Patrolling Grids with a Bit of Memory

Michael Amir[1,2], Dmitry Rabinovich[2], and Alfred M. Bruckstein[2]

[1] University of Cambridge, Cambridge, United Kingdom ma2151@cam.ac.uk
[2] Technion, Haifa, Israel {dmitry.ra@, freddy@cs.}technion.ac.il

Abstract. This work addresses the challenge of patrolling regular grid graphs of any dimension using a single mobile agent with minimal memory and limited sensing range. We show it is impossible to patrol some grid graphs with 0 bits of memory, regardless of sensing range, and give an exact characterization of those grid graphs that can be patrolled with 0 bits of memory and sensing range V. On the other hand, we show that an algorithm exists using 1 bit of memory and $V = 1$ that patrols any d-dimensional grid graph. This result is surprising given that the agent must be able to move in $2d$ distinct directions to patrol, while 1 bit of memory allows specifying only two directions per sensory input. Our 1-bit patrolling algorithm handles this by carefully exploiting a small state-space to access all the needed directions while avoiding getting stuck. Overall, our results give concrete evidence that extremely little memory is needed for patrolling high-dimensional regular environments.

1 Introduction

Patrolling is a key problem in robotics wherein a mobile agent or team of mobile agents are tasked with repeatedly visiting every vertex of a graph environment by traversing edges. This task has well-known applications to navigation, warehouse management, web crawling, and swarm intelligence [14]. Patrolling has been studied under diverse sets of assumptions regarding e.g. the number of agents, the capabilities of each agent, and the underlying graph environment [17]. A central question related to patrolling is the *space complexity* of patrolling an environment: the amount of memory required by the agent(s) to patrol the environment [17,12]. Rectangular grid graphs are fundamental in robotics, and are the subject of many works in patrolling [5,8,11,15]. Despite this fact, and despite the space complexity of patrolling being widely studied, the space complexity of patrolling grid graphs by a single agent has not yet been established. Hence, the goal of this work is to establish the space complexity of patrolling d-dimensional grid graphs with a single mobile agent that has limited visibility.

Specifically, suppose a mobile agent with fixed orientation, $\mathcal{R}$, is placed somewhere inside a grid graph. We assume it has b bits of state memory persisting between steps and the ability to see locations at Manhattan distance V or less from itself. $\mathcal{R}$ must act based only on this information, i.e., it is a *finite automaton enhanced with local geometric information*. For what values of b and V does there exist an algorithm that enables $\mathcal{R}$ to patrol the grid?

One might expect that existing space complexity results for broader classes of environments give good complexity bounds for grid graphs as a special case - but this does not appear to be true. The strongest result we found in the literature

implies a d-dimensional grid graph G of diameter $\text{diam}(\mathcal{G})$ can be patrolled with $\mathcal{O}(\text{diam}(\mathcal{G}) \cdot log(d))$ bits [12,13]. Generally, existing works set the expectation that the complexity of patrolling a given graph environment grows with diameter and maximum vertex degree (see "Related Work"). On the contrary, we show that, irrespective of diameter and maximum vertex degree (i.e., dimension), there is an algorithm using 1 bit of memory that patrols all grid graphs.

Theorem 3.1 and Theorem 4.3 together characterize exactly which grid graphs can be patrolled with sensing range V and b bits of memory, settling our main question. Theorem 3.1 gives an exact characterization of d-dimensional grid graphs that can be patrolled with sensing range V and 0 bits of memory. These turn out to be grid graphs that are bounded in all dimensions but at most one, and that have an even number of "sensing regions" (regions of vertices that are indistinguishable to $\mathcal{R}$).

Theorem 4.3 establishes the correctness of an algorithm that patrols any d-dimensional grid graph with sensing range $V = 1$ and 1 bit of memory. This is despite the fact that, to patrol a d-dimensional grid graph, $\mathcal{R}$ is required to move in $2d$ directions (it must be able to increment and decrement its coordinate along any of the d axes of the grid), which might suggest that the minimum memory required should grow with d, and the fact that in a large grid graph, most locations are geometrically indistinguishable to $\mathcal{R}$, hence it is unable to localize itself most of the time. Hence, we consider this result quite unexpected.

The patrolling algorithm we describe is called `MakeMove`. We must be more careful when patrolling low-dimensional grids than high-dimensional grids, because low-dimensional grids have fewer unique grid boundaries, and so there is less geometric information for $\mathcal{R}$ to exploit. For this reason, `MakeMove` handles 1-, 2-, and 3-dimensional grids as special cases, but extends generically to higher dimensions by exploiting the increased number of grid boundaries. The core of the algorithm is in patrolling 2D sub-grids (subspaces of the d-dimensional grid of interest), and then incrementing or decrementing the robot's coordinate in some higher dimension. However, due to memory constraints, the sequence of higher-dimensional coordinate changes is not trivial, and the robot may, e.g., patrol part of a lower-dimensional subgrid only to depart and return to it later. This is to maintain a small economy of *transient states* that are used by the robot to access higher dimensions.

Several of our techniques can potentially be applied to non-grid graph settings: first, the idea of partitioning the environment into sensing regions readily generalizes (see [3], Appendix B) and may yield further space complexity results. Next, the observation that patrolling, counter-intuitively, does not require more memory in higher-dimensional grids (because their richer, more distinguishable geometry compensates for their dimensionality) seems exploitable, and may lead to space-efficient patrolling algorithms in more general settings.

Due to space constraints, some of our proofs **have been moved to the extended version of this paper [3].**

Related Work. Graph exploration by mobile agents is a key topic in algorithm design, robotics, and web crawling [14]. Exploration of an environment

with mobile agents presents a considerably different challenge from classical graph search settings. In classical settings, it is assumed that the entire graph can be accessed from memory. Mobile agents, on the other hand, must make decisions using limited memory and only local knowledge of the graph environment. Throughout the years, considerable work has been devoted to identifying the minimal capabilities required to enable exploration under a diverse set of assumptions, e.g., single agent [9] versus group of agents [5,7]; exploring graphs with termination conditions [13] versus without [23]; competitive [22] or cooperative [15]. In this work we focus on non-terminating grid exploration by a single finite automaton, i.e., *patrolling*, where the aim of the exploration is to ensure every graph vertex is periodically visited by the agent deterministically.

Exploration of known and unknown graphs using finite automata with limited visibility has been studied under various assumptions. In [12,13], it is shown that a graph G with maximum degree d and of diameter $\mathrm{diam}(\mathcal{G})$ can be patrolled with $\mathcal{O}(\mathrm{diam}(\mathcal{G}) \cdot log(d))$. Our problem setting can be considered a special case of [12,13]'s model that we analyse very carefully, attaining better bounds. In a seminal work by Budach et al. [6] it is shown that no finite automaton exists that can find its way out of any 2D maze. More recently, Kilibarda has shown that any automaton walking rectangular labyrinth can be reduced to an automaton following either left-, or right-hand-on-the wall approach [18]. It is known that finite automata can patrol graphs if the graph is pre-processed so as to label the edges [9]. Exploring arbitrary graphs without such pre-processing is known to require considerably more memory, and, in most problem settings, cannot be done by finite automata [10,12,13].

