Building a Nest by an Automaton

Jurek Czyzowicz
Département d'informatique, Université du Québec en Outaouais, Canada
jurek@uqo.ca

Dariusz Dereniowski
Faculty of Electronics, Telecommunications and Informatics,
Gdańsk University of Technology, Poland
deren@eti.pg.edu.pl

Andrzej Pelc
Département d'informatique, Université du Québec en Outaouais, Canada
pelc@uqo.ca

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 *field*, is initially connected. The (Manhattan) distance between the farthest cells of the field 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 field 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 field, in time $O(sz)$, where s is the span of the initial field and z is the number of bricks. We show that this complexity is optimal.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases finite automaton, plane, grid, construction task, brick, mobile agent, robot

Digital Object Identifier 10.4230/LIPIcs.ESA.2019.35

Related Version A full version of the paper is available at `https://arxiv.org/abs/1904.10850`.

Funding *Jurek Czyzowicz*: Supported in part by NSERC discovery grant.
Dariusz Dereniowski: Partially supported by National Science Centre (Poland) grant number 2015/17/B/ST6/01887.
Andrzej 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.

1 Introduction

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 *field*, is initially connected. The (Manhattan) distance between the farthest cells of the field is called its *span*. Notice that the span of any current field may be much smaller than its diameter as a subgraph of the grid. In fact, this diameter may be sometimes undefined, if the field becomes disconnected. The robot starts at a full cell. It

© Jurek Czyzowicz, Dariusz Dereniowski, and Andrzej Pelc;
licensed under Creative Commons License CC-BY

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 field, 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 field 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.

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 field, in time $O(sz)$, where s is the span of the initial field and z is the number of bricks. The time is defined as the number of moves of the robot. We show that this complexity is optimal.

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. This approach ensures the optimal complexity. (In order to show where the problem is, we also sketch a much simpler algorithm that uses another approach but has significantly larger complexity). There are two major difficulties to carry out this plan. The first is that the span of the initial field 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. The second problem is that, while we temporarily disconnect the field during the execution of the algorithm, special care has to be taken so that the connected components of intermediary fields be close to each other, to prevent the robot from getting lost in large empty spaces.

To the best of our knowledge, the task of constructing structures from available material using an automaton, has never been studied before in the algorithmic setting. It is interesting to compare this task to that of exploration of mazes by automata, that is a classic topic with over 50 years of history (see the section "Related work"). It follows from the result of Budach [9] (translated to our terminology) that if an automaton can only navigate in the field without moving bricks then it cannot explore all connected fields, even without the stop requirement, i.e., it cannot even see all bricks. 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.

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 *field*. Any maximal connected subgraph of the current field 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 field is called its *size*, and the distance between two farthest cells of a field is called its *span*. A nest of size z is a field that has the minimum span among all fields of size z.

We are given a mobile entity (robot) starting in some cell of the initial field and traveling in the grid. The robot has a priori no knowledge of the field, of its size or of its span. The robot is formalized as a finite deterministic Mealy automaton $\mathcal{R} = (X, Y, \S, \delta, \lambda, S_0, S_{\mathrm{f}})$. $X = \{e, f\} \times \{l, h\}$ 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 S. Given this state and the input x, the robot outputs the symbol $\lambda(x, S) \in \{N, E, S, W\} \times \{e, f\} \times \{l, h\}$ 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:

$\lambda(S, e, l)$ must be $(\cdot, e, l)$

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

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

$\lambda(S, f, h)$ must be $(\cdot, f, h)$

Seeing the input symbol x and being in a current state 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 $\delta(x, S)$. 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}.

Related work

Problems concerning exploration and navigation performed by mobile agents or robots in an unknown environment have been studied for many years (cf. [7, 21, 26]). 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 [12, 14, 28]. Robots, 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 [1, 3, 4, 13, 19] 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 [2, 5, 9, 15, 16, 17, 25, 27] 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., [13, 17, 25]), or it is assumed that nodes are anonymous (as in, e.g., [3, 4, 9, 10, 27]). 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., [25]), 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., [15, 20, 19]). 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., [6, 9, 15, 20, 23, 27]. In particular, it was shown in [27] 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 fields in this paper). Hoffmann [22] proved that one pebble does not help to do it. By contrast, Blum and Kozen [6] 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 [20].

