Tight Bounds for Deterministic High-Dimensional Grid Exploration

Sebastian Brandt
ETH Zürich, Switzerland
brandts@ethz.ch

Julian Portmann
ETH Zürich, Switzerland
pjulian@ethz.ch

Jara Uitto
Aalto University, Finland
jara.uitto@aalto.fi

Abstract

We study the problem of exploring an oriented grid with autonomous agents governed by finite automata. In the case of a 2-dimensional grid, the question how many agents are required to explore the grid, or equivalently, find a hidden treasure in the grid, is fully understood in both the synchronous and the semi-synchronous setting. For higher dimensions, Dobrev, Narayanan, Opatrny, and Pankratov [ICALP'19] showed very recently that, surprisingly, a (small) constant number of agents suffices to find the treasure, independent of the number of dimensions, thereby disproving a conjecture by Cohen, Emek, Louidor, and Uitto [SODA'17]. Dobrev et al. left as an open question whether their bounds on the number of agents can be improved. We answer this question in the affirmative for deterministic finite automata: we show that 3 synchronous and 4 semi-synchronous agents suffice to explore an n-dimensional grid for any constant n. The bounds are optimal and notably, the matching lower bounds already hold in the 2-dimensional case.

Our techniques can also be used to make progress on other open questions asked by Dobrev et al.: we prove that 4 synchronous and 5 semi-synchronous agents suffice for *polynomial-time* exploration, and we show that, under a natural assumption, 3 synchronous and 4 semi-synchronous agents suffice to explore *unoriented* grids of arbitrary dimension (which, again, is tight).

2012 ACM Subject Classification Computing methodologies → Mobile agents

Keywords and phrases Mobile agents, finite automata, grid search

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.13

Related Version A full version of the paper is available at `https://arxiv.org/abs/2005.12623`.

Funding *Julian Portmann*: Supported by the Swiss National Foundation, under project number200021_184735.

1 Introduction

Grid search by mobile agents is one of the fundamental primitives in swarm robotics and a natural abstraction of foraging behavior of animals. For example in the case of cost-efficient robots or insects, a single agent has relatively limited computation and communication capabilities and hence, many independent agents are required to efficiently solve tasks. To understand such collective problem solving better, knowledge from distributed computing has proven valuable. For instance, Feinerman et al. gave tight bounds on the time complexity of a collective grid search problem inspired by desert ants [17]. In this paper, we focus on the minimum *number* of agents required to solve the grid search problem. A series of papers [7,9,15] nailed down the exact complexity of the 2-dimensional case, that is, discovered the exact number of synchronous/semi-synchronous and deterministic/randomized finite

34th International Symposium on Distributed Computing (DISC 2020).
Editor: Hagit Attiya; Article No. 13; pp. 13:1–13:16
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

automata needed to explore a 2-dimensional grid. However, the approaches in these works do not generalize (well) to higher dimensions. The only known tight bound achieved by such a generalization is obtained by the recent protocol for the deterministic semi-synchronous 3-dimensional setting by Dobrev, Narayanan, Opatrny, and Pankratov [13].

The authors of [13] also gave a more general result: they showed how to implement a stack data structure using only a constant number of agents governed by finite automata. By employing this stack in their search protocols, they show how to explore an n-dimensional grid using only a (small) constant number of agents, for any positive integer n. In particular, the number of agents is independent of the dimension n.

For the case of a 2-dimensional grid the required number of agents is fully understood. However, for higher dimensions there are still gaps between the best upper and lower bounds. Indeed, Dobrev et al. left as open questions the tight complexities of exploring high-dimensional grids in the synchronous/semi-synchronous and deterministic/randomized settings. In this work, we answer these questions for the deterministic setting. Moreover, building on our techniques we make progress on other open questions by Dobrev et al.

1.1 Results and Techniques

Similarly to the approach by Dobrev et al. [13], our search protocols rely on an efficient implementation of a stack data structure. One agent is dedicated to do the actual search while the remaining agents implement a stack (together with the searching agent that indicates the base of the stack) with their positions on the grid. On a high level, the size of the stack encodes the cell the searching agent is supposed to explore next, relative to the current position of the searching agent. Both our protocol and the protocol from [13] explore the grid by repeatedly reading the stack, moving the searching agent to the cell indicated by the stack, moving the searching agent back to its original cell, and incrementing the stack. The difficult part is to be able to effectively read the stack (without destroying the stack in the process) despite the fact that the size of the stack grows arbitrarily far beyond the number of states in the finite automaton reading the stack. The authors of [13] managed to implement this data structure using 4 agents in the synchronous and 5 agents in the semi-synchronous setting and showed how to explore oriented grids with as many agents.

One of our main contributions is to implement this stack and the operations required for reading it with only 3 (synchronous) agents (including the searching agent), which is optimal given the grid exploration lower bound by Emek et al. [15]. We achieve this by an encoding scheme that transforms the location of a cell to be explored into a single integer (that can be represented by the stack size) by interpreting the coordinates of the cell (relative to the current location of the searching agent) as exponents of distinct prime factors. This scheme is also known as Gödel's encoding. One crucial advantage of this specific encoding is that there is a way to read the stack (using 3 synchronous agents), i.e., to repeatedly provide the searching agent with different parts of the encoded information, that does not destroy the encoded information, but instead changes the encoding slightly: replacing the base (prime) for one of those exponents by a different prime (and then switching to the next base prime). The technical details why such replacement operations can be performed by 3 synchronous agents and why they allow the searching agent to obtain the desired information are covered in Sections 3 and 4. Moreover, by adding one agent as a synchronizer, the protocol can be made to work in the semi-synchronous setting.

▶ **Theorem 1.** *For any positive integer n, the n-dimensional (oriented) grid can be explored by* 3 *synchronous finite automata, resp.* 4 *semi-synchronous finite automata.*

Unoriented Grids. An underlying assumption of the setting considered so far is that the agents are aware of the $2n$ cardinal directions, i.e., they know for each of the n dimensions of the grid which two adjacent cells are neighbors in that dimension, and each dimension is oriented. Or, to put it simply, the agents know which directions are north, south, etc.; in particular the directions are globally consistent. In contrast, in the *unoriented* setting considered in [13], each cell is endowed with a labeling that indicates for each cell which neighbor is north, south, etc. (and for each of the $2n$ directions there is exactly one neighbor), but there is no consistency guarantee between the directions indicated by the labels of different cells, e.g. by going north twice, an agent could end up in the cell it started.

In their work, Dobrev et al. also ask "How many additional agents are necessary to solve the problem in *unoriented grids*?". We show, perhaps surprisingly, that the unoriented case is no harder than the oriented case given the following (natural) assumption: If we follow some fixed direction, we never end up back in the same cell where we started.

▶ **Theorem 2** (Simplified). *Under a natural assumption, for any positive integer n, 3 synchronous finite automata, resp. 4 semi-synchronous finite automata, suffice to explore any n-dimensional unoriented grid.*

The key idea to obtain Theorem 2 is that, even without a globally consistent orientation, we can implement a (virtual) stack. Due to the missing consistency, the same cell may occur repeatedly in the stack, but we can show that we can bound the number of occurrences for each cell and that the agents can distinguish between the different occurrences of the same cell. In essence, we will show that the stack corresponds to (a part of) a DFS exploration of an infinite tree consisting of those edges (between cells) that point north.