The problem of low-complexity patrolling of regular grid graphs that we study in this paper has surprising applications in autonomous traffic management [20], where it can be used for high-level coordination of fleets of cars in a low-complexity, error-resilient fashion, and has been studied in the context of optimal paths for lawn mowing by a single robot [4].

Our results show that, for the purposes of grid exploration, an agent with 1 bit of memory is significantly more powerful than an agent with 0 bits of memory (also called an "oblivious" agent). Some other recent works have shown similar gaps between 0- and 1-bit agents in graph search tasks such as deciding graph properties [17] and exploring cactus graphs [23]. The authors find fascinating this enormous leap in capabilities between memoryless agents and agents that have any internal memory at all, even a single bit.

2 Problem Statement

A d-dimensional grid graph $\mathcal{Q}$ is a graph whose vertex set is $[n_1] \times [n_2] \times \ldots [n_d]$, $\forall i, n_i > 1$, where $[n]$ is the set of integers k such that $1 \leq k \leq n$, and where there is an edge between any two vertices at Manhattan distance 1 from each other. A mobile agent, $\mathcal{R}$, is initialized at some arbitrary vertex $v_0 \in \mathcal{Q}$ and is tasked with visiting every vertex in $\mathcal{Q}$ by traversing edges. Hence, $\mathcal{R}$ traverses $\mathcal{Q}$ by incrementing or decrementing one of d different coordinates in unit steps.

Our model of movement and sensing is standard in previous works on graph patrolling (when these works' setting is restricted to a grid) and on robot nav-

igation [14,16,1,2,21]. We assume $\mathcal{R}$ has fixed orientation (i.e., a "compass" - it distinguishes between the d axes of $\mathcal{Q}$ and chooses which one it wants to move along) and that, when $\mathcal{R}$ is located at p, it *senses* vertices of $\mathcal{Q}$ at Manhattan distance V or less from p. $\mathcal{R}$ knows the *position relative to* p of every vertex of $\mathcal{Q}$ it senses, given by the set:

$$\mathbb{S}(V,p) := \{\, p' - p \mid p' \in \mathcal{Q}, \|p' - p\|_1 \leq V \,\}$$

where $p' - p$ is the position of p' relative to p. V is called $\mathcal{R}$'s *sensing* or *visibility range*. We further assume $\mathcal{R}$ has access to b bits of persistent state memory which it can access and modify - these bits represent a persistent state that $\mathcal{R}$ maintains between steps. At every step, $\mathcal{R}$ must decide its next action *deterministically* based only on $\mathbb{S}(V,p)$ and its b bits of memory. Hence, it can be considered a finite automaton enhanced with local geometric information. More formally, $\mathcal{R}$'s algorithm is restricted to be a function $\mathrm{ALG}(\mathbb{S}, mem)$ that takes as the input $\mathcal{R}$'s current sensing data $\mathbb{S}(V,p)$ and memory state, and outputs $\mathcal{R}$'s next step and memory state.

When two vertices p and p' have $\mathbb{S}(V,p) = \mathbb{S}(V,p')$, they are indistinguishable from $\mathcal{R}$'s perspective and thus must be treated identically by its traversal algorithm. Figure 1 illustrates, in 2- and 3-dimensional grid graphs, regions of vertices for which $\mathbb{S}(V,p)$ is identical given sensing range $V = 1 \ldots 3$. The closer $\mathcal{R}$ is to a boundary of $\mathcal{Q}$ the smaller the set $\mathbb{S}(V,p)$ becomes, because $\mathbb{S}(V,p)$ only contains locations inside $\mathcal{Q}$. Hence, the closer $\mathcal{R}$ is to a boundary, the more information it has about its current position.

The central question we wish to answer is: given sensing range V and b bits of memory, does there exist a *deterministic* function $ALG(\cdot,\cdot)$ that enables $\mathcal{R}$ to visit every vertex in $\mathcal{Q}$? Formally:

Definition 2.1. *An algorithm* ALG *is said to* ***patrol*** $\mathcal{L} \subset \mathcal{Q}$ *if, given any initial position* $v_1 \in \mathcal{L}$ *and initial memory state, executing* ALG *causes* $\mathcal{R}$ *to visit all vertices* $v \in \mathcal{L}$ *within a finite number of steps.*

Note that Definition 2.1 implies, in particular, that ALG will visit each vertex of $\mathcal{L}$ an unlimited number of times (if we let it run indefinitely), hence the term "patrolling". We are generally interested in the case $\mathcal{L} = \mathcal{Q}$ (i.e., patrolling the entire grid graph), but shall sometimes let $\mathcal{L}$ be a strict subset of $\mathcal{Q}$ in proofs.

3 Patrolling with 0 Bits of Memory

We first consider the problem of visiting every vertex in $\mathcal{Q}$ assuming $\mathcal{R}$ has sensing range V and 0 bits of internal state memory. We shall show that, in this setting, whether $\mathcal{R}$ is capable of patrolling $\mathcal{Q}$ depends on the dimensions of $\mathcal{Q}$. Our main result is the following exact characterization of the set of grid graphs that can be patrolled:

Theorem 3.1. *Let* $\mathcal{Q} = [n_1] \times [n_2] \times \ldots [n_d]$. *There exists an algorithm that patrols* $\mathcal{Q}$ *with* 0 *bits of memory and* V *sensing range if and only if:*

1. *There is at most one index* i *such that* $n_i > 2V + 1$, *and*

2. $\prod_{i=1}^{d} \min\left(n_i, 2V+1\right)$ *is even or equals* 1 *(i.e.,* $\mathcal{Q}$ *contains a single vertex).*

Condition (1) of Theorem 3.1 says that patrollable d-dimensional grids are bounded in all dimensions but at most one, and Condition (2) says they must have an even number of "sensing regions". Sensing regions - to be formally defined later in this section - are regions of vertices that are indistinguishable to $\mathcal{R}$ (see Figure 1). The idea behind the proof of Theorem 3.1 is to partition $\mathcal{Q}$ into sensing regions and to show that if there is an odd number of such regions, or if any such region contains a 2×2 subregion, $\mathcal{Q}$ cannot be patrolled. Vice versa, when the conditions of Theorem 3.1 hold, we shall describe explicit algorithms for patrolling $\mathcal{Q}$: algorithms 1 and 2.

One caveat regarding algorithms 1 and 2 is that they require the dimensions of $\mathcal{Q}$ to be known to $\mathcal{R}$ in advance. This does not falsify Theorem 3.1, since the Theorem is concerned with whether an 0-bit algorithm exists for patrolling a *given* grid graph, and we may embed information about said grid graph in the algorithm. However, it is desirable to find algorithms that can patrol $\mathcal{Q}$ without knowing its dimensions. Later, in Section 4, we shall give an algorithm for patrolling any arbitrary d-dimensional grid graph using 1 bit of memory that works generically, without requiring knowledge of $\mathcal{Q}$'s dimensions in advance.

