Dispersion is (Almost) Optimal under (A)synchrony

Ajay D. Kshemkalyani
ajay@uic.edu
University of Illinois Chicago
Chicago, IL, 60607, USA

Manish Kumar*
manishsky27@gmail.com
Indian Institute of Technology Madras
Chennai, 600036, India

Anisur Rahaman Molla†
anisurpm@gmail.com
Indian Statistical Institute
Kolkata, 700108, India

Debasish Pattanayak
drdebmath@gmail.com
University of Ottawa
Ottawa, Ontario, Canada

Gokarna Sharma
sharma@cs.kent.edu
Kent State University
Kent, OH, 44242, USA

Abstract

The dispersion problem has received much attention recently in the distributed computing literature. In this problem, $k \leq n$ agents placed initially arbitrarily on the nodes of an n-node, m-edge anonymous graph of maximum degree Δ have to reposition autonomously to reach a configuration in which each agent is on a distinct node of the graph. Dispersion is interesting as well as important due to its connections to many fundamental coordination problems by mobile agents on graphs, such as exploration, scattering, load balancing, relocation of self-driven electric cars (robots) to recharge stations (nodes), etc. The objective has been to provide a solution that optimizes simultaneously time and memory complexities. There exist graphs for which the lower bound on time complexity is $\Omega(k)$. Memory complexity is $\Omega(\log k)$ per agent independent of graph topology. The state-of-the-art algorithms have (i) time complexity $O(k \log^2 k)$ and memory complexity $O(\log(k+\Delta))$ under the synchronous setting [DISC'24] and (ii) time complexity $O(\min\{m, k\Delta\})$ and memory complexity $O(\log(k+\Delta))$ under the asynchronous setting [OPODIS'21]. In this paper, we improve substantially on this state-of-the-art. Under the synchronous setting as in [DISC'24], we present the first optimal $O(k)$ time algorithm keeping memory complexity $O(\log(k+\Delta))$. Under the asynchronous setting as in [OPODIS'21], we present the first algorithm with time complexity $O(k \log k)$ keeping memory complexity $O(\log(k+\Delta))$, which is time-optimal within an $O(\log k)$ factor despite asynchrony. Both the results were obtained through novel techniques to quickly find empty nodes to settle agents, which may be of independent interest.

CCS Concepts

• **Theory of computation** **Distributed algorithms**; • **Computer systems organization** **Robotics**; • **Computing methodologies** **Distributed computing methodologies**.

Keywords

Distributed algorithms, Mobile agents, Local communication, Dispersion, Time and memory complexity

ACM Reference Format:
Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, Debasish Pattanayak, and Gokarna Sharma. 2025. Dispersion is (Almost) Optimal under (A)synchrony. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28-August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3694906.3743317

1 Introduction

We consider the dispersion problem which has been extensively studied in the distributed computing literature recently. The *dispersion* problem asks $k \leq n$ mobile agents placed initially arbitrarily on the nodes of an n-node anonymous graph to reposition autonomously to reach a configuration in which each agent is on a distinct node of the graph. This problem is interesting as well as important due to its connections to many fundamental agent coordination problems, such as exploration, scattering, load balancing, relocating self-driven electric cars (agents) to recharge stations (nodes), etc. The objective has been to provide a solution that optimizes simultaneously time and memory complexities. Time complexity is measured as the time duration to achieve dispersion starting from any initial configuration. Memory complexity is measured as the number of bits stored in the persistent memory at each agent (graph nodes are memory-less and cannot store any information). There exist graphs for which any dispersion algorithm exhibits time complexity $\Omega(k)$. Suppose two nodes of a graph are at a distance of $\Omega(k)$, then to disperse from one node to the furthest node takes $\Omega(k)$ time (e.g., line graph). Memory complexity is $\Omega(\log k)$ for any solution since at least one agent needs $\Omega(\log k)$ bits to store its unique ID.

There are two state-of-the-art algorithms, one in the synchronous setting due to Sudo, Shibata, Nakamura, Kim, and Masuzawa [40] appeared recently in [DISC'24] and another in the asynchoronous setting due to Kshemkalyani and Sharma [28] appeared in [OPODIS'21]. In the *synchronous* setting ($\mathcal{SYNC}$), all agents perform their operations simultaneously in synchronized rounds (or steps), and hence time complexity (of the algorithm) is measured in rounds. However, in the *asynchronous* setting ($\mathcal{ASYNC}$), agents become active at arbitrary times and perform their operations in arbitrary duration, and hence time complexity is measured in epochs – an *epoch* represents the time interval in which each

*M. Kumar is supported by CyStar at IIT Madras.
†A. R. Molla is supported, in part, by ANRF-SERB Core Research Grant, file no. CRG/2023/009048 and R. C. Bose Centre's internal research grant.

SPAA '25, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/2025/07
https://doi.org/10.1145/3694906.3743317

Model	Paper	Time Complexity	Memory Complexity
Any	-	$\Omega(k)$	$\Omega(\log k)$
Rooted Initial Configurations			
$\mathcal{SYNC}$	[40]	$O(k)$	$O(\Delta + \log k)$
	[40]	$O(k \log k)$	$O(\log(k+\Delta))$
	Theorem 5.1	$O(k)$	$O(\log(k+\Delta))$
$\mathcal{ASYNC}$	[27]	$O(D\Delta(D+\Delta))$	$O(D + \Delta \log k)$
	[28]	$O(\min\{m, k\Delta\})$	$O(\log(k+\Delta))$
	Theorem 6.1	$O(k \log k)$	$O(\log(k+\Delta))$
General Initial Configurations			
$\mathcal{SYNC}$	[40]	$O(k \log^2 k)$	$O(\log(k+\Delta))$
	Theorem 7.1	$O(k)$	$O(\log(k+\Delta))$
$\mathcal{ASYNC}$	[28]	$O(\min\{m, k\Delta\})$	$O(\log(k+\Delta))$
	Theorem 7.2	$O(k \log k)$	$O(\log(k+\Delta))$

Table 1: An illustration of the proposed dispersion results and comparison with the state-of-the-art.

agent becomes active at least once. In $\mathcal{SYNC}$, an epoch is a round. In particular, Sudo, Shibata, Nakamura, Kim, and Masuzawa [40] [DISC'24] presented an algorithm with time complexity $O(k \log^2 k)$ rounds and memory complexity $O(\log(k + \Delta))$ bits in $\mathcal{SYNC}$. Kshemkalyani and Sharma [28] [OPODIS'21] presented an algorithm with time complexity $O(\min\{m, k\Delta\})$ epochs and memory complexity $O(\log(k + \Delta))$ bits in $\mathcal{ASYNC}$. Here m and Δ are the number of edges and the maximum degree of the graph respectively.

Contributions. In this paper, we improve substantially on this state-of-the-art through two results, one in $\mathcal{SYNC}$ as in [40] and another in $\mathcal{ASYNC}$ as in [28]. Table 1 compares and contrasts our results with the state-of-the-art. We say initial configuration *rooted*, if all k robots are initially at the same node, *general* otherwise. Our algorithms have the same time and memory complexities for both rooted and general initial configurations. However, in Table 1, we list them separately to compare better with the previous bounds of Sudo, Shibata, Nakamura, Kim, and Masuzawa [40] [DISC'24] which differ by $O(\log k)$ factor in time complexities.

- Under $\mathcal{SYNC}$ as in Sudo, Shibata, Nakamura, Kim, and Masuzawa [40] [DISC'24], we present the first time optimal $O(k)$-round algorithm with memory complexity $O(\log(k + \Delta))$ bits, which is an $O(\log^2 k)$ factor improvement on [40].
- Under $\mathcal{ASYNC}$ as in Kshemkalyani and Sharma [28] [OPODIS'21], we present the first algorithm with time complexity $O(k \log k)$ epochs and memory complexity $O(\log(k+\Delta))$ bits. Time is optimal within an $O(\log k)$ factor and a substantial improvement over $O(\min\{m, k\Delta\})$ epochs in [28].

The above results are possible from two techniques, one for $\mathcal{SYNC}$ and another for $\mathcal{ASYNC}$, we develop in this paper, which we describe in a nutshell, in Section 3 (Overview of Challenges and Techniques). Our technique for $\mathcal{SYNC}$ finds a fully unsettled empty neighbor (if exists) in $O(1)$ rounds and handles meetings of two DFSs with $O(1)$ overhead to achieve $O(k)$-round algorithm with $O(\log(k + \Delta))$ memory; a *fully unsettled* node is the one that has no agent positioned on it yet. Our technique for $\mathcal{ASYNC}$ finds a fully unsettled empty neighbor (if exists) in $O(\log k)$ epochs and handles meetings of two DFSs with $O(1)$ overhead to achieve $O(k \log k)$-epoch algorithm with $O(\log(k + \Delta))$ memory.

Related work. The state-of-the-art results on dispersion are listed in Table 1 as well as the established results. The state-of-the-art in $\mathcal{SYNC}$ is the result due to [40] appeared in [DISC'24], whereas the result due to [28] appeared in [OPODIS'21] is the state-of-the-art in $\mathcal{ASYNC}$. These two results solve the problem starting from any initial configuration (rooted or general).

For the rooted initial configurations, some better bounds were obtained. In $\mathcal{SYNC}$, there are two results [40] [DISC'24]: (i) time complexity optimal $O(k)$ with memory complexity $O(\Delta+\log k)$ and (ii) time complexity $O(k \log k)$ with memory complexity $O(\log(k + \Delta))$. Our $\mathcal{SYNC}$ result achieves time optimality keeping memory $O(\log(k+\Delta))$. In $\mathcal{ASYNC}$, there is one result [27] [JPDC'22]: time complexity $O(D\Delta(D+\Delta))$ with memory complexity $O(D+\Delta \log k)$. Our $\mathcal{ASYNC}$ result makes time optimal within an $O(\log k)$ factor keeping memory $O(\log(k + \Delta))$.

The state-of-the-art results discussed in the previous two paragraphs were the culmination of the long series of work [1, 3, 4, 6, 10, 15–19, 24–26, 28, 30, 35, 36, 38]. The majority of works considered the faulty-free case, except [4, 31, 32] which considered *Byzantine* faults (where agents might act arbitrarily) and [3, 4, 6, 33] considered *crash* faults (where some agents might stop working permanently at any time). Moreover, most of the works considered the *local communication* model where only agents co-located at a node can communicate at any epoch/round, except Kshemkalyani *et al.* [27] who considered the *global communication* model in which agents can communicate irrespective of their positions on the graph at any epoch/round. Furthermore, most of the works considered static graphs, except [26, 35, 36] which considered different models of dynamic graphs. Moreover, most of the works presented deterministic algorithms except [10, 30] where randomness is used to minimize the memory complexity. Dispersion is considered with restricted local communication among co-located robots in [15]. Reaching restricted final configurations, such as no two adjacent nodes can contain agents, are considered in [17]. The majority of papers considered arbitrary graphs except for some papers where special graphs were considered, for example, grid [3, 4, 25], ring [1, 32], and trees [1, 27]. Finally, results in [6, 24] were established assuming parameters n, m, Δ, k known to the agents a priori.

