Near-Linear Time Dispersion of Mobile Agents

Yuichi Sudo
Hosei University, Tokyo, Japan

Masahiro Shibata
Kyushu Institute of Technology, Fukuoka, Japan

Junya Nakamura
Toyohashi University of Technology, Aichi, Japan

Yonghwan Kim
Nagoya Institute of Technology, Aichi, Japan

Toshimitsu Masuzawa
Osaka University, Osaka, Japan

Abstract

Consider that there are $k \leq n$ agents in a simple, connected, and undirected graph $G = (V, E)$ with n nodes and m edges. The goal of the dispersion problem is to move these k agents to mutually distinct nodes. Agents can communicate only when they are at the same node, and no other communication means, such as whiteboards, are available. We assume that the agents operate synchronously. We consider two scenarios: when all agents are initially located at a single node (rooted setting) and when they are initially distributed over one or more nodes (general setting). Kshemkalyani and Sharma presented a dispersion algorithm for the general setting, which uses $O(m_k)$ time and $\log(k + \Delta)$ bits of memory per agent [OPODIS 2021], where m_k is the maximum number of edges in any induced subgraph of G with k nodes, and Δ is the maximum degree of G. This algorithm is currently the fastest in the literature, as no $o(m_k)$-time algorithm has been discovered, even for the rooted setting. In this paper, we present significantly faster algorithms for both the rooted and the general settings. First, we present an algorithm for the rooted setting that solves the dispersion problem in $O(k \log \min(k, \Delta)) = O(k \log k)$ time using $O(\log(k + \Delta))$ bits of memory per agent. Next, we propose an algorithm for the general setting that achieves dispersion in $O(k \log k \cdot \log \min(k, \Delta)) = O(k \log^2 k)$ time using $O(\log(k + \Delta))$ bits. Finally, for the rooted setting, we give a time-optimal (i.e., $O(k)$-time) algorithm with $O(\Delta + \log k)$ bits of space per agent. All algorithms presented in this paper work only in the synchronous setting, while several algorithms in the literature, including the one given by Kshemkalyani and Sharma at OPODIS 2021, work in the asynchronous setting.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases mobile agents, autonomous robots, dispersion

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.38

Related Version *Full Version*: `https://doi.org/10.48550/arXiv.2310.04376` [19]

Funding JSPS KAKENHI 20KK0232.
Yuichi Sudo: JSPS KAKENHI 20H04140, JST FOREST Program JPMJFR226U.
Masahiro Shibata: JSPS KAKENHI 21K17706 and 23K28037.
Junya Nakamura: JSPS KAKENHI 22K11971.

1 Introduction

In this paper, we focus on the dispersion problem involving mobile entities, referred to as mobile agents, or simply, *agents*. At the start of an execution, k agents are arbitrarily positioned at nodes of an undirected graph $G = (V, E)$ with n nodes and m edges. The objective is to ensure that all agents are located at mutually distinct nodes. This problem

© Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa;
licensed under Creative Commons License CC-BY 4.0

Table 1 Dispersion of mobile agents on an arbitrary graph ($\tau = \min(k, \Delta)$) The algorithm of [7] needs to know an asymptotically tight upper bound on m_k in advance. (Since $m_k \le \min(m, k\Delta, \binom{k}{2})$), knowing tight upper bounds on m, k, and Δ is sufficient, but it increases the running time to $O(\min(m, k\Delta, \binom{k}{2}) \log k)$.) The $\log k$ term can be eliminated from the space complexities marked with daggers (†) if you choose to disregard the memory space required for each agent to store its own identifier. For example, the proposed algorithm mentioned in Theorem 8 requires only $O(\log \Delta)$ bits per agent. The space complexity of the first algorithm given by [6], which is marked with a double dagger (‡), can be decreased to $O(k \log \Delta)$ if we assume that the number of possible agent-identifiers is $O(k)$. All algorithms listed in this table are deterministic.

	Memory per agent	Time	General/Rooted	Async./Sync.
[2]	$O(\log(k+\Delta))$ †	$O(m_k)$	rooted	async.
[10]	$O(D + \Delta \log k)$	$O(D\Delta(D+\Delta))$	rooted	async.
Theorem 8	$O(\log(k+\Delta))$ †	$O(k \log \tau)$	rooted	sync.
Theorem 12	$O(\Delta + \log k)$ †	$O(k)$	rooted	sync.
[6]	$O(k \log(k+\Delta))$ ‡	$O(m_k)$	general	async.
[6]	$O(D \log \Delta + \log k)$ †	$O(\Delta^D)$	general	async.
[6]	$O(\log(k+\Delta))$	$O(m_k \cdot k)$	general	async.
[7]	$O(\log(k+\Delta))$	$O(m_k \log k)$	general	sync.
[16]	$O(\log(k+\Delta))$	$O(m_k \log k)$	general	sync.
[11]	$O(\log(k+\Delta))$	$O(m_k)$	general	async.
Theorem 10	$O(\log(k+\Delta))$	$O(k \log^2 k)$	general	sync.
Lower bound	any	$\Omega(k)$	any	any

was originally proposed by Augustine and Moses Jr. [2] in 2018. A particularly intriguing aspect of this problem is the unique computation model. Unlike many other models involving mobile agents on graphs, we do not have access to node identifiers, nor can we use local memory at each node. In this setting, an agent cannot retrieve or store any information from or on a node when it visits. However, each of the k agents possesses a unique identifier and can communicate with each other when they are at the same node in the graph. The agents must collaboratively solve the dispersion problem through this direct communication.

Several algorithms have been introduced in the literature to solve the dispersion problem. This problem has been examined in two different contexts within the literature: the *rooted setting* and the *general setting*. In the rooted setting, all k agents initially reside at a single node. On the other hand, the general setting imposes no restrictions on the initial placement of the k agents. For any $i \in [1, n]$, we define m_i as the maximum number of edges in any i-node induced subgraph of G. The parameter m_k, where k is the number of agents, serves as an upper bound on the number of edges connecting two nodes, each hosting at least one agent, in any configuration. Consequently, m_k frequently appears in the time complexities of dispersion algorithms. This is because (i) solving the dispersion problem essentially requires finding k distinct nodes, and (ii) the simple depth-first search (DFS), employed as a submodule by many dispersion algorithms, needs to explore m_k edges to find k nodes.

Table 1 provides a summary of various dispersion algorithms found in the literature, all designed for arbitrary graphs. Here, Δ and D are the maximum degree and the diameter of a graph, respectively, and $\tau = \min(k, \Delta)$. Augustine and Moses Jr. [2] introduced a simple algorithm, based on depth-first search (DFS), for the rooted setting. This algorithm solves the dispersion problem in $O(m_k)$ time using $O(\log \Delta)$ bits of space per agent. Kshemkalyani and Ali [6] provides two algorithms that accomplish dispersion in the general setting: an $O(m_k)$-time and $O(k \log(k+\Delta))$-space algorithm, and an $O(m_k \cdot k)$-time and $O(\log(k+\Delta))$-

space algorithm, offering a trade-off between time and space. The first is faster but needs more space, while the second is slower but more memory-efficient. Kshemkalyani, Molla, and Sharma [7] found a middle ground with an algorithm that runs in $O(m_k \log k)$ time and uses $O(\log(k + \Delta))$ bits of each agent's memory. This algorithm, however, requires a priori global knowledge, asymptotically tight upper bounds on m_k, to attain its time upper bound. Shintaku, Sudo, Kakugawa, and Masuzawa [16] managed to eliminate this requirement for global knowledge. More recently, Kshemkalyani and Sharma [11] removed the $\log k$ factor from the running time. This algorithm also works in an asynchronous setting, meaning the agents do not need to share a common clock. Any dispersion algorithm requires at least $\Omega(k)$ time, which is almost trivial, but we will provide a proof for completeness in this paper. No other lower bounds on the time complexity of dispersion have been established in the literature. Thus, there is still a significant gap between the best known upper bound $O(m_k)$ and this lower bound of $\Omega(k)$ because $m_k = \Theta(k^2)$ holds in many graph classes. Note that $m_k = \Theta(k^2)$ may hold even in a sparse graph when $k = O(\sqrt{n})$.

All the algorithms mentioned above are based on DFS. However, a few algorithms [6, 10] are designed based on BFS (breadth-first search) and exhibit different performance characteristics. Notably, their upper bounds on running time do not depend on the number of agents k, but depend on diameter D and the maximum degree Δ of a graph.

▶ Note 1 (Space Complexity). Conforming to the convention in the studies of mobile agents [5], this paper, including Table 1, evaluates the space complexity of an algorithm as the maximum size of *persistent memory* needed by an agent during its execution. Persistent memory refers to the information an agent carries when it moves from one node to another and does not include the *working memory* used for local computations at nodes. This persistent memory includes the space required to store its own identifier. Since the k agents are labeled with unique identifiers, every algorithm requires $O(\log k)$ bits per agent.

▶ Note 2 (Parameter m_k). The parameter m_k is introduced in this paper and has not been previously utilized in the literature. Traditionally, the running times of DFS-based algorithms are represented using the parameter $\min(m, k\Delta)$ or $\min(m, k\Delta, \binom{k}{2})$, which are always greater than or equal to m_k. The parameter m_k may be better to represent them because there are some graph classes where $m_k = o(k^2)$ while $\min(m, k\Delta, \binom{k}{2}) = \Omega(k^2)$.

1.1 Our Contribution

In this paper, we drastically reduce the gap between the upper bound $O(m_k)$ and the lower bound $\Omega(k)$ mentioned earlier. As previously noted, $m_k = \Theta(k^2)$ may hold even in sparse graphs, making this gap significant. Let $\tau = \min(k, \Delta)$. We present two algorithms: one for the rooted setting that achieves dispersion in $O(k \log \tau) = O(k \log k)$ time using $O(\log \Delta)$ bits, and the other for the general setting that achieves dispersion in $O(k \log k \cdot \log \tau) = O(k \log^2 k)$ time using $O(\log(k + \Delta))$ bits. The upper bounds obtained here match the lower bounds in both the rooted and the general settings when ignoring poly-logarithmic factors.