Before proving Theorem 3.1, let us build some intuition about patrolling with 0 bits of memory. In particular, let us show that any algorithm that patrols the grid $\mathcal{Q}$ also finds a Hamiltonian cycle of $\mathcal{Q}$.

Definition 3.2. *Let* ALG *be an algorithm that patrols* $\mathcal{Q}$*. Suppose* $\mathcal{R}$ *executes* ALG *in* $\mathcal{Q}$ *starting from position* v_1*, and let* v_i *be the vertex of* $\mathcal{R}$ *after* i *steps. The* ***walk from*** v_1 ***induced by*** ALG*, denoted* $\mathcal{W}_{\mathrm{ALG}}(v_1)$*, is the walk* $v_1 \dots v_T$ *where* T *indicates the first step at which* $\mathcal{R}$ *has visited all vertices of* $\mathcal{Q}$*. The* ***length of*** $\mathcal{W}_{\mathrm{ALG}}(v_1)$ *is* T*.*

Lemma 3.3. *Suppose* ALG *is an algorithm that patrols* $\mathcal{Q}$ *using* 0 *bits of memory and sensing range* V*. Let* $\mathcal{W}_{\mathrm{ALG}}(v_1) = v_1 \dots v_T$*. Then* $v_1 \dots v_T v_1$ *is a Hamiltonian cycle of* $\mathcal{Q}$*, and* $T = \prod_{i=1}^{d} n_i$*.*

Given some patrolling algorithm ALG, we shall refer to $v_1 \dots v_T v_1$ as the **Hamiltonian cycle from** v_1 **induced by** ALG, denoted $\mathcal{C}_{\mathrm{ALG}}(v_1)$.

Lemma 3.3 says that any algorithm that visits all vertices of $\mathcal{Q}$ with 0 bits of memory must repeatedly traverse a Hamiltonian cycle of $\mathcal{Q}$. As a preliminary result, this fact immediately allows us to prove a special case of Theorem 3.1:

Corollary 3.4 (Special case of Theorem 3.1). *Let* $\mathcal{Q} = [n_1] \times [n_2] \times \dots [n_d]$*. If* $\prod_{i=1}^{d} n_i$ *is odd and greater than* 1*, no algorithm exists that can patrol* $\mathcal{Q}$ *with* 0 *bits of memory.*

Proof. Let $\mathcal{Q} = [n_1] \times [n_2] \times \dots [n_d]$. If $\prod_{i=1}^{d} n_i$ is odd, it is known that no Hamiltonian cycle of $\mathcal{Q}$ exists [19]. But if some algorithm patrols $\mathcal{Q}$ using 0 bits of memory, Lemma 3.3 says a Hamiltonian cycle of $\mathcal{Q}$ exists–contradiction. □

Suppose $\mathcal{R}$ is located at $p = (x_1, x_2 \ldots, x_d)$. We need a way to represent what $\mathcal{R}$ senses along some dimension i at position p. To this end, we project $\mathbb{S}(V, p)$ onto dimension i and measure the distance to the boundaries of $\mathcal{Q}$:

Definition 3.5. *We define a "projection" of* $\mathbb{S}(V, p)$ *onto dimension* i *as follows:*

$$\mathbb{P}_i(V, p) := \{\, p' - p \in \mathbb{S}(V, p) \mid p' \in \mathcal{Q}, p' - p = (0, \ldots 0, x'_i - x_i, 0, \ldots 0) \,\} \quad (1)$$

We further define the ***boundary distances from*** p ***along dimension*** i *as* (l_i, r_i) *where* $l_i = |\{\, p' - p \in \mathbb{P}_i(V, p) \mid p' \in \mathcal{Q}, x'_i < x_i \,\}|$ *and* $r_i = |\{\, p' - p \in \mathbb{P}_i(V, p) \mid p' \in \mathcal{Q}, x'_i > x_i \,\}|$.

l_i and r_i are the number of negative and positive unit steps (respectively) that $\mathcal{R}$ can move in dimension i before hitting a boundary of $\mathcal{Q}$, but they are bounded above by V. When $l_i = r_i = V$, $\mathcal{R}$ cannot tell how far it is from a boundary of $\mathcal{Q}$ along dimension i. When $l_i < V$ or $r_i < V$, however, $\mathcal{R}$ can sense how close it is to a boundary of $\mathcal{Q}$ along dimension i, which helps localization.

Because $\mathcal{R}$ has a sensing range of V, when $\mathcal{Q}$ is large there will inevitably exist indistinguishable vertices p and p' for which $\mathbb{S}(V, p) = \mathbb{S}(V, p')$. Let us define an equivalence relation on such vertices:

Definition 3.6. *Let* $p, p' \in \mathcal{Q}$. *We define the relation* $p \sim p'$ *if* $\mathbb{S}(V, p) = \mathbb{S}(V, p')$. *The equivalence class of* p, *denoted* $\tilde{p} := \{\, p' \mid p' \sim p \,\}$, *is called* ***the sensing region induced by*** p.

Definition 3.7. *Let* $G(\mathcal{Q}, V)$ *be the graph whose vertices are the different sensing regions of* $\mathcal{Q}$ *given sensing range* V, *and where there is an edge* $(\tilde{p}, \tilde{p'})$ *iff there exist vertices* $v \in \tilde{p}$ *and* $v' \in \tilde{p'}$, *where* $\tilde{p}$ *and* $\tilde{p'}$ *are distinct sensing regions, such that* (v, v') *is an edge of* $\mathcal{Q}$. $G(\mathcal{Q}, V)$ *is called* ***the sensing region graph of*** $\mathcal{Q}$ ***given visibility*** V.

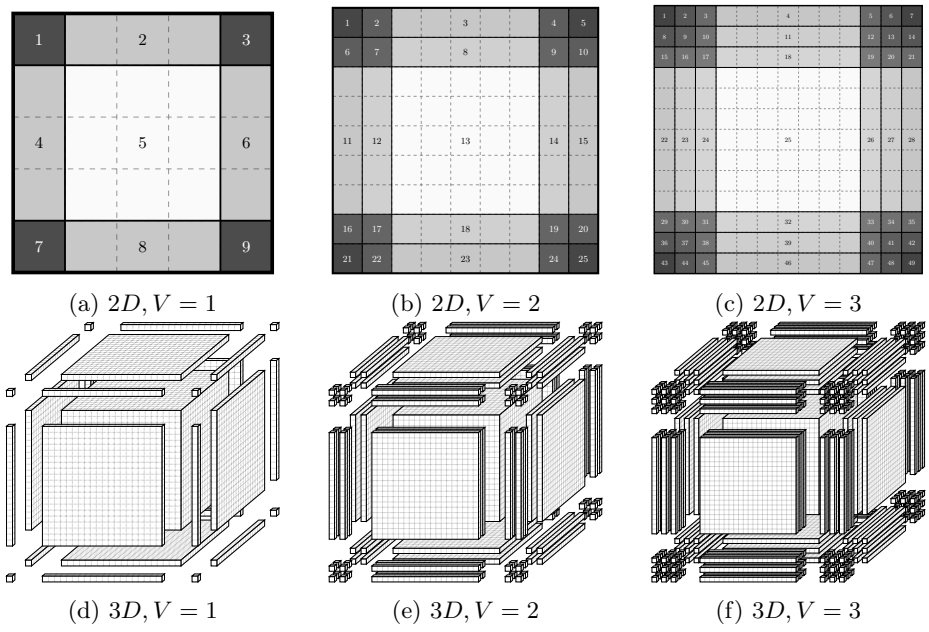
The sensing regions of 2- and 3-dimensional grid graphs are illustrated in Figure 1. We note two straightforward facts about sensing regions:

