Overcoming Obstacles with Ants

Barbara Keller, Tobias Langner, Jara Uitto, Roger Wattenhofer

ETH Zürich, Switzerland

Abstract

Consider a group of mobile finite automata, referred to as agents, located in the origin of an infinite grid. The grid is occupied by *obstacles*, i.e., sets of cells that can not be entered by the agents. In every step, an agent can sense the states of the co-located agents and is allowed to move to any neighboring cell of the grid not blocked by an obstacle. We assume that the circumference of each obstacle is finite but allow the number of obstacles to be unbounded. The task of the agents is to cooperatively find a treasure, hidden in the grid by an adversary.

In this work, we show how the agents can utilize their simple means of communication and their constant memory to systematically explore the grid and to locate the treasure in finite time. As integral part of the agents' behavior, we present a method that allows a group of six agents to follow a straight line, even if the line is partially obscured by obstacles, and to discover all cells along this line. In total, our search protocol requires nine agents.

1 Introduction

How do ants find that crumb of chocolate dropped on the kitchen floor? And how do they navigate through that huge Lego castle built by the children to get to the crumb? General knowledge is that this amazing achievement can be explained by so-called pheromones, a chemical factor used by ants to mark the terrain. However, as it turns out, many ant species do not use pheromones at all, and instead communicate with their antennae when bumping into each other [20]. So how do they do it – are ants pretty intelligent after all?

In this paper, we model a single ant with a mobile version of a finite state machine. If two ants meet, they can influence their states, no other form of communication is allowed. We show that a small group of nine of our ants will collaboratively be able to find a treasure in an arbitrarily obstructed environment. Our ants use only a constant amount of memory, independent of the distance from the nest to the treasure, and the number and size of the obstacles.

Related Work. Recently, scientists in biology and computing have been flirting with each other. Distributed computing in particular seems to be a valuable tool towards understanding biological phenomena, as both often deal with networks of simple nodes, collaborating by means of minimal communication. Please see the recent survey from Navlahka and Bar-Joseph for more details [18].

Ants in particular have been a focus of interest in the Computer Science community. As an example, Feinerman et. al. modeled the foraging behavior of ants as an exploration problem, where n agents are collaboratively searching the plane and the goal is to find an adversarially hidden treasure [13,14]. Our model is a variant of their model, where the agents are controlled by finite state machines instead of Turing machines. Without obstacles, it was shown that asymptotically there is no penalty when ants are restricted to finite state machines [12]. In the case of an infinite grid without obstacles, it was discovered by Emek et al. that two deterministic finite state machines cannot discover every cell [34]. In the same work, it was also shown that a randomized finite state machine requires infinite time in expectation and that 4 (deterministic) finite state machines are always enough to discover the treasure. Since the unobstructed infinite grid is a special case of our setting, the same lower bounds hold for our problem. However, it seems that introducing obstacles fundamentally changes the picture. In this paper we show that it is still possible to discover the treasure in this more challenging setting and we derive an upper bound for the number of ants required for it.

Our work also has connections to graph exploration, as the problem we are studying is a variant of it. In the general graph exploration setting, the goal is to visit all the nodes or all the edges of a graph starting from any node. In our work the unobstructed cells can be interpreted as nodes and their connections to their neighbors as edges. The task of the ants is to discover all unobstructed cells. Graph exploration has been extensively studied in the literature and the studies can be divided into two settings. One of the settings is to assume that the graphs are directed, i.e., an edge can only be traversed to one direction, not vice versa [1,3,7]. In the other, the edges can be traversed to both directions [2,8,10]. Our work belongs to the second setting, as the ants can move back and forth between neighboring cells.

Furthermore, there are two main types of performance measures regarding graph exploration. The first measure is the time complexity, i.e., how long does it take for the agent(s) to finish the exploration task [19]. The other one is to measure the bit complexity, i.e., how many bits of memory does the agent(s) require to solve the exploration task [15]. Furthermore, the aforementioned graph exploration tasks can be considered with return, stop, or perpetual properties, i.e., whether the agent is required to return to the starting cell, stop the search after finishing, or if the agent is not required to terminate [8, 16]. Note that even though in [16] a finite automaton explores a graph, this automaton is equipped with a memory linear in size of the diameter of the graph. In our work we show that our finite automata only need constant memory to solve the task.

Since we are restricting our underlying graph to $\mathbb{Z}^2$ and the obstacles in our domain essentially block the agents from entering specific cells, our graphs correspond to a concept widely studied in literature called *labyrinths* [4, 9]. Exploration of a labyrinth corresponds to the task of getting as far from the starting point as possible, for any starting point. It was shown by Budach that a single automaton cannot explore every finite labyrinth, where a finite labyrinth has only a finite amount of blocked cells [6]. On the positive side, it is known that every finite labyrinth can be explored by a finite automaton using 4 pebbles and that all co-finite (number of non-blocked cells is finite) labyrinths can be explored with a finite state machine using 2 pebbles [5]. A pebble can be seen as a marker, which can be put down/picked up and moved by the automaton. Finally, Hoffman showed that the problem cannot be solved in neither finite nor co-finite labyrinths by using only 1 pebble [17]. Note that our goal differs from the one of labyrinth exploration, i.e., our goal is to visit all non-blocked cells.

2 Model

We consider the asynchronous version of the ANTS problem variant described in [12], where a set of mobile *agents* search the infinite grid for an adversarially hidden treasure. The agents are controlled by asynchronous finite state machines with a common sense of direction and communicate only with agents sharing the same grid cell.

More formally, we consider a set A of mobile agents that explore $\mathbb{Z}^2$. In the beginning of the execution, all agents are positioned in a designated grid cell referred to as the *origin*; the cell with coordinates $(0, 0) \in \mathbb{Z}^2$. We denote the cells with either x- or y-coordinate being 0 as *north/east/south/west-axis*, depending on their location. The *distance* between two grid cells $(x, y), (x', y') \in \mathbb{Z}^2$ is defined with respect to the ℓ_1 norm (a.k.a. Manhattan distance), that is, $|x - x'| + |y - y'|$. Two cells are called *neighbors* if the distance between them is 1.

The set of cells $B \subset \mathbb{Z}^2$ represents the *blocked* cells, which cannot be entered by an agent. All other cells are called *free*. For simplicity, we assume that B neither contains the origin nor any of the cells within distance at most 3 from the origin. We note that assuming the origin free is necessary and that our protocols can easily be modified to work in an environment without assuming that the nearby cells around the origin are free. This assumption merely allows for a cleaner and more reader friendly initialization of our protocol.

To make the exploration of the grid feasible, we require that the cells in B do not fully enclose any free cell, i.e., that any free cell is reachable from any other free cell by a path of neighboring free cells. The set B induces a set $\mathcal{O}$ of *obstacles*. An obstacle $O \in \mathcal{O}$ is a maximal set of connected cells, where two cells are connected if both their x- and y-coordinates each differ by at most one (diagonally adjacent cells are connected!). We require each obstacle to be of finite size.