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 [18] 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 [24].

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. To the best of our knowledge, such construction tasks performed by automata have never been studied previously in the algorithmic setting.

2 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 is called *special*, if it is either a leaf, or has at least two full cells adjacent to it, sharing a 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. Clearly the robot can remember its orientation, using its states. We refer to cardinal directions with respect to this current 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 $\prec$ on the set of all cells. For cells $c = (x, y)$ and $c' = (x', y')$, $c \prec c'$ holds if and only if either $y < y'$, or $y = y'$ and $x \le x'$. We denote by $|S|$ the number of cells in a sequence or a set S of cells.

We define a *disc* of radius $r \ge 0$ 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 $z_r = 2(1+3+5+\cdots+(2r-1))+(2r+1) = 2r^2 + 2r + 1$ cells and has span $2r$. A *rough disc* of size z, where $z_r \le z < z_{r+1}$ is defined as follows. If $z = z_r$, 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 $z - z_r$ cells belonging to F, starting from the North neighbor of the East-most cell of D and going counterclockwise. If $z_r < z \le z_r + 2r + 2$ then the rough disc of size z has span $2r + 1$ and if $z_r + 2r + 2 < z < z_{r+1}$ then the rough disc of size z has span $2r + 2$, the same as the disc of radius $r + 1$ that has size z_{r+1}.

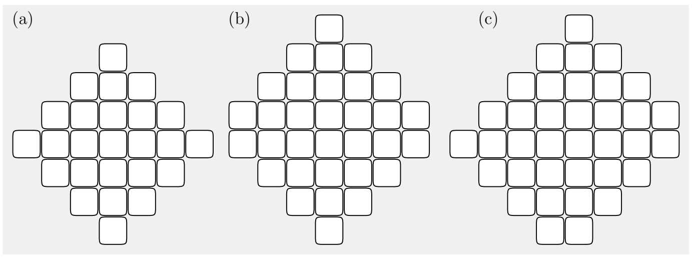

Figure 1 (a) disc of size z_r for $r = 3$; (b) a rough disc of size $z_r + 7$ and span $2r + 1$, $r = 3$; (c) a rough disc of size $z_r + 11$ and span $2r + 2$, $r = 3$.

The proofs of the next two propositions are omitted due to space limitation.

▶ **Proposition 1.** *Any rough disc is a nest.*

The nests built by our automaton will be rough discs. The following proposition shows that the complexity of our nest-building algorithm is optimal, regardless of the relation between the size of the initial field and its span (recall that, by definition, the span must be smaller than the size z and it must be in $\Omega(\sqrt{z})$). 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 field.

▶ **Proposition 2.** *Let $s' < z$ be positive integers such that $s' \in \Omega(\sqrt{z})$. There exists an initial field of size z and span $s \in \Theta(s')$, such that any algorithm that builds a nest starting from this field must use time $\Omega(sz)$.*

3 The algorithm

The robot will move bricks from the original field 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 pseudo-code 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 field, 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 come from our desire to keep the complexity of the algorithm optimal, i.e., $O(sz)$. If complexity were not an issue, the following much simpler algorithm would be sufficient. The robot first builds a horizontal line at the level of a South-most cell of the initial field, by gradually squeezing down the field, keeping it connected at all times. Then it transforms the line into a nest. The squeezing down can be performed by iteratively repeating the following steps. First, the robot makes sure that it is not on the lowest level (if the line is not yet constructed, this can be done by finding a full cell with a full South neighbor). Then the robot goes to some North-West extremity of the field, i.e., to any full cell that has empty cells to the North and to the West of it. Then it picks the brick from this cell and drops it one level down, ensuring the connectivity. This can be done by first moving one level down and then iteratively going West until an empty cell is encountered, where the robot drops the brick. When the field is squeezed down to a line, the robot will recognize this and easily transform it into a nest.