To achieve this upper bound, we introduce a new technique. Like many existing algorithms, our algorithms are based on Depth-First Search (DFS). That is, we let agents run DFS on a graph and place or *settle* an agent at each unvisited node they find. Each time unsettled agents find an unvisited node v, one of the agents settles at v, and the others try to find an unvisited neighbor of v. If such a neighbor exists, they move to it. If no such neighbor exists, they go back to the parent of v in the DFS tree. To find an unvisited neighbor, all DFS-based dispersion algorithms in the literature make the unsettled agents visit those neighbors sequentially, i.e., one by one. This process obviously requires $\Omega(\tau)$ time. We break this barrier and find an unvisited neighbor of v in $O(\log \tau) = O(\log k)$ time, with the help of the agents already settled at neighbors of the current location v.

Our goal here is to find any one unvisited neighbor of v if it exists, not to find all of them. Consider the case where there are only two agents a and b at v, a is settled at v, and b is still an unsettled agent. Agent b visits a neighbor of v and if b finds a settled agent at that node, b brings that agent to v. Consequently, there are two agents on v, excluding a, so we can use these two to visit two neighbors of v in parallel. Again, if there are settled agents on both nodes, those agents will be brought to v. Importantly, the number of agents at v, excluding a, doubles each time this process is repeated until an unvisited neighbor is found. Therefore, over time, we can check neighbors of v in parallel with an exponentially increasing number of agents. As a result, we can finish this search or *probing* process in $O(\log \tau)$ time. Thereafter, we allow the helping agents we brought to v to return to their original nodes, or their *homes*. Since we perform the probing process only $O(k)$ times in total throughout DFS, a simple analysis shows that dispersion can be achieved in $O(k \log \tau)$ time in the rooted setting. We call the resulting DFS the *HEO (Helping Each Other)-DFS* in this paper.

In the general setting, like in existing studies, we conduct multiple DFSs in parallel, each starting from a different node. While the DFS performed in existing research requires $\Theta(m_k)$ time, we use HEO-DFS, thus each DFS completes in $O(k \log \tau)$ time. Thus, at first glance, it seems that dispersion can be achieved in $O(k \log \tau)$ time. However, this analysis does not work so simply because each DFS interferes with each other. Our proposed algorithm employs the method devised by Shintaku et al. [16] to efficiently merge multiple DFSs and run HEO-DFSs in parallel with this method. The merge process incurs an $O(\log k)$ overhead, so we solve the dispersion problem in $O(k(\log k) \cdot (\log \tau)) = O(k \log^2 k)$ time.

It might seem that the overhead can be eliminated by using the DFS parallelization method proposed by Kshemkalyani and Sharma [11], instead of the method of Shintaku et al. [16]. However, this is not the case because our HEO-DFS is not compatible with the parallelization method of Kshemkalyani and Sharma. Specifically, their method entails a process such that one DFS absorbs another when multiple DFSs collide. During this process, it is necessary to gather the agents in the absorbed side to a single node, which requires $\Theta(m_k)$ time. Our speed-up idea effectively works for finding an unvisited neighbor, but it does not work for the acceleration of gathering agents dispersed on multiple nodes. Therefore, it is unlikely that our HEO-DFS can be combined with the method of Kshemkalyani and Sharma.

The two algorithms mentioned above are nearly time-optimal, i.e., requiring $O(k \cdot \log^c k)$ time for some constant c. We also demonstrate that in the rooted setting, a time-optimal algorithm based on the HEO-DFS can be achieved if significantly more space is available, specifically $O(\Delta)$ bits per agent.

To the best of our knowledge, HEO-DFS is a novel approach, and no similar techniques have been used in the literature concerning mobile agents and mobile robots. While we demonstrate that HEO-DFS significantly reduces the running time of dispersion algorithms, this technique may also prove useful for addressing other fundamental problems such as exploration and gathering.

A drawback of our HEO-DFS is that it fundamentally requires a synchronous model, even in the rooted setting; i.e., it does not function in an asynchronous model. In HEO-DFS, we attempt to find an unvisited neighbor of the current location with the help of agents settled on other neighbors. These agents must return to their homes once the probing process is completed. In an asynchronous model, unsettled agents (and/or helping agents) may visit those homes before their owners return, disrupting the consistent behavior of HEO-DFS. Therefore, the algorithm provided by Kshemkalyani and Sharma [11] remains the fastest for the asynchronous model. It is still an open question whether there exists a $o(k^2)$-time algorithm that accommodates asynchronicity.

▶ Note 3 (Termination). In this paper, we do not explicitly mention how the agents terminate the execution of a given algorithm. In many cases, termination is straightforward without any additional assumptions in the rooted setting, while in the general setting, additional assumptions are required. Specifically, in the general setting, all algorithms listed in Table 1, except for the $O(\Delta^d)$-time algorithm presented in [8] [1], require both a synchronous setting and global knowledge such as (asymptotically tight upper bounds on) m_k and k. With these assumptions, the agents can easily terminate simultaneously after a sufficiently large number of steps, e.g., $\Theta(k \log^2 k)$ steps in our algorithm for the general setting. Thus, when termination is required, our general setting algorithm no longer exhibits disadvantages compared to existing algorithms: all existing algorithms, except for the $O(\Delta^D)$-time one [8], also require a synchronous setting (and some global knowledge).

For completeness, we present how the agents terminate in our algorithms for the rooted setting, which is almost trivial, in the arXiv version [19].

1.2 Further Related Work

The dispersion problem has been studied not only for arbitrary undirected graphs but also for graphs with restricted topologies such as trees [2], grids [7, 9], and dynamic rings [1]. Additionally, several studies have explored randomized algorithms to minimize the space complexity of dispersion [13, 4], and others have focused on fault-tolerant dispersion [12, 3]. Kshemkalyani et al. [10] introduced the *global communication model*, where all agents can communicate with each other regardless of their locations. In contrast, the standard model, where only the agents co-located at the same node can communicate with each other, is sometimes referred to as the *local communication model*. All algorithms listed in Table 1 assume the local communication model and are deterministic.

Exploration by a single mobile agent is closely related to the dispersion problem. The exploration problem requires an agent to visit all nodes of a graph. Many studies have addressed the exploration problem, and numerous efficient algorithms, both in terms of time and space, have been presented in the literature [15, 14, 17, 18]. In contrast to exploration, the dispersion problem only requires finding k nodes, and we can use k agents to achieve this. Our HEO-DFS take advantage of these differences to solve the dispersion problem efficiently.

2 Preliminaries

Let $G = (V, E)$ be any simple, undirected, and connected graph. Let $n = |V|$ and $m = |E|$. We denote the set of *neighbors* of node $v \in V$ by $N(v) = \{u \in V \mid \{u, v\} \in E\}$ and the degree of a node v by $\delta_v = |N(v)|$. Let $\Delta = \max_{v \in V} \delta_v$, i.e., Δ is the maximum degree of G. The nodes are anonymous, i.e., they do not have unique identifiers. However, the edges incident to a node v are locally labeled at v so that an agent located at v can distinguish those edges. Specifically, those edges have distinct labels $0, 1, \ldots, \delta_v - 1$ at node v. We call these local labels *port numbers*. We denote the port number assigned at v for edge $\{v, u\}$ by $p_v(u)$. Each edge $\{v, u\}$ has two endpoints, thus has labels $p_u(v)$ and $p_v(u)$. Note that these labels are independent, i.e., $p_u(v) \neq p_v(u)$ may hold. For any $v \in V$, we define $N(v, i)$ as the node $u \in N(v)$ such that $p_v(u) = i$. For simplicity, we define $N(v, \perp) = v$ for all $v \in V$.

[1] However, in this algorithm, the agents do not terminate simultaneously, and they require the ability to detect whether or not there is a terminated agent at the current location.

We consider that k agents exist in graph G, where $k \leq n$. The set of all agents is denoted by A. Each agent is always located at some node in G, i.e., the move of an agent is *atomic* and an agent is never located at an edge at any time step (or just *step*). The agents have unique identifiers, i.e., each agent a has a positive integer as its identifier $a.\mathtt{ID}$ such that $a.\mathtt{ID} \neq b.\mathtt{ID}$ for any $b \in A \setminus \{a\}$. The agents know a common upper bound $\mathrm{id}_{\max} \geq \max_{a \in A} a.\mathtt{ID}$ such that $\mathrm{id}_{\max} = \mathrm{poly}(k)$, thus the agents can store the identifier of any agent on $O(\log k)$ space. Each agent has a read-only variable $a.\mathtt{pin} \in \{0, 1, \dots, \Delta - 1\} \cup \{\bot\}$. At time step 0, $a.\mathtt{pin} = \bot$ holds. For any $t \geq 1$, if a moves from u to v at step $t-1$, $a.\mathtt{pin}$ is set to $p_v(u)$ (or the port of v incoming from u) at the beginning of step t. If a does not move at step $t-1$, $a.\mathtt{pin}$ is set to $\bot$. We call the value of $a.\mathtt{pin}$ the incoming port of a. The values of all variables in agent a, excluding its identifier $a.\mathtt{ID}$ and special variables $a.\mathtt{pin}, a.\mathtt{pout}$, constitute the state of a. (We will see what is $a.\mathtt{pout}$ later.)

The agents are synchronous and are given a common algorithm $\mathcal{A}$. An algorithm $\mathcal{A}$ must specify the initial state s_{init} of agents. All agents are in state s_{init} at time step 0. Let $A(v,t) \subseteq A$ denote the set of agents located at node v at time step $t \geq 0$. At each time step $t \geq 0$, each agent $a \in A(v,t)$ is given the following information as the inputs: (i) the degree of v, (ii) its identifier $a.\mathtt{ID}$, and (iii) a sequence of triples $((b.\mathtt{ID}, s_b, b.\mathtt{pin}))_{b \in A(v,t)}$, where s_b is the current state of b. Note that each $a \in A(v,t)$ can obtain its current state s_a and $a.\mathtt{pin}$ from the sequence of triples since a is given its ID as the second information. Then, it updates the variables in its memory space in step t, including a variable $a.\mathtt{pout} \in \{\bot, 0, 1, \dots, \delta_v - 1\}$, according to algorithm $\mathcal{A}$. Finally, each agent $a \in A(v,t)$ moves to node $N(v, a.\mathtt{pout})$. Since we defined $N(v, \bot) = v$ above, agent a with $a.\mathtt{pout} = \bot$ stays in v in step t.