Polynomial Time Protocol. The task of exploring the entire grid is equivalent to finding a treasure located at some distance D from the starting point. This allows us to discuss the *efficiency* of a protocol, i.e., its runtime with respect to D. We observe that our encoding scheme for the oriented grid using only 3 synchronous, resp. 4 semi-synchronous, agents might result in exponential time. However, with one additional agent, certain useful stack operations can be extended to work for non-constant values. This allows us to use a different exploration scheme, proposed by Dobrev et al. [13], resulting in a polynomial runtime.

▶ **Theorem 3.** *For any positive integer n, the n-dimensional (oriented) grid can be explored by: (1) 4 synchronous agents in time $O(V(D)^2)$, and (2) 5 semi-synchronous agents in time $O(V(D)^3)$, where $V(D) = \Theta(D^n)$ is the volume of the ℓ_1-ball of radius D.*

Due to space constraints, our discussion of polynomial-time exploration, as well as the proofs of all lemmas and theorems are deferred to the full version.

1.2 Further Related Work

In a typical graph exploration setting, we are given a graph where initially, one or more mobile agents are placed on some vertices of the graph. The agents are able to traverse along the edges and their goal is to *explore* the graph, that is, visit every node or edge of the graph. Equivalently, one can think of searching for a treasure hidden on an edge or a node of the graph. Graph exploration has been widely studied in the literature (see, for example, [1,10,11,18,21,22]) and it comes in many variants.

A classic setting is the *cow-path* problem, where a single agent, the cow, is searching for adversarially hidden food on a path [3,4]. The goal for the cow is to minimize the number of edge-traversals until the food is found. It is known that a simple spiral search is optimal and this algorithm also generalizes to the case of grids. This problem was also studied in the case of many cows [20]. Closely related to our work is the exploration of *labyrinths*, i.e., 2-dimensional grids where some cells are blocked [8]. It is known that two finite automata or one automaton with two pebbles (movable marker) suffice for co-finite labyrinths, where a finite amount of cells are not blocked [6]. Finite labyrinths, where a finite amount cells are blocked, can be explored with one automaton and four pebbles, whereas one automaton and one pebble is not enough [5, 19]. An agent with $\Theta(\log\log n)$ pebbles can explore all graphs and this bound is tight [12].

In the case of many agents, the agents typically operate in *look-compute-move* cycles. First, the agents simultaneously take a local snapshot, then decide on the next operation, and finally, execute the operation. Graph exploration can be divided into *synchronous* ($\mathcal{FSYNC}$), *semi-synchronous* ($\mathcal{SSYNC}$), and *asynchronous* ($\mathcal{ASYNC}$) variants [23–25]. In the $\mathcal{FSYNC}$ setting, the execution is divided into synchronous rounds, where in every round, every agent executes one cycle. The execution in $\mathcal{SSYNC}$ consists of discrete time steps, where in each step, a subset of the agents executes one atomic cycle. In the $\mathcal{ASYNC}$ setting, the cycles are not (necessarily) atomic. In this paper, we study the $\mathcal{FSYNC}$ and the $\mathcal{SSYNC}$ settings.

For finite graphs, a *random walk* provides a simple algorithm that explores the graph in polynomial time [2]. In the case of an infinite n-dimensional grid, a random walk finds the treasure with probability 1 if $n \leq 2$. However, the expected hitting time, i.e., the time to find the treasure, is infinite. Cohen et al. showed that even for the case of two (collaborating) randomized agents governed by finite automata, one cannot achieve any finite hitting time for $n \geq 2$ [9]. Very recently, Dobrev et al. showed that 3 randomized $\mathcal{FSYNC}$ and 4 randomized $\mathcal{SSYNC}$ agents suffice to achieve a finite hitting time for any n [13]. In this work, we achieve the same bounds with deterministic agents.

This work follows a series of papers inspired the work by Feinerman et al., where they studied the time it takes to find a treasure in a 2-dimensional grid by k non-communicating agents governed by Turing machines [17]. They showed that the time complexity of this task is $\Theta(D^2/k + D)$, where D is the distance from the origin to the treasure. This bound can be matched by finite automata that are allowed to communicate within the same cell [16]. Emek et al. asked what is the minimum number of finite automata agents required to find the treasure [15]. They showed that at least 3 synchronous deterministic agents are required and that 3 synchronous deterministic, 4 semi-synchronous deterministic, and 3 semi-synchronous randomized agents are enough. Cohen et al. [9] and Brandt et al. [7] showed the matching lower bounds for the randomized and deterministic semi-synchronous cases, respectively.

2 Preliminaries

Grids. We consider the problem of exploring the infinite n-dimensional grid, whose vertices are the elements of $\mathbb{Z}^n$, which we refer to as *cells*. A cell $c = (c_1, \dots, c_i, \dots, c_n)$ is described by its coordinates and two cells c and c' are connected by an edge if there is a dimension i such that $|c_i - c'_i| = 1$ and $c_j = c'_j$ for $j \neq i$. When talking about distance, we will use the ℓ_1 or *Manhattan distance*, which is defined as $d(c, c') = \sum_i |c_i - c'_i|$.

In the *oriented* case, we assume that there is a consistent labeling of the edges by both of its endpoints, which in the 2-dimensional case can be thought of as the directions of a compass: north, south, east, and west. In general, an edge (c, c') is labeled by $(+1, i)$ from the side of c (and thus $(-1, i)$ from the side of c') if we have that $c_i + 1 = c'_i$.

For *unoriented* grids, we assume that each endpoint of an edge has a label from $\{1, \dots, 2n\}$. We will also refer to these labels as the *ports* of a cell. The only assumption we make is that the labels around each cell are pairwise distinct, i.e., each cell has every port from 1 to $2n$ exactly once. Thus, each edge can receive any pair of labels from $\{1, \dots, 2n\}$.

Exploration. The exploration is performed by m agents, $a_1, \ldots, a_m$, which are initially all placed in the same cell, called the *origin*. W.l.o.g. we assume the origin to have coordinates $(0, \ldots, 0)$. The agents cannot distinguish different cells (including the origin); in particular, they do not know the coordinates of the cell they are in. Their behavior and movement is controlled by a deterministic finite automaton. While we require all agents to use the same automaton, they may start in different initial states. (As we only consider protocols with constantly many agents, one can equivalently assume each agent to be controlled by an individual automaton, as we can combine m automata into one by using disjoint state spaces.) Agents can only communicate if they are in the same cell: each agent senses the states for which there is an agent that occupies the same cell, and performs its next move and state transition based on this information. For oriented grids, such a move is described by a direction and dimension to move in.