Additionally, dispersion is closely related to graph exploration which has been quite extensively studied for specific as well as arbitrary graphs, e.g., [2, 7, 12, 14, 29]. Another related problem is scattering which has been studied for rings [13, 37] and grids [5, 11, 34]. Dispersion is also related to the load balancing problem, where a given load has to be (re-)distributed among several nodes, e.g., [9, 39]. Recently, dispersion solutions have been used in [20, 21, 23]

electing a leader among agents and solving graph-level tasks such as computing minimum spanning tree (MST), maximal independent set (MIS), and minimal dominating set (MDS) problems.

Paper organization. We discuss model and some preliminaries in Section 2. We overview challenges and techniques developed in Section 3. We build some techniques crucial for our $\mathcal{SYNC}$ and $\mathcal{ASYNC}$ algorithms in Section 4. We discuss the $\mathcal{SYNC}$ and $\mathcal{ASYNC}$ rooted dispersion algorithms in Sections 5 and 6, respectively. We then discuss algorithms handling general initial configurations in Section 7. Finally, we conclude in Section 8.

2 Model, Notations, and Preliminaries

Graph. Let $G = (V, E)$ be a simple, undirected, and connected arbitrary graph with $n = |V|$ nodes, $m = |E|$ edges, and diameter D. We denote the set of *neighbors* of a node $v \in V$ by $N(v) = \{u \in V \mid \{u, v\} \in E\}$. We denote the *degree* of $v \in V$ by $\delta_v = |N(v)|$. We have that $\Delta = \max_{v \in V} \delta_v$, i.e., Δ represents the *maximum degree* of G. The graph G is *anonymous*, meaning that the nodes in V do not have identifiers. However, G is *port-labeled* meaning that the edges incident to a node v are locally labeled at v such that an agent located at v can uniquely distinguish these edges. Specifically, these edges are assigned distinct labels $1, 2, \ldots, \delta_v$ at node v. We refer to these local labels as *port numbers*. The port number at v for an edge $\{v, u\}$ is denoted by $p_v(u)$. Since each edge $\{v, u\}$ has two endpoints, it is assigned two labels, $p_v(u)$ and $p_u(v)$, at nodes v and u, respectively. Note that these labels are independent, meaning $p_u(v) \neq p_v(u)$ may hold, and hence there is no correlation between port numbers assigned for any two nodes $u, v \in G$. 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 $v \in V$. Each node $v \in V$ is memory-less and cannot retain any information.

Agents. We consider the set $\mathcal{A} = \{a_1, \ldots, a_k\}$ of $k \leq n$ agents positioned initially on the nodes of G. The agents have identifiers, i.e., each agent a_i has a positive integer as its identifier $a_i.\texttt{ID}$. The identifiers are unique meaning that $a_i.\texttt{ID} \neq a_j.\texttt{ID}$ for any $a_i, a_j \in \mathcal{A}$, $j \neq i$. We assume that $a_i.\texttt{ID} \in [1, k^{O(1)}]$. In $\mathcal{ASYNC}$, an agent a_i might see another agent a_j on an edge of G but when an epoch (defined formally later) finishes, it is guaranteed that no agent is on an edge, i.e., all agents are on the nodes of G. In $\mathcal{SYNC}$, no agent sees another agent at any edge of G. Each agent a_i positioned at node v has a read-only variable $a_i.\texttt{pin} \in \{1, 2, \ldots, \delta_v\} \cup \{\perp\}$. The execution starts at time $t = 0$. At $t = 0$, $a_i.\texttt{pin} = \perp$ holds. For any time $t \geq 1$, if a_i moves from v to u, then $a_i.\texttt{pin}$ is set to $p_u(v)$ (the port of u incoming from v) at the beginning of round t. We refer to the value of $a_i.\texttt{pin}$ as the *incoming port* of a_i. The values of all variables in the agent a_i, including its identifier $a_i.\texttt{ID}$ and the special variables $a_i.\texttt{pin}$ and $a_i.\texttt{pout}$, constitute the state of a_i. The $a_i.\texttt{pout}$ variable denotes the *outgoing port* which, for agent a_i current at node v, is the port used by a_i to exit v. We use notations $\alpha(a_i)$ and $\alpha(w)$ to denote node where agent a_i resides and agent that resides at node w, respectively, i.e., for agent a_i currently residing at node w, $\alpha(a_i) = w$ and $\alpha(w) = a_i$.

Time cycle. At any time, an agent a_i could be active or inactive. When becomes active, a_i performs the "Communicate-Compute-Move" (CCM) cycle as follows. **Communicate:** Agent a_i positioned at node u can observe the memory of another agent a_j positioned at node u. Agent a_i can also observe its own memory. **Compute:** Agent a_i may perform an arbitrary computation using the information observed during the "communicate" portion of that cycle. This includes the determination of a port to use to exit u and the information to store in the agent a_j that is at u. **Move:** At the end of the cycle, a_i writes new information (if any) in the memory of an agent a_j at u, and exits u using the computed port and reaches a neighbor of u.

Round, epoch, time, and memory complexity. In $\mathcal{SYNC}$, time complexity is measured in rounds (a cycle is a round). In $\mathcal{ASYNC}$, time complexity is measured in epochs. An *epoch* is the smallest time interval within which each robot is active at least once [8]. We will use the term "time" generically to mean rounds for $\mathcal{SYNC}$ and epochs for $\mathcal{ASYNC}$. Memory complexity is the number of bits stored at any agent over one CCM cycle to the next. The temporary memory needed during the Compute phase is considered free.

3 Overview of Challenges and Techniques

3.1 Challenges

The literature on dispersion relied mostly on search techniques, depth-first-search (DFS) and breadth-first-search (BFS). DFS has been preferred over BFS since it facilitated optimizing memory complexity along with time complexity. DFS time complexity depends on how quickly a fully unsettled empty neighbor node can be found from the current node to settle an agent. Recall that a fully unsettled node is the one that has no agent positioned on it yet. Prior to the result of Sudo *et al.* [40] appeared in [DISC'24], the fully unsettled neighbor search needed $O(\Delta)$ time. Additionally, $k - 1$ different nodes need to be visited to settle k agents (one per node). Therefore, the dispersion was solved in $O(k\Delta)$ time; precisely $O(\min\{m, k\Delta\})$ time since DFS finishes after examining all the edges in the graph. Kshemkalyani and Sharma [28] proved that this bound applies to both $\mathcal{SYNC}$ and $\mathcal{ASYNC}$ and is the state-of-the-art bound in $\mathcal{ASYNC}$. Sudo *et al.* [40] developed a clever technique in $\mathcal{SYNC}$ to find a fully unsettled empty neighbor node in $O(\log k)$ time. Using this technique, they achieved dispersion for rooted initial configurations in $O(k \log k)$ time, improving significantly on $O(\min\{m, k\Delta\})$ time complexity.

In general initial configurations, let ℓ be the number of nodes with agents on them (for the rooted case, $\ell = 1$). There will be ℓ DFSs initiated from those ℓ nodes. It may be the case that two or more DFSs *meet*. The meeting situation needs to be handled in a way that ensures it does not increase the time required to find fully unsettled neighbor nodes. Kshemkalyani and Sharma [28] showed that such meetings can be handled in additional time proportional to $O(\min\{m, k\Delta\})$ in both $\mathcal{SYNC}$ and $\mathcal{ASYNC}$, and hence overall time complexity remained $O(\min\{m, k\Delta\})$ for general initial configuration, the same time complexity as in rooted initial configurations. Sudo *et al.* [40] handled such meetings in $\mathcal{SYNC}$ with $O(\log k)$ factor overhead compared to $O(k \log k)$ time for the rooted initial configurations and hence time complexity became $O(k \log^2 k)$ for general initial configurations in $\mathcal{SYNC}$.

DFS starts in the forward phase and works alternating between forward and backtrack phases until $k - \ell$ fully unsettled nodes are visited (ℓ nodes already have agents and one on each such nodes

can settle there). During each forward phase from one node to another, one such fully unsettled node becomes settled. To visit k different fully unsettled nodes, even when starting from a rooted initial configuration, a DFS must perform at least $k-1$ forward phases, and at most $k-1$ backtrack phases. Therefore, the best DFS time complexity is $2(k-1) = O(k)$, which is asymptotically optimal since in graphs (such as line) exactly $k-1$ forward phases are needed in the worst-case (consider the case of all k agents are on either end node of the line graph). Therefore, the challenge is how to guarantee only $k-1$ forward phases and each forward phase to finish in $O(1)$ time to obtain $O(k)$ time bound.

In this paper, in $\mathcal{SYNC}$, we devise techniques to find, from any node, a fully unsettled empty neighbor (if exists) in $O(1)$ rounds and handle meetings of two DFSs with $O(1)$ overhead, achieving time-optimal $O(k)$-round algorithm with $O(\log(k+\Delta))$ memory per agent. Despite the efficacy of the developed techniques in $\mathcal{SYNC}$ achieving time optimality with $O(\log(k+\Delta))$ memory, we are not able to extend the $\mathcal{SYNC}$ techniques to $\mathcal{ASYNC}$, which we describe later. Nevertheless, we are able to devise techniques to find, from any node, a fully unsettled empty neighbor (if exists) in $O(\log k)$ epochs. Interestingly, our $\mathcal{ASYNC}$ technique extends the $\mathcal{SYNC}$ technique of Sudo *et al.* [40] that achieved $O(k \log^2 k)$-round solution for dispersion in $\mathcal{SYNC}$. Additionally, our technique handles meetings of two DFSs with $O(1)$ overhead even in $\mathcal{ASYNC}$, achieving $O(k \log k)$-epoch algorithm with $O(\log(k+\Delta))$ memory in $\mathcal{ASYNC}$. This result is interesting and important since it answers an open question from Sudo *et al.* [40] in the affirmative about whether $o(\min\{m, k\Delta\})$-time algorithm could be designed for dispersion in $\mathcal{ASYNC}$ with $O(\log(k+\Delta))$ memory. Furthermore, our $\mathcal{ASYNC}$ result improves the $\mathcal{SYNC}$ time bound of Sudo *et al.* [40] by an $O(\log k)$ factor. It remains open whether our $\mathcal{SYNC}$ technique can be extended to achieve $O(k)$-epoch algorithm with $O(\log(k+\Delta))$ memory in $\mathcal{ASYNC}$.

3.2 Techniques for $\mathcal{SYNC}$

As discussed above, to solve dispersion in $O(k)$ rounds, the number of forward phases should be $O(k)$ and each forward phase should take $O(1)$ rounds. To be able to run the forward phase in $O(1)$ rounds at a node v, a fully unsettled neighbor node of v should be found in $O(1)$ rounds, where an agent can settle. A simple idea to guarantee $O(1)$ rounds for the forward phase at v is to probe all δ_v neighbors of v in parallel (which we call *synchronous probing*). To probe δ_v neighbors in parallel, v needs at least δ_v agents positioned. However, let $k' \leq k$ be the number of agents at v. When $k' < o(\delta_v)$, synchronous probing cannot finish in $O(1)$ rounds. We know that using the probing technique of Sudo *et al.* [40], synchronous probing can be done in $O(\log \delta_v)$ rounds, even when $k' < o(\delta_v)$. Therefore, dispersion needs $O(k + \Delta \log \Delta)$ time in the worst-case, which is $O(k \log k)$, when $\Delta = \Theta(k)$, matching [40].