All agents are controlled by the same asynchronous *finite automaton* (FA). This means that the individual agent has a constant memory and thus, in general, can not store coordinates in $\mathbb{Z}^2$. Since we design a protocol for a constant number of agents, we allow each agent to run a different individual protocol. This is modeled by assigning to each agent an individual initial state in the shared automaton An agent a positioned in cell $z \in \mathbb{Z}^2$ can communicate with all other agents positioned in cell z at the same time. This communication is quite limited though: agent a merely senses for each state q of the finite state machine, whether there exists at least one agent $a' \neq a$ in cell z whose current state is q. In each step of the execution, agent a positioned in cell $(x, y) \in \mathbb{Z}^2$ can either move to one of the four neighboring cells $(x, y+1), (x, y-1), (x+1, y), (x-1, y)$, or stay put in cell (x, y). The former four *position transitions* are denoted by the corresponding cardinal directions $\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}$, whereas the latter (stationary) position transition is denoted by P. For convenience, we also identify the four directions $\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}$ with the unit vectors in the corresponding directions and, e.g., $z = (x, y) + \mathrm{N} = (x, y+1)$. We point out that the agents have a common sense of orientation, i.e., the cardinal directions are aligned with the corresponding grid axes for every agent in every cell.

The agents operate in an asynchronous environment. Each agent's execution progresses in discrete (asynchronous) steps indexed by the non-negative integers and we denote the time at which agent a completed step $i > 0$ by $t_a(i) > 0$. Following the common practice, we assume that the time stamps $t_a(i)$ are determined by the policy ψ of an adversary that knows the protocol whereas the agents do not have any sense of time.

Formally, the agents' protocol is captured by the 3-tuple $\Pi = \langle Q, s_0^a, \delta \rangle$, where Q is the finite set of *states*; $s_0^a \in Q$ is the *initial state* of agent a; and

$$\delta : Q \times 2^Q \times \{\top, \bot\}^4 \to 2^{Q \times \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}, \mathrm{P}\}}$$

is the *transition function*. At time 0, all agents are positioned at the origin and their FAs are in the respective initial states. Suppose that at time $t_a(i)$, agent a is in state $q \in Q$ and positioned in cell $z \in \mathbb{Z}^2$. Then, the state $q' \in Q$ of a at time $t_a(i+1)$ and its corresponding position transition $\tau \in \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}, \mathrm{P}\}$ are determined by the transition function $\delta(q, Q_a, b) = (q', \tau)$, where $Q_a \subseteq Q$ contains state $p \in Q$ if and only if there exists some (at least one) agent $a' \neq a$ such that a' is in state p and positioned in cell z at time $t_a(i)$, and b is a 4-tuple indicating which of the neighboring cells N/E/S/W are blocked ($\top$) or free ($\bot$). If the transition function dictates that an agent enters a blocked cell, the agent stays put instead. For simplicity, we assume that while the state subset Q_a (input to δ) is determined based on the status of cell z at time $t_a(i)$, the actual application of the transition function δ occurs instantaneously at the end of the step, i.e., agent a is considered to be in state q and positioned in cell z throughout the time interval $[t_a(i), t_a(i+1))$.

The goal is to locate an adversarially hidden *treasure*, i.e., to bring at least one agent to the *free* cell in which the treasure is positioned. The distance to the treasure from the origin is denoted by D.

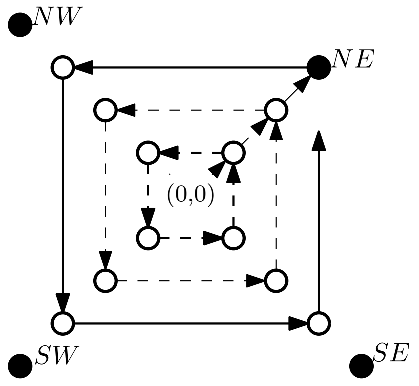


Figure 1: The filled black dots represent the corner agents (N, E, S, W), marking the next spot, where the exploring group should turn counter-clockwise in order to walk a square. The hollow dots represent where the corner agents were in earlier stages. The arrows present the way the exploring group was taking so far.

3 Basic Idea

In order to find the treasure, the agents have to visit every free cell. The high level idea is that the agents walk in growing squares counter-clockwise around the origin. To this end, each agent is given a specific task. An *explorer* explores the plane by walking along squares of increasing sizes, whereas four other agents, called *guides*, mark the four corners of the square that the explorer should walk along. We identify the four guides by the cardinal direction of their respective corner NE, NW, SW, SE. Upon entering a cell with a guide, the explorer accompanies the guide to the correct position for the next square before continuing the search. After updating the position of the last guide, the explorer starts a new search along the next bigger square. We define *square*(d) as the square given by the four corner cells $(d, d), (d, -d), (-d, -d), (-d, d)$.

In the presence of obstacles, the subroutines get more involved. Obstacles can obstruct the path of the explorer or hinder a guide to mark the cell it is supposed to. To solve the former of the aforementioned problems we provide a subroutine that essentially allows the explorer to walk "through" the obstacle. For the second problem we change the conditions for the guides. Instead of marking the corner of the square, a guide has to either mark the correct y-coordinate or the correct x-coordinate, depending on the guide. The NE- and SW-guides mark the y-coordinates of the corners of the square whereas the NW- and SE-guides mark the x-coordinates of said corners (see Figure 2).

Let us describe the new condition for the NE-guide. Consider the NE-guide that is supposed to mark the cell $z = (d, d)$ for some value of d and assume further that z is blocked. Then, the *surrogate cell* for the cell z is given by $z' = (x', d)$ where $x' = \min\{x \mid x \geq d \wedge (x, d) \notin B\}$. Informally, z' is the first free cell with the same y-coordinate as z further away from the origin. As the obstacles are of finite size we can guarantee that such a cell always exists. With this condition, we make sure that the guide is either on the corner (if it is free) or outside of the square on which the explorer is walking.

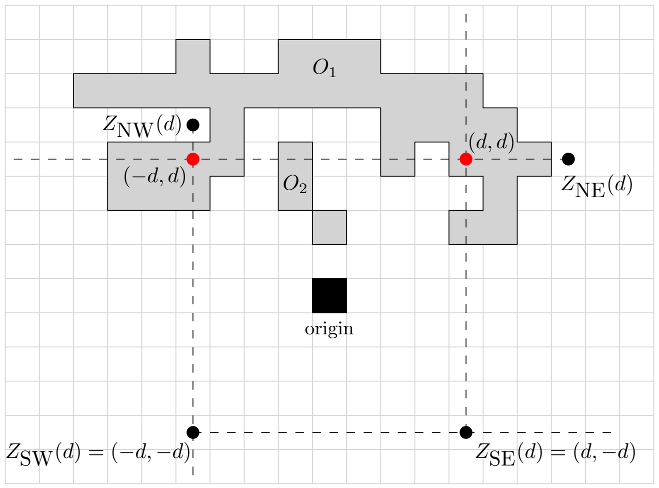


Figure 2: The grey area describes the obstacles O_1, O_2 and the red dots indicate where the NE- and NW-guide would be if there was no obstacle. The black dots indicate the cells, that the guides actually mark. The dashed lines indicate the side of the square that the respective guide is marking and altogether mark the square that the explorer is supposed to walk along.