In the case of unoriented grids, we assume that agents can also see both labels of each incident edge, and perform their decisions based on this information as well. A move is then described by choosing a port of the current cell and moving along this edge. Previous work by Dobrev et al. [13] used an essentially equivalent definition: Each agent could only see the label on its side of each incident edge, but once it arrived in the new cell by traversing some edge, it would obtain the information about the second label on the edge it traversed. We choose to formalize the model in a slightly different way, as it will simplify the description of our algorithms. However, we emphasize that for our purposes, the two models can be used interchangeably since within $2n$ steps in the model of Dobrev et al., the agents can learn all information that we assume the agents can immediately see.

Formally, we have a state space Q, a transition function δ, and an initial state q_i^0 for every agent a_i. For oriented grids, the transition function has the form: $\delta : Q \times 2^Q \to Q \times (\{-1, +1\} \times \{0, 1, \ldots, n\})$. The function maps an agent in state $q \in Q$, which observes the set of states for which there is an agent occupying the same cell, to a new state $q' \in Q$ and a movement, which is described by the direction (-1 or $+1$) and the dimension (from 1 to n) along which the agent moves to the respective neighboring cell, where an agent can also choose to *stay* in the same cell which is described by dimension 0. We will say that an agent moves *north* if its movement is $(+1, 1)$, and *south* if it is $(-1, 1)$.

For unoriented grids, we change the definition of the transition function slightly to $\delta : Q \times 2^Q \times \{1, \ldots, 2n\}^{2n} \to Q \times (\{-1, +1\} \times \{0, 1, \ldots, n\})$. The function maps an agent in state $q \in Q$, which observes both the set of states for which there is an agent occupying the same cell, and, for each port, the other label on the edge corresponding to that port, to a new state $q' \in Q$ and a movement, which is specified by the port via which the agent leaves the current cell, or 0, in which case the agent does not move.

The Schedule. Time is divided into discrete units, where in each time step, a set of *active* agents performs a *look-compute-move* cycle. First, an agent senses the states of all agents in the same cell (and in the case of unoriented grids both of the labels on all incident edges), then it applies the transition function to its own state and all sensed information, and finally it changes its state and moves as indicated by the result. We assume that one such cycle is atomic, i.e., cycles that start at different times do not overlap.

For the *synchronous* or $\mathcal{FSYNC}$ model, we assume that all agents are active at every time step. We call the system *semi-synchronous*, or the $\mathcal{SSYNC}$ variant, if at every time step only a subset of agents, chosen by an adversary, is active. While the adversary knows all information about the agents and their behavior, it must schedule each agent infinitely often, to avoid trivial impossibilities.

Exploration Cost. Finally, if we discuss the efficiency of a protocol, we consider the following problem, which is equivalent to exploring the grid: the agents are tasked to find a treasure, which is hidden at some distance D from the origin (without the agents knowing the value of D). This enables us to measure the time or *exploration cost* it takes to find the treasure with respect to D. In the synchronous setting, we measure the exploration cost as the number of time steps needed for an agent to arrive at the cell containing the treasure. As, in the semi-synchronous model, this number of steps depends on the schedule, we instead define the exploration cost as the total distance traveled by all agents in this setting.

3 Building Blocks

Encoding Information as a Stack. Dobrev et al. [13] introduced the idea of using multiple agents to implement a *stack*. In its simplest form, a stack is just a pair of agents, whose distance encodes some information. However, to allow for manipulations of the stack, more agents are needed. Our protocol for exploring n-dimensional grids with 3 synchronous, resp. 4 semi-synchronous, agents will consist of subroutines that involve manipulations of the stack. The relevant parameter will be the *stack size*, denoted by X, which is defined as the distance between the *base* of the stack and the *end* of the stack. The base of the stack is the location of agent a_1, and the end of the stack is the location of the other agents. We will only be interested in the stack and its size at the very beginning and very end of each subroutine; at these points in time all agents except a_1 are guaranteed to be in the same cell, and this cell is guaranteed to be reachable from the cell containing a_1 by going repeatedly north, making the notion of a stack well-defined. Whenever we refer to the base, end, or size of the stack *during* some subroutine, we mean the respective notion at the beginning of the subroutine.

In this section, we will describe the subroutines that form the building blocks of our exploration algorithm. Moreover, we will show for both the synchronous and the semi-synchronous setting how to implement the subroutines with the desired number of agents.

In [13], the authors show how to multiply the current stack size by 2, resp. divide it by 2, using 3 synchronous agents. This also provides a way to check whether the current stack size is divisible by 2. The idea behind the implementation is simple: while agent a_1 stays at the base of the stack, the other two agents, initially located at the end of the current stack, move with different speeds[1], a_3 either away from or towards the base of the stack, and a_2 first towards the base, and then reversing direction when the base is reached. The operation is completed when a_2 and a_3 meet again (after a_2 visited the base). By choosing a speed of 1 for a_2, and a speed of $1/3$ for a_3, and letting move a_3 *towards* the base, we achieve that the stack size is halved; by choosing the same speeds and letting move a_3 *away from* the base, we achieve that the stack size is doubled.

We will need similar subroutines as building blocks for our synchronous and semi-synchronous protocols. More precisely, given a positive integer $k \geq 2$, we want the agents to be able to perform the following operations.

- MultiplyStackSize(k): Multiply the stack size by k.
- IsDivisible(k): Check whether the current stack size is divisible by k.
- DivideStackSize(k): If the stack size is divisible by k, divide the stack size by k.

We will only require the agents to be able to perform these operations for constantly many k, where the constant depends (only) on the dimension n of the grid.

---

[1] An agent moves with speed $1/j$ in some direction if it repeatedly performs the following behavior: first it takes one step in the chosen direction, and then it waits for $j-1$ steps. Note that our speed of $1/j$ is the same as speed j in [13].

To implement these operations, we simply adapt the protocols for the case $k = 2$ from [13] by choosing the speeds of $1/(k-1)$ (instead of 1) for a_2 and $1/(k+1)$ (instead of $1/3$) for a_3. More precisely, we implement the desired operations using 3 synchronous agents as follows.

MultiplyStackSize(k). While it is usually easier to understand the behavior of an agent if it is described without specifying the exact states and the transition function, we will provide the latter for subroutine $\textsc{MultiplyStackSize}(k)$ to give an example how to translate the agents' behaviors described in this work into the formal specification of a finite automaton. Let $k \geq 2$ be a positive integer. As usual we assume that a_2 and a_3 are in the same cell c', and a_1 is in a cell $c \neq c'$ such that c' can be reached from c by going north repeatedly (i.e., c and c' differ only in the first coordinate, and c has a smaller first coordinate than c').