Our proposed technique runs synchronous probing in $O(1)$ rounds. Particularly, we guarantee that, during DFS, $\lceil k/3 \rceil$ agents are available for synchronous probing. We make $\lceil k/3 \rceil$ agents available leaving $\geq \lceil k/3 \rceil$ fully unsettled nodes empty during DFS, i.e., the agents cannot settle on $\geq \lceil k/3 \rceil$ nodes of the associated DFS tree T_{DFS} until DFS finishes. Therefore, the $\lceil k/3 \rceil$ agents can finish synchronous probing at any node in $O(1)$ rounds. If node v has $\delta_v \leq \lceil k/3 \rceil$, then $\lceil k/3 \rceil$ agents can probe all δ_v neighbors of v in 2 rounds. For $\delta_v > \lceil k/3 \rceil$, notice that probing only $\min\{k, \delta_v\}$ neighbors of v is enough to find a fully unsettled empty neighbor of v (if exists). This is because no more than k nodes are going to be occupied with agents at any time since there are k agents. Therefore, with $\lceil k/3 \rceil$ agents, probing at v can finish in at most 3 iterations, i.e., $O(1)$ rounds.

We provide an illustration of synchronous probing in Fig. 5. Although having $\lceil k/3 \rceil$ agents helps in finishing probing in $O(1)$ rounds, guaranteeing the availability of $\lceil k/3 \rceil$ agents creates three major challenges Q1–Q3 below. We need some notations. Suppose each agent maintains a variable $\texttt{settled} \in \{\bot, \top\}$. Initially, the $\lceil k/3 \rceil$ agents start as *seeker* and the remaining $\lfloor 2k/3 \rfloor$ agents start as *explorer*. An explorer becomes a 'settler' once DFS settles it at a node. A settler is 'non-oscillating' if it never leaves the node where it is settled. A settler may become 'oscillating' if it leaves the node where it is settled to help run DFS (i.e., doing an oscillation trip). After DFS finishes, an oscillating settler returns to the node where it is settled initially, becomes non-oscillating, and remains there. These concepts will be more apparent during our discussions later. We say an explorer agent a_i a *settler* when $a_i.\texttt{settled} = \top$. The seeker agents are used in synchronous probing and they settle only after synchronous probing is not needed anymore.

Q1. Which DFS tree nodes to leave empty? Since we allocate $\lceil k/3 \rceil$ agents as seeker agents, they cannot settle during DFS. Additionally, the DFS needs to visit k nodes, i.e., the DFS tree T_{DFS} should have size k. For this, we need to leave at least $\lceil k/3 \rceil$ nodes of T_{DFS} empty. The challenge is how to meet such requirements. We carefully leave the nodes of T_{DFS} empty such that each such empty node has a settler within 2 hops in T_{DFS}. The idea is to leave the nodes of T_{DFS} at odd depth (depth of the root is 0) empty. This is sufficient when no node in T_{DFS} is a branching node (i.e., a degree more than 2). For branching nodes (depending on their depth), we carefully decide on whether to (i) put extra settlers on their empty children nodes or (ii) remove some settlers from their non-empty children nodes. We prove that our approach leaves at least $\lceil k/3 \rceil$ nodes of T_{DFS} empty until DFS finishes. Therefore, DFS finishes with having k nodes in T_{DFS} but with only at most $\lfloor 2k/3 \rfloor$ nodes of T_{DFS} occupied with settlers. This guarantees $\lceil k/3 \rceil$ seeker agents we allocated are available to do synchronous probing until DFS finishes. We discuss the algorithm in detail in Section 4 (Algorithm to Select Empty Nodes) with illustrations (Fig. 1).

Q2. How to successfully run DFS despite being some of its tree nodes empty? Suppose a settler a_j is positioned on a node of T_{DFS}. Let that node be a_j's *home node*. We ask settler a_j to *oscillate* in a round-robin manner covering the empty nodes starting from and ending at its home node. We call settler a_j that oscillates an *oscillating settler*. We guarantee that an oscillating settler at depth d (the root of T_{DFS} is at depth 0) only needs to cover at most 3 empty children nodes at depth $d+1$ or at most 2 sibling nodes at the same depth d in T_{DFS}. This guarantees the matching of settlers to empty nodes such that each settler's round-robin oscillation trip finishes in (at most) 6 rounds. Some of the settlers may

never oscillate and they are called *non-oscillating settlers*. A non-oscillating settler never leaves its home node. Some settlers may initially be non-oscillating, become oscillating over time, and finally become non-oscillating. When an oscillating settler is not at its home, we say that it is at its *oscillating home node*. We provide illustrations in Figs. 2 and 4.

Q3. How to settle seeker agents to the empty nodes after DFS finishes? After having k nodes in T_{DFS}, the DFS finishes. The $\lceil k/3 \rceil$ seeker agents (and remaining explorers, if any) first go to the root of T_{DFS} as a group following the parent pointers in T_{DFS}. Then, they re-traverse T_{DFS}, starting from the root node, to position themselves on the empty nodes in T_{DFS}. We make this re-traversal process finish in $O(k)$ time with $O(\log(k+\Delta))$ memory through the use of a *sibling pointer* technique which we develop. Without sibling pointers, the memory requirement for re-traversal becomes $O(\Delta \log(k+\Delta))$[1]. An oscillating setter, once its job is done, stops oscillating, returns to its home node, transforms itself as a non-oscillating settler and settles.

Handling general initial configurations. So far we discussed techniques to achieve $O(k)$ time complexity for rooted initial configurations. In general initial configurations, there will be ℓ DFSs initiated from ℓ nodes (ℓ not known). Each DFS follows the approach as in the rooted case. Let a node has k_1 agents running DFS D_1. We show that having $\geq \lceil k_1/3 \rceil$ agents as seekers is enough to finish D_1 if D_1 does not meet any other DFS, say D_2. This provides the guarantee that, if no two DFSs ever meet, dispersion is achieved. In case of a meeting, we develop an approach that handles the meeting of two DFSs D_1 and D_2 with overhead the size of the larger DFS between D_1 and D_2. In other words, $k_1 + k_2$ agents belonging to D_1 and D_2 disperse in $O(k_1 + k_2)$ rounds, if a meeting with the third DFS D_3 never occurs during DFS. If that meeting occurs, we show that the time complexity becomes $O(k_1+k_2+k_3)$ rounds. Therefore, the worst-case time complexity starting from any ℓ nodes becomes $O(k)$ rounds. Specifically, to achieve this runtime, we extend the *size-based subsumption* technique of Kshemkalyani and Sharma [28] [OPODIS'21] which works as follows. Suppose DFS D_1 meets DFS D_2 at node w (notice that w belongs to D_2). Let $|D_i|$ denote the number of agents settled from DFS D_i (in other words, the number of nodes in the DFS tree T_{D_i} built by D_i so far). D_1 subsumes D_2 if and only if $|D_2| < |D_1|$, otherwise D_2 subsumes D_1. The agents settled from subsumed DFS are collected and given to the subsuming DFS to continue with its DFS, which essentially means that the subsumed DFS does not exist anymore. This subsumption technique guarantees that one DFS out of ℓ' met DFSs (from ℓ nodes) always remains subsuming and grows monotonically until all agents settle.

3.3 Techniques for $\mathcal{ASYNC}$

Although the techniques discussed in Section 3.2 provided optimal $O(k)$-round solution in $\mathcal{SYNC}$ with $O(\log(k+\Delta))$ memory, we are not able to make it work to obtain the same $O(k)$-epoch solution in $\mathcal{ASYNC}$. Since some of the nodes are left empty in the algorithm, there are two kinds of empty nodes: (Type A) fully unsettled empty and (Type B) empty because the robot settled at a node is not currently present at that node. The $\mathcal{SYNC}$ technique relies on oscillation to differentiate Type A nodes from Type B nodes. In $\mathcal{SYNC}$, this decision can be made if an agent does not visit a node in 6 rounds; that is, if the node is of type B, then an agent will visit that node in every 6 rounds, otherwise it is a Type A node. In $\mathcal{ASYNC}$, making this decision is difficult since agents do not have an agreement on the duration and start/end of each cycle (i.e., no agreed-upon round definition). Surprisingly, we are able to extend the $\mathcal{SYNC}$ technique of Sudo *et al.* [40] to $\mathcal{ASYNC}$ such that finding a fully unsettled neighbor node and moving to that neighbor finishes in $O(\log k)$ epochs. This extension indeed shows that the probing technique of Sudo *et al.* [40] is not inherently dependent on synchrony assumption.

We first discuss the challenge for such an extension and how we overcome it. Suppose the DFS is currently at node w with at least three agents a_1, a_2, a_3 on it, a_1 settled at w and a_2, a_3 need to be settled on other nodes. To settle a_2, a fully unsettled neighbor node of w needs to be found. The idea of Sudo *et al.* [40] for $\mathcal{SYNC}$ is as follows. a_2 leaves w via port-1 to visit port-1 neighbor. Suppose a_2 returns with the knowledge that port-1 neighbor is empty, which is in fact the fully unsettled neighbor node of w due to $\mathcal{SYNC}$. The DFS then makes a forward move to port-1 neighbor, a_2 settles at that node, and the DFS continues at that node to find a fully unsettled node to settler a_3. Suppose a_2 returns with the knowledge that port-1 neighbor is non-empty (has an agent settled). While returning to w, a_2 brings that agent (say a_{p1}) to w. Next time, a_2 and a_{p1} visit in parallel port-2 and port-3 neighbors of w and return either with settled agents at those neighbors or empty-handed. This process continues until possibly $\min\{k, \delta_w\}$ neighbors are visited. This finishes in $O(\log \min\{k, \delta_w\}) = O(\log k)$ rounds, since every next iteration of probing is done by double the number of agents.

The question is how to run this approach in $\mathcal{ASYNC}$. We synchronize agents running probing by waiting for all the agents doing probing in the current iteration to return to w to start the next iteration. This can be easily done since w can have a count on how many are left and how many are returned. Figure illustrating these ideas is omitted due to space constraints which can be found in Fig. 7 in [22]. Now suppose a fully unsettled neighbor is found (if exists) or all neighbors are visited and no empty neighbor is found. The question is how to return the settled helper agents back to their home nodes. The *settled helper agents* are the settled agents at w's neighbors brought at w to help with probing. In $\mathcal{SYNC}$, after a fully unsettled neighbor node is found, a_2, a_3 go to that neighbor, say v, where a_2 settles and the settled helper agents go to their respective nodes, which finished in a round. However, in $\mathcal{ASYNC}$, it may not be the case, i.e., it might hamper the search of a_3 to find a fully unsettled neighbor node of v (if exists) to settle. To illustrate, let u be the common neighbor of w and v. Let the agent a_u originally settled at u is in transit from w to u and has not arrived at u yet. Let a_3 start probing the neighbors from v and visit u before a_u reaches u. a_3 finds u as a fully unsettled neighbor node (since empty) and decides to settle at u, violating the dispersion configuration, since the settled helper agent a_u which is in transit to its home node u arrives at u, there will be two settlers at u.