The condition for the other three guides is analogous. Consider square(d) and a guide responsible for the corner $x \in \{\mathrm{NE}, \mathrm{NW}, \mathrm{SW}, \mathrm{SE}\}$ of said square. Then, we denote by $Z_x(d)$ the cell where this guide will be positioned during the exploration of the square.

$$\begin{aligned}
Z_{\mathrm{NE}}(d) &= (x', d) \text{ where } x' = \min\{x'' \mid x'' \geq d \wedge (x'', d) \notin B\}, \\
Z_{\mathrm{NW}}(d) &= (-d, y') \text{ where } y' = \min\{y'' \mid y'' \geq d \wedge (-d, y'') \notin B\}, \\
Z_{\mathrm{SW}}(d) &= (x', -d) \text{ where } x' = \max\{x'' \mid x'' \leq -d \wedge (x'', -d) \notin B\}, \\
Z_{\mathrm{SE}}(d) &= (d, y') \text{ where } y' = \max\{y'' \mid y'' \leq -d \wedge (d, y'') \notin B\}
\end{aligned}$$

4 Basic Capabilities

Our protocol requires the agents and in particular the explorer to be able to perform various advanced maneuvers. They have to be able to walk along the boundary of an obstacle, memorize their offsets from other cells, be able to find back to a cell they previously occupied, update the position of a guide to the next square, and, most importantly, to virtually walk through an obstacle. In this section we present the basic routines which are then combined in Section 5 to obtain the more complex ones.

4.1 Walking Around an Obstacle

Consider an agent a that currently walks into direction h where h can be N/E/S/W and is called the *heading* of a. We say that a turns right or left as shorthand for a changing its heading to an

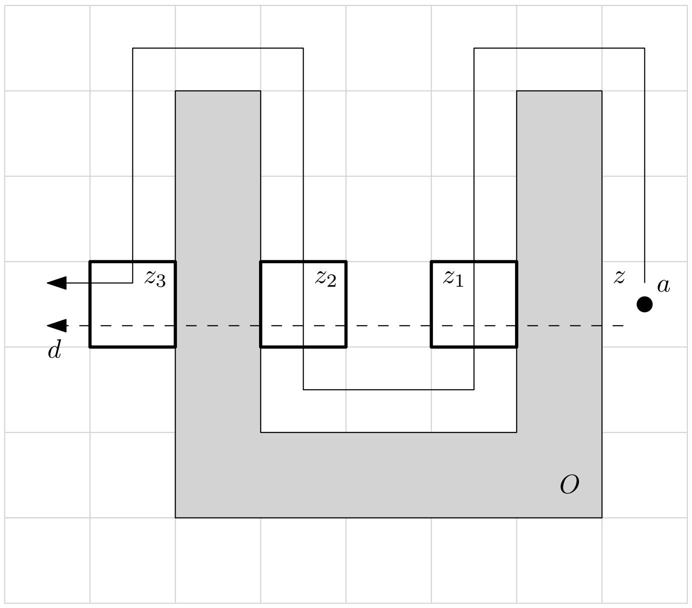


Figure 3: Agent a wants to "walk through" an obstacle in a straight line in direction d, which is accomplished by tracing the boundary of the obstacle along the path p to locate the cell where the straight line exists the obstacle and then continue walking straight.

adjacent cardinal direction. Now suppose that agent a is in cell $z = (x, y)$ and the cell $z + h$ is blocked by the obstacle O that a intends to walk around. In the very first step, a turns right so that the obstacle is on its left side — an invariant that will be maintained during the process of walking around the obstacle. Then, in every following step, a first checks if the cell on the left side with respect to the current heading is blocked. If this is the case, a walks once towards its heading, if possible. In case the cell towards the heading is also blocked, a turns right. In the case that the cell on the left is free, a turns left and walks once towards the new heading. We can verify that this case only occurs if in the previous step, a moved towards its current heading and therefore, the cell on the left was blocked. Therefore, the obstacle will again be on the left side of a in the next step. The details of the method StepCounterClockwise for a single step are given in Procedure 1,the method assumes that agent a is positioned in a cell along the border of the obstacle O and the cell left of a (with respect to h) is blocked by the obstacle O. As the procedure ensures the aforementioned invariant, agent a can execute it repeatedly to traverse the complete boundary of the obstacle.

4.2 Bounded Offset Counter

In this section we explain how the agents can simulate a bounded counter. As the agents have only a constant number of states, they can not remember arbitrarily large numbers, such as how many steps north they went along an obstacle. In order to circumvent this lack of memory, the ants collaboratively implement one or more offset counters. The counter is suitable to

Procedure 1: StepCounterClockwise()

```
Agent $a$ is located in $(x, y)$ and has heading $h$
if cell on left is free then
    turn left
else if $(x, y) + h$ is blocked then
    while $(x, y) + h$ is blocked do
        turn right
move once towards $h$
return $h$
```

memorize offsets to cells while moving along the boundary of an obstacle. The counter provides the basic operations On, Off, IsNull, IsPositive, IsNegative, Increment, and Decrement, which activate/deactivate the counter, allow the agent to determine whether the offset is zero/positive/negative, or to increment/decrement it, respectively. It is important to note that our implementation of the offset counter is only available while the agent is adjacent to an obstacle and while this obstacle stays the same. As soon as the agent moves to a cell that is not adjacent to the obstacle anymore, the value of the counter becomes invalid. Hence, our protocols ensure that the counter is always turned off before leaving an obstacle. Moreover, the value of the counter only works correctly as long as its value is bounded by the circumference of the obstacle. This does not pose a problem, however, as all offsets that the agents need to store in our protocol are bounded appropriately.

We first give an informal description of our implementation and then specify how the basic operations can be implemented. Consider an agent a located in a cell (x, y) adjacent to an obstacle O. Agent a is equipped with the counter c represented by the auxiliary agents a_c, a_b, and a_m called *count agent*, *base agent*, and *messenger agent*, respectively. When the counter is turned off, the auxiliary agents are in the *follow mode*, which implies that they simply follow agent a and do not perform any specific task. When the counter is turned on, the auxiliary agents enter the *counter mode* and perform special tasks. The job of a_b is to mark the cell where the counter has been turned on the last time. Agent a_c's task is to store an offset value v by residing in the cell that is reached when starting in the cell containing a_b and walking $|v|$ cells clockwise along the boundary of the obstacle O. In order to distinguish positive and negative offsets, a_c encodes the sign of v in its states. Agent a_m generally resides in the same cell as agent a and moves to a_c and a_b when the counter is to be changed or read. Either of the basic operations can only be executed when the previous operation has been completed, which is the case when a_m is in the same cell as a.

For the purpose of argumentation, we denote the value represented by counter c as $val(c)$. We remark, however, that this value is not directly accessible to any of the agents.