In subroutine $\textsc{MultiplyStackSize}(k)$, we denote the starting state of each agent a_i by $\textsc{Mult}^0_{i,k}$. Apart from state $\textsc{Mult}^0_{i,k}$, we will use $2k-2$ other states for agent a_2, denoted by $\textsc{Mult}^1_{2,k}, \dots, \textsc{Mult}^{k-2}_{2,k}$, $\textsc{MultBack}^0_{2,k}, \dots, \textsc{MultBack}^{k-2}_{2,k}$ and $\textsc{Mult}^{\text{fin}}_{2,k}$, and $k+1$ other states for agent a_3, denoted by $\textsc{Mult}^1_{3,k}, \dots, \textsc{Mult}^k_{3,k}$, and $\textsc{Mult}^{\text{fin}}_{3,k}$. Agent a_1 always stays in state $\textsc{Mult}^0_{1,k}$ and cell c. Agents a_2 moves and changes its state according to the following rules, where "stay" indicates that the agents does not move to another cell.

$$\begin{array}{rlr}
(\textsc{Mult}^0_{2,k}, S) \to (\textsc{Mult}^1_{2,k}, \text{south}) & & \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{1,k} \notin S \\
(\textsc{Mult}^0_{2,k}, S) \to (\textsc{MultBack}^1_{2,k}, \text{north}) & & \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{1,k} \in S \\
(\textsc{Mult}^j_{2,k}, S) \to (\textsc{Mult}^{j+1}_{2,k}, \text{stay}) & & \text{for any } 1 \leq j \leq k-3 \text{ and any } S \in 2^Q \\
(\textsc{Mult}^{k-2}_{2,k}, S) \to (\textsc{Mult}^0_{2,k}, \text{stay}) & & \text{for any } S \in 2^Q \\
(\textsc{MultBack}^0_{2,k}, S) \to (\textsc{MultBack}^1_{2,k}, \text{north}) & & \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{3,k} \notin S \\
(\textsc{MultBack}^0_{2,k}, S) \to (\textsc{Mult}^{\text{fin}}_{2,k}, \text{stay}) & & \text{for any } S \in 2^Q \text{ satisfying } \textsc{Mult}^0_{3,k} \in S \\
(\textsc{MultBack}^j_{2,k}, S) \to (\textsc{MultBack}^{j+1}_{2,k}, \text{stay}) & & \text{for any } 1 \leq j \leq k-3 \text{ and any } S \in 2^Q \\
(\textsc{MultBack}^{k-2}_{2,k}, S) \to (\textsc{MultBack}^0_{2,k}, \text{stay}) & & \text{for any } S \in 2^Q
\end{array}$$

For agent a_3, the rules are as follows.

$$\begin{array}{lr}
(\textsc{Mult}^0_{3,k}, S) \to (\textsc{Mult}^1_{3,k}, \text{north}) & \text{for any } S \in 2^Q \text{ satisfying } \textsc{MultBack}^0_{2,k} \notin S \\
(\textsc{Mult}^0_{3,k}, S) \to (\textsc{Mult}^{\text{fin}}_{3,k}, \text{stay}) & \text{for any } S \in 2^Q \text{ satisfying } \textsc{MultBack}^0_{2,k} \in S \\
(\textsc{Mult}^j_{3,k}, S) \to (\textsc{Mult}^{j+1}_{3,k}, \text{stay}) & \text{for any } 1 \leq j \leq k-1 \text{ and any } S \in 2^Q \\
(\textsc{Mult}^k_{3,k}, S) \to (\textsc{Mult}^0_{3,k}, \text{stay}) & \text{for any } S \in 2^Q
\end{array}$$

The protocol terminates when both a_2 and a_3 are in states $\textsc{Mult}^{\text{fin}}_{2,k}$ and $\textsc{Mult}^{\text{fin}}_{3,k}$, respectively. The design of the protocol (in particular, of the two rules leading to the two terminal states) ensures that a_2 and a_3 terminate at the same point in time. As the rules of the protocol specify that a_2 walks with speed exactly $1/(k-1)$, and a_3 with speed exactly $1/(k+1)$, we see that the first time a_2 and a_3 are in the same cell in states $\textsc{MultBack}^0_{2,k}$, resp. $\textsc{Mult}^0_{3,k}$ (which is the configuration leading to termination in the next step), they are in a cell in distance kX from the base of the stack. The meeting happens after a_2 traversed $(k+1) \cdot X$ cells (X towards the base, kX away from the base), whereas a_3 traversed $(k-1) \cdot X$ cells.

DivideStackSize(k). Analogously, we can implement division by k by letting a_3 walk *towards* the base, instead of away from the base, i.e., by replacing the first rule for a_3 by

$$(\textsc{Mult}^0_{3,k}, S) \to (\textsc{Mult}^1_{3,k}, \text{south}) \quad \text{for any } S \in 2^Q \text{ satisfying } \textsc{MultBack}^0_{2,k} \notin S$$

while leaving all other rules (for all agents) unchanged. However, the two rules leading to the terminal states require a_2 and a_3 to be in states $\textsc{Multback}^0_{2,k}$ and $\textsc{Mult}^0_{3,k}$, respectively, to ensure termination. If the initial stack size X is divisible by k, then the states of the two agents will align perfectly in the cell c'' in distance X/k from the base of the stack: after $(k-1)(k+1)$ time steps, a_2 has traversed $k+1$ cells with speed $1/(k-1)$, and a_3 has traversed $k-1$ cells with speed $1/(k+1)$, hence both are in cell c'' in the states leading to the terminal states. If, however, X is not divisible by k, then the states of the two agents do not align when they meet again after a_2 visited the base.

IsDivisible(k). Hence, before dividing by k, we will always check whether the current stack size is divisible by k. This can be achieved by having a_2 walk towards the base with speed 1 while increasing a counter modulo k each time it takes a step. If the counter is at 0 when a_2 reaches a_1, the stack size is divisible by k; if not, then the stack size is not divisible by k. The subroutine of checking for divisibility by k terminates after a_2 has walked back to a_3 and informed it whether the current stack size is divisible by k or not.

Further Building Blocks. In order to be able to write our synchronous exploration protocol concisely, it will be useful to define a few other subroutines. As before, we will assume that, in the beginning of the subroutines, agents a_2 and a_3 will be in the same cell c', representing the end of the stack, and a_1 is in a cell c representing the base of the stack that differs from c' only in that its coordinate in dimension 1 is strictly smaller. The only exception will be the subroutine $\textsc{InitializeStackSize}(k)$ that initializes the stack to some positive integer k by having a_2 and a_3 walk k steps away from a_1 – here, all three agents are initially in the same cell. Apart from $\textsc{InitializeStackSize}(k)$, we define the subroutines $\textsc{IncreaseStackSize}(k)$ for positive integers k, and $\textsc{MoveStack}(g, i)$, where $g \in \{-1, 1\}$ and $i \in \{1, \dots, n\}$. Subroutine $\textsc{IncreaseStackSize}(k)$ simply increases the stack size by k (additively) by having a_2 and a_3 walk k steps away from a_1.

A subroutine similar to our $\textsc{MoveStack}(g, i)$ was already introduced in [13]. The purpose of this subroutine is to move the whole stack in some direction specified by dimension i and sign g. In our definition, $\textsc{MoveStack}(g, i)$ moves every agent to a new cell that differs from the old cell only by having its ith coordinate increased by g, i.e., effectively each agent takes one step in dimension i. However, one has to be a bit careful when implementing this subroutine as we want to be able to concatenate it with other subroutines. In particular, in all other subroutines, agent a_1 does not know when the subroutine is started or terminates, while the other agents do know. In order to also obtain this property for $\textsc{MoveStack}(g, i)$, we implement the desired movement by having a_2 walk towards a_1, notifying it about the desired step and the chosen direction (upon which a_1 performs the step) and then returning to a_3, where both a_2 and a_3 perform the desired step as well.

