S is of length larger than 2, then the robot makes two moves forward along S in line 5: the first of these moves brings the robot to c and thus again no switch occurs when the robot is at c. Hence we have proved that no lost component is created while backtracking on $\mathcal{W}$.

We finally prove that the robot is at the marker at the end of procedure ReturnToMarker. By Lemma 5.5, the robot traverses $\varphi(W)$ in line 7 of this procedure. Thus, it becomes located at its end, which is the beginning of $\mathcal{W}$, and hence it can correctly check that it is at distance 4 from the marker, by the definition of switch-traversal. More precisely, this is guaranteed by condition (S1′) that terminates $\varphi(\mathcal{W})$. Then the robot gets to the marker executing line 8 of procedure ReturnToMarker.

□

5.3 Analysis of the Rough Disc Construction

The following lemma shows that procedure Sweep does not create lost components and makes sufficient room around the rough disc. Moreover, the cost of each call of the procedure is $O(s)$. The technical proof essentially follows the actions of procedure Sweep, proving that each action is correctly defined. To summarize the proof, note that the correctness of detection of any full cell at a bounded distance from the current rough disc D comes from an observation that the robot can make a full traversal around D by starting and stopping at the marker. The same holds for checking if there are any free components left. To analyze the overall time spent in procedure Sweep, note that each walk around the rough disc takes time $O(s)$ and there are $O(n)$ such events. Thus, it remains to count the total number of steps done in line 7—we call each such traversal a *sweep-walk*. We then consider two cases. Such a sweep-walk can be short, i.e., of fixed length, and it occurs mostly due to shifting (such a shifting occurs next to the gap and thus moves bricks into the gap) and thus we may get $O(s)$ such short sweep-walks in each iteration of the main loop of algorithm Nest, giving $O(sn)$ robot's moves in total. On the other hand, we argue that long

walks can also happen as a result of shifting but only one per an iteration of the main loop, thus giving $O(n)$ such events in total. Finally note that procedure Sweep also removes bricks from the gap that have not been placed there as a result of executing procedures FindNextBrick and ReturnToMarker but were originally present there. But there are at most n such bricks and the removal of each requires following a path of length $O(s)$, giving $O(sn)$ steps in total.

Lemma 5.7 *Suppose that the robot is at the marker and that there are no lost components. An execution of procedure Sweep takes time $O(s)$ and results in a strongly structured shape.*

Proof We first prove that the shape is structured after executing the procedure. The robot can correctly go to the nearest cell of the rough disc D in line 2. Indeed, in a call to procedure Sweep in line 5 of algorithm Nest this is because $|D| = 1$ and D is at distance 2 from the robot. In each call made in procedure ExtendRoughDisc this is because the robot is at the marker and, by Lemma 5.2(iii), the rough disc and the marker form two separate components. While traversing the border of the rough disc in line 3, the robot iterates over each cell c of the gap in all executions of the 'for' loop in line 4. Since the distance between the current location $c' \in D$ and c is at most 7, the robot can correctly execute instructions in line 6. In line 7, the robot finds the first empty cell in the direction d away from D. Then, in line 8, the robot drops the brick at this cell and returns to c'. We argue that the latter is feasible. This is done by first finding the closest empty cell e in the direction opposite to d. This cell is at distance at most 7 from the rough disc, due to the condition used in line 7. Thus, the robot can remember the relative position of e with respect to c' because the distance between them is bounded. The latter is due to the selected direction d. Indeed, the fact that direction d is away from D implies that the prefix of the walk in line 7 consisting of cells at distance at most 7 from c' is of bounded length.

The above proves that each cell at distance 7 from the rough disc, except the marker, is empty upon completion of the traversal in line 3. Since the 'for' loop in line 4 does not select cells at distance larger than 7 from the rough disc, the rough disc has a gap of width 7, i.e., the shape satisfies condition (F2). Moreover, this implies that no lost component is created during the traversal in line 3 because each brick dropped in line 8 is placed next to a full cell (see line 7).

Checking if there exists a free component $\mathcal{C}$ in line 9 can be done by traversing the border of the rough disc because, by assumption, there are no lost components. By the same argument, the robot can relocate the marker in line 10 of procedure Sweep. This also ensures that the shape satisfies condition (F1). Thus, the shape is structured, i.e., it satisfies condition (T1).

By assumption, there are no lost components, and each brick dropped in line 8 is placed next to a full cell. Thus, no lost component is created during procedure Sweep, which implies that the shape satisfies condition (T2). According to line 11 of procedure Sweep, upon its completion the robot is at the marker, which ensures condition (T3). Thus, the shape is strongly structured as required.