(A) Let ALG be $\mathcal{R}$'s traversal algorithm. As long as $\mathcal{R}$ is located within a given sensing region $\tilde{p}$, ALG can only move $\mathcal{R}$ in one direction.
(B) If $\mathcal{Q} = [n_1] \times [n_2] \times \ldots [n_d]$, then $G(\mathcal{Q}, V)$ is isomorphic to the d-dimensional grid graph $[m_1] \times [m_2] \times \ldots \times [m_d]$ where $m_i = \min\big(n_i, 2V + 1\big)$.

(A) follows from the fact that vertices in $\tilde{p}$ are indistinguishable to $\mathcal{R}$, and $\mathcal{R}$ has 0 bits of memory, hence ALG must always make the same decision as long as $\mathcal{R}$ is in $\tilde{p}$.

(B) is established by noting that two locations $p, p' \in \mathcal{Q}$ belong to the same sensing region if and only if p and p' have the same boundary distances $(l_1, r_1), \ldots (l_d, r_d)$ along each dimension (Definition 3.5). The set of possible boundary distances (l_i, r_i) is of size m_i, because $\mathcal{R}$ can sense vertices at distance less than or equal to V along dimension i. Each sensing region is connected to adjacent sensing regions in the same way grid vertices are (see Figure 1), making $G(\mathcal{Q}, V)$ isomorphic to $[m_1] \times [m_2] \times \ldots [m_d]$.

We can now establish some facts about whether a given d-dimensional grid is patrollable:



(a) $2D, V = 1$ (b) $2D, V = 2$ (c) $2D, V = 3$

(d) $3D, V = 1$ (e) $3D, V = 2$ (f) $3D, V = 3$

Fig. 1: The sensing regions of $2D$ and $3D$ grid graphs given sensing range $V = 1$, 2, or 3. For $2D$, regions are labelled from 1 to $(2V + 1)^2$.

Lemma 3.8. *Let $\mathcal{Q} = [n_1] \times \ldots \times [n_d]$. If $\prod_{i=1}^{d} m_i$, where $m_i = \min\left(n_i, 2V + 1\right)$ is odd and greater than 1, no algorithm exists that can patrol $\mathcal{Q}$ with 0 bits of memory.*

Proof. Suppose for contradiction that such an algorithm, ALG, exists and let $\mathcal{C}_{\text{ALG}}(v_1) = v_1 \ldots v_T v_1$ be the Hamiltonian cycle from v_1 induced by ALG. Let $\tilde{\mathcal{C}}$ be the walk $\tilde{v_1} \ldots \tilde{v_T} \tilde{v_1}$ in $G(\mathcal{Q}, V)$ after deleting duplicates (that is, vertices $\tilde{v_i}$ for which $\tilde{v_i} = \tilde{v_{i-1}}$). $\tilde{\mathcal{C}}$ is a cycle that traverses all of $G(\mathcal{Q}, V)$. In fact, $\tilde{\mathcal{C}}$ is a Hamiltonian cycle, since by (A), $\mathcal{R}$ cannot visit the same sensing region twice before visiting all others. But $G(\mathcal{Q}, V)$ is isomorphic to $[m_1] \times [m_2] \times \ldots [m_d]$, and $\prod_{i=1}^{d} m_i$ is odd, so no Hamiltonian cycle of it exists (see [19])–contradiction. □

Lemma 3.9. *Let $\mathcal{Q} = [n_1] \times \ldots \times [n_d]$. If there exist different i, j such that $n_i, n_j > 2V + 1$, no algorithm exists that can patrol $\mathcal{Q}$ with 0 bits of memory.*

Proof. Suppose without loss of generality that $i = 1, j = 2$. Suppose also, for contradiction, that an algorithm ALG exists that patrols $\mathcal{Q}$ with 0 bits of memory. Since $n_1, n_2 > 2V + 1$, the sensing region $\tilde{r} \in G(\mathcal{Q}, V)$ that contains $(V + 1, V + 1, 1, \ldots 1) \in \mathcal{Q}$ necessarily also contains $(V + 1, V + 2, 1, \ldots 1)$, $(V + 2, V + 1, 1, \ldots 1)$ and $(V + 2, V + 2, 1, \ldots 1)$. Let us denote these four vertices p_1, p_2, p_3, p_4 respectively. Let $\mathcal{C}_{\text{ALG}}(v_1) = v_1 \ldots v_T v_1$ and $\tilde{\mathcal{C}}$ be as in the proof of Lemma 3.8. Since $\tilde{\mathcal{C}}$ is a Hamiltonian cycle of $G(\mathcal{Q}, V)$, we know that the Hamiltonian path $v_1 \ldots v_T$ of $\mathcal{Q}$ enters each sensing region exactly once. Furthermore, by (A), $\mathcal{R}$ can only move in one direction within a given sensing region. However,

the sensing region $\tilde{r}$ contains the 2×2 subregion $\{p_1, p_2, p_3, p_4\}$, hence requires $\mathcal{R}$ to move in two directions (along the x_1 and x_2 axes) to visit all its vertices. Consequently, it cannot be fully visited by the path $v_1 \dots v_T$ - contradiction. □

Lemmas 3.8 and 3.9 establish that if the conditions of Theorem 3.1 are unfulfilled, no algorithm exists that can patrol $\mathcal{Q}$ with 0 bits of memory. It remains to show that if the conditions are fulfilled, such algorithms exist. We shall explicitly describe such algorithms. Specifically, Theorem 3.1 requires us to construct, for every V, an algorithm that patrols a grid graph $\mathcal{Q}$ satisfying:

1. There is at most one index i such that $n_i > 2V + 1$, and
2. $\prod_{i=1}^{d} \min\left(n_i, 2V + 1\right)$ is even or equals 1.

Note that this problem is not strictly equivalent to finding *any* Hamiltonian cycle of the sensing region graph, because sensing regions may contain more than 1 vertex, and an arbitrary Hamiltonian cycle may require the robot to move in several different directions inside the same sensing region. Hence, we need to identify a particular Hamiltonian cycle that works.

We give two different algorithms for the cases $V = 1$ and $V > 1$ (the algorithm for $V > 1$ constructs a different kind of Hamiltonian cycle and must be handled separately). Both algorithms assume, without loss of generality, that $n_1 \geq n_2 \geq \dots n_d$.