Semi-Synchronous Agents. All of the above subroutines can also be performed by (at most) 4 semi-synchronous agents, as we show in the following. Similar to the approach in [13], we will use one agent (a_4) to effectively synchronize the behavior of the other agents, which allows us to essentially execute the 3-agent synchronous subroutines described above with the remaining 3 agents. In more detail, agent a_4 will visit the other agents in a suitable order, and each of the other agents will only move when they are in the same cell as a_4 (while a_4 will not leave the cell of the agent it wants to move next until the agent actually left the cell). We start by showing how this can be achieved for subroutine $\textsc{MultiplyStackSize}(k)$.

As in the synchronous version of the subroutine, we would like the two agents a_2 and a_3 to move with (relative) speeds $1/(k-1)$ (first towards a_1 and, after meeting a_1, away from a_1) and $1/(k+1)$ (away from a_1), respectively, while a_1 simply stays at the base of the stack. The purpose of this design – that when a_2 and a_3 meet next, they are in a cell that has the k-fold distance to a_1 as they have currently – can also be achieved by having a_3 move $k-1$ steps, then having a_2 move $k+1$ steps, and so on, always alternating between the two agents, until they are both in the same cell again (which, by their relative "speeds" must have the desired distance to the base of the stack). This behavior can be ensured by using a_4:

Agents a_2 and a_3 follow their designated route, but they only take one step of those routes if they are in the same cell as a_4 and a_4 is in a state indicating that a_2, resp. a_3 should move (the latter condition is not strictly necessary, but simplifies things by ensuring that a_2 and a_3 never move at the same time). Agent a_4 alternates between visiting a_2 and a_3, during each "visit" making sure that the respective agent takes the desired number of steps (i.e., $k-1$ or $k+1$). It does so by going to the cell of the respective agent a_i ($i \in \{2,3\}$), indicating that a_i should take a step of its route, waiting until a_i takes a step and leaves the cell, incrementing an internal counter, following agent a_i to the next cell, and repeating this behavior until the counter indicates that the desired number of steps has been taken by a_i, upon which a_4 visits the other agent a_{5-i}. Note that a_4 always knows in which direction it has to move to find the desired agent as the coordinates of a_2 and a_3 only differ in dimension 1, and a_2 always has a smaller (or equally large) first coordinate. Moreover, a_4 also knows in which direction it has to go to follow the agents to the next cell as the only change in direction is performed by a_2 and the reason for the change, namely meeting a_1, is an information known to a_4 since when a_2 meets a_1, it stays in the cell containing a_1 until a_4 also arrives there.

In an analogous fashion, we can implement DivideStackSize(k) with 4 semi-synchronous agents. For the other four subroutines, the picture is even simpler: it is straightforward to check that these subroutines can already be implemented by 3 semi-synchronous agents by having the agents perform the same steps as in the respective synchronous subroutines. The reason that these subroutines also work in the semi-synchronous setting is that either the synchronous version already contain one agent that effectively acts as a synchronizer (in the sense that every action is performed by that agent or directly instigated by a visit of that agent), as in IsDivisible(k) and MoveStack(g, i), or the actions of the agents are independent of each other, as in InitializeStackSize(k) and IncreaseStackSize(k). For these four subroutines, we will simply assume that a_4 is treated the same as a_3; in particular, at the beginning and end of each subroutine, we will always have a_2, a_3, and a_4 in the same cell, indicating the end of the stack.

We have to be careful with the termination of each subroutine as we want to be able to concatenate the subroutines. To this end, we will again use a_4 as a synchronizer: before terminating itself, a_4 will wait that a_2 and a_3 (which are in the same cell at the end of each subroutine) have terminated. Similarly, we can assume that a_4 will initialize the next subroutine by changing its state suitably, thereby making sure that the start and end of the subroutines align across all agents. A last detail is that in the semi-synchronous version of MoveStack(g, i) (which is the only subroutine where a_1 moves), after meeting a_1, agent a_2 has to wait until a_1 takes its step before moving back to a_3 and a_4, in order to make sure that a_4 does not terminate and initialize the next subroutine before a_1 takes its step.

4 The Exploration Protocol

In this section, we will combine the building blocks of Section 3 to a protocol that allows 3 synchronous, resp. 4 semi-synchronous, agents to explore the n-dimensional (oriented) grid, and prove the protocol's viability. Our protocol is given by algorithm Explore.

Algorithm 1 Explore.

```
InitializeStackSize(3)
repeat
    for each function $g : \{1, \dots, n\} \to \{-1, 1\}$ do
        FollowRoute($g$)
        FollowRoute($-g$)
    end for
    IncreaseStackSize(2)
procedure FollowRoute($g$)
    for $i = 1$ to $n$ do
        while IsDivisible($p_i$) do
            DivideStackSize($p_i$)
            MultiplyStackSize(2)
            MoveStack($g(i), i$)
        end while
        while IsDivisible(2) do
            DivideStackSize(2)
            MultiplyStackSize($p_i$)
        end while
    end for
end procedure
```

The underlying idea of algorithm Explore is the same as in the algorithms from [13]: We generate each (non-zero) n-dimensional vector $(v_1, \dots, v_n)$ with non-negative integer coordinates, and for each such vector, we let one agent walk from the origin to each cell $(c_1, \dots, c_n)$ such that $c_i \in \{v_i, -v_i\}$ for all $1 \le i \le n$, and then back to the origin. More precisely, in each execution of FollowRoute(g), agent a_1 walks to the respectively specified cell $(c_1, \dots, c_n)$, and in each execution of FollowRoute($-g$), a_1 walks back to the origin. To generate $(v_1, \dots, v_n)$, a counter, represented by the stack size, is used that is incremented gradually, thereby iterating through the positive integers. Each time the counter is incremented, the new value X will be transformed into some n-dimensional vector, $(v_1, \dots, v_n)$, where the design of the transformation has to make sure that every (non-zero) vector with non-negative integer coordinates is generated by some value X. We require the stack size to be odd, as we will be using powers of 2 to store some intermediate values. Thus, we will always increase the counter by 2, while still ensuring that all vectors are generated.

However, as we have one fewer agent available than in the protocols in [13], our protocol requires a new way to implement this idea. In particular, we avoid using a separate agent to remember the stack size when the stack is read, instead making sure that even after the stack is read, no information about the previous stack size(s) is lost[2]. To this end, we define the vector $(v_1, \dots, v_n)$ we want to transform X into as follows. Let $p_1, \dots, p_n$ denote the first n odd primes, where $p_1 < \dots < p_n$. For all $1 \le i \le n$, we define v_i to be the largest non-negative integer such that $p_i^{v_i}$ divides X. In other words, v_i represents how often p_i occurs as a prime factor of X.

---

[2] Note that such information is still required after reading the stack: we will need it both to guide a_1 back to the origin and to retrieve the counter value X that we want to increase repeatedly.