Now we analyze the time needed to perform an execution of procedure Sweep. For each cell c selected in the 'for' loop in line 4, all the following steps take time

$O(1)$, possibly except for the walk performed in line 7 and the return from it in line 8. The cells traversed in line 7 are called a *sweep-walk*. Thus, to estimate the execution time of procedure Sweep, we estimate the total length of all such walks that occur during the execution.

The call in line 5 in algorithm Nest takes time $O(s)$ because, for the initial rough disc of size 1, $O(1)$ cells need to be moved to ensure property (F2) of the shape, and for each such cell, the sweep-walk is of length $O(s)$.

We now consider one execution of procedure Sweep called in procedure ExtendRoughDisc. In order to analyze the time of executing this call, we divide sweep-walks depending on their length. Let W_2 be all sweep-walks of length at most 2 and let W_* be all the remaining sweep-walks. A sweep-walk X is traversed by the robot because condition (F2) is violated, i.e., there is a full cell c at distance less than 7 from the rough disc. We consider two cases.

In the first case, some brick has been dropped at c in the iteration of the 'while' loop of algorithm Nest in which the considered call to procedure Sweep occurs. The event of dropping the brick occurred after the previous call to procedure Sweep because this previous call, as we have proved above, left the shape structured. Thus, this occurred during an execution of procedure FindNextBrick or procedure ReturnToMarker. The cell c does not become full as a result of a switch because, by Lemma 5.6, all switch-components are reconnected with the component in which the agent is performing a return switch-traversal in procedure ReturnToMarker. Thus, c becomes full during shifting. Let S be the last segment traversed by the robot during the execution of procedure FindNextBrick. Note that the brick at c has been moved from a cell c' of S during shifting. The cell c is at distance 2 from c'. If the walk X is perpendicular to S, then it has 2 cells because the neighbor of c in S is empty by the definition of shifting. If the walk X is parallel to S, then c is the single cell, among those filled during shifting, that results in a sweep-walk, due to the direction away from D in procedure Sweep. Thus, in this first case, we have $O(s)$ sweep-walks in W_2 and at most one sweep-walk in W_*.

In the second case, the cell c has been full in the iteration of the 'while' loop of algorithm Nest in which the considered call to procedure Sweep occurs. Thus, this cell results in a sweep-walk because the robot extended the rough disc. However, for each such extension, there are $O(1)$ such cells c. Hence, in the second case we have $O(1)$ sweep-walks in W_2 and $O(1)$ sweep-walks in W_*.

Note that the length of a sweep-walk in W_* is $O(\sqrt{n} + s) = O(s)$ because the span of the final rough disc is $O(\sqrt{n})$. Thus, in each iteration of the 'while' loop of algorithm Nest, the total length of all sweep-walks is $O(s)$. □

Whenever procedure ExtendRoughDisc is called in our algorithm, the robot is at the marker. The following lemma says that it can correctly execute the procedure.

Lemma 5.8 *If the robot is at the marker, carries a brick, and there are no lost components, then the robot can correctly execute procedure ExtendRoughDisc in time* $O(s)$.

Proof In order to execute the instruction in line 1 of procedure ExtendRoughDisc, it is enough for the robot to remember whether the current rough disc is a disc. Adding a brick to the rough disc requires performing one traversal along its border to find the correct location for the brick. This takes time $O(s)$ because the size of the border of a rough disc is $O(\sqrt{n})$ which is $O(s)$. By Lemma 5.7, sweeping in line 2 can be correctly executed because the robot is at the marker when it is called and, by assumption, there are no lost components. Moreover, the time needed to execute procedure Sweep is $O(s)$. □

We are now able to prove our main result, i.e., Theorem 4.1.

Proof of Theorem 4.1 We argue by induction on the number of iterations of the 'while' loop in algorithm Nest that at the beginning of the i-th iteration, for $i \geq 1$, the shape is strongly structured. The claim holds for $i = 1$ because, by Lemma 5.7, the call to procedure Sweep in line 5 results in a shape that is structured. Assuming that the assertion holds for some $i \geq 1$, consider the $(i + 1)$-th iteration. By Lemma 5.6, after executing procedures FindNextBrick and ReturnToMarker in the $(i + 1)$-th iteration, there are no lost components, and the robot is at the marker. We argue that the robot is carrying a brick. If procedure FindNextBrick did not perform shifting then (see line 5) the robot is carrying a brick at the end of its execution due to Lemma 5.2(i). If shifting has been performed, then the robot is carrying a brick at the end of shifting, which is the end of the execution of procedure FindNextBrick (see line 7), due to Lemma 5.1(iii). Note that if the robot performing the return switch-traversal in procedure ReturnToMarker is carrying a brick prior to a switch, then it is carrying a brick upon completion of the switch. Thus, by an inductive argument we can state that the fact that the robot is carrying a brick at the beginning of the return switch-traversal implies that the robot is carrying a brick at the end of it. Thus in particular, the robot is carrying a brick when procedure ExtendRoughDisc is called in line 9 of algorithm Nest. This implies that the assumptions of Lemma 5.8 are satisfied. By this lemma, the execution of procedure ExtendRoughDisc extends the rough disc by one brick. The call to procedure Sweep in line 2 of procedure ExtendRoughDisc results in a shape that is strongly structured, in view of Lemma 5.7. This completes the inductive argument for the $(i + 1)$-th iteration. Thus we showed that, at the beginning of each iteration of the 'while' loop in algorithm Nest the shape is strongly structured.