[1]This memory bound can be made $O(\Delta + \log k)$ by just storing the bit information to separate the ports at a node belonging to the DFS tree T_{DFS}, instead of storing the port numbers themselves. This is exactly what Sudo *et al.* [40] did to have memory $O(\Delta + \log k)$ for their $\mathcal{SYNC}$ rooted dispersion algorithm running in $O(k)$ rounds. Through our sibling pointer technique, we were able to keep the time complexity $O(k)$ with memory only $O(\log(k+\Delta))$.

We overcome this difficulty as follows. Before, a_2 leaves w to settle at v, we guarantee that settled helper agents reach their respective home nodes, which are the neighboring nodes of w. Let there be $\alpha \leq \min\{k, \delta_w\}$ settled helper agents at w after probing finishes. We pair agents and send a pair each at $\lfloor \alpha/2 \rfloor$ neighbors of w. For each neighbor, one agent for which it is its home stays at that neighbor and one returns to w, which guarantees that $\lfloor \alpha/2 \rfloor$ settled helper agents now reached their nodes, despite asynchrony. We wait until one robot each from $\lfloor \alpha/2 \rfloor$ pairs returns to w. We then send 2 agents each to $\lfloor \alpha/4 \rfloor$ neighbors of w and $\lfloor \alpha/4 \rfloor$ agents return. Finally, repeating this procedure, there will be only one settled helper agent at w which will be settled at its node by sending it with a_2 and a_2 returns to w. Fig. 6 illustrates these ideas. This process finishes in $O(\log k)$ epochs. Therefore, when a_3 runs its procedure to find a fully unsettled neighboring node and that procedure finds an empty neighbor of v, then that empty node is indeed a fully unsettled node. Thus, we can solve dispersion in $\mathcal{ASYNC}$ for rooted initial configurations in $O(k \log k)$ epochs.

Handling general initial configurations. We proceed with the technique used in our $\mathcal{SYNC}$ general initial configurations to handle general initial configurations in $\mathcal{ASYNC}$. Interestingly, the size-based subsumption technique of Kshemkalyani and Sharma [28] works even in $\mathcal{ASYNC}$ with $O(\log(k + \Delta))$ memory. Since we have an algorithm for rooted initial configuration running in $O(k \log k)$ epochs, we merge these two ideas to achieve $O(k \log k)$-epoch solution for the general initial configurations in $\mathcal{ASYNC}$ with $O(\log(k + \Delta))$ memory.

4 Empty Nodes, Oscillation, and $\mathcal{SYNC}$ and $\mathcal{ASYNC}$ Probing

We first discuss empty node selection and oscillation techniques used in our $\mathcal{SYNC}$ algorithms. We then discuss neighbor probing techniques for both $\mathcal{SYNC}$ and $\mathcal{ASYNC}$ algorithms.

4.1 Empty Node Selection

We describe here an algorithm `Empty_Node_Selection()` (pseudocode is omitted due to space constraints and can be found in Algorithm 1 in full version [22]) that decides which nodes of T_{DFS} to leave empty in our $\mathcal{SYNC}$ algorithms (Sections 5 and 7.1). Let T be an arbitrary T_{DFS} with root r. We assume here that T is known and `Empty_Node_Selection()` is a centralized algorithm. We later show how `Empty_Node_Selection()` can be implemented during DFS when T_{DFS} is monotonically growing in its size. Let the depth of T be $d_{\max}$ (root r at depth 0).

The algorithm is as follows. We put a settler agent each on all the nodes of T at depth $0, 2, 4, \ldots$. We then look at the nodes of T and either (A) remove extra settlers or (B) put new settlers. Fig. 1 provides an illustration. Let lf_i be a leaf node of T. Let lf_i has depth d^i; the depth of T is $d_{\max} := \max_i d^i$. Depending on whether d^i is even or odd, lf_i has either a settler or is empty.

- **Case A – Remove extra settlers:** Consider the leaves of T which have settlers (the children of node w, except w', in Fig. 1 (left)). Let lf_i be one such leaf on T. Let lf_{i-1} be a node in T which is the parent of lf_i (node w in Fig. 1 (left)). Note that lf_{i-1} does not have a settler since it is the parent of lf_i with a settler. Let x be the total number of children of lf_{i-1} that are leaves of T ($x = 4$ for w in Fig. 1 (left)). Note that all these x children must have a settler each. If $x = 1$, we do nothing (the only child of z' in Fig. 1 (left)). If $x > 1$, we remove settlers from $\lfloor \frac{2x}{3} \rfloor$ of those children and hence only $\lceil \frac{x}{3} \rceil$ children are left with a settler each. For w in Fig. 1 (right)), we removed two setters from children of w leading through ports 2 and 3 of w.
- **Case B – Put new settlers:** Let $lf_{\leq 1}$ be a non-leaf node in T at any even depth from r. Notice that $lf_{\leq 1}$ has a settler since it is at even depth. The nodes r and z in Fig. 1. Let x be the number of children of $lf_{\leq 1}$ (pick node z). Note that all these x children do not have a settler since they are at odd depths. If $x > 3$, we put one settler each on $\lceil \frac{x-3}{3} \rceil$ children of $lf_{\leq 1}$ As shown in Fig. 1 (right), we put a settler on the child of z leading through port 4 of z.

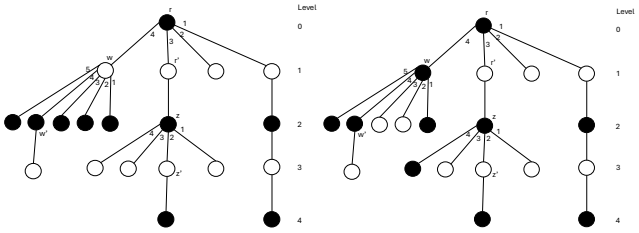

Figure 1: An illustration of which nodes of an arbitrary tree T are occupied with a settler and which nodes are left empty. (left) Tree T after nodes at even depth are occupied with a settler. (right) Tree T after adjustment through removing some settlers and putting new settlers.

We prove the following for `Empty_Node_Selection()`.

Lemma 1. `Empty_Node_Selection()` *leaves* $\geq \lceil \frac{k}{3} \rceil$ *nodes empty in any tree* T *of size* $k \geq 3$.

Proof. Consider $k = 3$. Let T be a line with one endpoint, the root r. The depth of T is 2. The root node and the depth-2 node will have a settler. Therefore, we have $\geq \lceil k/3 \rceil$ nodes empty. Consider now the case of $k \geq 4$. We show that at least $\lfloor k/2 \rfloor$ nodes remain empty. If T is a line, this is immediate. If T is not a line (or when T is a line but the root is not at either end), there must be at least a branch. A branch starts from a node which we call a *branching* node. Order the branching nodes based on their depth from r (including r). Consider the branching node closest to r (including r). Let that branching node be b_i at depth i. Node b_i has at least two children at depth $i + 1$. Since we settle agents at even depth nodes of T, if i is even, b_i has a settler and its children are empty. Let x be the number of those children. If $x \leq 3$, we do not put any settler. If $x > 4$, we divide the children into $\lceil \frac{x-3}{3} \rceil$ groups (one group may have at least one and at most 3 children and assign one settler for each group). Therefore, at least half of the children at depth $i + 1$ remain empty for that branching node.

If i is odd, b_i is empty and its children have a settler each. We classify those children according to whether they are leaves of T or not. For non-leaf children, we do nothing. For leaf children (if there are at least two), we again divide them into groups of three

(one group may have at least one and at most 3 children, and others have 3 children each) and remove all the settlers except one from each group. Therefore, at least half of those nodes at depth $i+1$ remain empty for that branching node.

Let n' be the empty nodes in T and let $n'' = k - n'$ be the non-empty nodes. Now we examine the branching nodes and non-branching nodes in combination at level i. For the non-branching nodes, there is a child at level $i+1$ (which is empty if i is even and non-empty if i is odd). For the branching nodes at level i, we have that at level $i+1$, there are as many empty nodes as non-empty nodes. Therefore, for any T from depth 1 up to $d_{\max}$, $n' \geq n''$. Since root r has a settler, we have that $n' \geq n'' - 1$. Since $n' + n'' = k$, the lemma follows for any $k \geq 3$. □

4.2 Matching Empty Nodes to Settlers and Oscillation

We saw that `Empty_Node_Selection()` leaves $\geq \lceil \frac{k}{3} \rceil$ nodes empty in any arbitrary tree T of size k. We now prove a property such that the empty nodes could be covered by $\leq \lceil \frac{2k}{3} \rceil$ agents settled on the nodes of T. Consider an agent s_d positioned at a depth-d node in T. The agent s_d covers (I) either at most 3 empty children nodes at depth $d+1$ or (II) at most 2 empty sibling nodes at depth d. Fig. 2 provides an illustration where it is shown how agents (shown in the circle with slanted lines) inside each group of dashed boundaries cover empty children or sibling nodes inside the group. For Case I, for the agent s_d at depth d, let the empty nodes at depth d+1 be a, b, c. An oscillation trip for agent s_d would be $s_d - a - s_d - b - s_d - c - s_d$. Notice that the order in which empty nodes are visited does not impact the length of the oscillation trip. For Case II, let a, b be the empty sibling nodes at depth d. Let $p(s_d)$ be the parent of s_d at depth $d-1$ (and also of a, b) by construction. An oscillation trip is $s_d - p(s_d) - a - p(s_d) - b - p(s_d) - s_d$. The node $p(s_d)$ (at depth $d-1$) if empty will be covered by a settler agent at depth $d-2$ and hence we do not consider $p(s_d)$ as covered by s_d in its oscillation trip although the oscillation trip of s_d passes through $p(s_d)$. The order on which a, b are visited has no impact on the oscillation trip length for s_d.