Consider procedure FollowRoute(g). The for loop of this procedure iterates through the n dimensions. For each dimension i, the first while loop repeatedly replaces one prime factor p_i by prime factor 2, by dividing by p_i and multiplying by 2. Each time such a replacement is performed, the whole stack is moved one cell w.r.t. dimension i (either increasing or decreasing the respective coordinate by 1, depending on the value of $g(i)$). After all (i.e., v_i) occurrences of p_i as prime factors have been replaced by factors 2, the stack manipulations are reversed in the second while loop, resulting in the original stack size X. Note that in the very beginning of algorithm Explore, the stack size is initialized to 3, and each time the counter represented by the stack size is increased, it is increased by 2; hence, before starting the first while loop, the stack size is odd, ensuring that the second while loop goes through exactly the same number of iterations as the first one. Note further that we do not revert the steps that a_1 took (yet) when reversing the stack manipulations. After iterating through all dimensions, agent a_1 is now in cell $(c_1, \ldots, c_n)$, and we can consider this cell as explored, concluding the execution of FollowRoute(g).

The execution of FollowRoute($-g$) is identical to the execution of FollowRoute(g), except that each step of a_1 is performed in the opposite direction. Hence, at the end of the execution of FollowRoute($-g$), agent a_1 is back at the origin, while the stack size is (again) X. The (outer) for loop in algorithm Explore simply iterates through all possible assignments of signs $\in \{-1, +1\}$ to the dimensions, making sure that for each generated vector $(v_1, \ldots v_n)$, each corresponding cell $(v'_1, \ldots, v'_n)$ is explored.

In the following, we state our main result for oriented grids. The proof is given in the full version of this work.

▶ **Theorem 1.** *For any positive integer n, the n-dimensional (oriented) grid can be explored by 3 synchronous finite automata, resp. 4 semi-synchronous finite automata.*

5 Unoriented Grids

In [13], the authors showed that any protocol for the oriented grid can be transformed into a protocol for unoriented grids by adding sufficiently many agents such that, at all times, each original agent moving across a non-constant distance is accompanied by one of the additional agents. In particular, for both their protocol and our improved protocol, this implies that 2 additional agents are required in the synchronous case and 1 additional agent in the semi-synchronous case (since in the protocols for the oriented grid, 2 synchronous agents are traversing non-constant distances at the same time, while in the semi-synchronous case only 1 agent does so). Hence, our protocol for the oriented grid improves also the state of the art for the minimum number of required agents on *unoriented* grids from 6 to 5 (in both the synchronous and the semi-synchronous setting).

On an informal level, it seems unlikely that our upper bound of 5 can be improved since intuitively, as Dobrev et al. [14, Section 7, arXiv] write, “a lone agent cannot cross any non-constant distance, as the irregular nature of the port labels would lead it astray, never to meet any other agent”. The tightness of our bound on the oriented grid combined with the perceived necessity of having moving agents accompanied by a partner seems to indicate that we cannot do better. However, there is no formal proof of any lower bound beyond the synchronous 3-agent and semi-synchronous 4-agent lower bounds [7, 15] that carry over from the case of the oriented grid. Admittedly, as such a formal lower bound might require us to find a “bad” input instance (i.e., a bad input edge labelings of the infinite n-dimensional grid) for every potential protocol with more than 3 synchronous, resp. 4 semi-synchronous, agents, it is not particularly surprising that we do not have better lower bounds – yet, making at least some progress would be desirable.

In this section, we will show that under a natural assumption the current lower bounds are actually optimal by providing tight upper bounds. Our assumption states that you cannot walk in a cycle if you always follow the same direction, or, more formally:

▶ **Assumption 4.** *Let* $\ell \in \{1, \dots, 2n\}$ *be any port,* z *any positive integer, and* $c_0, \dots, c_z$ *any sequence of cells such that, for each* $0 \leq j \leq z - 1$*, we reach cell* c_{j+1} *by leaving cell* c_j *via port* ℓ*. Then* $c_0 \neq c_z$*.*

Surprisingly, this assumption does not contradict the intuition about agents traveling alone discussed above, yet it still allows us to prove upper bounds for unoriented grids matching the lower bounds obtained on *oriented* grids. While our upper bounds answer the question for the minimally required number of agents in a natural[3] setting very close to truly unoriented grids, we think that they also constitute a useful step on the way to a lower bound construction for the general unoriented setting (assuming that the current lower bounds are not optimal): any such construction necessarily has to contain cycles that violate Assumption 4.

Our Approach. The general idea behind our approach is to find a way to construct a stack also for unoriented grids. The natural idea of simply selecting one port ℓ and interpreting the sequence of cells obtained by successively leaving cells via port ℓ as the stack does not work: while it is easy for an agent to traverse the stack in the direction away from the base (it just has to leave each cell via port ℓ), traversing the stack in the opposite direction runs into the problem that there might be different neighboring cells from which the current cell can be reached via port ℓ and the traversing agent cannot know which is the one that belongs to the intended stack. Instead, we will build the desired virtual stack by constructing an auxiliary (infinite) directed labeled forest and then traversing (a part of) the forest from some starting cell in a DFS-like fashion, which will ensure that agents can traverse the stack in both directions. In particular, the same cell can occur in the stack several times; to distinguish the occurrences (and make it possible for an agent to traverse the virtual stack), our stack will formally consist of pairs (cell, integer), where the integers come from the set $\{1, \dots, 2n\}$. For an illustration of the auxiliary graph and the virtual stack, we refer to Figures 1 and 2.

The Auxiliary Graph. We start by defining our auxiliary graph $G = (V, E)$. The vertices of G are the cells of our grid, and we have a directed edge (c, c') between two cells c, c' if c and c' are neighbors in the grid and cell c' is reached by leaving cell c via port 1.[4] In particular, this implies that each cell c has exactly one outgoing edge in G; we call the cell c' reached by traversing this edge the *parent of* c, and c a *child of* c'. Note that Assumption 4 ensures that G does not contain cycles, and hence, is an infinite forest. In particular, for any two neighboring cells c, c', at most one of the two possible edges (c, c') and (c', c) is present in E.

Let indegree(c) denote the *indegree* of a cell c, i.e., the number of edges from E incoming to c. We assign to each edge $e = (c, c')$ a level $L(e)$ as follows. For each cell c', we order the incoming edges (c, c') increasingly by the corresponding port of c', and then assign (distinct) *levels* from 1 to indegree(c') to the edges according to this order. For instance, if c' has two incoming edges (c, c') and (c'', c'), corresponding to ports 5 and 3 of c', respectively, then the order of the edges will be (c'', c'), (c, c'), and we will assign level 1 to (c'', c'), and level $2 = \text{indegree}(c')$ to (c, c').

---

[3] After all, it seems like a reasonable minimal requirement for a sense of direction that if you go north (or in any other direction) repeatedly, then you do not return to the starting point.

[4] The choice of port 1 here is arbitrary; choosing any other label from $\{1, \dots, 2n\}$ works equally well.