This idea potentially requires time $\Theta(z)$ to lower a brick by one level. Since there are z bricks possibly on $\Theta(s)$ levels, the entire time would be $\Theta(sz^2)$ in the worst case, which is suboptimal. Thus we proceed with the detailed description of the optimal algorithm Nest whose high-level idea was described before.

3.1 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 field. 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: 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 (which can be done with stop, using 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

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 $c \notin D \cup \{M\}$ 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 $\mathcal{C}$ **then**
10     pick the brick from marker and place it at distance 3 from the rough disc and at
         distance 4 from $\mathcal{C}$, creating a new marker
11 go to the marker
```

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 priority North, East, South, West.

3.2 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. 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 below. 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 $O(sz)$ 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 later. 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, we will describe how the desired brick can be found at the end of $\mathcal{W}$.

3.2.1 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.

(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).

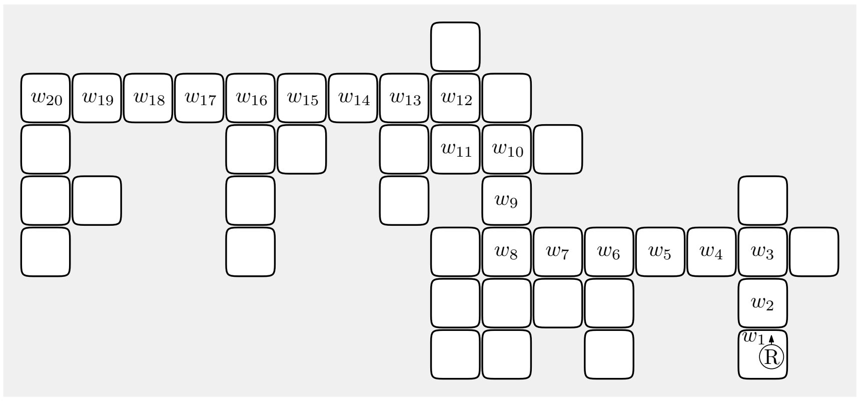

Figure 2 An example of a search walk $\mathcal{W} = (w_1, \ldots, w_{20})$ that is constructed by the robot initially located at w_1 and facing North. This search walk is left-oriented, and has three left-free segments $S_1 = (w_1, w_2, w_3)$, $S_3 = (w_8, w_9, w_{10})$, $S_5 = (w_{11}, w_{12})$ and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

3.2.2 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 field 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 field 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).

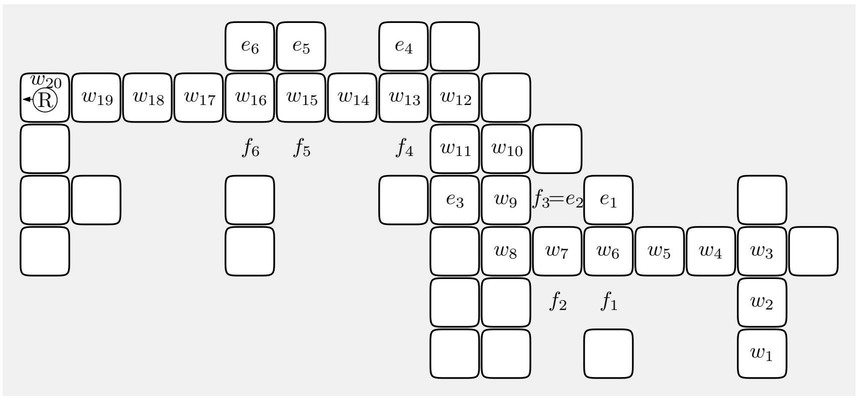

Figure 3 The field from Figure 2 after switch-traversal of the search walk from Figure 2. The cells $w_6, w_7, w_9, w_{13}, w_{15}$ and w_{16} are the break points at which the robot moves a brick from a cell f_i to e_i for $i \in \{1, \dots, 6\}$. Note that a brick is moved from f_2 to e_2 while traversing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

3.2.3 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 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 field 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 (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.

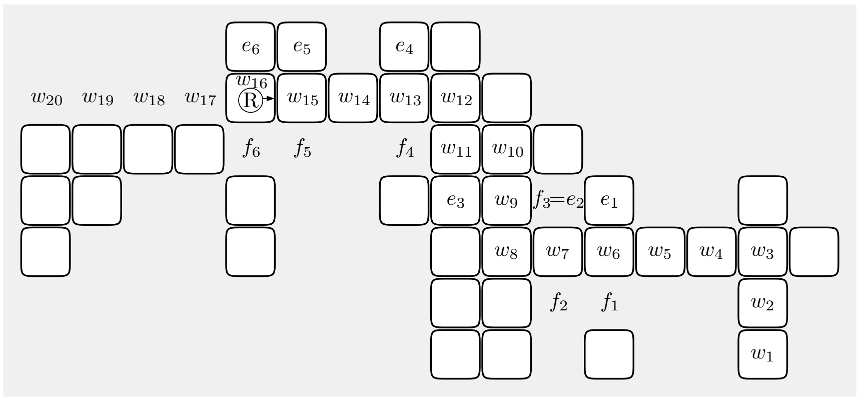

Figure 4 The field from Fig. 3 at the end of shifting. The shifting ends with a right-free segment, at the cell w_{16} because it has a full neighbor, the cell e_6. There is one fewer brick than in Fig. 3 and this is the free brick obtained and carried by the robot.

Procedure FindNextBrick finding a free brick.