Lemma 2. *There will only be groups of two types such that: (I) an agent a node with at most 3 empty children nodes or (II) an agent at a node with at most 2 empty sibling nodes.*

Proof. The proof follows immediately by how empty nodes are selected in `Empty_Node_Selection()`. □

Lemma 3. *An oscillation trip for an agent s_d settled at a node at depth d finishes in 6 rounds.*

Proof. The agent s_d at a node w needs to visit (I) either 3 children nodes of w at depth $d+1$ or (II) 2 sibling nodes at depth d. For case I, starting from w, a roundtrip of 2 rounds takes w to one empty children and back to w. Therefore, at most 3 children are visited in 6 rounds with s_d starting from w and ending at w. For case II, let $p(s_d)$ be the parent of s_d. $p(s_d)$ must be the parent of the 2 empty sibling nodes s_d needs to visit. s_d reaches $p(s_d)$ in 1 round, it then visits two siblings from $p(s_d)$ in 4 rounds roundtrip. It then returns to w from $p(s_d)$ in 1 round. To illustrate this further, we invite the reader to the left of Fig. 3. Consider the group with ports 1, 5, and 6. The agent (striped circle) visits its (at most) 3 children (empty circle) in a total of 6 rounds. In the same figure, consider the middle group. The agent in the group (striped circle) visits other two sibling nodes (empty circle) in 6 rounds: one round to go to an intermediate node (not in group) then 4 rounds to visit the nodes in the group from the intermediate node, and one round to finally return from the intermediate node to its location. Lemma 2 proves that there will only be two groups. Therefore, in total 6 rounds for oscillation for both cases I and II. □

The agent at a node which performs an oscillation trip is an oscillating settler. The agent at a node which does not need to perform an oscillation trip is a non-oscillating settler. In Fig. 4, the black agents are non-oscillating settlers for the given tree T. However, while running DFS, T_{DFS} monotonically grows and we have to decide on which nodes to leave empty and which agents to assign for oscillation to cover those empty nodes in real-time. At that time, a settler may start as non-oscillating, then it becomes oscillating, and then eventually after DFS finishes becomes non-oscillating. Fig. 4 provides an illustration in which the agents at nodes y, w, w'' which were non-oscillating in Fig. 2 now become oscillating since the tree T grew during DFS.

Lemma 4. *A settler $\alpha(w)$ at a node w of T is an oscillating settler if and only if:*

- *w is an even depth node and*
  - *w is a non-leaf node of T.*
  - *w is a leaf node of T with at least one sibling leaf node not covered by another agent.*
- *w is an odd depth leaf node of T with at least one sibling leaf node not covered by another agent.*

Otherwise, $\alpha(w)$ is a non-oscillating settler.

Proof. The proof follows from the working principle of `Empty_Node_Selection()`. Suppose w is an even depth node in T. We differentiate two sub-cases. If w is a non-leaf node in T, then it must have a child which is empty. $\alpha(w)$ then covers that child. If w is a leaf node in T and there is at least one other even depth sibling node, $\alpha(w)$ covers it if it was not covered by another sibling agent. If w is an odd depth node and a leaf in T and there is another sibling leaf node, $\alpha(w)$ needs to cover it if it was not covered by another sibling agent. The agent that does not satisfy any of the above conditions does not oscillate. □

We now discuss how we form the groups for oscillation. Fig. 3 illustrates these ideas. We follow `Empty_Node_Selection()` and arrange them as groups according to the increasing port numbers until group size limit (3 children or 2 sibling to cover) is reached.

4.3 Empty Node Selection and Oscillation during DFS

DFS starts from source $s \in V$ where all k agents are initially located. The largest ID agent a_{max} takes the responsibility of running DFS. DFS settles one agent at s. Regarding other fully unsettled nodes the DFS visits, it settles an agent on each such node that is at even depth in T_{DFS} built by the DFS so far. If no branching node, Lemma 1 immediately satisfies. If there is a branching node, the DFS adjusts while at the child node of that branching node by either putting

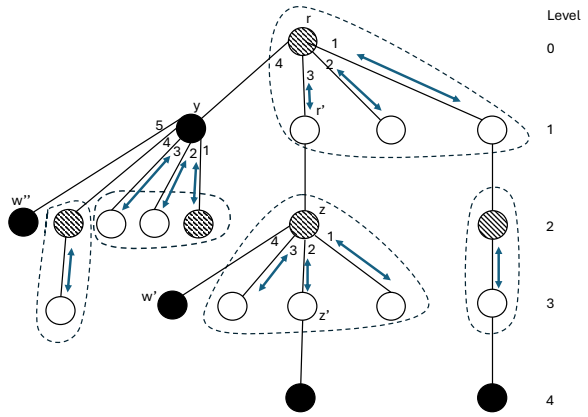

Figure 2: An illustration of how settlers (shown in circle with slanted lines) cover the empty nodes through oscillation shown as groups inside dashed boundaries. There is exactly one oscillating settler in each group that is responsible for oscillation to cover the empty nodes (at most 2 for the sibling case, otherwise at most 3). Oscillations are two types: (i) An oscillating settler at even depth d covers at most 3 empty nodes at depth $d+1$ (ii) An oscillating settler at even/odd depth d covers at most 2 empty sibling nodes at same depth d. The settlers shown as solid black circles are non-oscillating settlers for this figure.

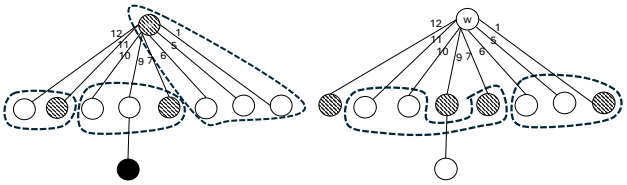

Figure 3: An illustration of how groups are formed for oscillation for a branching node w. Groups are formed according to the port numbers in increasing orders for the children that satisfy the oscillation criteria. There are two types of groups. The first type has an agent at a node with at most three children nodes empty. The second type agent has an agent at a node with at most two sibling nodes empty.

an extra settler (branching node is at any even depth) or removing a settler (branching node is at odd depth and at least one of its children is a leaf of T_{DFS}). Fig. 4 illustrates these ideas. Consider the agents at nodes y, w', w''. They were non-oscillating in Fig. 2. Suppose DFS made a forward move from node y through port 6. The agent at w'' now sees there is one sibling node added in T_{DFS} and there is no other agent to cover it. The agent at w'' becomes an oscillating settler to cover that new sibling node. Therefore, whether to put an extra settler or remove a settler whether a settler oscillates or not, and when it oscillates can be done during DFS.

Observation 1. `Empty_Node_Selection()` *can be implemented during DFS even when T_{DFS} is monotonically growing.*

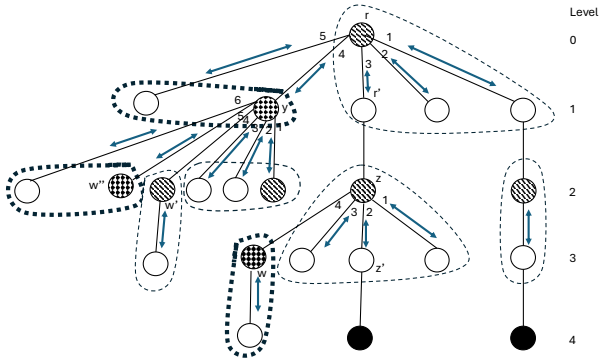

Figure 4: An illustration of how non-oscillating settlers become oscillating when DFS progresses adding more nodes in T_{DFS}. Three non-oscillating settlers (y, w, and w'') in Fig. 2 now became oscillating (shown in circle with diamonds) covering empty nodes in their respective groups.

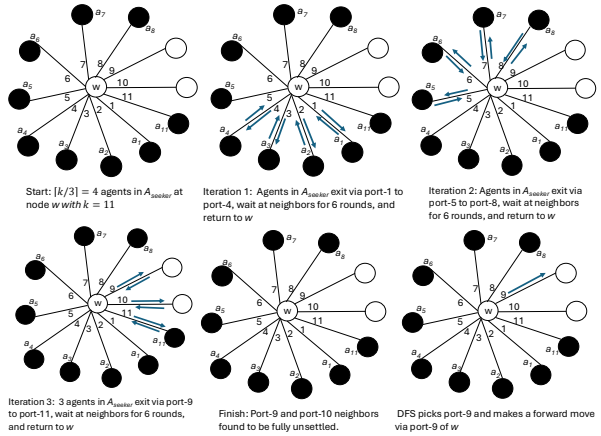

Figure 5: An illustration of how `Sync_Probe()` (Algorithm 1) finds a fully unsettled neighbor at a node w (if exists one). In the figure, port-9 and port-10 neighbors of w were found to be fully unsettled. DFS does its forward move by exiting through port-9 of w.

4.4 Synchronous Probing

We present an algorithm `Sync_Probe()` (pseudocode in Algorithm 1) which finishes probing at a node w in $\mathcal{SYNC}$ in $O(1)$ rounds. Fig. 5 provides an illustration. The goal in `Sync_Probe()` is to verify whether there is at least a neighbor of w that is fully unsettled and return the port of w leading to that node (if there exits one). `Sync_Probe()` uses agents in A_{seeker} present on w to search for a fully unsettled node in $N(w)$. We have that $|A_{seeker}| = \lceil k/3 \rceil$. Since there are $k \leq n$ agents, T_{DFS} never has more than k nodes (both empty and non-empty). Since $\geq \lceil k/3 \rceil$ agents in A_{seeker} run `Sync_Probe()` at w, running `Sync_Probe()` (at most) three times at w should cover all neighbors in $N(w)$, taking total 6 rounds with 2 roundtrip round per iteration. However, since some nodes of T_{DFS} may be empty (Algorithm 1 in [22]) the empty neighbor in $N(w)$ found by `Sync_Probe()` may be the one that was already

Algorithm 1: Sync_Probe()

```
1  (α(w).next, α(w).checked) ← (⊥, 0);
2  while α(w).checked ≠ δ_w do
3      {a_1, a_2, ..., a_x} ∈ A_seeker with x = |A_seeker|;
4      Δ' ← min(x, δ_w − α(w).checked);
5      for j ← 1 to Δ' do
6          assign a_j to the port N(w, j + α(w).checked);
7          a_j leaves w using its port (suppose node reach is u_j),
             waits at u_j for 6 rounds, and returns to w;
8      if there exists a_j that did not meet an agent α(u_i) then
9          α(w).next ← j + α(w).checked;
10         break the while loop;
11     α(w).checked ← α(w).checked + Δ';
```

visited. To avoid this, we ask our $\geq \lceil k/3 \rceil$ seeker agents to return only after waiting at the neighboring nodes for 6 rounds. While waiting for 6 rounds, if that node was previously visited by DFS, it is guaranteed that a seeker agent will meet an oscillating settler doing its trip at that node in those 6 rounds. We implement Sync_Probe() with a variable $\alpha(w)$.checked $\in [0, \delta_w]$. The variable $\alpha(w)$.checked stores the most recently checked port number, initially, $\alpha(w)$.checked is set to 0. $\alpha(w)$.checked $= l$ implies that the neighbors $N(w,1), N(w,2), \ldots, N(w,l)$ are not fully unsettled. Let $x = |A_{seeker}|$. In the first iteration of Sync_Probe(), the $\min\{x, \delta_w\}$ agents visit $\min\{x, \delta_w\}$ neighbors in parallel (one to one mapping), wait at the respective neighbors they reached for 6 rounds, and return to w (Lines 5–7). If there is an agent in A_{seeker} that does not find a settler at the neighbor of w it visited, then the node visited by that agent must be fully unsettled. In such a case, the port used by one of those agents (picked smallest port neighbor in case of multiple) is stored in $\alpha(w).next$, and the while loop terminates (Line 10). If all agents find a settler (no empty node), we repeat Sync_Probe() until $\min\{k, \delta_w\}$ ports are checked. We have following guarantees.