Operation On(c). When a activates the counter, it signals this to the auxiliary agents using a special state, upon which they enter their respective counter mode states.

Operation Off(c). Agent a_m moves clockwise around the obstacle, instructs a_c and a_b to move along the obstacle to the cell containing a, and finally does the same. The auxiliary agents then enter the follow mode.

Operation IsNull(c). Agent a_m walks clockwise until it locates the cell containing agent a_b. It checks whether agent a_c occupies the same cell and reports this information to agent a.

Operation IsPositive/IsNegative(c). Agent a_m walks clockwise until it locates the cell containing the agent a_c. If the cell also contains agent a_b — the value of the counter is zero — agent a_m reports `false` to a . Otherwise, a_m senses the sign of c through the state of a_c and reports the result to a accordingly.

Operation Increment/Decrement(c). Agent a_m walks clockwise until it locates the cell containing agent a_c. It then instructs a_c to increment/decrement and returns to agent a. Depending on whether the state of a_c corresponds to a positive or negative sign, a_c moves one cell clockwise or counter-clockwise along the obstacle. If a_c resides in the same cell as a_b, it also needs to change its sign state accordingly.

These operations complete the specification of the counter functionality. Please note that all these operations make only use of a constant number of states.

4.3 Combining Offset Counters

The agents in our protocol sometimes employ a constant number of offset counters c_1 to c_k on the same obstacle, where the respective counters are activated in the same cell. This functionality can be provided by having one base agent a_b and one messenger agent a_m and k count agents for the different counters. To ensure that the messenger interacts with the correct count agent, they encode an index in their states such that the messenger agent can distinguish them. Correspondingly, the messenger agent encodes the index of the counter that it is operating on in its state. As only a constant number of offsets are used, this is possible with a constant finite automaton. We distinguish the count agents of different counters by their index as superscript, i.e., a_c^i is the count agent of the counter c_i.

When an agent uses several counters, it has access to two additional operations. Operation LessThan(c_i, c_j) compares the value of two counters and returns a boolean indicating whether $\text{val}(c_i) < \text{val}(c_j)$. The operation Set($c_i, c_j$) sets the value of counter c_i to $\text{val}(c_j)$.

Operation LessThan(c_i, c_j). Agent a_m moves clockwise around the obstacle until it locates the cell containing a_b. Then, a_m walks further clockwise around the obstacle until having located both a_c^i and a_c^j. Based on the signs encoded in the states of a_c^i and a_c^j and the order in which these agents were located, a_m infers the result of the comparison, then returns to a and signals it.

Operation Set(c_i, c_j). Agent a_m walks along the obstacle to the cell containing a_c^i and instructs a_c^i to walk to the cell containing a_c^j, while a_m accompanies a_c^i on its way. When a_c^i enters the cell containing a_c^j, agent a_c^i updates its sign to the sign of a_c^j and agent a_m returns to a to finish the operation.

5 Advanced Procedures

In this section, we combine the basic functionalities described in the previous section into the complex procedures, that eventually constitute our search protocol. The most important functionality is the ability to virtually walk through an obstacle following a horizontal or vertical straight line. The agents do this by locating the closest cell that lies on the straight line through the obstacle and then continue the walk from there. This functionality is realized by the procedures Shift and Probe that will be described next.

5.1 Shifting the Position Along an Obstacle

The procedure Shift(c_x, c_y) allows an agent a positioned in cell $z = (x, y)$ next to the obstacle O and equipped with two counters c_x and c_y to move to the cell $z' = (x + \text{val}(c_x), y + \text{val}(c_y))$, where z' must be also next to O. During the process, agent a continuously updates the counters to reflect the new offsets, so that when a has reached cell z', the values of both counters c_x and c_y are zero. Consequently, both counters are then turned off. Procedure 2 gives a pseudo-code description.

```
Procedure 2: Shift($c_x, c_y$)
  while ¬IsNull($c_x$) ∨ ¬IsNull($c_y$) do
      $h$ ← StepCounterClockwise()
      Increment($c_x$) / Decrement($c_x$) according to $h$
      Increment($c_y$) / Decrement($c_y$) according to $h$
  Off($c_x$); Off($c_y$)
```

5.2 Probing Target Cells

While the procedure StepCounterClockwise allows the agent a to walk around an obstacle O, it still needs to figure out which of the cells visited along the walk is the next free cell t along the straight path through O. There are two main difficulties that we face when trying to identify t. First, the circumference of O can be arbitrarily large and therefore, a single agent cannot keep track of its relative location with respect to its starting cell $z = (x_b, y_b)$. Second, there might be many possible cells along the edges of O that are hit by the straight line through O. We refer to all these cells along the border of O as *potential target cells* (cf. Figure 4).

The procedure Probe allows an agent a located at cell z to locate the closest potential target cell z^* in direction of its initial heading h and returns a counter representing the distance of z^* relative to z. The exact formulation of Probe depends on the heading h of a. Procedure 3 gives a pseudo-code description for the case of $h = W$. The other cases are analogous.

The idea is that agent a employs three counters c_x, c_y and $c_{\min}$ while walking along the boundary of O. The counters c_x and c_y track the offset of a from the initial cell (x_b, y_b). Whenever c_y is zero, a has located a cell with the same y-coordinate and the value of c_x is stored in $c_{\min}$ if it

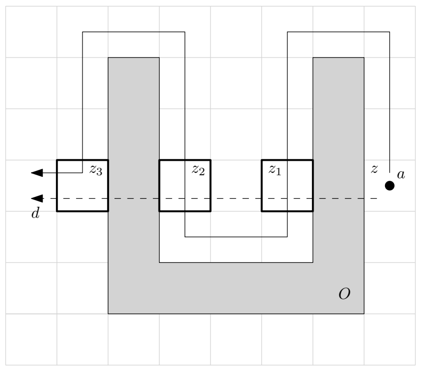


Figure 4: Agent a wants to walk west but the direct path (dashed arrow) is obstructed by an obstacle O. Thus, a walks counter-clockwise around the boundary of O (continuous arrow) and uses offset counters to detect the potential target cells z_1, z_2, and z_3.

is smaller than the previous $c_{\min}$. This process is iterated until the agent returns to the starting position (it meets agent a_b again). Then it turns off counters c_x and c_y and returns $c_{\min}$.

```
Procedure 3: ProbeW()
  On($c_x$); On($c_y$); On($c_{\min}$);
  repeat
      $h \leftarrow$ StepCounterClockwise()
      Increment($c_x$) / Decrement($c_x$) according to $h$
      Increment($c_y$) / Decrement($c_y$) according to $h$
      if IsNull($c_y$) $\wedge$ (IsNull($c_{\min}$) $\vee$ LessThan($c_x, c_{\min}$)) then
          Set($c_{\min}, c_x$)
  until $a$ meets $a_b$;
  Off($c_x$); Off($c_y$);
  return $c_{\min}$
```

5.3 Procedure Scan

A detail that we have to be careful with is, when traveling from one guide to another, that each cell along the current square gets discovered and that we eventually reach the guide. To this end, the explorer visits each cell on the boundary of an obstacle that it meets using the procedure Scan.