```
1 $\mathcal{W}$ := 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 $\mathcal{W}$
4 if the robot is at a leaf then
5     pick the brick
6 else
7     perform shifting
```

3.3 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 $\mathcal{W}$ := the search walk traversed in the last call to FindNextBrick
2 let $S_1, \dots, S_l$ be the segments in $\mathcal{W}$
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 $\varphi(\mathcal{W})$
8 go to the marker
```

3.4 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 $D \cup \{e\}$ 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 field.

```
1 if the span of the field is at most 2 then
2     exit ▷ the field 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 $\mathcal{C}$ 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 $D \cup \{e\}$ is a rough disc
```

The following is the main result of this paper. Its proof is omitted due to space limitation.

▶ **Theorem 3.** *Algorithm Nest builds a nest starting from any connected field of size z and span s in time $O(sz)$. This time is worst-case optimal.*

4 Conclusion

We designed a finite deterministic automaton that builds the most compact structure starting from any connected field 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 field 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 Susanne Albers and Monika Rauch Henzinger. Exploring Unknown Environments. *SIAM J. Comput.*, 29(4):1164–1188, 2000. `doi:10.1137/S009753979732428X`.

2 Baruch Awerbuch, Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal Graph Exploration by a Mobile Robot. *Inf. Comput.*, 152(2):155–172, 1999. `doi:10.1006/inco.1999.2795`.

3 Michael A. Bender, Antonio Fernández, Dana Ron, Amit Sahai, and Salil P. Vadhan. The Power of a Pebble: Exploring and Mapping Directed Graphs. *Inf. Comput.*, 176(1):1–21, 2002. `doi:10.1006/inco.2001.3081`.

4 Michael A. Bender and Donna K. Slonim. The Power of Team Exploration: Two Robots Can Learn Unlabeled Directed Graphs. In *35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 75–85, 1994. `doi:10.1109/SFCS.1994.365703`.

5 Margrit Betke, Ronald L. Rivest, and Mona Singh. Piecemeal Learning of an Unknown Environment. *Machine Learning*, 18(2-3):231–254, 1995. `doi:10.1007/BF00993411`.

6 Manuel Blum and Dexter Kozen. 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)*, pages 132–142, 1978. `doi:10.1109/SFCS.1978.30`.

7 Manuel Blum and William J. Sakoda. On the Capability of Finite Automata in 2 and 3 Dimensional Space. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 147–161, 1977. `doi:10.1109/SFCS.1977.20`.

8 Sebastian Brandt, Jara Uitto, and Roger Wattenhofer. A Tight Lower Bound for Semi-Synchronous Collaborative Grid Exploration. In *32nd International Symposium on Distributed Computing (DISC)*, pages 13:1–13:17, 2018. `doi:10.4230/LIPIcs.DISC.2018.13`.