$V = 1$, 0-bit Patrolling Algorithm Theorem 3.1 allows for exactly one of the dimensions of $\mathcal{Q}$ to be greater than $2V + 1$. Hence, we assume w.l.o.g. that $n_2, n_3 \dots n_d \leq 2V + 1$ (so x_1 is the unique dimension allowed to be larger).

Our algorithm for the $V = 1$ case, Algorithm 1, is straightforward. Its pseudocode returns a *step* vector based on $\mathcal{R}$'s sensing data, which $\mathcal{R}$ adds to its current coordinates to determine its next location. It works by patrolling $\mathcal{Q}$ in a "zig-zag" fashion, moving in a fixed direction along the x_1 axis until it hits a boundary, and subsequently flipping direction and incrementing or decrementing x_2; moving in a fixed direction along x_2 until it hits a boundary and subsequently flipping direction and incrementing or decrementing x_3; and so on. A subroutine called *Parity* is used to keep track of whether $\mathcal{R}$ needs to increment or decrement its coordinate along a given axis via a variable called `down`. The behavior of Algorithm 1 over $\mathcal{Q} = [5] \times [3] \times [3] \times [2]$ is illustrated in Figure 2a.

Assuming Theorem 3.1's conditions are fulfilled, Algorithm 1 finds a Hamiltonian path of $\mathcal{Q}$, without violating any assumptions in the problem setting:

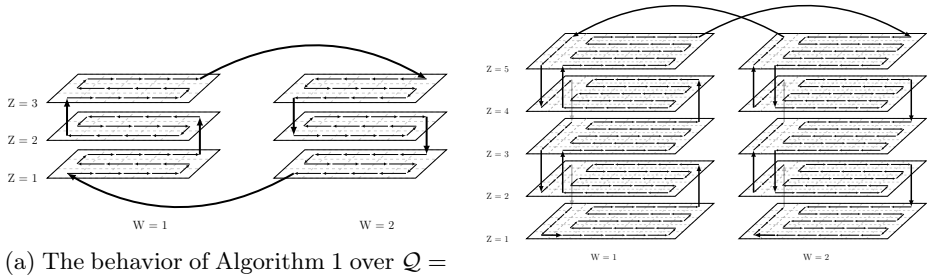
- **No memory or sensing range constraints are violated:** As can be seen from the pseudocode, the algorithm receives as (constant) inputs the boundary distances of $\mathcal{R}$ and the dimensions of $\mathcal{Q}$, but uses no persistent state memory.
- **The algorithm successfully patrols $\mathcal{Q}$:** When $V = 1$, Theorem 3.1's conditions imply some dimension n_i of $\mathcal{Q}$ equals exactly 2. Due to the parity bit `down`, the path of $\mathcal{R}$ is reversed when $x_d = 2$ compared to when $x_d = 1$ (see Figure 2a). Hence, Algorithm 1 alternates between two opposite Hamiltonian walks, one covering the vertices of $\mathcal{Q}$ which have $x_d = 1$ and one covering vertices which have $x_d = 2$. Stitching these walks together forms a Hamiltonian cycle of $\mathcal{Q}$.

Algorithm 1: A memoryless, $V = 1$ sensing range patrolling algorithm for d-dimensional grids that adhere to the constraints of Theorem 3.1.

```
1  Function Parity
       Input: V, n, (l, r)
2      if l < V then
3          return l mod 2
4      return n − r + 1 mod 2
5  Function MakeMoveMemoryless
       Input: V = 1 - agent sensing range.
       Input: n_1, ..., n_d - the dimensions of Q
       Input: {(l_1, r_1), ... (l_d, r_d)} - see Definition 3.5
6      step = (0, ..., 0)                          // d - dimensional zero vector
7      for j = 1 to d do
8          down = Σ_{i>j}^{d} Parity(V, n_i, (l_i, r_i)) mod 2      // 0 when j = d
9          if down = 0 and r_j > 0 then
10             step[j] = 1                          // move up in the jth dimension
11             return step
12         if down = 1 and l_j > 0 then
13             step[j] = −1                         // move down in the jth dimension
14             return step
15     step[d] = −1                                 // move down in the last dimension
16     return step
```

***V* > 1, 0-bit Patrolling Algorithm** Our algorithm for the $V > 1$ case, Algorithm 2, needs to patrol a greater variety of grid graphs than the algorithm we gave for the $V = 1$ case, since the conditions asserted by Theorem 3.1 depend on V. As before, we assume w.l.o.g. that $n_2, n_3 \dots n_d \leq 2V + 1$.

Similar to Algorithm 1, it can be seen from the pseudocode that Algorithm 2 is memoryless and does not violate the sensing constraints of $\mathcal{R}$. Algorithm 2 works by partitioning $\mathcal{Q}$ into two disjoint sub-graphs, $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2$. $\mathcal{Q}_1$ contains all vertices in $\mathcal{Q}$ whose 1st coordinate is 1, i.e., all vertices of the form $(1, *, *, \dots *)$. $\mathcal{Q}_2$ contains all vertices whose 1st coordinate is greater than 1, i.e., $\mathcal{Q}_2 = \mathcal{Q} \setminus \mathcal{Q}_1$. The behavior of Algorithm 2 is illustrated by Figure 2b. As seen in the Figure, the Hamiltonian cycle constructed by Algorithm 2 is split into (i) a Hamiltonian path of $\mathcal{Q}_2$ and (ii) a Hamiltonian path of $\mathcal{Q}_1$. The Hamiltonian path of $\mathcal{Q}_2$ goes in a zig-zag, similar to Algorithm 1, either increasing or decreasing x_1 until a coordinate of the form $(2, *, \dots *)$ or $(n_1, *, \dots *)$ is met (this is why Algorithm 2 requires $V > 1$ - otherwise we could not uniquely distinguish coordinates of the form $(2, *, \dots *)$), taking a step along some non-x_1 coordinate, and then going in reverse along the x_1 axis. The conditions of Theorem 3.1 imply $\prod_{i=2}^{d} n_i$ is even when $d > 1$. This implies that the Hamiltonian path of $\mathcal{Q}_2$ shall end up adjacent to the start of $\mathcal{Q}_1$'s Hamiltonian path. Hence the two paths can be joined to form a Hamiltonian cycle of $\mathcal{Q}$.



(a) The behavior of Algorithm 1 over $\mathcal{Q} = [5] \times [3] \times [3] \times [2]$. Arrows denote the next step of $\mathcal{R}$ from a given location.

(b) The behavior of Algorithm 2 over $\mathcal{Q} = [7] \times [5] \times [5] \times [2]$.

Fig. 2: Memoryless patrolling algorithms.

Algorithm 1 and Algorithm 2, together with Lemmas 3.8 and 3.9, establish Theorem 3.1. □

4 Patrolling with 1 Bit of Memory

Section 3 established that no 0-bit algorithm can patrol all grid graphs no matter how large $\mathcal{R}$'s sensing range is. In contrast, in this section we give a 1-bit, $V = 1$ algorithm that patrols all d-dimensional grid graphs. Additionally, unlike Algorithms 1 and 2, we shall describe an algorithm that does not depend on the dimension sizes $(n_1, \ldots n_d)$ of $\mathcal{Q}$, thus $\mathcal{R}$ does not need to know these in advance.