When an agent a executes Scan, it first activates two counters c_x and c_y. Then, it walks once around the obstacle by repeatedly invoking StepCounterClockwise and updating c_x and c_y according to its actual movements. If a meets the next guide along the way, it does not update the counters anymore. When a returns to the cell containing the base agent a_b of its counter, the walk is finished. If both c_x and c_y equal 0, no guide was not found during Scan. Otherwise, the values of the counters represent the offset to the guide and the procedure “returns” the two counters c_x and c_y. Since a might meet different guides, it stores the index of the next guide that it is supposed to meet according to the protocol in its state, thereby allowing it to ignore all other guides.

5.4 Procedure Update

As the last building block of our algorithm, we establish the procedure Update(M) that updates the location $Z_M(d)$ of some guide M to $Z_M(d+1)$ for any $d > 0$.

Lemma 1. *The procedure Update(M) enables the the explorer to move from cell $Z_M(d)$ to cell $Z_M(d+1)$ and back to cell $Z_M(d)$, for any $d \geq 1$.*

5.5 Procedure Update(M, c)

In this section, we establish the procedure Update that allows the explorer to find the cell $Z_M(d+1)$ starting from the cell $Z_M(d)$ of some guide M for any $d > 0$. Consider Update in the case of the NW-guide currently occupying cell $Z_{\text{NW}}(d) = (-d, y^*)$. We assume that the explorer has access to a counter c_y, denoting the y-offset to the line $y = d$. To initialize the update, the explorer leaves another agent to mark $Z_{\text{NW}}(d)$ and instructs the NW-guide to follow the explorer. A lengthy pseudo-code representation of the Update for the NW-guide can be found in Procedure 4, where the Update$_{\text{NW}}(c)$ stands for the special case of the NW-guide. To locate the cell $Z_{\text{NW}}(d+1)$, our first task is to find a cell $z \in L$, where L is the set of cells whose x-coordinate equals to $-(d+1)$ and whose y-coordinate is at least as large as $d+1$, i.e.,

$$L = \{(i, j) \in \mathbb{Z}^2 \mid (i = -(d+1)) \wedge (j \geq d+1))\} \ .$$

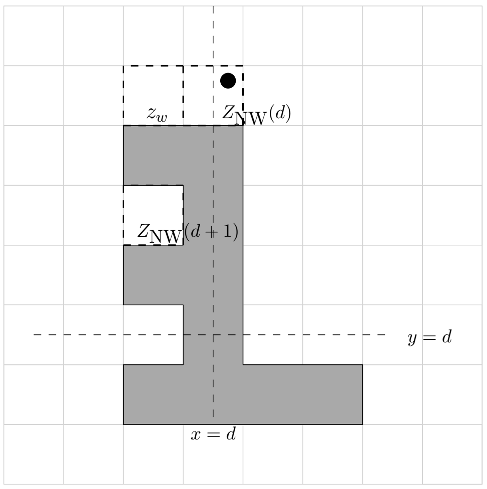
We divide our description of Update into several cases. First, we consider the case that the cell $z_w = (-(d+1), y^*)$ west to $Z_{\text{NW}}(d)$ is blocked by obstacle O. This induces that $Z_{\text{NW}}(d)$ and $Z_{\text{NW}}(d+1)$ are on the border of the same obstacle O. Refer to Figure 5b for an illustration. The explorer turns on the c_x counter. Then, it increments its value by 1 to correspond to the offset from $Z_{\text{NW}}(d+1)$. Also the c_y counter is decremented by 1, to mark the next desired y-coordinate.

To reach a cell $z \in L$, the explorer now simply turns its heading to north to initialize a walk counter-clockwise around O. Now since O is finite, it has to be the case that there is at least one cell from L on the boundary of O. The explorer successively executes StepCounterClockwise, updates counters c_x and c_y accordingly, and always checks if $c_x = 0$ and if c_y is positive. If the check returns true, the explorer has reached a cell $z \in L$.

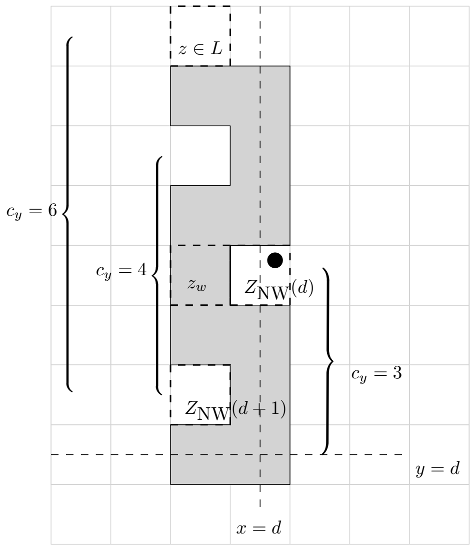
To now find the cell $Z_{\text{NW}}(d+1)$, the explorer first turns its heading towards south and then successively executes Shift(0, Probe()), and updates c_y accordingly during every Shift, until Probe returns a value greater than the current c_y. If the next cell found by Probe is further away than c_y we know that we are in the cell $Z_{\text{NW}}(d+1)$ at the moment. As the last step of this case, the explorer instructs the NW-guide to remain in this cell, and walks counter-clockwise around O until it finds the agent denoting cell $Z_{\text{NW}}(d)$.