A node does not have any local memory accessible by the agents. Thus, the agents can coordinate only by communicating with the co-located agents. No agents are given any global knowledge such as m, Δ, k, and m_k in advance.

A function $C : A \to \mathcal{M}_{\mathcal{A}} \times V \times \{\bot, 0, 1, \dots, \Delta - 1\}$ is called a global state of the network or a *configuration* if $C(a) = (s, v, q)$ yields $q = \bot$ or $q < \delta_v$ for any $a \in A$, where $\mathcal{M}_{\mathcal{A}}$ is the (possibly infinite) set of all agent-states. A configuration specifies the state, location, and incoming port of each $a \in A$. In this paper, we consider only deterministic algorithms. Thus, if the network is in a configuration C at a time step t, a configuration C' in the next step $t+1$ is uniquely determined. We denote this configuration C' by $\mathit{next}_{\mathcal{A}}(C)$. The execution $\Xi_{\mathcal{A}}(C_0)$ of algorithm $\mathcal{A}$ starting from a configuration C_0 is defined as an infinite sequence $C_0, C_1, \dots$ of configurations such that $C_{t+1} = \mathit{next}_{\mathcal{A}}(C_t)$ for all $t = 0, 1, \dots$. We say that a configuration C_0 is *initial* if the states of all agents are s_{init} and the incoming ports of all agents are $\bot$ in C_0. Moreover, in the rooted setting, we restrict the initial configurations to those where all agents are located at a single node.

▶ **Definition 4** (Dispersion Problem). *A configuration C of an algorithm $\mathcal{A}$ is called* legitimate *if (i) all agents in A are located in different nodes in C, and (ii) no agent changes its location in execution $\Xi_{\mathcal{A}}(C)$. We say that $\mathcal{A}$ solves the dispersion problem if execution $\Xi_{\mathcal{A}}(C_0)$ reaches a legitimate configuration for any initial configuration C_0.*

We evaluate the *time complexity* or *running time* of algorithm $\mathcal{A}$ as the maximum number of steps until $\Xi_{\mathcal{A}}(C_0)$ reaches a legitimate configuration, where the maximum is taken over all initial configurations C_0. Let $\mathcal{M}'_{\mathcal{A}} \subseteq \mathcal{M}_{\mathcal{A}}$ be the set of all agent-states that can appear in any possible execution of $\mathcal{A}$ starting from any initial configuration. We evaluate the *space complexity* or *memory space* of algorithm $\mathcal{A}$ as $\log_2 |\mathcal{M}'_{\mathcal{A}}| + \log_2 \mathrm{id}_{\max}$, i.e., the maximum number of bits required to represent an agent-state that may appear in those executions, plus the number of bits required for each agent to store its own identifier. This implies that we exclude the size of the working memory used for deciding the destination and updating states, as well as the space for storing input information, except for the agent's own identifier.

Throughout this paper, we denote by $[i, j]$ the set of integers $\{i, i+1, ..., j\}$. We have $[i, j] = \emptyset$ when $j < i$. When the base of a logarithm is not specified, it is assumed to be 2. We frequently use $\tau = \min(k, \Delta)$. We define $\nu(a, t)$ as the node where agent a resides at time step t. We also omit time step t from any function in the form $f(*, t)$ and just write $f(*)$ if t is clear from the context. For example, we just write $A(v)$ and $\nu(a)$ instead of $A(v, t)$ and $\nu(a, t)$.

We have the following remark considering the fact that G can be a simple path.

▶ Remark 5. For any dispersion algorithm $\mathcal{A}$, there exists a graph G such that an execution of $\mathcal{A}$ requires $\Omega(k)$ time steps to achieve dispersion on both the rooted and the general settings.

In the two algorithms we present in this paper, **RootedDisp** and **GeneralDisp**, each agent maintains a variable $a.\texttt{settled} \in \{\bot, \top\}$. We say that an agent a is a *settler* when $a.\texttt{settled} = \top$, and an *explorer* otherwise. All agents are explorers initially. Once an explorer becomes a settler, it never becomes an explorer again. Let t be the time at which an agent a becomes a settler. Thereafter, we call the location of a at that time, i.e., $\nu(a, t)$, the *home* of a. Formally, a's home at $t' \geq 0$, denoted by $\xi(a, t')$, is defined as $\xi(a, t') = \bot$ if $t' < t$ and $\xi(a, t') = \nu(a, t)$ otherwise. It is worth mentioning that a settler may temporarily leave its home. Hence $\xi(a, t') = \nu(a, t')$ may not always hold even after a becomes a settler, i.e., even if $t' \geq t$. However, by definition, no agent changes its home. We say that an agent a *settles* when it becomes a settler.

When a node u is a home of an agent at time step t, we call this agent the settler of u and denote it as $\psi(u, t)$. Formally, if there exists an agent a such that $\xi(a, t) = u$, then $\psi(u, t) = a$; otherwise, $\psi(u, t) = \bot$. This function ψ is well defined for the two presentented algorithms because they ensure that no two agents share a common home. We say that a node u is *unsettled* at time step t if $\psi(u, t) = \bot$, and *settled* otherwise.

3 Rooted Dispersion

In this section, we present an algorithm, **RootedDisp**, that solves the dispersion problem in the rooted setting. That is, it operates under the assumption that all agents are initially located at a single node $s \in V$. This algorithm straightforwardly implements the strategy of the HEO-DFS, which we presented in Section 1. The time and space complexities of this algorithm are $O(k \log \tau)$ steps and $O(\log(k + \Delta))$ bits, respectively.

In an execution of Algorithm **RootedDisp**, the agent with the largest ID, denoted as $a_{\max}$, serves as the leader. Note that every agent can easily determine whether it is $a_{\max}$ or not at time step 0 by comparing the IDs of all agents. Then, $a_{\max}$ conducts a depth-first search (DFS), while the other agents move with the leader and one of them settles at an unsettled node when they visit it. If $a_{\max}$ encounters an unsettled node without any accompanying agents, $a_{\max}$ settles itself on that node, achieving dispersion. During a DFS, $a_{\max}$ must determine (i) whether there is an unsettled neighbor of the current location, and (ii) if so, which neighbor is unsettled. To make this decision, all DFS-based algorithms in the literature have $a_{\max}$ visit neighbors one by one until it finds an unsettled node, which clearly requires $\Omega(\tau)$ steps. **RootedDisp**, in contrast, makes this decision in $O(\log \tau)$ steps with the help of the agents that have already settled on the neighbors of the current location.

The pseudocode for Algorithm **RootedDisp** is shown in Algorithm 1. This pseudocode consists of two parts: the main function (lines 1–12) and the function `Probe()` (lines 13–23). As mentioned in the previous section, every agent a maintains a variable $a.\texttt{settled} \in \{\bot, \top\}$, which decides whether a is an explorer or a settler. In addition, the settler $\psi(w)$ of a node w

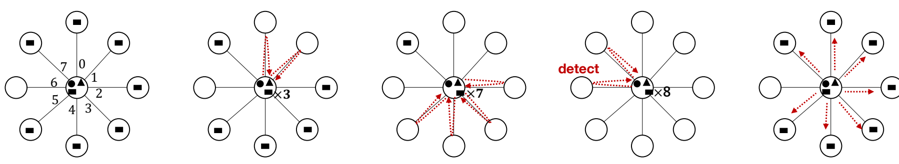

Figure 1 The behavior of the agents when the leader $a_{\max}$ invokes `Probe`() at the center node w in **RootedDisp**. A black circle, triangle, and rectangle represent a leader ($a_{\max}$), a non-leader explorer, and a settler, respectively. The integers in the leftmost figure represents port numbers. In every two time steps, the number of agents on w excluding $\psi(w)$ doubles (i.e., $2 \to 4 \to 8$) until some agent detects an unsettled neighbor of w. After that, $a_{\max}$ lets the helping settlers go back to their homes.

maintains two variables, $\psi(w).\texttt{parent}, \psi(w).\texttt{next} \in [0, \delta_w - 1] \cup \bot$ for the main function. As we will see later, the following are guaranteed each time $a_{\max}$ invokes `Probe`() at node w:

- If there exists an unsettled node in $N(w)$, the corresponding port number will be stored in $\psi(w).\texttt{next}$. More precisely, an integer i such that $N(w,i) = u$ and $\psi(u) = \bot$ is assigned to $\psi(w).\texttt{next}$.
- If all neighbors are settled, $\psi(w).\texttt{next}$ will be set to $\bot$.
- `Probe`() will return in $O(\log \tau)$ time.

The main function performs a depth-first search using function `Probe`() to achieve dispersion. At the beginning of the execution, all agents are located at the same node s. Initially, the agent with the smallest ID settles at node s, and $\psi(s).\texttt{parent}$ is set to $\bot$ (lines 1–2). Then, as long as there are unsettled nodes in $N(\nu(a_{\max}))$, all explorers move to one of those nodes together (lines 7–8). We call this kind of movements *forward moves*. After each forward move from a node w to u, the agent with the smallest ID among $A(u)$ settles on u, and $\psi(u).\texttt{parent}$ is set to i with $N(u,i) = w$ (lines 9–10). For any node $u \in V$, if $\psi(u).\texttt{parent} \neq \bot$, we say that $w = N(u, \psi(u).\texttt{parent})$ is a *parent* of u. By line 9–10, each of the nodes except for the starting node s will have its parent as soon as it becomes settled. When the current location has no unsettled neighbors, all explorers move to the *parent* of the current location (lines 11–12). We call this kind of movements *backward moves* or *retreats*. Finally, $a_{\max}$ terminates when it settles (line 3).