The above implies that there will be $n - 2$ iterations of the 'while' loop in algorithm Nest and, upon the exit from the loop, there are only two components: the rough disc of size $n - 1$ and the marker. By Proposition 3.1, adding the brick from the marker in line 10 to the rough disc results in a nest.

By the definition of a search walk, the length of any search walk is $O(s)$. The time needed to execute the two procedures that traverse the search walk and backtrack on it is linear in the length of the walk. Using Lemmas 5.7 and 5.8, the overall time of algorithm Nest is $O(sn)$. By Proposition 3.2, this time is worst-case optimal. □

6 Conclusion

We designed a finite deterministic automaton that builds the most compact structure starting from any connected shape of bricks, and does it in optimal time. An interesting problem yielded by our research is to characterize the classes of target structures that can be built by a single automaton, starting from any connected shape of bricks in the grid. Another problem is that of how the building task parallelizes, i.e., how much time many automata use to build some structure.

References

1. Akitaya, H.A., Arkin, E.M., Damian, M., Demaine, E.D., Dujmovic, V., Flatland, R.Y., Korman, M., Palop, B., Parada, I., van Renssen, A., Sacristán, V.: Universal reconfiguration of facet-connected modular robots by pivots: the O(1) musketeers. In: 27th Annual European Symposium on Algorithms, ESA 2019, September 9–11, 2019, Munich/Garching, Germany, pp. 3:1–3:14 (2019)
2. Albers, S., Henzinger, M.R.: Exploring unknown environments. SIAM J. Comput. **29**(4), 1164–1188 (2000)
3. Awerbuch, B., Betke, M., Rivest, R.L., Singh, M.: Piecemeal graph exploration by a mobile robot. Inf. Comput. **152**(2), 155–172 (1999)
4. Bender, M.A., Fernández, A., Ron, D., Sahai, A., Vadhan, S.P.: The power of a pebble: exploring and mapping directed graphs. Inf. Comput. **176**(1), 1–21 (2002)
5. Bender, M.A., Slonim, D.K.: The power of team exploration: two robots can learn unlabeled directed graphs. In: 35th Annual Symposium on Foundations of Computer Science (FOCS), pp. 75–85 (1994)
6. Betke, M., Rivest, R.L., Singh, M.: Piecemeal learning of an unknown environment. Mach. Learn. **18**(2–3), 231–254 (1995)
7. Blum, M., Kozen, D.: On the power of the compass (or, why mazes are easier to search than graphs). In: 19th Annual Symposium on Foundations of Computer Science (FOCS), pp. 132–142 (1978)
8. Brandt, S., Uitto, J., Wattenhofer, R.: A tight lower bound for semi-synchronous collaborative grid exploration. In: 32nd International Symposium on Distributed Computing (DISC), pp. 13:1–13:17 (2018)
9. Budach, L.: Automata and labyrinths. Math. Nachr. **86**, 195–282 (1978)
10. Chalopin, J., Das, S., Kosowski, A.: Constructing a map of an anonymous graph: applications of universal sequences. In: 14th International Conference on Principles of Distributed Systems (OPODIS), pp. 119–134 (2010)
11. Cohen, L., Emek, Y., Louidor, O., Uitto, J.: Exploring an infinite space with finite memory scouts. In: 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 207–224 (2017)