Lemma 5. *At the end of* Sync_Probe() *at a node $w \in V$, it is guaranteed that: (i) if there exists a fully unsettled node in $N(w)$, then $N(w, \alpha(w).next)$ is unsettled, and (ii) if there are no fully unsettled nodes in $N(w)$, then $\alpha(w).next = \perp$ holds true.* Sync_Probe() *at a node w finishes in $O(1)$ rounds.*

Proof. We prove the first two cases by contradiction. Consider first Case (i). Suppose there exists a fully unsettled node $u \in N(w)$ which Sync_Probe() classifies as settled. For that to happen, u must have been a node in T_{DFS} and u either has an agent settled or covered through oscillation by the agent at a node previously visited. This is a contradiction since u was not previously visited by the DFS and hence cannot be in T_{DFS}. Therefore, Sync_Probe() cannot classify it as settled since there is no agent settled or covered it through oscillation. Case (ii) follows from a similar argument.

We now prove the runtime of Sync_Prove(). Since there are $k \leq n$ agents, the size of T_{DFS}, $|T_{DFS}| = k$. Consider Sync_Prove() at node w. Irrespective of degree δ_w of w, there must be a neighbor in $N(w)$ that is fully unsettled found by Sync_Probe() after probing at most $k-1$ other nodes. Since $|A_{seeker}| = \lceil \frac{k}{3} \rceil$, (at most) $k-1$ neighbors of w are probed in 3 repetitions of Sync_Probe(). Each Sync_Probe() finishes in 8 rounds, 6 rounds wait at the neighbor and 2 rounds roundtrip from w to w's neighbor. Therefore, Sync_Probe() at w finishes in $24 = O(1)$ rounds. □

4.5 Asynchronous Probing

We present an algorithm Async_Probe() (pseudocode is omitted due to space constraints and can be found in Algorithm 6 in full version [22]) which finishes probing at a node w in $\mathcal{ASYNC}$ in $O(\log k)$ epochs. A figure proving an illustration is also omitted due to space constraints. The goal in Async_Probe() is to verify whether there is at least a neighbor of w that is fully unsettled and return the port of w leading to that node (if there exists one). Async_Probe() uses agents present at w as well as the settled helper agents from the previously visited non-empty neighbors in $N(w)$ while running Async_Probe() at w to search for a fully unsettled node in $N(w)$. Let $x = |A(w) \setminus \{\alpha(w)\}|$, i.e., there are x agents at w when Async_Probe() is invoked, excluding the agent settled at $\alpha(w)$. We have that $x \geq 1$, otherwise the DFS is complete and dispersion is achieved at w. The $\min\{x, \delta_w\}$ agents visit in parallel $\min\{x, \delta_w\}$ neighbors and then return to w. Each agent brings back the agent $\alpha(u_i)$ settled (if exists) at the neighbor u_i it visited ($\alpha(u_i)$ is the settled helper agent). If no such agent $\alpha(u_i)$ at u_i, then that neighbor u_i must be fully unsettled. This property is guaranteed through Guest_See_Off() (pseudocode is omitted due to space constraints and can be found in Algorithm 7 in full version [22]). If w is the first node where Async_Probe() is executing, it is indeed true that an empty neighbor is fully unsettled. If w is not the first node (let the previous node be w', then Guest_See_Off() executed at that node guarantees that a neighbor found empty while Async_Probe() is running at w is indeed fully unsettled. If a neighbor in $N(w)$ is found empty, the port of w leading to that neighbor is stored in $\alpha(w).next$ and Async_Probe() is complete. Otherwise, if all x agents bring back one agent each (i.e., no visited neighbor is fully unsettled), then there are $2x$ agents on w, excluding $\alpha(w)$. In the second iteration of Async_Probe(), these $2x$ agents visit the next $2x$ neighbors in search of fully unsettled neighbors. As long as no fully unsettled neighbor is discovered, the number of agents on w, excluding $\alpha(w)$, doubles with each iteration of Async_Probe(). Async_Probe() stops as soon as an empty neighbor is found or all neighbors of w are visited.

Lemma 6. *At the end of* Async_Probe() *at each node w, it is guaranteed that: (i) if there exists a fully unsettled node in $N(w)$, $N(w, \alpha(w).next)$ leads to that node, (ii) if there are no fully unsettled nodes in $N(w)$, then $\alpha(w).next = \perp$ holds true.* Async_Probe() *at w finishes in $O(\log k)$ epochs.*

Proof. Cases (i) and (ii) follow similarly as in Lemma 5. For the runtime, each iteration of Async_Probe() takes exactly two epochs. Since there are at most $\min\{k, \Delta\}$ settled nodes in $N(w)$ and in each subsequent operation double the number of neighbors in $N(w)$ can be probed, after running Async_Probe() for at most $O(\log \min\{k, \Delta\}) = O(\log k)$ times, either a fully unsettled neighbor node of w will be found, or the search will be concluded with no fully unsettled neighbor. □

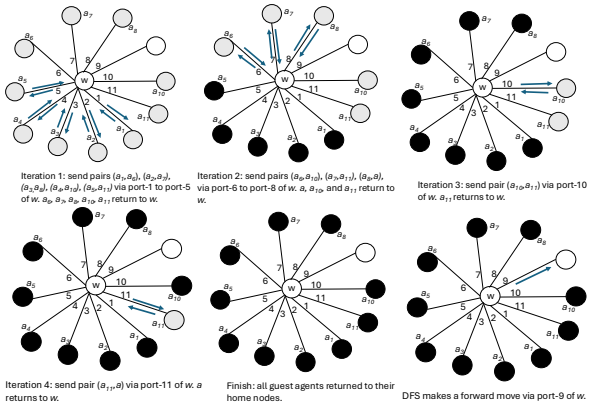

Figure 6: An illustration of how agents brought at node w during `Async_Probe()` (Algorithm 6 in [22]) are sent back to their homes executing `Guest_See_Off()` (Algorithm 7 in [22]). DFS does its forward move via exiting through port-9 of w (which was found to be fully unsettled during `Async_Probe()`). If no neighbor was found fully unsettled, DFS would go to the parent of w via a backtrack move.

After `Async_Probe()` completes at w, we need to send back each settled helper agent $\alpha(u_i)$ brought to w to help with probing back to their homes. Only after that the agents in $|A(w) \setminus \{\alpha(w)\}|$ can exit w. This guarantees that when `Async_Probe()` executes at the next node, due to asynchrony, the home nodes of the guests at w were not found as empty. This sending back of guests to their home nodes is done via `Guest_See_Off()`. Let $A_{guest}(w)$ be the set of such settler helper agents brought to w. $|A_{guest}(w)| \leq \min\{k, \Delta\}$. For this implementation, we pair the agents in $A_{guest}(w)$. Let a, b be a pair (for odd $|A_{guest}(w)|$, a non-paired agent remains at w). We send a, b to the home of a using the port of w from which a entered w from its home $h(a)$ during `Async_Probe()`. Each agent in $A_{guest}(w)$ stores such information in its memory. b returns to w after making sure a reached its home $h(a)$. After one agent from each pair returns to w, we have $|A_{guest}(w)| = |A_{guest}(w)|/2$. We then again make pairs among the agents in $A_{guest}(w)$ and settle half of the agents to their homes. This process continues until either $A_{guest}(w) = \emptyset$ (even) or $|A_{guest}(w)| = 1$ (odd). For the case of $|A_{guest}(w)| = 1$, $\alpha(w)$ is used as a pair to see off the only agent in $A_{guest}(w)$ to its home. This careful implementation of settling the guests to their homes plays a crucial role in guaranteeing that the empty neighbors found in `Async_Probe()` are in fact the fully unsettled empty neighbors.

Lemma 7. *At the end of* `Guest_See_Off()` *at node w, each agent brought to w goes backs to its home node and* `Guest_See_Off()` *at w finishes in $O(\log k)$ epochs.*

Proof. Let u_i be a neighboring node of w. Agent $\alpha(u_i)$ while brought to w (the first time coming to w) remembers the port of w from which it entered w. When it is sent back during `Guest_See_Off()`, it uses the same port to exit w. And hence, $\alpha(u_i)$ reaches u_i.

Algorithm 2: RootedSyncDisp

1 $a_{\min}.settled \leftarrow \top$ and $a_{\min}.parent \leftarrow \perp$ at node s;
2 $A_{seeker} \leftarrow$ the $\lceil \frac{k}{3} \rceil$ agents in A with largest IDs, except $a_{\max}$;
3 $A_{explorer}(s) \leftarrow A(s) \setminus (A_{seeker} \cup \{a_{min}\})$;
4 agents in $A_{explorer}(s) \cup A_{seeker}$ exit s via port-1 neighbor;
5 **while** k *different nodes visited* **do**
6 $w \leftarrow$ the node $\alpha(a_{\max})$ where a_{max} is positioned;
7 `Sync_Probe()` at w;
8 **if** $\alpha(w).next \neq \perp$ **then**
9 `Forward_Move()`;
10 **else**
11 `Backtrack_Move()`;
12 $A^{un}_{explorer} \leftarrow$ agents in $A_{explorer}(w)$ not yet settled;
13 agents in $A^{un}_{explorer} \cup A_{seeker}$ (and others not yet settled) follow *parent* pointers at each node until reaching s, the root of T_{DFS};
14 agents in $A^{un}_{explorer} \cup A_{seeker}$ (and others not yet settled) re-traverse T_{DFS} through using sibling pointers and settle at empty nodes;

Regarding runtime, we have that $|A_{guest}(w)| \leq \min\{k, \Delta\}$. Each iteration of `Guest_See_Off()` settles $\lfloor \frac{|A_{guest}(w)|}{2} \rfloor$ guests at node w to their home nodes. Therefore, executing `Guest_See_Off()` for $\lceil \log |A_{guest}(w)| \rceil + 1$ iterations settles all guests in $A_{guest}(w)$ to their home nodes. It is easy to see that each iteration of `Guest_See_Off()` finishes in 2 epochs. Therefore, the total time is $O(\log \min\{k, \Delta\}) = O(\log k)$ epochs. □

5 $\mathcal{SYNC}$ Rooted Dispersion Algorithm

In this section, we present our first algorithm, **RootedSyncDisp**, that solves the dispersion of $k \leq n$ agents initially located at a single node $s \in V$ in $O(k)$ rounds with $O(\log(k + \Delta))$ bits per agent in $\mathcal{SYNC}$. The algorithm is DFS-based which constructs the DFS tree, T_{DFS}. In **RootedSyncDisp** (Algorithm 2), the largest ID agent, denoted as $a_{\max}$, serves as the leader and conducts DFS. The non-leader agents move with the leader and one of them settles at a fully unsettled node when they visit it. If $a_{\max}$ encounters a fully unsettled node alone, it settles itself on that node, achieving dispersion. During DFS, $a_{\max}$ currently at node w must determine

(i) whether there is a fully unsettled neighbor node of w, and
(ii) if so, which neighbor of w is fully unsettled.

$a_{\max}$ makes this decision through function `Sync_Probe()` (Algorithm 1). $a_{\max}$ uses $\lceil \frac{k}{3} \rceil$ seeker agents to execute `Sync_Probe()` in $O(1)$ rounds.