Denote the current memory state of $\mathcal{R}$ "mem". Since we assume $\mathcal{R}$ possesses 1 bit of memory, mem is either 0 or 1. The primary challenge we face is that, with just 1 bit of memory, each sensing region of $\mathcal{Q}$ can be exited in only 2 directions - one direction when $mem = 0$ and one direction when $mem = 1$. However, patrolling a d-dimensional grid requires moving in $2d$ distinct directions (as we need to be able to increase and decrease each of our d coordinates). Consequently, we must somehow "diffuse" our directions of motion across the sensing regions of $\mathcal{Q}$ in a manner such that the in- and out- degree of each sensing region not exceed 2, but $\mathcal{R}$ nevertheless manages to visit every vertex in $\mathcal{Q}$.

The number of sensing regions grows exponentially with d: it is between 2^d and $(2V + 1)^d$ for a given V. Since $\mathcal{R}$'s algorithm can act differently in each sensing region, we have more "leeway" in higher dimensions. Hence, we find that low-dimensional grid graphs require "ad hoc" patrolling strategies, but that the patrolling strategy for higher dimensional graphs can be defined in a more principled way, using induction.

Figures 3a and 3b visually illustrate algorithms for patrolling a $1D$ and $2D$ grid graphs using 1 bit of memory. We shall refer to these algorithms as `MakeMove1D` and `MakeMove2D` respectively. Pseudocode for these algorithms is available in [3]. We use these lower-dimensional patrolling algorithms as subroutines in our higher-dimensional patrolling algorithms. Running `MakeMove1D` and `MakeMove2D` on higher-dimensional grid graphs will result in $\mathcal{R}$ patrolling a 1- or 2-dimensional sub-grid of $\mathcal{Q}$, respectively.

We note that the visual descriptions of `MakeMove1D` and `MakeMove2D` given in Figure 3b are not fully determined: some vertices have only one outgoing arrow.

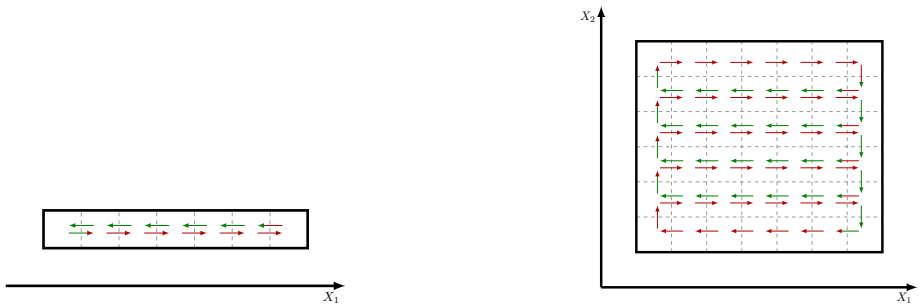
Algorithm 2: A memoryless, $V > 1$ sensing range patrolling algorithm for d-dimensional grids that adhere to the constraints of Theorem 3.1.

```
1  Function MakeMoveMemoryless
       Input: As in Algorithm 1
2      step = (0, ..., 0)                          // d - dimensional zero vector
3      if l_1 = 0 then // return back on the x_1 = 1 strip
4          for j = 2 to d do
5              up = Σ_{i>j}^{d} Parity(V, n_i, (l_i, r_i)) mod 2        // 0 when j = d
6              if up = 0 and l_j > 0 then
7                  step[j] = −1
8                  return step
9              if up = 1 and r_j > 0 then
10                 step[j] = 1
11                 return step
12         step[1] = 1
13     else // execute a zig-zag motion
14         for j = 1 to d do
15             down = Σ_{i>j}^{d} Parity(V, n_i, (l_i, r_i)) mod 2      // 0 when j = d
16             if down = 0 and r_j > 0 then
17                 step[j] = 1
18                 return step
19             if down = 1 and l_j > 0 then
20                 if j = 1 and l_1 = 1 then            // skip x_1 = 1 strip
21                     continue
22                 step[j] = −1
23                 return step
24         step[1] = −1
25     return step
```

For example, in Figure 3b, if $\mathcal{R}$ is located at $(1, 1)$ with $mem = 1$, its next move is undefined. Although these states are simple to handle, we deliberately do not depict them in Figure 3 because they are *transient*: $\mathcal{R}$ never returns to these states after exiting them.

Because $\mathcal{R}$ has just 1 bit of memory, there can be at most two outgoing arrows per vertex $v \in \mathcal{Q}$ in Figure 3b, thus transient states give us important "degrees of freedom" when we want to extend our patrolling algorithms to higher dimensions. For example, in our algorithm for patrolling 3-dimensional grid graphs, `MakeMove3D`, we turn some transient states into non-transient states that send $\mathcal{R}$ up and down the x_3 axis. To explain how `MakeMove3D` works we define "floors":

Definition 4.1. *Let $\mathcal{Q}$ be a d-dimensional grid graph. A k-**floor of** $\mathcal{Q}$ with coordinates $(q_{k+1}, q_{k+2}, \ldots, q_d)$ is the sub-graph of $\mathcal{Q}$ defined by $F_k(q_{k+1}, \ldots, q_d) = \{ (x_1, \ldots, x_k, q_{k+1}, \ldots, q_d) \in \mathcal{Q} \mid (x_1, \ldots, x_k) \in [n_1] \times \ldots [n_k] \}$.*



(a) (b)

Fig. 3: Patrolling 1D and 2D grid graphs with 1 bit of memory using (a) `MakeMove1D` and (b) `MakeMove2D`. Each arrow's body and head take on one of two colors: red or green. When $\mathcal{R}$'s bit of memory is set to 0, it searches for an arrow with a red body at its current location, and moves to where it points. Likewise, when $\mathcal{R}$'s bit of memory is set to 1, it searches for an arrow with a green body, and moves to where it points. The color of an arrow head determines what $\mathcal{R}$ should set its bit of memory to after moving: 0 if red, 1 if green.

We introduce the notation $(q_1, q_2, \ldots, q_d|m)$ to refer to the state of $\mathcal{R}$ where $x_i = q_i$ and $mem = m$. We shall also write $(q_1, \ldots, q_k, *|m)$ to refer to the set of states where $x_1 = q_1, \ldots x_k = q_k$, but $x_{k+1}, \ldots, x_d$ can take on any value. Finally, we use the notation $\neg q$ to say "any value except q". For example, $(\neg q_1, q_2, \ldots, q_d|m)$ refers to a set of states where $mem = m$, $x_1 \neq q_1$, and $x_2 = q_2, \ldots, x_d = q_d$.

`MakeMove3D` works by moving $\mathcal{R}$ according to `MakeMove2D` in every 2-floor of $\mathcal{Q}$, but it moves $\mathcal{R}$ up and down x_3 whenever $\mathcal{R}$ reaches the states $(1, n_2, \neg n_3, *|0)$ and $(\neg 1, n_2, \neg 1, *|1)$, respectively. It also reserves the state $(1, n_2, n_3, *|0)$ for transitioning between the latter two states. Pseudocode and diagrams for `MakeMove3D` are available: Algorithm 3 and Figure 4.