Since the number of agents is k, the DFS-traversal stops after $a_{\max}$ makes a forward move $k-1$ times. The agent $a_{\max}$ makes a backward move at most once from any node. Therefore, excluding the execution time of `Probe`(), the execution of the main function completes in $O(k)$ time. Furthermore, the function `Probe`() is invoked at most $2(k-1)$ times, once after each forward move and once after each backward move. Since a single invocation of `Probe`() requires $O(\log \tau)$ time, the overall execution time of **RootedDisp** can be bounded by $O(k \log \tau)$ time.

Let us describe the behavior of the function `Probe`(), assuming that it is invoked on node w at time step t. Figure 1 may help the readers to understand the behavior. In the execution of `Probe`(), the leader $a_{\max}$ employs the explorers present on w and (a portion of) the settlers at $N(w)$ to search for an unsettled node in $N(w)$. We implement this process with a variable $\psi(w).\texttt{checked} \in [-1, \delta_w - 1]$ for the settler $\psi(w)$. Specifically, explorers at node w verify whether the neighbors of w are unsettled or not in the order of port numbers and store the most recently checked port number in $\psi(w).\texttt{checked}$. Consequently, $\psi(w).\texttt{checked} = \ell$ implies that the neighbors $N(w,0), N(w,1), \ldots, N(w,\ell)$ are settled. Let $x = |A(w,t) \setminus \{\psi(w)\}|$, i.e., there are x agents at w when `Probe`() is invoked, excluding $\psi(w)$.

Algorithm 1 RootedDisp.

```
1  b.settled ← ⊤, where b is the agent with the smallest ID in A(s)
                                          // b settles at the starting node s
2  b.parent ← ⊥
3  while a_max.settled = ⊥ do
4      Probe()
5      Let w = ν(a_max)
6      if ψ(w).next ≠ ⊥ then
7          Let u = N(w, ψ(w).next)               // u is an unsetteled node here
8          All explorers in A(w) go to u
9          b'.settled ← ⊤, where b' is the agent with the smallest ID in A(u)
10         b'.parent ← a_max.pin
11     else
12         All explorers in A(w) go back to node N(w, ψ(w).parent).
13 function Probe():
14     Let w = ν(a_max).
15     (ψ(w).next, ψ(w).checked) ← (⊥, −1)
16     while ψ(w).checked ≠ δ_w − 1 do
17         Let a_1, a_2, ..., a_x be the agents in A(w) \ {ψ(w)}, and let
           Δ' = min(x, δ_w − 1 − ψ(w).checked). For each i = 1, 2, ..., Δ', assign a_i to
           the neighboring node u_i = N(w, i + ψ(w).checked), and let a_i make a round
           trip between w and u_i. In other words, make a_i move in the order
           w → u_i → w. If a_i finds a settler at u_i, it will bring the settler ψ(u_i) back to
           w.
18         if there exists a_i that did not bring ψ(u_i) back to w then
19             ψ(w).next ← i + ψ(w).checked                  // u_i must be unsettled
20             Break the while loop.
21         else
22             ψ(w).checked ← ψ(w).checked + Δ'
23     Let all settlers except for ψ(w) go back to their homes.
```

In the first iteration of the while loop (lines 16–22), the $\min(x, \delta_w)$ agents concurrently visit $\min(x, \delta_w)$ neighbors and then return to w (lines 17–18). This entire process takes exactly two time steps. These agents bring back all the settlers, at most one for each neighbor, they find. If there is an agent that does not find a settler, then the node visited by that agent must be unsettled. In such a case, the port used by one of these agents is stored in $\psi(w)$.next, and the while loop terminates (lines 20–21).[2] If all x agents bring back one agent each, then there are $2x$ agents on w, excluding $\psi(w)$. In the second iteration of the while loop, these $2x$ agents visit the next $2x$ neighbors and search for unsettled neighbors in a similar way. As long as no unsettled neighbors are discovered, the number of agents on w, excluding $\psi(w)$,

[2] For simplicity, we reset $\psi(w)$.checked to -1 each time we invoke Probe() at w, so we do not use the information about which ports were already checked in the past invocation of Probe(). As a result, the value of $\psi(w)$.next computed by Probe does not have to be the minimum port leading to an unsettled neighbor of w.

doubles with each iteration of the while loop. Since there are at most $\tau = \min(k, \Delta)$ settled nodes in $N(w)$, after running the while loop at most $O(\log(\tau/x)) = O(\log \tau)$ times, either an unsettled node will be found, or the search will be concluded without finding any unsettled nodes. In the latter case, since $\psi(w)$.`next` is initialized to $\perp$ when `Probe()` is called (line 15), $\psi(w)$.`next` $= \perp$ will also be valid at the end of the while loop, allowing $a_{\max}$ to verify that all neighbors of w are settled. After the while loop ends, the settlers brought back to w return to their homes (line 23). This process of "returning to their homes" requires the agents to remember the port number leading to their home from w. However, we exclude this process from the pseudocode because it can be implemented in a straightforward manner, and it requires only $O(\log \Delta)$ bits of each agent's memory. In conclusion, we have the following lemma.

▶ **Lemma 6.** *Each time* `Probe()` *is invoked on node* $w \in V$*,* `Probe()` *finishes in* $O(\log \tau)$ *time. At the end of* `Probe()`*, it is guaranteed that: (i) if there exists an unsettled node in* $N(w)$*, then* $N(w, \psi(w)$.`next`$)$ *is unsettled, and (ii) if there are no unsettled nodes in* $N(w)$*, then* $\psi(w)$.`next` $= \perp$ *holds true.*

▶ **Lemma 7.** *Each agent requires* $O(\log(k + \Delta))$ *bits of memory to execute* **RootedDisp***.*

Proof. In this algorithm, an agent handles several $O(\log \Delta)$-bit variable, `next`, `checked`, `parent`, as well as the port number that the settler $\psi(u)$ needs to remember in order to return to node u from node w at line 23 after coming at line 17. Every other variable can be stored in a constant space. Therefore, the space complexity is $O(\log(k + \Delta))$ bits, adding the memory space to store the agent's identifier. ◀

▶ **Theorem 8.** *In the rooted setting, algorithm* **RootedDisp** *solves the dispersion problem within* $O(k \log \tau)$ *time using* $O(\log(k + \Delta))$ *bits of space per agent.*

Proof. As long as there is an unsettled neighbor of the current location, $a_{\max}$ makes a forward move to one of those nodes. If there is no such neighbor, $a_{\max}$ makes a backward move to the parent node of the current location. Since the graph is connected, this DFS-traversal clearly visits k nodes with exactly $k - 1$ forward moves and at most $k - 1$ backward moves. Thus, the number of calls to `Probe()` is at most $2(k - 1)$ times. By Lemma 6, the execution of **RootedDisp** achieves dispersion within $O(k \log \tau)$ time. ◀

4 General Dispersion

4.1 Overview

In this section, we present an algorithm **GeneralDisp** that solves the dispersion problem in $O(k \log \tau \cdot \log k) = O(k \log^2 k)$ time, using $O(\log(k + \Delta))$ bits of each agent's memory, in the general setting. Unlike the rooted setting, the agents are deployed arbitrarily. In **GeneralDisp**, we view the agents located at the same starting node as a single *group* and achieve rapid dispersion by having each group perform a HEO-DFS in parallel, sometimes merging groups. We show that by employing the group merge method given by Shintaku et al. [16], say *Zombie Method*, we can parallelize HEO-DFS by accepting an additive factor of $\log k$ to the space complexity and a multiplicative factor of $\log k$ to the time complexity. We have made substantial modifications to the Zombie Method to avoid conflicts between the function `Probe()` of HEO-DFS and the behavior of the Zombie Method.

As defined in Section 2, agents a with a.`settled` $= \top$ are called settlers, and the other agents are called explorers. In addition, in **GeneralDisp**, we classify explorers to two classes, *leaders* and *zombies*, depending on a variable `leader` $\in [0, \mathrm{id}_{\max}]$. We call an

explorer a a leader if $a.\texttt{leader} = a.\texttt{ID}$, otherwise a zombie. Each agent a initially has $a.\texttt{leader} = a.\texttt{ID}$, so all agents are leaders at the start of an execution of **GeneralDisp**. As we will see later, a leader may become a zombie and a zombie will eventually become a settler, whereas a zombie never becomes a leader again, and a settler never becomes a leader or zombie again. Among the agents in $A(v,t)$, the set of leaders (resp., zombies, settlers) staying at v in time step t is denoted by $A_L(v,t)$ (resp., $A_Z(v,t), A_S(v,t)$). By definition, $A(v,t) = A_L(v,t) \cup A_Z(v,t) \cup A_S(v,t)$.

We introduce a variable $\texttt{level} \in \mathbb{N}$ to bound the execution time of **GeneralDisp**. We call the value of $a.\texttt{level}$ the *level* of agent a. The level of every agent is 1 initially. The pair $(a.\texttt{leader}, a.\texttt{level})$ serves as the group identifier: when agent a is a leader or settler, we say that a belongs to a group $(a.\texttt{leader}, a.\texttt{level})$. By definition, for any $(\ell, i) \in \mathbb{N}^2$, a group (ℓ, i) has at most one leader. A zombie does not belong to any group. However, when it accompanies a leader, it joins the HEO-DFS of that leader. We define a relationship $\prec$ between any two non-zombies a and b using these group identifiers as follows:

$$a \prec b \quad \Leftrightarrow \quad (a.\texttt{level} < b.\texttt{level}) \lor (a.\texttt{level} = b.\texttt{level} \land a.\texttt{leader} < b.\texttt{leader}).$$

We say that agent a is *weaker* than b if $a \prec b$, and that a is *stronger* than b otherwise.