12. Das, S., Flocchini, P., Santoro, N., Yamashita, M.: On the computational power of oblivious robots: forming a series of geometric patterns. In: 29th Annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 267–276 (2010)
13. Daymude, J.J., Derakhshandeh, Z., Gmyr, R., Porter, A., Richa, A.W., Scheideler, C., Strothmann, T.: On the runtime of universal coating for programmable matter. Nat. Comput. **17**(1), 81–96 (2018)
14. Daymude, J.J., Gmyr, R., Hinnenthal, K., Kostitsyna, I., Scheideler, C., Richa, A.W.: Convex hull formation for programmable matter. CoRR, abs/1805.06149 (2018)
15. Daymude, J.J., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Improved leader election for self-organizing programmable matter. In: Algorithms for Sensor Systems—13th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2017, Vienna, Austria, September 7-8, 2017, Revised Selected Papers, pp. 127–140 (2017)
16. Daymude, J.J., Hinnenthal, K., Richa, A.W., Scheideler, C.: Computing by Programmable Particles, pp. 615–681. Springer, Cham (2019)
17. Deng, X., Papadimitriou, C.H.: Exploring an unknown graph. J. Graph Theory **32**(3), 265–297 (1999)
18. Derakhshandeh, Z., Dolev, S., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Brief announcement: amoebot—a new model for programmable matter. In: 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14, Prague, Czech Republic—June 23–25, 2014, pp. 220–222 (2014)
19. Derakhshandeh, Z., Gmyr, R., Richa, A.W., Scheideler, C., Strothmann, T.: Universal shape formation for programmable matter. In: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2016, Asilomar State Beach/Pacific Grove, CA, USA, July 11–13, 2016, pp. 289–299 (2016)
20. Di Luna, G.A., Flocchini, P., Santoro, N., Viglietta, G., Yamauchi, Y.: Shape formation by programmable particles. Distrib. Comput. **33**(1), 69–101 (2020)
21. Dieudonné, Y., Petit, F., Villain, V.: Leader election problem versus pattern formation problem. In: 24th International Symposium on Distributed Computing (DISC), pp. 267–281 (2010)
22. Diks, K., Fraigniaud, P., Kranakis, E., Pelc, A.: Tree exploration with little memory. J. Algorithms **51**(1), 38–63 (2004)
23. Dudek, G., Jenkin, M., Milios, E.E., Wilkes, D.: Robotic exploration as graph construction. IEEE Trans. Robot. Autom. **7**(6), 859–865 (1991)
24. Duncan, C.A., Kobourov, S.G., Anil Kumar, V.S.: Optimal constrained graph exploration. ACM Trans. Algorithms **2**(3), 380–402 (2006)
25. Emek, Y., Langner, T., Stolz, D., Uitto, J., Wattenhofer, R.: How many ants does it take to find the food? Theor. Comput. Sci. **608**, 255–267 (2015)
26. Fekete, S.P., Gmyr, R., Hugo, S., Keldenich, P., Scheffer, C., Schmidt, A.: Cadbots: algorithmic aspects of manipulating programmable matter with finite automata. arXiv, arXiv:1810.06360 (2018)
27. Fekete, S.P., Niehs, E., Scheffer, C., Schmidt, A.: Connected assembly and reconfiguration by finite automata. arXiv, arXiv:1909.03880 (2019)
28. Fraigniaud, P., Ilcinkas, D.: Digraphs exploration with little memory. In: 21st Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp. 246–257 (2004)
29. Fraigniaud, P., Ilcinkas, D., Peer, G., Pelc, A., Peleg, D.: Graph exploration by a finite automaton. Theor. Comput. Sci. **345**(2–3), 331–344 (2005)
30. 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 2018, August 27–31, 2018, Liverpool, UK, pp. 52:1–52:15 (2018)
31. Gmyr, R., Hinnenthal, K., Kostitsyna, I., Kuhn, F., Rudolph, D., Scheideler, C., Strothmann, T.: Forming tile shapes with simple robots. In: 24th International Conference on Computing and Molecular Programming (DNA), pp. 122–138 (2018)
32. Hemmerling, A.: Labyrinth Problems: Labyrinth-Searching Abilities of Automata. Teubner-Texte zur Mathematik, p. 114. B. G. Teubner Verlagsgesellschaft, Leipzig (1989)
33. Hoffmann, F.: One pebble does not suffice to search plane labyrinths. In Fundamentals of Computation Theory (FCT), pp. 433–444 (1981)
34. Kozen, D.: Automata and planar graphs. In: Fundamentals of Computation Theory (FCT), pp. 243–254 (1979)
35. Langner, T., Keller, B., Uitto, J., Wattenhofer, R.: Overcoming obstacles with ants. In: 19th International Conference on Principles of Distributed Systems (OPODIS), pp. 9:1–9:17 (2015)
36. Michail, O., Skretas, G., Spirakis, P.G.: On the transformation capability of feasible mechanisms for programmable matter. J. Comput. Syst. Sci. **102**, 18–39 (2019)

37. Panaite, P., Pelc, A.: Exploring unknown undirected graphs. J. Algorithms **33**(2), 281–295 (1999)
38. Rao, N., Kareti, S., Shi, W., Iyengar, S.: Robot navigation in unknown terrains: introductory survey of non-heuristic algorithms. Technical report ORNL/TM-12410, Oak Ridge National Lab. (1993)
39. Rollik, H.-A.: Automaten in planaren graphen. Acta Inform. **13**, 287–298 (1980)
40. Suzuki, I., Yamashita, M.: Distributed anonymous mobile robots: formation of geometric patterns. SIAM J. Comput. **28**(4), 1347–1363 (1999)