The idea behind `MakeMove3D` seems natural, as it simply runs `MakeMove2D` until it finishes patrolling a 2-floor and then goes to a different 2-floor. However, because we have just 1 bit of memory, there is more nuance to this than might be suspected at a glance: the states $(1, n_2, \neg n_3, *|0)$ and $(\neg 1, n_2, \neg 1, *|1)$ were specifically chosen because $\mathcal{R}$ can transition between them in a single move, and because $(\neg 1, n_2|1)$ is a transient state of `MakeMove2D` that can be repurposed to enable $\mathcal{R}$ to move in the x_3 axis. Likewise, to use `MakeMove3D` as a subroutine in our higher-dimensional patrolling algorithm, `MakeMove(k)D`, we must find other transient states that do not interfere with the ones used by `MakeMove3D`. Hence, there is *a small economy of transient states* that must be exploited; it took the authors some time to find a construction that works.

In general, our Algorithm for patrolling $k+1$-dimensional grid graphs shall be called `MakeMove(k+1)D`. This algorithm is built recursively, calling `MakeMove(k)D` as a subroutine. The concept of `MakeMove(k+1)D` is similar to `MakeMove3D`: it has a small number of *specially-designated states* whose purpose is to increment and

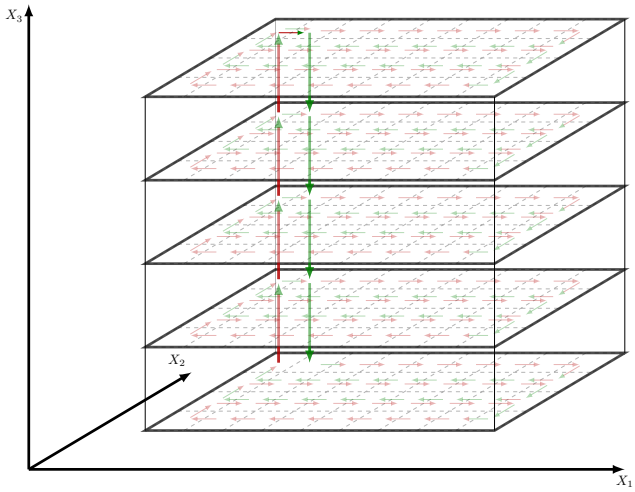


Fig. 4: Patrolling a 3D grid graph with 1 bit of memory using `MakeMove3D`. To be read the same way as Figure 3.

decrement x_{k+1}, and at all other times it executes `MakeMove(k)D`. Pseudocode for `MakeMove(k+1)D` is available - see Algorithm 4. The algorithm can also be defined non-inductively, i.e., without calling `MakeMove(k)D`: see Appendix A in the extended version of this paper [3]. The full algorithm is summarized in Table 1.

The main difference between `MakeMove(k)D`, $k > 3$ and `MakeMove3D` is that the specially-designated states of `MakeMove(k)D` are $(n_1, 1, \ldots, 1, n_{k-1}, n_k, *|1)$, $(n_1, 1, \ldots, 1, n_{k-1}, \neg n_k, *|1)$ and $(\neg n_1, 1, \ldots, 1, n_{k-1}, \neg 1, *|1)$, whereas `MakeMove3D` specially designates the states $(1, n_2, n_3, *|0)$, $(1, n_2, \neg n_3, *|0)$ and $(\neg 1, n_2, \neg 1, *|1)$. To understand why, note that at the lowest level of recursion, `MakeMove(k+1)D` executes `MakeMove2D`- and it only deviates from `MakeMove2D` in a small number of states. Thus, `MakeMove(k+1)D` can be understood as executing `MakeMove2D` until, in a given 2-floor, it hits a specially-designated state of some higher-dimensional `MakeMove` algorithm. Consequently, it is crucial that these designated states do not break `MakeMove2D`'s ability to patrol 2-floors. The x_1 and x_2 coordinates of the specially-designated states of `MakeMove(k)D`, $k > 3$, are always $(n_1, 1)$ and $(\neg n_1, 1)$. However, if instead of calling `MakeMove3D` as a subroutine of `MakeMove(k)D` when $k = 4$, we naively tried to call `MakeMove(k)D` with $k = 3$, we see from the pseudocode that its designated states' x_1 and x_2 coordinates would have been (n_1, n_2) and $(\neg n_1, n_2)$ - requiring 2-floors to specially handle these coordinates in addition to $(n_1, 1)$ and $(\neg n_1, 1)$. It is difficult to repurpose these coordinates without breaking `MakeMove2D`, so instead, we have `MakeMove3D` interact with the $(1, n_2)$ and $(\neg 1, n_2)$ states of 2-floors.

Lemma 4.2. *Let $\mathcal{Q}$ be a d-dimensional grid graph and let v be $\mathcal{R}$'s current location.* `MakeMove(k)D` *patrols the k-floor of $\mathcal{Q}$ that contains v.*

Let us give the name "`MakeMove`" to the algorithm that patrols a d-dimensional grid graph $\mathcal{Q}$ by running `MakeMove(k)D` with $k = d$ (where d is inferred from $\mathcal{R}$'s sensing data, $\mathbb{S}(1, p)$). Lemma 4.2 directly implies our main result:

Algorithm 3: A 3-dimensional grid graph patrolling algorithm, using sensing range $V = 1$ and 1 bit of internal memory.

```
Function MakeMove3D
    Input: {(l_1, r_1), (l_2, r_2), (l_3, r_3)}, mem ∈ {0, 1}.
    step = (0, ..., 0)                          // d - dimensional zero vector
    if l_1 = 0 and r_2 = 0 and mem = 0 then
        if r_3 = 0 then
            step[1] = 1          // special state that occurs when x_3 is
             maximized
        else
            step[3] = 1                                          // move up x_3
        mem = 1
        return step, mem
    if l_1, l_3 ≠ 0 and r_2 = 0 and mem = 1 then
        step[3] = −1                                         // move down x_3
        return step, mem
    return MakeMove2D ({(l_1, r_1), (l_2, r_2)}, mem)
```

Algorithm 4: A $(k+1)$-dimensional grid graph patrolling algorithm, using sensing range $V = 1$ and 1 bit of internal memory.

```
Function MakeMove(k+1)D
    Input: {(l_1, r_1), ..., (l_{k+1}, r_{k+1})}, mem ∈ {0, 1}.
    step = (0, ..., 0)                          // d - dimensional zero vector
    if ⋀_{j=2}^{k−1} l_j = 0 and r_k = 0 and mem = 1 then
        if r_1 = 0 then
            if r_{k+1} = 0 then
                step[1] = −1
            else
                step[k + 1] = 1                                  // move up x_{k+1}
                mem = 0
            return step, mem
        if l_{k+1} ≠ 0 then
            step[k + 1] = −1                                 // move down x_{k+1}
            return step, mem
    return MakeMove(k)D ({(l_1, r_1), ..., (l_k, r_k)}, mem)
```

Theorem 4.3. *Let $\mathcal{Q}$ be any grid graph.* `MakeMove` *patrols $\mathcal{Q}$, visiting every vertex in at most $2|\mathcal{Q}|$ steps, using $V = 1$ sensing range and 1 bit of memory.*

As mentioned earlier, `MakeMove` can also be written explicitly, without recursive calls to `MakeMove(k)D`–see Appendix B of the extended paper [3].