Initially, all agents are leaders and each forms a group of size one. In the first time step, the strongest agent at each node turns all the other co-located agents into zombies (if exists). From then on, each leader performs a HEO-DFS while leading those zombies. For any leader a, we define the *territory* of a as

$$V_a = \{v \in V \mid \exists b \in A : \psi(v) = b \land b.\texttt{leader} = a.\texttt{ID} \land b.\texttt{level} = a.\texttt{level}\}.$$

Each time a leader a visits an unsettled node, it settles one of the accompanying zombies (if exists), giving it a's group identifier $(a.\texttt{leader}, a.\texttt{level})$. That is, a expands its territory. If a node outside a's territory is detected during the probing process of HEO-DFS, that node is considered unsettled even though it belongs to the territory of another leader. As a result, a may move forward to a node u that is inside another leader's territory. If that node u belongs to the territory of a weaker group, a incorporates the settler $\psi(u)$ into its own group by giving $\psi(u)$ its group identifier $(a.\texttt{leader}, a.\texttt{level})$. If a leader a encounters a stronger leader or a stronger settler during its HEO-DFS, a becomes a zombie and terminates its own HEO-DFS. If there is a leader at the current location $\nu(a)$ when a becomes a zombie, a joins the HEO-DFS of that leader. Otherwise, the agent a, now a zombie, chases a stronger leader by moving through the port $\psi(v).\texttt{next}$ at each node v. Unlike **RootedDisp**, a leader updates $\psi(v).\texttt{next}$ with the most recently used port even when it makes a backward move. This ensures that a catches up to a leader eventually, at which point a joins the HEO-DFS led by the leader.

Unlike **RootedDisp**, a leader does not settle itself at a node in the final stage of HEO-DFS. The leader a suspends the HEO-DFS if it visits an unsettled node but it has no accompanying zombies to settle at that time. A leader who has suspended the HEO-DFS due to the absence of accompanying zombies is called a *waiting leader*. Conversely, a leader with accompanying zombies is called an *active leader*. A waiting leader a resumes the HEO-DFS when a zombie catches up to a at $\nu(a)$. As we will see later, the execution of **GeneralDisp** ensures that all agents eventually become either waiting leaders or settlers, each residing at a distinct node. The agents have solved the dispersion once such a configuration is reached because thereafter no agent moves and no two agents are co-located.

When a leader a encounters a zombie z with the same level, a increments its level by one, and z resets its level to zero. This "level up" changes the identifier of a's group, i.e., from $(a.\texttt{ID}, i)$ to $(a.\texttt{ID}, i+1)$ for some i. By the definition of the territory, at this point, a loses

Table 2 Slot Assignments.

Slot Number	Role	Initiative	Pseudocode
Slot 1	Leader election	Leaders	2
Slot 2	Settle, increment level, etc.	Leaders	3
Slots 3	Move to join `Probe`()	Settlers	4
Slots 4–8	`Probe`()	Leaders	3
Slot 9–10	Chase for leaders	Zombies	5
Slot 11–12	Move forward/backward	Leaders	2

all nodes from its territory except for the current location. That is, each time a leader a increases its level, it restarts its HEO-DFS from the beginning. Note that this "level up" event also occurs when two leaders a, b ($b \prec a$) with the same level meet (and there is no stronger agent at the location) because then b becomes a zombie after it finds a stronger leader a, which results in the event that a leader a encounters a zombie with the same level, say b. We have the following lemma here.

▶ **Lemma 9.** *The level of an agent is always at most* $\log_2 k + 1$.

Proof. A level-up event requires one leader a and one zombie b with the same level. That zombie b will get level 0. Thereafter, b never triggers a level-up event again because the level of a leader is monotonically non-decreasing starting from level 1. Therefore, for any $i \geq 1$, the number of agents that can reach level i is at most $\lfloor k/2^{i-1} \rfloor$, leading the lemma. ◀

Therefore, each leader performs HEO-DFS at most $O(\log k)$ times. According to the analysis in Section 3, each HEO-DFS completes in $O(k \log \tau)$ time, which seems to imply that **GeneralDisp** finishes in $O((\log k) \cdot (k \log \tau)) = O(k \log^2 k)$ time. However, this analysis does not take into account the length of the period during which leaders suspend their HEO-DFS. Thus, it is not clear whether a naive implementation of the strategy described above would achieve the dispersion in $O(k \log^2 k)$ time. Following Shintaku et al. [16], we vary the speed of zombies chasing leaders based on a certain condition, which bounds the execution time by $O(k \log^2 k)$ time.

We give zombies different chasing speeds as follows. First, we classify zombies based on two variables $\texttt{level}_L$ and $\texttt{level}_S$ that each zombie manages. For any zombie z, we call $z.\texttt{level}_L$ and $z.\texttt{level}_S$ the *location level* and *swarm level* of z. When a leader z becomes a zombie, it initializes both $z.\texttt{level}_L$ and $z.\texttt{level}_S$ with its level, i.e., $z.\texttt{level}$. Thereafter, a zombie z copies the level of $\psi(\nu(z))$ to $z.\texttt{level}_L$ and updates $z.\texttt{level}_S$ to be $\max\{b.\texttt{level} \mid b \in A_Z(\nu(z))\}$ in every $O(1)$ time steps. Since a zombie only chases a leader with an equal or greater level, $z.\texttt{level}_S \leq z.\texttt{level}_L$ always holds. We say that a zombie z is *strong* if $z.\texttt{level}_S = z.\texttt{level}_L$; z is *weak* otherwise. Then, we exploit the assumption that the agents are synchronous and let weak zombies move twice as frequently as strong zombies to chase a leader. As we will prove later, this difference in chasing speed results in a desirable property of **GeneralDisp**, namely that $\min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\})$ is monotone non-decreasing and increases by at least one in every $O(k \log \tau)$ steps, where A_{AL} is the set of active leaders and A_Z is the set of zombies both in the whole graph, until $A_{AL} \cup A_Z$ becomes empty. Thus, by Lemma 9, $A_{AL} \cup A_Z$ becomes empty and the dispersion is achieved in $O(k \log \tau \cdot \log k) = O(k \log^2 k)$ steps.

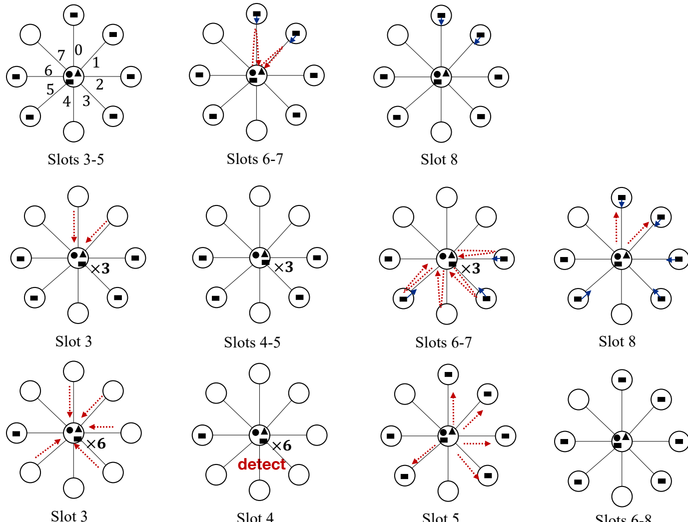

Figure 2 The behavior of explorers when their leader invokes `Probe`() at the center node w in **GeneralDisp**. A black circle, triangle, and rectangle represent a leader, a zombie, and a settler, respectively. The integers in the top left figure represents port numbers.

4.2 Implementation

In **GeneralDisp**, we group every 12 time steps into one unit, with each unit consisting of twelve *slots*. In other words, time steps $0, 1, 2, \dots$ are classified into twelve slots. Specifically, each time step $t \geq 0$ is assigned to slot $(t \bmod 12) + 1$. For example, time step 26 is in slot 3, and time step 47 is in slot 12. Dividing all time steps into twelve slots helps to reduce the interference of multiple HEO-DFSs and allows us to set different "chasing speeds" for weak and strong zombies. Table 2 summarizes the roles of each slot.

Essentially, slots 1–2 are designated for leader election (i.e., group merging), slots 3–8 for probing, slots 9–10 for zombie chasing, and slots 11–12 for forward and backward movement in DFS traversal. It is important to note that settlers always stay at their home during slots 1–2 and 9–12. Hence, once a settler leaves its home, it returns within $O(1)$ steps. This is not the case in **RootedDisp**, where a settler in helping mode does not return home until its leader completes the probing process. In **GeneralDisp**, this frequent return home enables leaders to detect collisions with other groups: if a leader enters another group's territory, it will certainly notice the intrusion during the next slot 1, as it encounters a settler from that group.

Thus, the probing process in **GeneralDisp** slightly differs from that in **RootedDisp**. Consider a leader a_l starting the probing process at a node w (refer to Figure 2). The objective here is to identify any neighboring node of w that lies outside a_l's territory, if such exists. During slots 6–7, explorers at w visit its neighbors and return in parallel. If an explorer b encounters a settler s at a node $u \in N(w)$ in slot 6, b does not bring s back to w in slot 7. Instead, b requests s to enter helping mode, wherein s records the port number to w in the variable $s.\texttt{help} \in \mathbb{N} \cup \bot$ (A settler s is in helping mode if and only if $s.\texttt{mode} \neq \bot$). The helping settler s moves to w via port $s.\texttt{help}$ in the subsequent slot 3, joins the probing in slots 6-7, and returns to its home u again in slot 8. Leader a_l expects that the exact $\psi(w).\texttt{checked}$ helping settlers arrives at w in each slot 3. If this does not occur, a_l detects a non-territorial neighbor in slot 4 by checking the `pin` variable of the helping settlers at

w. Similar to **RootedDisp**'s probing process, the total number of explorers and helping settlers at w doubles until such a neighbor is detected, concluding the process in $O(\log \tau)$ steps. Subsequently, a_l reverts the helping settlers at w to non-helping mode by setting their `help` variable to $\perp$ and instructs them to return to their homes in slot 5.

As mentioned earlier, we differentiate the chasing speed of weak zombies and strong zombies. Specifically, weak zombies move in both slots 9 and 10, while strong zombies move only in slot 10.

We left the detailed implementation of **GeneralDisp** including pseudocodes and the complete proofs of its correctness and time complexity to the appendix due to space constraints. We give only a proof sketch here for the following main theorem.