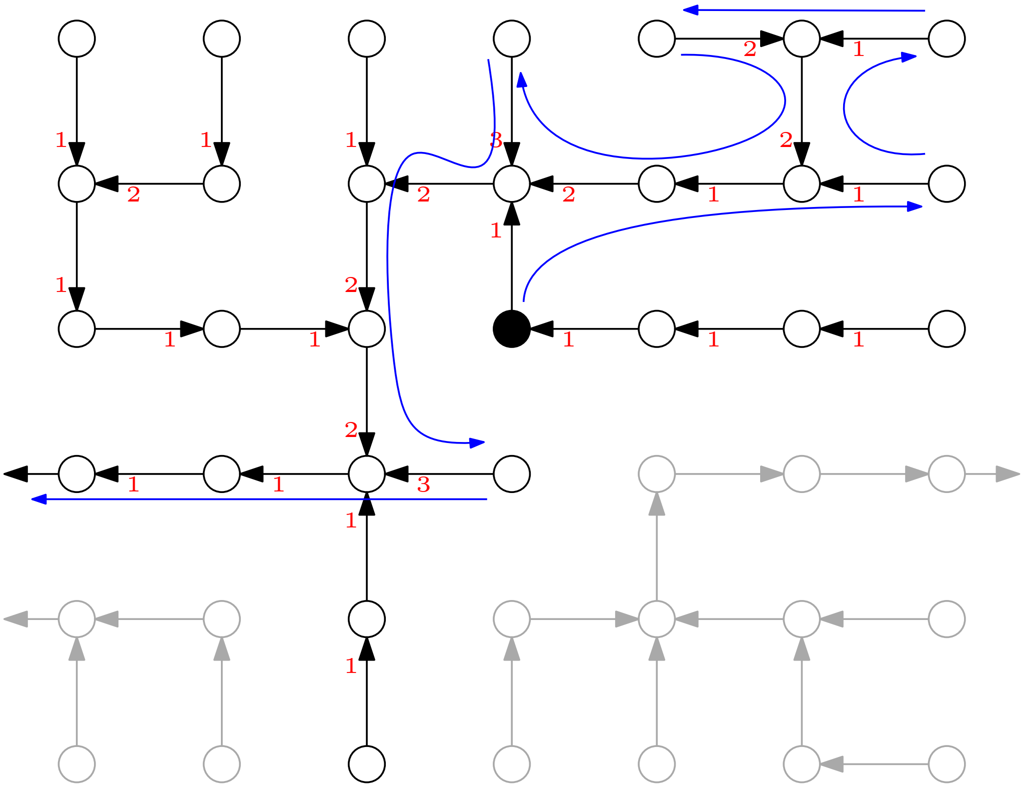

Figure 1 Figure 1 depicts a part of a possible auxiliary graph G for a 2-dimensional unoriented grid and the respective virtual stack. Vertices, i.e., cells, are represented by circles, and directed edges by arrows. The parts grayed out belong to different trees than the one containing the cell where a_1 is located (colored black). The edges are labeled with their respective levels. The physical cells of the virtual stack rooted in the black cell (i.e., the first component of the pairs the virtual stack consist of) are indicated by the route that starts in the black cell and follows the blue arrows. Each further step on this route leads to the physical cell corresponding to the next higher position in the stack, where the black cell indicates position 0. For each occurrence of a cell on this route, the corresponding level (i.e., the second component of the pairs the virtual stack consists of) is 1 if the current cell is a child of the previous cell, and equal to the level of the arrow traversed last plus 1 if the current cell is the parent of the previous cell. When the blue route goes from a cell to its parent, then the edge traversed next will have a level that is higher by 1 than the previously traversed edge; when the route goes from a cell to one of its children, then the edge traversed next has level 1. This leads to a virtual stack that corresponds to a part of a DFS exploration on the (infinite) tree containing the black cell, as can be seen in Figure 2, where the same route on the same auxiliary graph is depicted as a rooted tree.

The Virtual Stack. Using the auxiliary graph G, we now define, for each cell c, the *virtual stack* Virt_c *rooted in* c as follows. Recall that Virt_c consists of pairs (cell, integer). We will use the functions $\text{Cell}(\cdot)$ and $\text{Level}(\cdot)$ to retrieve the first, resp. second, component of such a pair. The base of the stack is defined as $\text{Virt}_c[0] := (c, \text{indegree}(c) + 1)$. For each integer $j \geq 1$, we inductively define $\text{Virt}_c[j]$ according to the following case distinction.

- If $\text{Level}(\text{Virt}_c[j-1]) = \text{indegree}(\text{Cell}(\text{Virt}_c[j-1])) + 1$, then
    - $\text{Cell}(\text{Virt}_c[j])$ is defined as the parent of $\text{Cell}(\text{Virt}_c[j-1])$, and
    - $\text{Level}(\text{Virt}_c[j]) := L((\text{Cell}(\text{Virt}_c[j-1]), \text{Cell}(\text{Virt}_c[j]))) + 1$.

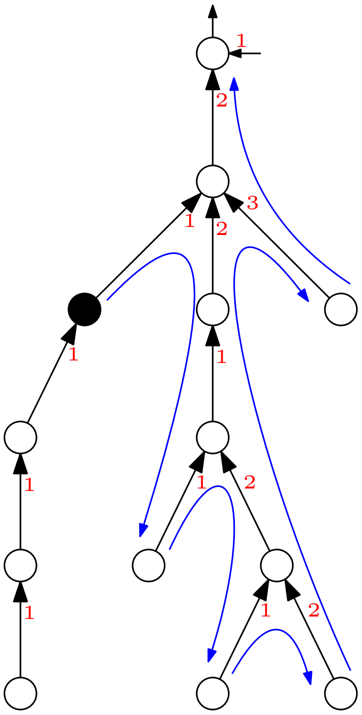

Figure 2 The same virtual stack as in Figure 1, depicted as a rooted tree.

- If $\mathrm{Level}(\mathrm{Virt}_c[j-1]) \leq \mathrm{indegree}(\mathrm{Cell}(\mathrm{Virt}_c[j-1]))$, then
  - $\mathrm{Cell}(\mathrm{Virt}_c[j])$ is defined as the child of $\mathrm{Cell}(\mathrm{Virt}_c[j-1])$ connected to $\mathrm{Cell}(\mathrm{Virt}_c[j-1])$ via an (outgoing) edge of level $\mathrm{Level}(\mathrm{Virt}_c[j-1])$, and
  - $\mathrm{Level}(\mathrm{Virt}_c[j]) := 1$.

In other words, we inductively build the virtual stack rooted in c as follows. We start in c and leave c via the unique outgoing edge. Each time we enter a cell c' via an incoming edge, i.e., coming from a child c'', the next cell we visit is the next higher child of c', i.e., the child that is connected to c' via an edge of level $L(c'', c') + 1$. If no higher child remains, i.e., if (c'', c') has level $\mathrm{indegree}(c')$, then the next cell we visit is the parent of c'. Each time we enter a cell c' from its parent, the next cell we visit is the first child of c', i.e., the child that is connected to c' via an edge of level 1. Hence, our stack corresponds to a DFS exploration of the tree in forest G containing c, where we assume that the part of the DFS that is executed before traversing the edge from c to its parent has already happened. As the tree is infinite, we may not reach every cell contained in the tree in finite time, but to use such a DFS exploration as a stack, this is not relevant. What is relevant, however, is that once we traverse an edge from a child to its parent, the DFS will never return to the child in finite time as Assumption 4 ensures that the parent chain starting from c (and therefore also any parent chain starting from any other cell visited by the partial DFS) is infinite. Combining this fact with the cyclic fashion in which each visited cell iterates through its children and parent to determine the neighbor visited next, we obtain the following observation.