RootedSyncDisp runs DFS either in *forward* phase or *backtrack* phase. The forward phase is handled by `Forward_Move()` (Algorithm 3) and the backtrack phase is handled by `Backtrack_Move()` (Algorithm 4). The forward phase is executed at w, if `Sync_Probe()` at w finds at least one neighbor of w fully unsettled, otherwise the backtrack phase. After DFS completes (i.e., T_{DFS} has k nodes), the seeker agents then go to the

Algorithm 3: `Forward_Move()`

1 **if** $\alpha(w).firstchild = \perp$ **then**
2 $\alpha(w).firstchild \leftarrow \alpha(w).next$;
3 $\alpha(w).latestchild \leftarrow \alpha(w).next$;
4 **else**
5 $a_{\max}$ moves to $\alpha(w).latestchild$, sets $\alpha(w).latestchild.nextsibling \leftarrow \alpha(w).next$, and returns to w;
6 $A(w) \leftarrow$ agents present at w, except the settler $\alpha(w)$, if any;
7 all agents in $A_{explorer}(w) \cup A_{seeker}$ exit w through port $N(w, \alpha(w).next)$ to reach node u; u must be a fully unsettled node;
8 **if** u *is at odd depth* **then**
9 $a_{min} \leftarrow$ agent with smallest ID in $A(u)$;
10 $x \leftarrow$ the number of times `Sync_Probe()` found $\alpha(w).next \neq \perp$ at w so far;
11 **if** $x \leq 3$ **then**
12 ask $\alpha(w)$ to include u in its oscillation trip;
13 **else**
14 **if** $x \mod 3 = 1$ **then**
15 $a_{\min}.settled \leftarrow \top$, $a_{\min}.parent \leftarrow a_{\max}$.`pin` (which leads to w);
16 **else**
17 **if** $(x-1) \mod 3 = 1$ **then**
18 $\alpha(u') \leftarrow$ the agent settled at node u' reached through $(x-1)$-th $\alpha(w).next$ port at w;
19 **if** $(x-1) \mod 3 = 2$ **then**
20 $\alpha(u') \leftarrow$ the agent settled at node u' reached through $(x-2)$-th $\alpha(w).next$ port at w;
21 ask (sibling settler) $\alpha(u')$ to include u in its oscillation trip;
22 **else**
23 $a_{min} \leftarrow$ agent with smallest ID in $A(u)$;
24 $a_{\min}.settled \leftarrow \top$, $a_{\min}.parent \leftarrow a_{\max}$.`pin`;
25 $\alpha(u)$.`treelabel` $\leftarrow a_{\max}$.`treelabel`;

root of T_{DFS} and then re-traverse T_{DFS} to settle at its empty nodes. After all this is done, the k nodes of T_{DFS} finally have each node occupied and dispersion is achieved.

We now provide details. In **RootedSyncDisp**, every agent a maintains a variable a.`settled` $\in \{\perp, \top\}$. Let $\alpha(w)$ be the settler/oscillating agent at w. Agent $\alpha(w)$ maintains the following variables, $\alpha(w).parent$ (the parent of w), $\alpha(w).firstchild$ (the first child of w), $\alpha(w).sibling1$, $\alpha(w).sibling2$ (two siblings of a child of w), $\alpha(w).latestchild$ (the latest child of w to facilitate sibling traversal), $\alpha(w).depth$ (the depth of w in T_{DFS}), and $\alpha(w).next \in [1, \delta_w] \cup \{\perp\}$. In `Sync_Probe()` at node w, if there exists a fully unsettled node in $N(w)$, the corresponding port number will be in $\alpha(w).next$. More precisely, an integer i such that $N(w, i) = u$ and $\alpha(u) = \perp$ is assigned to $\alpha(w).next$. If all neighbors $N(w)$ are settled, $\alpha(w).next$ will be set to $\perp$.

Algorithm 4: `Backtrack_Move()`

1 **if** w *is at even depth* **then**
2 $p_w \leftarrow$ parent of w which is reached via $a_{\max}$.`pin`;
3 let w be the x-th leaf children of $\alpha(p_w)$ in the DFS tree;
4 **if** $x > 1$ **then**
5 **if** $x \mod 3 = 0$ **then**
6 $\alpha(u') \leftarrow$ the agent settled at node u' which is the $(x-2)$-th leaf children of $\alpha(p_w)$;
7 **if** $x \mod 3 = 2$ **then**
8 $\alpha(u') \leftarrow$ the agent settled at node u' which is the $(x-1)$-th leaf children of $\alpha(p_w)$;
9 remove agent $\alpha(w)$ settled at w setting $\alpha(w).settled \leftarrow \perp$;
10 ask (sibling settler) $\alpha(u')$ to include node w in its oscillation trip;
11 agents in $A_{explorer}(w) \cup A_{seeker} \backslash \{\alpha(w)\}$ leave w via port $a_{\max}$.`pin` from w;

At round 0, all agents are located at node s. Let $A_{seeker} \subset A$ be the $\lceil k/3 \rceil$ agents at s with largest IDs, except $a_{\max}$. The agents in A_{seeker} will run `Sync_Probe()` at each node. Let $A_{explorer}(w) \in A \backslash A_{seeker}$ denote the agents on any node w except $\alpha(w)$ settled at w. Let $a_{\min}(w)$ be the smallest ID agent at w among the agents in $A_{explorer}(w)$. Agent $a_{\min}(s)$ settles at node s and sets $a_{\min}(s).parent \leftarrow \perp$.

Forward move. The DFS starts at s in the forward phase. As long as there is at least a fully unsettled node in $N(s)$ (for multiple such nodes, pick one associated with the smallest port), all agents in $A_{explorer}(s) \cup A_{seeker}$ move to one of those nodes together (Line 7 of `Forward_Move()`). We call this kind of movement *forward move*.

Suppose a forward move from node w brought agents $A_{explorer}(u) \cup A_{seeker}$ to u (w is the parent of u). The following happens at u:

1. Suppose u is the even depth node in T_{DFS}. $a_{\min}(u) \in A_{explorer}(u)$ settles at u and sets $a_{\min}(s).settled \leftarrow \top$ and $a_{\min}(s).parent \leftarrow a_{\max}$.`pin`.
2. Suppose u is the odd depth node in T_{DFS}. Whether an agent settles at u or not depends on how many times `Forward_Move()` has been successful so far from w, u's parent. Let x be that number. If $x \leq 3$, we leave u empty and ask $\alpha(w)$ to oscillate to u (i.e., $\alpha(w)$ will have at most three children nodes to oscillate to). If $x \geq 4$, we settle an agent at u if $(x \mod 3) = 1$. If $(x-1) \mod 3 \in [1, 2]$, we leave u empty. The agent settled at $(x \mod 3 = 1)$-th children of w oscillates to $x+1$-th and $x+2$-th children of w (i.e., an agent at a sibling node covers two other empty sibling nodes).

Backtrack move. When the current location w has no fully unsettled neighbor, all explorers $A_{explorer}(w) \cup A_{seeker}$ exit w through $\alpha(w).parent$ (Line 11 of `Backtrack_Move()`). We call this kind of movement *backtrack move*. Suppose there is an agent $\alpha(w)$ settled at node w. Before exiting w, depending on the situation, $a_{\max}$ needs to decide on whether to leave $\alpha(w)$ settled or make it an explorer

(i.e., add to $A_{explorer}(w)$). Let p_w be the parent node of $\alpha(w)$. Let w be the x-th leaf children of $\alpha(p_w)$ in T_{DFS} known so far. The leaf children means $\alpha(w)$ must be the agent positioned on the leaf of T_{DFS} (if `Sync_Probe()` does not return $\alpha(w).next \neq \perp$ even once at w, $\alpha(w)$ must be the leaf) and `Backtrack_Move()` is needed. If $x = 1$, we leave $\alpha(w)$ settled. If $x > 1$, we leave $\alpha(w)$ settled only when $(x \mod 3) = 1$ (i.e., $x = 4, 7, 10$, etc.) In all other values of x, we make $\alpha(w)$ the explorer. The agent settled at $(x \mod 3 = 1)$-th children of w oscillates to $(x+1)$-th and $(x+2)$-th children of w (i.e., an agent at a sibling node covers two other empty sibling nodes). This transitioning of a settled agent to an unsettled one is to make sure that we never need agents from A_{seeker} to settle while running DFS.

Lemma 8. *For any $k \geq 3$, during DFS, only (at most) $\lfloor \frac{2k}{3} \rfloor$ agents settle.*

The proof of Lemma 8 is omitted here; can be found in [22].

Lemma 9. RootedSyncDisp *solves dispersion in $O(k)$ rounds using $O(\Delta \log(k + \Delta))$ bits per agent.*

Proof. Let the DFS be currently at node w. As long as there is (at least) a fully unsettled neighbor node of w, $a_{\max}$ makes a forward move to that node. If there is no such neighbor, $a_{\max}$ makes a backward move to the parent node of w. Since G is a connected graph, the DFS visits k nodes with exactly $k-1$ forward moves and at most $k-1$ backward moves. Thus, the number of calls to `Sync_Probe()` is at most $2(k-1)$. From Lemma 5, `Sync_Probe()` at a node finishes in $O(1)$ rounds. Additionally, the agents in $A^{un}_{explorer} \cup A_{seeker}$ need $k-1$ rounds to reach the root node of T_{DFS} following parent pointers. The agents in $A^{un}_{explorer} \cup A_{seeker}$ reach to empty nodes of T_{DFS} and settle, re-traversing T_{DFS}, in $O(k)$ rounds following the information about all children of w. Therefore, **RootedSyncDisp** finishes in $O(k)$ rounds. Regarding memory, an agent at node w handles several $O(\log \Delta)$-bit variables, $firstchild$, $next$, checked, $parent$, $latestchild$, $sibling1$, $sibling2$, and the information about all the ports of w that lead to children of w in T_{DFS} (at most Δ such ports with each port needing $O(\log \Delta)$ bits). Other variables can be stored in $O(1)$ bits. An agent needs $\lceil \log k \rceil$ bits to store its ID. Therefore, the memory becomes $O(\Delta \log(k + \Delta))$ bits. □

Memory-efficient DFS tree re-traversal. We now discuss how the memory complexity of $O(\Delta \log(k+\Delta))$ bits per agent in Lemma 9 can be reduced to $O(\log(k + \Delta))$ bits per agent. Notice that this is due to the information to keep at the agent at node w in Line 14 of Algorithm 2 about all the ports of w that lead to children of w in T_{DFS} (at most Δ such ports with each port needing $O(\log \Delta)$ bits). This information is used by agents in $A^{un}_{explorer} \cup A_{seeker}$ to settle at empty nodes of T_{DFS}. We describe a technique of *sibling pointer* that allows to re-traverse T_{DFS} in $O(\log \max\{k, \Delta\})$ bits, reducing the memory in Lemma 9 to $O(\log \max\{k, \Delta\})$ bits. Let $C_w = \{c_{w1}, c_{w2}, \ldots, c_{wv}\}$ be the v children of w in T_{DFS}. Consider i-th child $c_{w,i}$ of w.