▶ **Theorem 10.** *In the general setting, there exists an algorithm that solves the dispersion problem within $O(k \log \tau \cdot \log k)$ time using $O(\log(k + \Delta))$ bits of space per agent.*

▶ **Proof Sketch.** It suffices to show that $A_{AL} \cup A_Z$ becomes empty within $O(k \log \tau \cdot \log k)$ steps, at which point every agent is either a waiting leader or a settler, thereby achieving dispersion. We obtain this bound from Lemma 9 and the fact that $\alpha = \min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\})$ increases by at least one in every $O(k \log \tau)$ time steps unless $A_{AL} \cup A_Z$ becomes empty (Lemma 15 in Appendix). We can prove Lemma 15 roughly as follows. Suppose $\alpha = i$. First, all weak zombies at location level i vanish within $O(k)$ steps as they move faster than leaders and strong zombies, eventually encountering a leader, higher-level settlers or stronger zombies. Thereafter, no new weak zombies at location level i are created. Subsequently, without weak zombies at location level i, waiting leaders at level i do not resume active HEO-DFS without increasing its level, which leads to the disappearance of active leaders at level i within $O(k \log \tau)$ steps. Finally, strong zombies chasing leaders at level i catch up to those leaders within $O(k)$ steps or find higher-level settlers, resulting in the increase of their location level. Hence, all zombies with location level i and active leaders at level i are eliminated within $O(k \log \tau)$ steps. ◀

5 For Further Improvement in Time Complexities

In the previous sections, we introduced nearly time-optimal dispersion algorithms: an $O(k \log \tau)$-time algorithm for the rooted setting and an $O(k \log^2 k)$-time algorithm for the general setting. This raises a crucial question: is it possible to develop a truly time-optimal algorithm, specifically an $O(k)$-time algorithm, even if it requires much more space? In this section, we affirmatively answer this question for the rooted setting. We present an $O(k)$-time algorithm that utilizes $O(\Delta + \log k)$ bits of space per agent. However, the feasibility of an $O(k)$-time algorithm in the general setting remains open.

We refer to the new algorithm as **RootedOpt** in this section. In **RootedDisp**, with $O(\log(k + \Delta))$ bits of space, each settler $\psi(w)$ cannot memorize the exact set of settled neighbors of w. Instead, it only remembers the maximum i such that the first i neighbors of w are settled. In contrast, **RootedOpt** allows each settler $\psi(w)$ to remember all settled neighbors of w using $O(\Delta)$ bits, which significantly helps to eliminate an $O(\log \tau)$ factor from the time complexity. However, somewhat surprisingly, both the design of the new algorithm and the analysis of its execution time are non-trivial.

Below, we outline the modifications made to **RootedDisp** to obtain **RootedOpt**. We expect that most readers will grasp the behavior of **RootedOpt** simply by reviewing the following key differences, while we provide the pseudocode for **RootedOpt** in the arXiv version [19].

- In **RootedOpt**, each settler $\psi(w)$ maintains an array variable $\psi(w).\mathtt{checked}$ of size δ_w. Each element of $\psi(w).\mathtt{checked}[i]$ takes a value from the set $\{0, 1, \perp\}$. The assignment $\psi(w).\mathtt{checked}[i] = 0$ (respectively, 1) indicates that the neighbor $N(w, i)$ is unsettled (respectively, settled). The value $\perp$ is utilized exclusively during the probing process, meaning that the neighbor $N(w, i)$ has yet to be checked for its settled status. For any $c \in \{0, 1, \perp\}$, we define $U_c(u) = \{N(u, i) \mid i \in [0, \delta_w - 1], \psi(w).\mathtt{checked}[i] = c\}$.
- Consider that the unique leader $a_{\max}$ invokes the probing process Probe() at a node w. (Remember that $a_{\max}$ is the unique leader that has the maximum identifier at the beginning of the execution.) In **RootedDisp**, the probing process terminates as soon as any agent finds an unsettled neighbor. However, in **RootedOpt**, the process only ends when all of w's neighbors are probed or when at least ℓ unsettled neighbors are found, where ℓ is the number of explorers. In the former case, the probing process is now complete: $U_\perp(w)$ is empty, and $U_0(w)$ equals the set of unsettled neighbors of w. In the latter case, the explorers go to distinct ℓ unsettled nodes and settle there, thereby achieving dispersion.
- Consider an agent a making a round trip $w \to u \to w$ during Probe(), where $u = N(w, p)$ for some $p \in [0, \delta_w - 1]$. If a does not encounter a settler at u, it simply sets $\psi(w).\mathtt{checked}[i]$ to 0. On the other hand, if a settler is found at u, a sets $\psi(w).\mathtt{checked}[i]$ to 1 and additionally sets $\psi(u).\mathtt{checked}[q]$ to 1, where $q \in [0, \delta_u - 1]$ is the port number such that $w = N(u, q)$. This modification ensures that $U_0(u)$ remains equal to the set of unsettled neighbors of u when the explorers go back to u.
- In **RootedDisp**, the leader $a_{\max}$ invokes Probe() after each forward or backward move. However, in **RootedOpt**, $a_{\max}$ only invokes Probe() after making a forward move. This change does not compromise the correctness of **RootedOpt** because when $a_{\max}$ makes a backward move to a node w, $\psi(w)$ accurately remembers its unsettled neighbors due to the modification mentioned earlier.

One might think that the probing process Probe() in **RootedOpt** could take longer time than in **RootedDisp**, as it only finishes after all neighbors of the current location have been probed or after finding ℓ unsettled neighbors, where ℓ is the number of explorers. Particularly, there seems to be a concern that during the probing at a node w, the number of agents, excluding $\psi(w)$, may not always double: this event occurs when some agents discover an unsettled neighbor. Despite that, we deny this conjecture at least asymptotically, that is, we have the following lemma.

▶ **Lemma 11.** *Assume that $a_{\max}$ invokes* Probe() *at node w during the execution of* **RootedOpt**, *and exactly ℓ explorers including $a_{\max}$ exists at the time. Then,* Probe() *finishes within $O(1) + \max(0, 2\lceil \log \tau - \log \ell \rceil)$ time.*

Proof. If $\ell \geq \tau$, we have $\ell \geq \min(\Delta, k) = \Delta$ because $\ell < k$. Then, the lemma trivially holds: Probe() finishes in a constant time. Thus, we consider the case $\ell < \tau$. Let t be the time step at which $a_{\max}$ invokes Probe() at a node w, and let $z = \lceil \log \tau - \log \ell \rceil + 1$. It suffices to show that $U_\perp(w, t') = \emptyset$ or $|U_0(w, t')| \geq \ell$ holds for some $t' \in [t, t + 2z + O(1)]$. Assume for contradiction that this does not hold. For any $r \in [0, z]$, we define $f(r) = X_r \cdot 2^{r-1} + Y_r$, where $X_r = |U_0(w, t + 2r)|$ and $Y_r = |A(w, t + 2r) \setminus \{\psi(w)\}|$. By definition, $f(0) = 0 \cdot 2^{0-1} + \ell = \ell$. Under the above assumption, for any $r = 0, 1, \dots, z - 1$, the agents in $A(w, t + 2r)$ move to distinct neighbors in $U_\perp(w, t + 2r)$ in time step $t + 2r$, and bring back all settlers they find, at most one for each neighbor, in time step $t + 2r + 1$. Let α be the number of those settlers i.e., $\alpha = Y_{r+1} - Y_r$. Note that $X_{r+1} - X_r = Y_r - \alpha$ holds here. Then, irrespective of α, we

obtain

$$f(r+1) = X_{r+1} \cdot 2^r + Y_{r+1} = (X_r + Y_r - \alpha) \cdot 2^r + Y_r + \alpha$$
$$= X_r \cdot 2^r + (2^r - 1)(Y_r - \alpha) + 2 \cdot Y_r \geq 2(X_r \cdot 2^{r-1} + Y_r) = 2f(r),$$

where we use $2^r \geq 1$ and $Y_r - \alpha = X_{r+1} - X_r \geq 0$ in the above inequality. Therefore, we have $f(z) \geq \ell \cdot 2^z$, whereas we have assumed (for contradiction) that $X_z = |U_0(w, t+2z)| < \ell$, thus $f(z) = X_z \cdot 2^{z-1} + Y_z \leq (\ell - 1)2^{z-1} + Y_z$ holds. This yields $|A(w, t+2z) \setminus \{\psi(w)\}| = Y_z \geq \ell \cdot 2^z - (\ell-1)2^{z-1} = (\ell+1)2^{z-1} \geq (\ell+1)\tau/\ell > \tau$. Since $\tau = \min(\Delta, k)$, we have $\tau = \Delta$ or $\tau = k$. In the former case, $Y_k > \Delta$ agents at w are enough to visit all neighbors in $U_\perp(w, t+2z)$ in time step $t+2z$, thus $U_\perp(w, t+2z+2) = \emptyset$ holds, a contradiction. In the latter case, there are $|A(w, t+2z)| \geq k+2$ agents in w at time step $t+2z$, a contradiction. Therefore, $U_\perp(w, t') = \emptyset$ or $|U_0(w, t')| \geq \ell$ holds at time step $t' \leq t + 2z + 2$. ◀

▶ **Theorem 12.** *In the rooted setting, algorithm* **RootedOpt** *solves the dispersion problem within $O(k)$ time using $O(\Delta + \log k)$ bits of space per agent.*

Proof. The unique leader $a_{\max}$ invokes `Probe`() only when it settles an agent, except for when $a_{\max}$ itself becomes settled. Therefore, $a_{\max}$ invokes `Probe` exactly $k-1$ times, with precisely $k-i$ explorers present at the i-th invocation. By Lemma 11, the total number of steps required for the $k-1$ executions of `Probe`() is at most $\sum_{\ell=1}^{k-1}(\log k - \log \ell + O(1)) = k \log k - (\log(k!) - \log k) + O(k) = O(k)$, where we apply Stirling's formula, i.e., $\log(k!) = k \log k - k + O(\log k)$. As demonstrated in Section 3, both forward and backward moves also require a total time of $O(k)$. Thus, **RootedOpt** completes in $O(k)$ time. Regarding space complexity, the array variable `checked` is the primary factor, needing $O(\Delta)$ bits per agent. Other variables require only $O(\log \Delta)$ bits. ◀

6 Discussion

It is worth mentioning that while HEO-DFS does not function in a *fully* asynchronous model, where movement between two nodes may require an unbounded period, it does not require a *fully* synchronous model in the rooted setting. Specifically, **RootedDisp** and **RootedOpt** can operate under an asynchronous scheduler if every movement of agents between nodes is *atomic*, i.e., each agent is always located at a node and never on an edge at any time step. Under this scheduler, after the probing process is completed at a node v, unsettled agents can wait for all helping settlers to leave v before they themselves depart. Since every movement is atomic, when unsettled agents visit the home node of one of these settlers, the settler has already returned. Thus, **RootedDisp** and **RootedOpt** functions under any fair scheduler that guarantees every movement is atomic. However, this move-atomicity is not sufficient for **GeneralDisp**, because this algorithm, designed for the general setting, differentiates the moving speeds of agents based on their roles – leader, strong zombie, or weak zombie – which inherently requires a fully synchronous scheduler.