Table 1: Overview of `MakeMove(k+1)D` (Algorithm 4) based on $\mathcal{R}$'s current coordinate and *mem*. "Lines" refers to the relevant lines in the pseudocode.

State	Lines	Action
$(1, n_2, n_3, *\|0)$	Algorithm 3, Line 5	Increase x_1 and set $mem = 1$
$(1, n_2, \neg n_3, *\|0)$	Algorithm 3, Line 7	Increase x_3 and set $mem = 1$
$(\neg 1, n_2, \neg 1, *\|1)$	Algorithm 3, Line 11	Decrease x_3
$(n_1, 1, \ldots, 1, n_k, n_{k+1}, *\|1)$, $k < d$	Algorithm 4, Line 6	Decrease x_1
$(n_1, 1, \ldots, 1, n_k, \neg n_{k+1}, *\|1)$, $k < d$	Algorithm 4, Line 8	Increase x_{k+1} and set $mem = 0$
$(\neg n_1, 1, \ldots, 1, n_k, \neg 1, *\|1)$, $k < d$	Algorithm 4, Line 12	Decrease x_{k+1}
Else	-	Run `MakeMove2D`

5 Discussion

We set out to answer whether a given d-dimensional grid graph can be patrolled with sensing range V and b bits of memory. Our results settle this question: Theorem 4.3 shows that $b = 1$ and $V = 1$ are sufficient to patrol any grid graph, and Theorem 3.1 characterizes the values of V for which a grid graph can be patrolled with $b = 0$ bits of memory. Our 1-bit patrolling result is surprising because it defies the intuition, based on existing complexity bounds, that memory requirements should grow with the graph's maximum degree. We are able to exploit the fact that higher-dimensional grids have more distinct local geometric features to design an algorithm that does not grow with grid dimension.

An important direction in patrolling is to establish the space complexity of patrolling broader classes of highly structured environments beyond grids, such as other subgraph families of $\mathbb{Z}^d$. As noted in the introduction, our 1-bit algorithm, and the techniques we developed here, such as analyzing sensing regions, provide a foundation to potentially obtain improved space complexity results for more general environments, such as Cartesian products of general Z^d subspaces. More broadly, mapping the space complexity landscape of patrolling in structured environments is an important challenge where progress seems possible by exploiting inherent regularities in the environments under consideration.

References

1. N. Agmon. *Multi-robot patrolling and other multi-robot cooperative tasks: An algorithmic approach.* Bar Ilan University, 2009.
2. M. Amir and A. M. Bruckstein. Minimizing Travel in the Uniform Dispersal Problem for Robotic Sensors. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 113–121. IFAAMAS, 2019.
3. M. Amir, D. Rabinovich, and A. M. Bruckstein. Patrolling grids with a bit of memory. *arXiv:2307.09214 (`https://arxiv.org/abs/2307.09214`)*, 2023.
4. E. M. Arkin, S. P. Fekete, and J. S. Mitchell. Approximation algorithms for lawn mowing and milling. *Computational Geometry*, 17(1-2):25–50, 2000.
5. E. Bampas, J. Czyzowicz, L. Gąsieniec, D. Ilcinkas, and A. Labourel. Almost optimal asynchronous rendezvous in infinite multidimensional grids. In *Distributed Computing: 24th International Symposium, DISC 2010. Proceedings 24*, pages 297–311. Springer, 2010.
6. L. Budach. Automata and labyrinths. *Mathematische Nachrichten*, 86(1):195–282, 1978.

7. J. Chalopin and S. Das. Rendezvous of mobile agents without agreement on local orientation. In *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010.*, pages 515–526. Springer, 2010.
8. L. Cohen, Y. Emek, O. Louidor, and J. Uitto. Exploring an infinite space with finite memory scouts. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 207–224. SIAM, 2017.
9. R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg. Label-guided graph exploration by a finite automaton. *ACM Transactions on Algorithms (TALG)*, 4(4):1–18, 2008.
10. K. Diks, P. Fraigniaud, E. Kranakis, and A. Pelc. Tree exploration with little memory. *Journal of Algorithms*, 51(1):38–63, 2004.
11. S. Dobrev, L. Narayanan, J. Opatrny, and D. Pankratov. Exploration of high-dimensional grids by finite automata. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
12. P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg. Graph exploration by a finite automaton. *Theoretical Computer Science*, 345(2-3):331–344, 2005.
13. P. Fraigniaud, D. Ilcinkas, S. Rajsbaum, and S. Tixeuil. Space lower bounds for graph exploration via reduced automata. In *Structural Information and Communication Complexity: 12 International Colloquium, SIROCCO 2005.*, pages 140–154. Springer, 2005.
14. L. Gąsieniec and T. Radzik. Memory efficient anonymous graph exploration. In *Graph-Theoretic Concepts in Computer Science: 34th International Workshop, WG 2008.*, pages 14–29. Springer, 2008.
15. B. Haeupler, F. Kuhn, A. Martinsson, K. Petrova, and P. Pfister. Optimal strategies for patrolling fences. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.
16. T.-R. Hsiang, E. M. Arkin, M. A. Bender, S. P. Fekete, and J. S. Mitchell. Algorithms for rapidly dispersing robot swarms in unknown environments. *Algorithmic Foundations of Robotics V*, pages 77–93, 2004.
17. T. Izumi, K. Kakizawa, Y. Kawabata, N. Kitamura, and T. Masuzawa. Deciding a graph property by a single mobile agent: One-bit memory suffices. *arXiv preprint arXiv:2209.01906*, 2022.
18. G. Kilibarda. On reduction of automata in labyrinths. *Publications de l'Institut Mathematique*, 101(115):47–63, 2017.
19. C. Moore and S. Mertens. *The nature of computation*. OUP Oxford, 2011.
20. D. Rabinovich and A. M. Bruckstein. Emerging cooperation on the road by myopic local interactions. *arXiv:2208.03760*, 2022.
21. O. Rappel, M. Amir, and A. M. Bruckstein. Stigmergy-based, dual-layer coverage of unknown regions. In *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, pages 1439–1447, 2023.
22. A. Shats, M. Amir, and N. Agmon. Competitive ant coverage: The value of pursuit. In *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6733–6740. IEEE, 2023.
23. K. Shimoyama, Y. Sudo, H. Kakugawa, and T. Masuzawa. One bit agent memory is enough for snap-stabilizing perpetual exploration of cactus graphs with distinguishable cycles. In *Stabilization, Safety, and Security of Distributed Systems: 24th International Symposium, 2022*, pages 19–34. Springer, 2022.