Procedure 4: $\textsc{Update}_{\text{NW}}(c_y)$

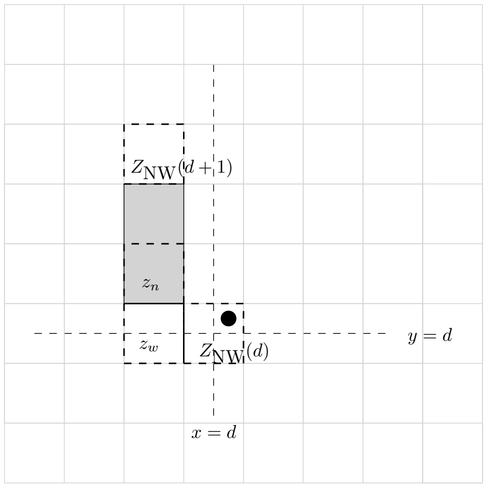
```
Agent $a$ is located in $Z_{\text{NW}}(d) = (-d, y^*)$, $z_w = (-(d+1), y^*)$, $z_n = (-(d+1), d+1)$
Mark $Z_{\text{NW}}(d)$ with an agent $a_{mark}$
**if** $z_w \in O$ **then**
    ▷ $z_w \in O \Rightarrow$ $Z_{\text{NW}}$(d) and $Z_{\text{NW}}$(d+1) are next to the same obstacle
    ▷ Figure 5b represents this case
    $h \leftarrow$ N; On($c_x$); Increment($c_x$); Decrement($c_y$);
    ▷ store offsets to the coordinate (-(d+1),d+1) instead to (-d,d)
    **repeat**
        $h \leftarrow$ StepCounterClockwise;
        Increment($c_x$) / Decrement($c_x$) according to $h$;
        Increment($c_y$) / Decrement($c_y$) according to $h$:
    **until** *IsNull*($c_x$) $\wedge$ *IsPositive*($c_y$);
    ▷ We found the cell z, cell $Z_{\text{NW}}$(d+1) is south to us
    $h \leftarrow$ S; Off($c_x$); On($c_0$);
    **repeat**
        $c_{y'} \leftarrow$ Probe();
        Shift($c_0, c_{y'}$) while updating $c_y$;
    **until** *LessThan*($c_y, c_{y'}$);
    turn off all counters; leave the NW-guide in this cell; follow the obstacle back to $c_{mark}$;
**else**
    $h \leftarrow$ W; move once towards $h$;  ▷ $z_w$ is free, walk one step west
    **if** *IsPositive*($c_y$) **then**
        ▷ (-d/d) is blocked and $Z_{\text{NW}}$(d) is further north
        ▷ $Z_{\text{NW}}$(d) and $Z_{\text{NW}}$(d+1) are next to the same obstacle
        Decrement$c_y$;
        **if** $\neg$*IsNull*($c_y$) **then**
            ▷ We are further north than needed for $Z_{\text{NW}}(d+1)$
            ▷ Figure 5a represents this case
            $h \leftarrow$ S; Off($c_x$); On($c_0$), $c_{y'} \leftarrow$ Probe()
            **while** *LessThan*($c_y, c_{y'}$) **do**
                $c_{y'} \leftarrow$ Probe()
                Shift($c_0, c_{y'}$) while updating $c_y$
        turn off all counters; leave the NW-guide in this cell, follow the obstacle back to
         $c_{mark}$
    **else**
        Off($c_x$); Off($c_y$); On($c_y$); Decrement($c_y$);
        $h \leftarrow$ N
        **if** $z_n \in O$ **then**
            ▷ $(-d, d)$ is free, $z_n$ is blocked, see Figure 5c
            On($c_0$); $c_{y'} \leftarrow$ Probe()
            Shift($c_0, c_{y'}$)
            turn off all counters; leave the NW-guide in this cell; reverse the movements to
             go back to $c_{mark}$
        **else**
            ▷ $(-d, d)$ and $(-(d+1), d+1)$ are both free
            move once towards $h$, leave the NW-guide in this cell
            turn off all the counters; move once south and once east to go back to $c_{mark}$
```



(a) The explorer is located in cell $Z_{\text{NW}}(d)$ and executes UPDATE(NW). Initially, $\text{val}(c_y) = 4 > 0$ and since z_w is free, the explorer moves directly to z_w and decrements c_y so that $\text{val}(c_y) = 3$. Then it performs PROBE() ($h = S$) that returns a counter with value 2. Thus, the explorer performs SHIFT$(0, 2)$ and updates c_y accordingly so that $\text{val}(c_y) = 1$ once the explorer reaches $Z_{\text{NW}}(d + 1)$. The following PROBE() returns a counter with value $2 > 1 = \text{val}(c_y)$ and therefore, the explorer knows that it currently occupies cell $Z_{\text{NW}}(d + 1)$.



(b) Initially, there is an offset of 3 from the north side of the square(d) (stored in the c_y counter), then c_y is decremented to 2. As a next step, the explorer locates cell z and then executes PROBE and SHIFT until $Z_{\text{NW}}(d + 1)$ is located. When $Z_{\text{NW}}(d + 1)$ is reached, the value of c_y is 0 and therefore smaller than the value of the counter returned by PROBE.



(c) The first cell to the west from $Z_{\text{NW}}(d)$ is free, c_y equals 0, and $Z_{\text{NW}}(d+1)$ is located by moving once west and then executing PROBE and SHIFT with heading N.

Figure 5: Special cases of UPDATE.

Then, consider the case that cell z_w is not blocked. We further split into two cases and we first consider the case that $c_y > 0$, which can be asserted by the explorer by checking if IsPositive(c_y) returns true. Then, it has to be the case that all cells $(-d, y^* - i)$, for $i \leq y^* - d$, are blocked by some obstacle O due to the invariant that $Z_{\text{NW}}(d)$ has the smallest y-coordinate among free cells $(-d, y \geq d)$. See Figure 5a for an illustration. Thus, the explorer can move to z_w and the counter c_y is still valid. Furthermore, cell $Z_{\text{NW}}(d+1)$ has to be on the boundary of O.

Next, the explorer decrements c_y by 1. If $c_y = 0$, then we have reached cell $Z_{\text{NW}}(d+1)$. Otherwise, similarly to the previous case, the explorer now turns its heading towards south and executes Shift(0, Probe()) until Probe returns a value greater than c_y. When Probe returns a value greater than c_y, the explorer has reached cell $Z_{\text{NW}}(d+1)$. Similarly to the previous case, the explorer instructs the NW-guide to mark this cell and travels back to $Z_{\text{NW}}(d)$ by walking around obstacle O.

Consider now the case where z_w is not blocked and $c_y \leq 0$. Note that due to the invariant that $Z_{\text{NW}}(d)$ has the smallest y-coordinate among free cells $(-d, y \geq d)$, we get that $c_y = 0$. Therefore, the explorer can turn off both counters c_x and c_y without losing any information. Then, the explorer along with the other agents, moves to cell z_w. After reaching z_w, the explorer turns its heading towards north and if $(-(d+1), d+1)$ is not blocked, it moves once north reaching the cell $Z_{\text{NW}}(d+1)$. After instructing NW to mark $Z_{\text{NW}}(d+1)$, the explorer can find back to $Z_{\text{NW}}(d)$ simply by reversing its movements.

If $(-(d+1), d+1)$ is blocked, the explorer executes Shift(0, Probe()) once, so that it reaches the free cell with the smallest y-coordinate at least $d+1$, i.e., the cell $Z_{\text{NW}}(d+1)$. Refer to Figure 5c for an illustration The explorer again instructs the NW-guide to remain in $Z_{\text{NW}}(d+1)$ and travels back to $Z_{\text{NW}}(d)$ by turning its heading south, executing Shift(0, Probe) once, and moving once east.

In all of the above cases, the guide was left in a cell $Z_{\text{NW}}(d+1)$ yielding the correctness of the update procedure for the NW-guide and the explorer found its way back to the cell $Z_{\text{NW}}(d)$. This concludes the description of Update for the NW-guide. The procedure Update works analogously for other guides. Note that when updating the NE-guide, the explorer does not return back to cell $Z_{\text{NE}}(d)$ and therefore does not leave an agent in that cell either. Thus, Lemma 1 follows.