References

1 Ankush Agarwalla, John Augustine, William K Moses Jr, Sankar K Madhav, and Arvind Krishna Sridhar. Deterministic dispersion of mobile robots in dynamic rings. In *Proceedings of the 19th International Conference on Distributed Computing and Networking*, pages 1–4, 2018. doi:10.1145/3154273.3154294.

2 John Augustine and William K. Moses Jr. Dispersion of mobile robots. *Proceedings of the 19th International Conference on Distributed Computing and Networking*, January 2018.

3 Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. Fault-tolerant dispersion of mobile robots. In *Conference on Algorithms and Discrete Applied Mathematics*, pages 28–40, 2023. doi:10.1007/978-3-031-25211-2_3.

4 Archak Das, Kaustav Bose, and Buddhadeb Sau. Memory optimal dispersion by anonymous mobile robots. *Discrete Applied Mathematics*, 340:171–182, 2023. doi:10.1016/J.DAM.2023.07.005.

5 Shantanu Das. Graph explorations with mobile agents. *Distributed Computing by Mobile Entities: Current Research in Moving and Computing*, pages 403–422, 2019. doi:10.1007/978-3-030-11072-7_16.

6 Ajay D Kshemkalyani and Faizan Ali. Efficient dispersion of mobile robots on graphs. In *Proceedings of the 20th International Conference on Distributed Computing and Networking*, pages 218–227, 2019. doi:10.1145/3288599.3288610.

7 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Fast dispersion of mobile robots on arbitrary graphs. In *International Symposium on Algorithms and Experiments for Sensor Systems, Wireless Networks and Distributed Robotics*, pages 23–40. Springer, 2019. doi:10.1007/978-3-030-34405-4_2.

8 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots in the global communication model. In *Proceedings of the 21st International Conference on Distributed Computing and Networking*, pages 1–10, 2020. doi:10.1145/3369740.3369775.

9 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots on grids. In *International Workshop on Algorithms and Computation*, pages 183–197. Springer, 2020. doi:10.1007/978-3-030-39881-1_16.

10 Ajay D Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots using global communication. *Journal of Parallel and Distributed Computing*, 161:100–117, 2022. doi:10.1016/J.JPDC.2021.11.007.

11 Ajay D. Kshemkalyani and Gokarna Sharma. Near-Optimal Dispersion on Arbitrary Anonymous Graphs. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, pages 8:1–8:19, 2021. doi:10.4230/LIPICS.OPODIS.2021.8.

12 Anisur Rahaman Molla, Kaushik Mondal, and William K Moses. Byzantine dispersion on graphs. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 942–951. IEEE, 2021.

13 Anisur Rahaman Molla and William K Moses Jr. Dispersion of mobile robots: The power of randomness. In *International Conference on Theory and Applications of Models of Computation*, pages 481–500, 2019. doi:10.1007/978-3-030-14812-6_30.

14 Petrişor Panaite and Andrzej Pelc. Exploring unknown undirected graphs. *Journal of Algorithms*, 33(2):281–295, 1999. doi:10.1006/JAGM.1999.1043.

15 Vyatcheslav B Priezzhev, Deepak Dhar, Abhishek Dhar, and Supriya Krishnamurthy. Eulerian walkers as a model of self-organized criticality. *Physical Review Letters*, 77(25):5079, 1996.

16 Takahiro Shintaku, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Efficient dispersion of mobile agents without global knowledge. In *22nd International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 280–294, 2020. doi:10.1007/978-3-030-64348-5_22.

17 Yuichi Sudo, Daisuke Baba, Junya Nakamura, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. A single agent exploration in unknown undirected graphs with whiteboards. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 98(10):2117–2128, 2015. doi:10.1587/TRANSFUN.E98.A.2117.

18 Yuichi Sudo, Fukuhito Ooshita, and Sayaka Kamei. Brief announcement: Self-stabilizing graph exploration by a single agent. In *38th International Symposium on Distributed Computing*, 2024.

19 Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. Near-linear time dispersion of mobile agents, 2024. arXiv:2310.04376, doi:10.48550/arXiv.2310.04376.

Algorithm 2 The behavior of a *leader* a.

```
1  while true do
2      /*************** Slot 1 begins ***************/
3      Let w = ν(a).
4      if ∃b ∈ A_L(w) ∪ A_S(w) : a ≺ b then
5          a.level_L ← a.level_S ← a.level
6          a.leader ← b.leader      // a becomes a zombie and stops Algorithm 2
7      /*************** Slot 2 begins ***************/
8      if A(w) ≠ {a} then                  // a is an active leader if A(w) ≠ {a}
9          if ψ(w) = ⊥ ∨ ψ(w) ≺ a then
10             if ψ(w) = ⊥ then
11                 Settle one zombie in A_Z(w) at w
12             ψ(w).parent ← a.parent                   // Initially, a.parent = ⊥
13         if ∃b ∈ A_Z(w) : a.level = b.level then
14             (a.level, b.level) ← (a.level + 1, 0)
15             ψ(w).parent ← ⊥
16             a.InitProbe ← true
17         (ψ(w).leader, ψ(w).level) ← (a.ID, a.level)
18         if a.InitProbe = true then              // Initially, a.InitProbe = true
19             (ψ(w).next, ψ(w).checked, ψ(w).help, ψ(w).done) ← (⊥, −1, ⊥, false)
20             a.InitProbe ← false
21         Probe(a)                                              // See Algorithm 3
22         if ψ(w).done = true then
23             /*************** Slot 11 begins ***************/
24             if ψ(w).next = ⊥ then
25                 ψ(w).next ← ψ(w).parent                   // for backward move
26             All agents in A(w) \ {ψ(w)} move to N(w, ψ(w).next)
27             /*************** Slot 12 begins ***************/
28             a.parent ← a.pin
29             a.InitProbe ← true
```

A Detail Implementation of General Dispersion

The pseudocode for the **GeneralDisp** algorithm is shown in Algorithms 2, 3, 4, and 5. In slots 1, 2, 4–8, 11, and 12, agents operate only under the instruction of a leader. Algorithms 2 and 3 define how each leader a operates and gives instructions in those slots. Algorithm 4 defines the behavior of settlers in slot 3. Algorithm 5 specifies the behavior of zombies in slots 9 and 10. Note that each agent needs to manage an $O(1)$-bit variable to identify the slot of the current time step, but for simplicity, the process related to its update is not included in the pseudocode because it can be implemented in a naive way.

First, we explain the behavior of a leader a. Let w be the node where a is located in slot 1. We make leader election in slot 1 (lines 4–6). Leader a becomes a zombie when it finds a stronger leader or settler on w. If a becomes a zombie, it no longer runs Algorithms 2 and 3, and runs only Algorithm 5. Consider that a survives the leader election in slot 1. In slot 2, if there are no agents other than a on w, a is a waiting leader and does nothing until the next slot 1. Otherwise, the leader a (i) settles one of the accompanying zombies if w is unsettled,

Algorithm 3 Probe(a).

30 /*************** Slot 4 begins ***************/
31 Let $w = \nu(a)$.
32 $\psi(w).\texttt{next}(w) \leftarrow \begin{cases} \min P & \text{if } P \neq \emptyset \\ \bot & \text{otherwise,} \end{cases}$
33 where $P = [0, \psi(w).\texttt{checked}] \setminus \{b.\texttt{pin} \mid b \in A_S(w) \setminus \{\psi(a)\}\}$
34 $b.\texttt{help} \leftarrow \bot$ for all $b \in A_S(w)$ with $b \prec a$.
35 Let all agents $b \in A_S(w)$ with $b \prec a$ go back to their homes
36 **if** $\psi(w).\texttt{next} \neq \bot \vee \psi(w).\texttt{checked} = \delta_w - 1$ **then**
37 /*************** Slot 5 begins ***************/
38 Execute $b.\texttt{help} \leftarrow \bot$ for each $b \in A_S(w) \setminus \{\psi(w)\}$
39 Let all agents in $A_S(w) \setminus \{\psi(w)\}$ go back to their homes.
40 $\psi(w).\texttt{done} \leftarrow$ **true**
41 **else**
42 /*************** Slot 6 begins ***************/
43 Let $\{a_1, a_2, \dots, a_x\}$ be the set of agents in $A(w) \setminus \{\psi(w)\}$
44 Let $\Delta' = \min(x, \delta_w - 1 - \psi(w).\texttt{checked})$
45 Let $u_i = N(w, i + \psi(w).\texttt{checked})$ for $i = 1, 2, \dots, \Delta'$
46 **for each** $a_i \in \{a_1, a_2, \dots, a_{\Delta'}\}$ **in parallel do**
47 a_i moves to u_i.
48 /*************** Slot 7 begins ***************/
49 **if** $(a_i.\texttt{leader}, a_i.\texttt{level}) = (\psi(u_i).\texttt{leader}, \psi(u_i).\texttt{level})$ **then**
50 $a_i.\texttt{found} \leftarrow$ **true**
51 $\psi(u_i).\texttt{help} \leftarrow a_i.\texttt{pin}$
52 **else**
53 $a_i.\texttt{found} \leftarrow$ **false**
54 Move to $N(u_i, a_i.\texttt{pin})$
55 /*************** Slot 8 begins ***************/
56 **if** $\exists i \in [1, \Delta'] : a_i.\texttt{found} =$ **false then**
57 $\psi(w).\texttt{next} \leftarrow i + \psi(w).\texttt{checked}$
58 $\psi(w).\texttt{checked} \leftarrow \psi(w).\texttt{checked} + \Delta'$
59 Let all agents in $A_S(w) \setminus \{\psi(w)\}$ go back to their homes.