▶ **Observation 5.** *Fix an arbitrary cell c. For any two non-negative integers $i \neq j$, we have $\mathit{Virt}_c[i] \neq \mathit{Virt}_c[j]$.*

In order to make use of the defined virtual stack, we need the agents to be able to represent their position in the stack in some way. However, given the specific design of the virtual stack, this is not difficult: each agent allocates a part of its state to keep track of the level $\mathrm{Level}(\mathrm{Virt}_c[j])$ of the current position $\mathrm{Virt}_c[j]$ in the stack, while the first component $\mathrm{Cell}(\mathrm{Virt}_c[j])$ of the current position in the stack is simply represented by the cell the agent currently occupies. An advantage of this design is that each agent a_i can determine which other agents are in the same stack position as a_i, and which are not (despite possibly being in the same physical cell). In other words, each agent has all the necessary information to evaluate its transition function, even for moving on the virtual stack.

However, there is one piece still missing for using the virtual stack similar to a physical stack: we have to show that even a lone agent can traverse the virtual stack in either direction, i.e., that a finite automaton is sufficient to determine the physical cell that corresponds to the previous, resp. subsequent, position in the virtual stack, and similarly, to determine the level of that stack position. The following lemma takes care of this.

▶ **Lemma 6.** *There is a finite automaton that, when located in cell* $\mathit{Cell}(\mathit{Virt}_c[j])$ *in state* $(i, \mathit{Level}(\mathit{Virt}_c[j]))$*, where* c *is an arbitrary cell,* $i \in \{-1, 1\}$*, and* $j \geq 1$ *an arbitrary integer, moves to cell* $\mathit{Cell}(\mathit{Virt}_c[j+i])$ *and changes its state to* $(i, \mathit{Level}(\mathit{Virt}_c[j+i]))$ *in* 2 *time steps.*

Note that when applying Lemma 6, the finite automaton from Lemma 6 will only constitute a part of the finite automaton governing our agents in the final protocol for the unoriented case. Using Observation 5, Lemma 6, and the so-called handrail technique discussed in the full version, we are finally set to prove Theorem 2.

▶ **Theorem 2.** *Suppose that Assumption 4 holds. Then, for any positive integer* n*,* 3 *synchronous finite automata, resp.* 4 *semi-synchronous finite automata, suffice to explore any* n*-dimensional unoriented grid.*

6 Open Problems

While we provided tight bounds for a number of settings, still a number of open questions remain: Can we prove a higher lower bound on the number of agents required to explore an unoriented grid, than we can for oriented grids? Is there a protocol that achieves both an optimal number of agents and polynomial time exploration? For $n \geq 3$, can we improve the semi-synchronous protocol using randomness (the best known lower bound states that at least 3 agents are required)? How much can we reduce the computational power of the agents without compromising the optimal bounds? In our protocols, we can, e.g., replace agent a_1 with a movable marker, and in the semi-synchronous protocols, we can replace all agents except one with a movable marker; can we allow further/other restrictions?

References

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

2 Romas Aleliunas, Richard M. Karp, Richard J. Lipton, Laszlo Lovasz, and Charles Rackoff. Random Walks, Universal Traversal Sequences, and the Complexity of Maze Problems. In *Foundations of Computer Science (FOCS)*, pages 218–223, 1979.

3 Ricardo A. Baeza-Yates, Joseph C. Culberson, and Gregory J. E. Rawlins. Searching in the Plane. *Information and Computation*, 106:234–252, 1993.

4 Anatole Beck. On the Linear Search Problem. *Israel Journal of Mathematics*, 1964.

5 M. Blum and W. J. Sakoda. On the Capability of Finite Automata in 2 and 3 Dimensional Space. In *Foundations of Computer Science (FOCS)*, pages 147–161, 1977.

6 Manuel Blum and Dexter Kozen. On the Power of the Compass (or, Why Mazes Are Easier to Search Than Graphs). In *Foundations of Computer Science (FOCS)*, pages 132–142, 1978.

7 Sebastian Brandt, Jara Uitto, and Roger Wattenhofer. A Tight Lower Bound for Semi-Synchronous Collaborative Grid Exploration. In *International Symposium on Distributed Computing (DISC)*, pages 13:1–13:17, 2018.

8 Lothar Budach. Automata and Labyrinths. *Mathematische Nachrichten*, 86(1):195–282, 1978.

9 Lihi Cohen, Yuval Emek, Oren Louidor, and Jara Uitto. Exploring an Infinite Space with Finite Memory Scouts. In *Symposium on Discrete Algorithms (SODA)*, pages 207–224, 2017.

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

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

12 Yann Disser, Jan Hackfeld, and Max Klimm. Undirected Graph Exploration with $\Theta(\log \log n)$ Pebbles. In *Symposium on Discrete Algorithms (SODA)*, pages 25–39, 2016.

13 Stefan Dobrev, Lata Narayanan, Jaroslav Opatrny, and Denis Pankratov. Exploration of High-Dimensional Grids by Finite Automata. In *International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 139:1–139:16, 2019.

14 Stefan Dobrev, Lata Narayanan, Jaroslav Opatrny, and Denis Pankratov. Exploration of high-dimensional grids by finite state machines. *CoRR*, abs/1902.03693, 2019. `arXiv:1902.03693`.

15 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.

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

17 Ofer Feinerman, Amos Korman, Zvi Lotker, and Jean-Sebastien Sereni. Collaborative Search on the Plane Without Communication. In *Principles of Distributed Computing (PODC)*, pages 77–86, 2012.

18 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.

19 Frank Hoffmann. One Pebble Does Not Suffice to Search Plane Labyrinths. In *FCT*, pages 433–444, 1981.

20 Alejandro López-Ortiz and Graeme Sweet. Parallel Searching on a Lattice. In *CCCG*, pages 125–128, 2001.

21 Petrişor Panaite and Andrzej Pelc. Exploring Unknown Undirected Graphs. In *Symposium on Discrete Algorithms (SODA)*, pages 316–322, 1998.

22 H. A. Rollik. *Automaten in Planaren Graphen*, pages 266–275. Springer Berlin Heidelberg, Berlin, Heidelberg, 1979.

23 Kazuo Sugihara and Ichiro Suzuki. Distributed Algorithms for Formation of Geometric Patterns with many Mobile Robots. *Journal of Robotic Systems*, 13(3):127–139, 1996.

24 Ichiro Suzuki and Masafumi Yamashita. Distributed Anonymous Mobile Robots: Formation of Geometric Patterns. *SIAM Journal on Computing*, 28(4):1347–1363, 1999.

25 Ichiro Suzuki and Masafurni Yamashita. Distributed Anonymous Mobile Robots - Formation and Agreement Problems. In *Structural Information and Communication Complexity (SIROCCO)*, pages 1347–1363, 1996.