9 Lothar Budach. Automata and labyrinths. *Math. Nachrichten*, 86:195–282, 1978.

10 Jérémie Chalopin, Shantanu Das, and Adrian Kosowski. Constructing a Map of an Anonymous Graph: Applications of Universal Sequences. In *14th International Conference on Principles of Distributed Systems (OPODIS)*, pages 119–134, 2010. `doi:10.1007/978-3-642-17653-1_10`.

11 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an Infinite Space with Finite Memory Scouts. In *28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 207–224, 2017. `doi:10.1137/1.9781611974782.14`.

12 Shantanu Das, Paola Flocchini, Nicola Santoro, and Masafumi Yamashita. On the computational power of oblivious robots: forming a series of geometric patterns. In *29th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–276, 2010. `doi:10.1145/1835698.1835761`.

13 Xiaotie Deng and Christos H. Papadimitriou. Exploring an unknown graph. *Journal of Graph Theory*, 32(3):265–297, 1999. `doi:10.1002/(SICI)1097-0118(199911)32:3<265::AID-JGT6>3.0.CO;2-8`.

14 Yoann Dieudonné, Franck Petit, and Vincent Villain. Leader Election Problem versus Pattern Formation Problem. In *24th International Symposium on Distributed Computing (DISC)*, pages 267–281, 2010. doi:10.1007/978-3-642-15763-9_26.

15 Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. Tree exploration with little memory. *J. Algorithms*, 51(1):38–63, 2004. doi:10.1016/j.jalgor.2003.10.002.

16 Gregory Dudek, Michael Jenkin, Evangelos E. Milios, and David Wilkes. Robotic exploration as graph construction. *IEEE Trans. Robotics and Automation*, 7(6):859–865, 1991. doi:10.1109/70.105395.

17 Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal constrained graph exploration. *ACM Trans. Algorithms*, 2(3):380–402, 2006. doi:10.1145/1159892.1159897.

18 Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How many ants does it take to find the food? *Theor. Comput. Sci.*, 608:255–267, 2015. doi:10.1016/j.tcs.2015.05.054.

19 Pierre Fraigniaud and David Ilcinkas. Digraphs Exploration with Little Memory. In *21st Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 246–257, 2004. doi:10.1007/978-3-540-24749-4_22.

20 Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, 2005. doi:10.1016/j.tcs.2005.07.014.

21 A. Hemmerling. Labyrinth Problems: Labyrinth-Searching Abilities of Automata. *Teubner-Texte zur Mathematik. B. G. Teubner Verlagsgesellschaft, Leipzig*, 114, 1989.

22 Frank Hoffmann. One Pebble Does Not Suffice to Search Plane Labyrinths. In *Fundamentals of Computation Theory (FCT)*, pages 433–444, 1981. doi:10.1007/3-540-10854-8_47.

23 Dexter Kozen. Automata and planar graphs. In *Fundamentals of Computation Theory (FCT)*, pages 243–254, 1979.

24 Tobias Langner, Barbara Keller, Jara Uitto, and Roger Wattenhofer. Overcoming Obstacles with Ants. In *19th International Conference on Principles of Distributed Systems (OPODIS)*, pages 9:1–9:17, 2015. doi:10.4230/LIPIcs.OPODIS.2015.9.

25 Petrisor Panaite and Andrzej Pelc. Exploring Unknown Undirected Graphs. *J. Algorithms*, 33(2):281–295, 1999. doi:10.1006/jagm.1999.1043.

26 N. Rao, S. Kareti, W. Shi, and S. Iyengar. Robot navigation in unknown terrains: Introductory survey of non-heuristic algorithms. Technical Report ORNL/TM-12410, Oak Ridge National Lab., 1993.

27 Hans-Anton Rollik. Automaten in planaren Graphen. *Acta Informatica*, 13:287–298, 1980. doi:10.1007/BF00288647.

28 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM J. Comput.*, 28(4):1347–1363, 1999. doi:10.1137/S009753979628292X.