6 Searching the Plane

When executing the search protocol SquareWalk, the agents begin the search by four agents moving into the cells $(1,1)$, $(-1,1)$, $(-1,-1)$, and $(1,-1)$, corresponding to $Z_{\text{NE}}(1)$, $Z_{\text{NW}}(1)$, $Z_{\text{SW}}(1)$, and $Z_{\text{SE}}(1)$. Recall that these agents, the guides, essentially mark the corners of the square that the explorer will explore next and that we identify each guide with the cardinal direction of its corner (NE, NW, SW, SE). The explorer e, equipped with a set of counters in follow mode, moves to the NE-guide in the cell $Z_{\text{NE}}(1)$. It then starts to explore square(1) by moving west until it meets the NW-guide in cell $Z_{\text{NW}}(1)$ and, together with the NW-guide, moves to cell $Z_{\text{NW}}(2)$. Then the explorer returns to $Z_{\text{NW}}(1)$ and moves south towards the SW-guide. It proceeds analogously with the other guides and eventually returns to the NE-guide. After moving the NE-guide to cell $Z_{\mathrm{NE}}(2)$, the explorer does *not* return to $Z_{\mathrm{NE}}(1)$ but instead starts to explore square(2)

Starting from the next iterations, things get more involved as obstacles might be in the way of the explorer or of the guides. Consider the situation that the next square to be searched by the explorer is square(d), every guide M is in the corresponding cell $Z_M(d)$, and the explorer is in cell $Z_{\mathrm{NE}}(d)$. We explain how e can walk from the NE-guide to the NW-guide and thereby explore the north side of square(d); the three other sides of the square are analogous. Procedure 5 gives a pseudo-code description in which $z_e = (x_e, y_e)$ denotes the current cell of the explorer while an explanation follows below.

```
Procedure 5: ExploreNorthSide
  $h \leftarrow \mathrm{W}$    ▷ set heading
  repeat
      if $(z_e + h) \notin B$ then
          move($h$)    ▷ next cell is free
      else
          $c_{\mathrm{probe}} \leftarrow$ Probe()
          $(c_x, c_y) \leftarrow$ Scan()
          if $(\textit{IsNull}(c_x) \wedge \textit{IsNull}(c_y)) \vee \textit{LessThan}(c_{probe}, c_x)$ then
              Off($c_y$); On($c_y$); Off($c_x$)    ▷ reset $c_y$ to zero and turn off $c_x$
              Shift($c_{\mathrm{probe}}, c_y$)    ▷ move to next free cell
          else
              Off($c_{\mathrm{probe}}$); On($c_{\mathrm{update}}$)    ▷ re-use agents from the $c_{\mathrm{probe}}$ counter
              Set($c_{\mathrm{update}}, c_x$)
              Shift($c_x, c_y$)    ▷ move to NW-guide
  until $e$ meets NW;
  Update(NW, $c_{\mathrm{update}}$)
```

The explorer e sets its heading towards west and, as long as the cell in front is free, moves forward. If e senses an obstacle in front in cell z, e executes Probe to find the next free cell z' in the direction of its heading, resulting in the counter c_{probe} representing the distance between z_e and z'. Then e scans the obstacle using Scan yielding the counters c_x and c_y. If Scan was not successful, i.e., the NW-guide was not located along the obstacle, the counters c_x and c_y are both zero. Now, e moves to z' using Shift($c_{\mathrm{probe}}, 0$) (c_y is reset and used as second parameter) if

(i) Scan was not successful, i.e., the NW-guide was not located along the obstacle (corresponding to (IsNull(c_x) $\wedge$ IsNull(c_y) $=$ `true`) or

(ii) Scan found the next guide but it is further west than the next target cell (corresponding to LessThan(c_{probe}, c_x) $=$ `true`) and repeats the above. If $\text{val}(c_{\text{probe}}) \geq \text{val}(c_x)$ corresponding to $\textsc{LessThan}(c_{\text{probe}}, c_x) = \texttt{false}$, the explorer executes $\textsc{Shift}(c_x, c_y)$ to move to Z_{NW} to meet the NW-guide.

Finally, e uses Update to update the position of the NW-guide from $Z_{\text{NW}}(d)$ to $Z_{\text{NW}}(d+1)$ and returns to $Z_{\text{NW}}(d)$. Then, it sets its heading to south, turns off all counters and starts the analogous procedure ExploreWestSide, this time walking south towards the SW-guide.

The above procedure is repeated for all four sides of the square until the explorer arrives back at the NE-guide and updates its position to $Z_{\text{NE}}(d+1)$. Now e does *not* return to $Z_{\text{NE}}(d)$ but instead starts a search of square$(d+1)$ using ExploreNorthSide.

Correctness. In this section, we establish the correctness of the protocol SquareWalk, i.e., that it guarantees that the explorer eventually visits all free cells of the grid. We define the concept of a *configuration* $C : A \mapsto \mathbb{Z}^2$ as an assignment of a cell to each agent. A configuration is a snapshot of the positions of the agents at a given time. The *start configuration for distance* d, denoted by $\mathbf{Z}(d)$, is the configuration where each guide M is in its corresponding cell $Z_M(d)$ and the explorer and the auxiliary agents are in cell $Z_{\text{NE}}(d)$ with the NE-guide. We furthermore define the set

$$F_i = \{(x, y) \notin B \mid (|x| = i \wedge |y| \leq i) \vee (|y| = i \wedge |x| \leq i)\}$$

as the free cells of square(i) for some $i \geq 1$. We are now ready to prove the following theorem which establishes the correctness of SquareWalk.

Theorem 1. *The protocol SquareWalk guarantees that every cell $z \in \mathbb{Z}^2$ is visited by the explorer within finite time.*

Proof. We show by induction over d that for any d, there is a time such that the explorer has visited all cells in $\mathcal{F}_d = \bigcup_{i \leq d} F_i$ and the agents are in $\mathbf{Z}(d+1)$.

The induction base holds by design of the protocol as the agents start the search in configuration $\mathbf{Z}(1)$ and the cells in distance 2 from the origin are free. Hence, the explorer visits all cells and afterwards the agents are in $\mathbf{Z}(2)$.

For the inductive step, assume that the agents are in configuration $\mathbf{Z}(d)$ and all cells in $\mathcal{F}_{d-1}$ have been explored. We consider the walk along the north side of square(d). Let $V_d^{\text{N}} = \langle z_0 = Z_{\text{NE}}(d), z_1 = (x_1, d), \dots, z_k = (x_k, d), z_{k+1} = Z_{\text{NW}}(d)\rangle$ be the sequence of free cells of the north side of square(d) excluding the corners $\{(-d, d), (d, d)\}$ extended by the cells $z_0 = Z_{\text{NE}}(d)$ and $z_{k+1} = Z_{\text{NW}}(d)$, ordered by descending x-coordinates. Initially, the explorer is located in z_0 and we show that for any $i < k$, the explorer moves to z_{i+1} in finite time.

Consider the case of $i \leq k-1$ and thus $z_{i+1} \neq Z_{\text{NW}}(d)$. If z_{i+1} is neighbor to z_i, the explorer moves to z_{i+1}. If the cell west of z_i is blocked, then Probe finds z_{i+1} and the explorer moves there.