Algorithm 4 The behavior of a *settler* s in Slot 3.

60 /*************** Slot 3 begins ***************/
61 Move to $N(\nu(s), s.\texttt{help})$ if $s.\texttt{help} \neq \bot$.

Algorithm 5 The behavior of a *zombie* z in Slots 9 and 10.

```
62 /*************** Slot 9 begins ***************/
63 (z.level_L, z.level_S) ← (ψ(w).level, max{z'.level | z' ∈ A_Z(ν(z))})
64 if A_L(ν(z)) = ∅ and z is a weak zombie then
65 |  Move to N(ν(z), ψ(ν(z)).next)
66 /*************** Slot 10 begins ***************/
67 if A_L(ν(z)) = ∅ then
68 |  Move to N(ν(z), ψ(ν(z)).next)
```

(ii) updates its level if it finds a zombie with the same level, and (iii) gives the settler $\psi(w)$ its group identifier $(a.\mathtt{ID}, a.\mathtt{level})$ (lines 9–17). Note that settlers may leave their homes only in slots 3–8 (to join Probe()), thus a can correctly determine whether $\psi(w) = \perp$ or not here (lines 9–10). If $\psi(w) \prec a$, this procedure incorporates $\psi(w)$ into a's group, i.e., expands the territory of a. Each leader a manages a flag variable $a.\mathtt{InitProbe} \in \{\mathbf{false}, \mathbf{true}\}$, initially set to **true**. This flag is raised each time a requires probing, i.e., after it makes a forward or backward move (line 29), and when it increases its level (line 16). If the flag is raised, it initializes the variables used for Probe(), say $\psi(w).\mathtt{next}$, $\psi(w).\mathtt{checked}$, $\psi(w).\mathtt{help}$, and $\psi(w).\mathtt{done}$ in slot 2 (line 19).

Thereafter, a invokes Probe() at the end of slot 2. This subroutine runs in slots 4–8. While Probe() in **RootedDisp** returns the control to the main function after completing the probing, i.e., determining whether or not an unsettled neighbor exists, Probe() in **GeneralDisp** returns the control each time slot 8 ends even if it does not complete the probing. Consider that there are $x - 1$ accompanying zombies when a leader a begins the probing. First, a leader a and the $x - 1$ accompanying zombies join the probing. Each of them, say b, moves from a node w to one of its neighbors $u \in N(w)$ in slot 6 (line 47) and goes back to w in slot 7 (line 54). If b finds a settler in the same group at u, it sets $\psi(u).\mathtt{help}$ to $b.\mathtt{pin}$ (line 51). As long as $s.\mathtt{help} \neq \perp$, a settler s at a node v goes to a neighbor $N(v, s.\mathtt{help})$ in slot 3 (line 60, Algorithm 4). Hence, in the next slot 3, that settler $\psi(u)$ goes to w. If there are $2x$ agents at w excluding $\psi(w)$, those $2x$ agents perform the same process in the next slots 6 and 7, that is, they go to unprobed neighbors, update the help of settlers in the same group (if exists), and go back to w. In slot 8, a sends the helping settlers back to their home. The number of agents joining the probing at w, i.e., $|A(w) \setminus \{\psi(w)\}|$, doubles at each iteration of this process until they find a node without a settler in the same group or finish probing all neighbors in $N(w)$. Thus, like **RootedDisp**, the probing finishes in $O(\log \tau)$ time steps. At this time, $\psi(w).\mathtt{next} = \perp$ holds if all neighbors in $N(w)$ are settled by settlers in the same group. Otherwise, $N(w, \psi(w).\mathtt{next})$ is unsettled or settled by a settler in another group. Then, in the next slot 5, a resets the help of all settlers at w to $\perp$ except for $\psi(w)$, lets them go back to their homes, and sets $\psi(w).\mathtt{done}$ to **true**, indicating that the probing is done (lines 38–40). The probing process described above may be prevented by a stronger leader b when b visits a node $v \in N(w)$ such that $\psi(v)$ belongs to a's group and $\psi(v).\mathtt{help} \neq \perp$. Then, b incorporates $\psi(v)$ into b's group, and set $\psi(v).\mathtt{help}$ to $\perp$ (line 19), so $\psi(v)$ never goes to w to help a's probing. However, this event actually speeds up a's probing: a identifies this event when noticing that $\psi(v)$ does not arrive at w in the next slot 4. As a result, a can set $\psi(w).\mathtt{next}$ to p where $N(w, p) = v$ (lines 32–34).

Note that, even during the probing process at node w, leader a might become a zombie if it meets a stronger leader b in slot 1. Some settlers might then move to w in the next slot 3 to help a, not knowing a is now a zombie. In these situations, b changes the help of these

settlers to $\perp$ and sends them back to their homes in slot 4. Thereafter, those settlers remain at their home at least until they are incorporated into another group.

If a leader a at w observes $\psi(w)$.done = **true**, it makes a forward or backward move in slot 11 (lines 22–29). Each time a makes a forward or backward move to a node u, it remembers a.pin in a.parent after the move (line 28). This port number will be stored on the variable $\psi(u)$.parent when a settles a zombie on u or a incorporates $\psi(u)$ from the territory of another group. Note that this event occurs only when the last move is forward. Thus, like **RootedDisp**, a constructs a DFS tree in its territory. It is inevitable to use a variable a.parent tentatively since a.pin is updated every step by definition of a special variable pin and a may become a waiting leader after moving to u. Unlike **RootedDisp**, a records the most recently used port to move in $\psi(\nu(a))$.next even when it makes a backward move (lines 24–25). This allows a zombie to chase a leader.

The behavior of zombies in slots 9 and 10 is very simple (lines 61–67). A zombie always updates its location and swarm levels in slot 9 (line 62). A zombie z not accompanying a leader always chases a leader by moving through the port $\psi(\nu(z))$.next. As mentioned earlier, we differentiate the chasing speed of weak zombies and strong zombies. Specifically, weak zombies move in both slots 9 and 10, while strong zombies move only in slot 10 (lines 63–67).

▶ **Lemma 13.** *The location level of a zombie is monotonically non-decreasing.*

Proof. Neither a leader nor a settler decreases its level in **GeneralDisp**. When a zombie z does not accompany a leader, it chases a leader through port $\psi(\nu(z))$.next. This port $\psi(\nu(z))$.next is updated only if a leader makes a forward or backward move from $\nu(z)$, and the leader updates the level of $\psi(N(\nu(z), \psi(\nu(z)).\texttt{next}))$ if it is smaller than its level. Thus, a zombie never decreases its location level by chasing a leader. When a zombie z accompanies a leader, the leader copies its level to $\psi(\nu(z))$.level in slot 2, which is copied to $z.\texttt{level}_L$ in slot 8. The leader that z accompanies may change but does not change to a weaker leader. Thus, a zombie never decreases its location level when accompanying a leader. ◀

B Proofs of Theorem 10

Remember that A_Z and A_{AL} are the set of zombies and the set of active leaders, respectively, in the whole graph. We have the following lemma.

▶ **Lemma 14.** *For any $i \geq 0$, the number of weak zombies with a location level i is monotonically non-increasing starting from any configuration where* $\min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\}) = i$.

Proof. Let C be a configuration where $\min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\}) = i$. When a leader with level i becomes a zombie, its location level is i (line 5). So, a leader with level i may become a strong zombie with a location level i but never becomes a weak zombie with a location level i. The swarm level of a zombie decreases only when the zombie accompanies a leader (and this leader settles another zombie). Thus, a strong zombie with a location level i that does not accompany a leader cannot become a weak zombie without increasing its location level. Moreover, starting from C, a strong zombie with location level i must increase its location level when it encounters a leader in slot 1. Hence, the number of weak zombies with a location level i is monotonically decreasing. ◀

▶ **Lemma 15.** $\min(\{a.\texttt{level} \mid a \in A_{AL}\} \cup \{z.\texttt{level}_L \mid z \in A_Z\})$ *is monotone non-decreasing and increases by at least one in every $O(k \log \tau)$ time steps unless $A_{AL} \cup A_Z$ becomes empty.*

Proof. Let i be an integer $i \geq 0$ and C a configuration where $\min(\{a.\mathtt{level} \mid a \in A_{AL}\} \cup \{z.\mathtt{level}_L \mid z \in A_Z\}) = i$. It suffices to show that leaders with level i and zombies with location level i disappear in $O(k \log \tau)$ time steps starting from C.

Consider an execution starting from C. By Lemma 14, a weak zombie with location level i is never newly created in this execution. Let z be any weak zombie with a location level i that does not accompany a leader in a configuration C. In every 12 slots, z moves twice, while a strong zombie and a leader move only once, excluding the movement for the probing. Therefore, z catches up to a strong zombie and becomes strong too, catches up to a leader with level i, or increases its location level in $O(k)$ time steps. When z catches up to a leader, it joins the HEO-DFS of the leader, or this leader becomes a zombie. In the latter case, z becomes a strong zombie. Thus, z settles or becomes a strong zombie (with the current leader) in $O(k \log \tau)$ time steps. Therefore, the number of weak zombies with location level i becomes zero in $O(k \log \tau)$ steps. After that, no waiting leader with level i resumes its HEO-DFS without increasing its level because there is no weak zombie with location level i. Therefore, every active leader with location level i becomes a zombie with location level at least $i + 1$ or a waiting leader in $O(k \log \tau)$ steps. Thus, active leaders with location level i also disappear in $O(k \log \tau)$ steps. From this time, no leader moves in the territory of a group with level i or less. Hence, every strong zombie with location level i increases its location level or catches up to a waiting leader. Since the level of a waiting leader is at least i, the latter event also increases z's level by at least one. ◀

Since an agent in **GeneralDisp** manages only a constant number of variables, each with $O(\log(k + \Delta))$ bits, Lemmas 9 and 15 yield Theorem 10.