Now consider the case of $i = k$ and thus $z_{i+1} = Z_{\text{NW}}(d)$. If $(-d, d) = Z_{\text{NW}}(d)$ and thus a free cell, the explorer moves there either directly or through Probe/Shift. If $(-d, d)$ is blocked, $Z_{\text{NW}}(d)$ is located along the boundary of the obstacle that blocks the cell $(-d, d)$ by definition. As the explorer explores the boundary of said obstacle using Scan, the explorer is guaranteed to arrive at $Z_{\text{NW}}(d)$. Consequently, all cells in V_d^{N} are visited by the explorer and by Lemma 1, we

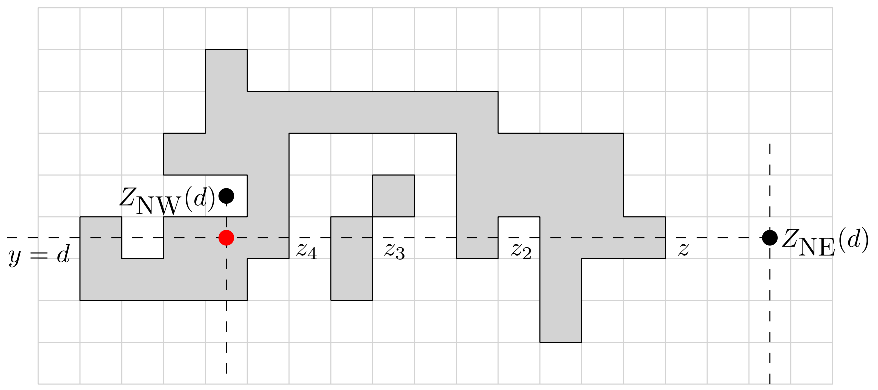


Figure 6: Even though agent e encounters the NW-guide already when it scans in cell c there are many more cells to be visited, even another obstacle has to be circumvented, before e turns south with the help of the guide.

know that the explorer can execute UPDATE in cell $Z_{\mathrm{NW}}(d)$ to move the NW-guide to $Z_{\mathrm{NW}}(d+1)$ and then return to $Z_{\mathrm{NW}}(d)$.

The argumentation for the three other sides of the square is analogous and thus the explorer visits all cells in F_d and then has visited all cells in $\mathcal{F}_d$. After moving the NE-guide to $Z_{\mathrm{NE}}(d+1)$, the explorer stays put. Hence, the agents are in configuration $\mathbf{Z}(d+1)$, which concludes the inductive step.

The design of our protocol ensures that the agents cannot enter an infinite loop and thus, in every time unit, at least one agent — and thus the execution of the protocol — progresses. Consequently, the explorer visits every cell in finite time. □

7 Conclusion

We presented the protocol SQUAREWALK that allows a group of finite state machines (with a constant number of states) to locate an adversarially hidden treasure in a plane obstructed by arbitrary obstacles of finite circumference. Our search protocol employs the weak communication capabilities of the agents to simulate a sufficient amount of memory to ensure progress in the search.

Our search protocol requires ten agents in total, where one of the agents acts as an explorer, who performs the searching. The protocol uses three offset counters, requiring five agents. The other four agents mark the sides of a square around the origin that bounds the area discovered so far. We remark that we can reduce the agent count to nine by using the triangle approach from [11]. But as this makes the specification of our protocol considerable more involved, we presented the simpler version employing the square approach.

References

[1] Susanne Albers and Monika Henzinger. Exploring Unknown Environments. *SIAM Journal on Computing*, 29:1164–1188, 2000.

[2] Baruch Awerbuch and Margrit Betke. Piecemeal Graph Exploration by a Mobile Robot. *Information and Computation*, 1999.

[3] Michael Bender, Antonio Fernandez, Dana Ron, Amit Sahai, and Salil Vadhan. The Power of a Pebble: Exploring and Mapping Directed Graphs. In *Proceedings of the 30th annual ACM Symposium on Theory of Computing (STOC)*, 1998.

[4] Manuel Blum and Dexter Kozen. On the Power of the Compass (or, Why Mazes Are Easier to Search Than Graphs). In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 132–142, 1978.

[5] Manuel Blum and William J. Sakoda. On the Capability of Finite Automata in 2 and 3 Dimensional Space. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 147–161, 1977.

[6] Lothar Budach. Automata and Labyrinths. *Mathematische Nachrichten*, pages 195–282, 1978.

[7] Xiaotie Deng and Christos Papadimitriou. Exploring an Unknown Graph. *Journal of Graph Theory*, 32:265–297, 1999.

[8] Krzysztof Diks, Pierre Fraigniaud, Evangelos Kranakis, and Andrzej Pelc. Tree Exploration with Little Memory. *Journal of Algorithms*, 51:38–63, 2004.

[9] Klemens Döpp. Automaten in Labyrinthen. *Elektronische Informationsverarbeitung und Kybernetik*, 7(2):79–94, 1971.

[10] Christian A. Duncan, Stephen G. Kobourov, and V. S. Anil Kumar. Optimal Constrained Graph Exploration. *ACM Transactions on Algorithms (TALG)*, 2(3):380–402, 2006.

[11] Yuval Emek, Tobias Langner, David Stolz, Jara Uitto, and Roger Wattenhofer. How Many Ants Does it Take to Find the Food? In *21th International Colloquium on Structural Information and Communication Complexity (SIROCCO)*, pages 263–278, 2014.

[12] Yuval Emek, Tobias Langner, Jara Uitto, and Roger Wattenhofer. Solving the ANTS Problem with Asynchronous Finite State Machines. In *Proceedings of the 41st International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 471–482, 2014.

[13] Ofer Feinerman and Amos Korman. Memory Lower Bounds for Randomized Collaborative Search and Implications for Biology. In *Proceedings of the 26th International Conference on Distributed Computing (DISC)*, pages 61–75, Berlin, Heidelberg, 2012. Springer-Verlag.

[14] Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative Search on the Plane Without Communication. In *Proceedings of the 31st ACM Symposium on Principles of Distributed Computing (PODC)*, pages 77–86, 2012.

[15] Pierre Fraigniaud and David Ilcinkas. Digraphs Exploration with Little Memory. In *21st Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 246–257, 2004.

[16] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph Exploration by a Finite Automaton. *Theoretical Computer Science*, 345(2-3):331–344, 2005.

[17] Frank Hoffmann. One Pebble Does Not Suffice to Search Plane Labyrinths. In *Fundamentals of Computation Theory*, pages 433–444. Springer Berlin Heidelberg, 1981.

[18] Saket Navlakha and Ziv Bar-Joseph. Distributed Information Processing in Biological and Computational Systems. *Communications of the ACM*, 58(1):94–102, 2014.

[19] Petrişor Panaite and Andrzej Pelc. Exploring Unknown Undirected Graphs. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 316–322, 1998.

[20] Noa Pinter-Wollman, Ashwin Bala, Andrew Merrell, Jovel Queirolo, Martin C Stumpe, Susan Holmes, and Deborah M Gordon. Harvester Ants Use Interactions to Regulate Forager Activation and Availability. *Animal behaviour*, 86(1):197–207, 2013.