- **Suppose $c_{w,i}$ is at odd depth:** For $i \leq 3$, $\alpha(w)$ stores the information about $c_{w,1}, c_{w,2}, c_{w,3}$, so that re-traversal visits them in order, and additionally it stores the first sibling pointer to $c_{w,4}$. Notice that $c_{w,4}$ has a settler. We store at $c_{w,4}$ the information about $c_{w,5}$ and $c_{w,6}$, and the next sibling pointer to $c_{w,7}$. Therefore, after DFS finishes re-traversing $c_{w,1}$ to $c_{w,3}$, it visits $c_{w,4}$ to collect information about $c_{w,5}$ and $c_{w,6}$ to visit and the next sibling pointer information to $c_{w,7}$. Since $c_{w,i}, i = 4, 7, \ldots$ have an agent positioned, $\alpha(w)$ can have information about (at most) four children at a time to re-traverse T_{DFS} without needing to store information about all v children.
- **Suppose $c_{w,i}$ is at even depth:** We only consider the subset of the children in $C_{wl} \subseteq C_w = \{c_{wl,1}, c_{wl,2}, \ldots, c_{wl,v'}\}$ which are the leaves in T_{DFS}. $\alpha(w)$ stores the information about $c_{wl,1}, c_{wl,2}, c_{wl,3}$, so that re-traversal visits them in order, and additionally it stores the first sibling pointer to $c_{wl,4}$. Notice that $c_{wl,4}$ has a settler. We store at $c_{wl,4}$ the information about $c_{wl,5}$ and $c_{wl,6}$, and the next sibling pointer to $c_{wl,7}$. Therefore, the children of $\alpha(w)$ can be re-traversed keeping information about (at most) four children at a time.

Lemma 10. *After DFS completes, re-traversal of T_{DFS} by agents in $A^{un}_{explorer} \cup A_{seeker}$ starting from root in $O(k)$ time can be done with only $O(\log(k + \Delta))$ bits per agent.*

Proof. We have that Algorithm 2, except Line 14, needs only $O(\log(k + \Delta))$ bits and finishes in $O(k)$ rounds. For Line 14, the sibling pointer idea asks to store information about 4 of the children of a node (at any depth of T_{DFS}) to re-traverse T_{DFS} by agents in $A^{un}_{explorer} \cup A_{seeker}$. This storage adds $O(\log \Delta)$ bits/agent. Therefore, the total memory complexity becomes $O(\log(k + \Delta))$ bits. □

We have the following main theorem from Lemmas 9 and 10.

Theorem 5.1. *Algorithm* **RootedSyncDisp** *solves dispersion within $O(k)$ rounds using $O(\log(k + \Delta))$ bits per agent in $\mathcal{SYNC}$.*

6 $\mathcal{ASYNC}$ Rooted Dispersion Algorithm

In this section, we present our second algorithm, **RootedAsyncDisp**, that solves the dispersion of $k \leq n$ agents initially located at a single node $s \in V$ in $O(k \log k)$ epochs with $O(\log(k + \Delta))$ bits per agent in $\mathcal{ASYNC}$. This algorithm is also DFS-based and extends the technique of [40] [DISC'24] developed for solving dispersion in $\mathcal{SYNC}$. We use our algorithm `Async_Probe()` (pseudocode is provided in Algorithm 6 in [22]) so that DFS at a node w finds a fully unsettled neighbor node of w in $O(\log k)$ epochs in $\mathcal{ASYNC}$. Sudo *et al.* [40] [DISC'24] provided the $O(\log k)$-round solution just for $\mathcal{SYNC}$.

The pseudocode for **RootedAsyncDisp** is omitted due to space constraints and can be found in Algorithm 8 in full version [22]). **RootedAsyncDisp** calls `Async_Probe()` (pseudocode is in Algorithm 6 in [22]). `Async_Probe()` finds a fully unsettled empty neighbor at any node w (if exists) and then calls `Guest_See_Off()` (pseudocode is in Algorithm 7 in [22]) to send the settled agents used in probing back to their home nodes. Every agent a maintains a variable a.`settled` $\in \{\perp, \top\}$, which decides whether a is an explorer or a settler. In addition, the settler $\alpha(w)$ at node w maintains two variables, $\alpha(w).parent, \alpha(w).next \in [1, \delta_w] \cup \perp$.

Again, $a_{\max}$ is the leader that runs DFS; non-leaders follow $a_{\max}$ until they settle. The following is guaranteed each time $a_{\max}$ invokes `Async_Probe()` at node w:

- If there exists a fully unsettled neighbor node in $N(w)$, `Async_Probe()` finds it and stores the corresponding port number leading to that neighbor in $\alpha(w).next$.
- If all neighbors have a settler (no neighbor fully unsettled), $\alpha(w).next = \perp$ is true.
- `Async_Probe()` will finish in $O(\log k)$ epochs.
- `Guest_See_Off()` will settle the collected helper agents back to their homes.
- `Guest_See_Off()` will finish in $O(\log k)$ epochs.

Forward and backtrack moves. In the beginning of **RootedAsyncDisp**, all agents in $\mathcal{A}$ are located at node s. The agent with the smallest ID $a_{\min}(s)$ settles at node s and sets $a_{\min}(s).parent \leftarrow \perp$. We denote $a_{\min}(s)$ as $\alpha(s)$, the agent settled at node s. Then, as long as there is (at least) a fully unsettled node in $N(s)$, all explorers move to that node together (which is a forward move). After each forward move from a node w to u, the agent with the smallest ID among $A(u)$ settles on u, and $\alpha(u).parent$ is set to the port of u leading to w. When the current node u has no fully unsettled node in $N(u)$, all explorers in $A(u)$ move to the parent of u (which is a backtrack move). Finally, $a_{\max}$ terminates when it settles (since $a_{\max}$ settles last) and dispersion is achieved. Since the number of agents is $k \leq n$, the DFS stops after $a_{\max}$ makes a forward move $k-1$ times. The agent $a_{\max}$ makes a backward move at most once from every visited node, i.e., total at most $k-1$ times. Therefore, excluding `Async_Probe()`, the execution of **RootedAsyncDisp** completes in $O(k)$ epochs. Notice that `Async_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 `Async_Probe()` requires $O(\log k)$ epochs, the time complexity of **RootedAsyncDisp** becomes $O(k \log k)$ epochs.

Regarding memory, an agent handles several $O(\log \Delta)$-bit variables, *next*, `checked`, *parent*, as well as the port number that the settler $\alpha(u)$ needs to remember in order to return to node u from node w during `Guest_See_Off()`. Every other variable can be stored in a constant space. An agent needs $\lceil \log k \rceil$ bits to store its ID. Therefore, the memory complexity is $O(\log(k+\Delta))$ bits.

Theorem 6.1. *Algorithm* **RootedAsyncDisp** *solves dispersion within $O(k \log k)$ epochs using $O(\log(k+\Delta))$ bits per agent.*

7 General Dispersion Algorithms

In this section, we discuss the dispersion of agents from general initial configurations, i.e., $k \leq n$ agents are initially on multiple nodes. Suppose agents are initially on ℓ nodes. The ℓ DFSs start in parallel from ℓ nodes. We extend the idea of merging from Kshemkalyani and Sharma [28] [OPODIS'21] to deal with merging the DFSs if one DFS meets another. We refer in this paper to this merging algorithm as the KS algorithm.

In the KS algorithm, multiple DFSs grow concurrently. The size of a DFS i at any point in time is the number of settled agents d_i in it. When one DFS meets another, the smaller sized DFS collapses and gets subsumed in the larger sized DFS. Subsumption and collapse essentially mean that the agents settled from the subsumed DFS are collected and given to the subsuming DFS to extend its traversal. This is done via a common module to re-traverse an already identified DFS component with nodes having the same *treelabel*. Such re-traversal occurs in procedures *Exploration*, *Collapse_Into_Child*, and *Collapse_Into_Parent*, and can be executed in $4d_i$ steps. After such a subsumption, the subsuming DFS continues growing from where it last left off, this time with added agents from the subsumed DFS(s). Operation in these two phases – growing and subsuming – continues until all agents are settled and there are no more meetings between DFSs. We term the DFS that subsumes other met DFS(s) at each meeting as the winner DFS. Thus in the KS algorithm, the winner DFS alternates between the two phases – growing, and subsuming. The sum of all subsuming times is $O(k)$ time[2], whereas the sum of all growing times is $O(\min\{m, k\Delta\})$.

7.1 $\mathcal{SYNC}$ General Dispersion Algorithm

Starting from an arbitrary initial configuration, multiple DFSs grow concurrently as per Algorithm **RootedSyncDisp** (Section 5). Essentially the KS algorithm is run but in the growing phase Algorithm **RootedSyncDisp** is followed. When there is a meeting between two DFS trees, the subsuming phase of the KS algorithm is run. The re-traversal of DFS i in the subsuming phase has to be adapted to account for the oscillating agents – specifically, when visiting a node in the re-traversal, the agent waits there for up to 6 rounds so that if the settled agent is an oscillating agent, the oscillating agent visits its home within that period – and this introduces an $O(1)$ factor in time complexity, thereby the overall re-traversal still takes $O(d_i)$ time. The sum of all the subsuming phases is thus $O(k)$ and, from the analysis of **RootedSyncDisp**, the sum of all the growing phases is $O(k)$. Thus, the general initial configuration can be solved in $O(k)$ rounds in $\mathcal{SYNC}$. This leads to the following result.

Theorem 7.1. *Starting from general initial configurations, dispersion can be solved in $O(k)$ rounds using $O(\log(k+\Delta))$ bits per agent in $\mathcal{SYNC}$.*

7.2 $\mathcal{ASYNC}$ General Dispersion Algorithm

The KS algorithm is run, but in the growing phases, we use the Algorithm **RootedAsyncDisp** (Section 6). As in the $\mathcal{SYNC}$ general algorithm, the growing phases (of Algorithm **RootedAsyncDisp**) and the subsuming phases in the KS algorithm need to be modified to deal with an empty node which is the home node of a settled agent that is currently helping with probing. In particular, how to detect that DFS i meets DFS j (and deal with it) when the "meeting node" does not have an agent from DFS j because that agent is helping with probing. The modifications required are detailed below.

In Lines 10-11 of procedure `Async_Probe()` (please refer pseudocode in Algorithm 6 in [22]), if a_i finds that the settler at u_i, $\alpha(u_i)$ belongs to a different DFS (determined by having a different *treelabel* t' than that of a_i which is t), then the DFS t meets DFS t'; in this case the procedures *Exploration*, *Collapse_Into_Child* and *Collapse_Into_Parent* for DFS t are invoked as in the KS algorithm after executing `Guest_See_Off()` (Algorithm 7 in [22]) for all the agents that were collected during `Async_Probe()`. (If there are multiple such trees t', the *treelabel* t' settler at the u_i reachable by the lowest numbered port is considered.) With this change, the

[2] In [28], the subsuming time assumed a loose bound of $O(\min\{m, k\Delta\})$ because that did not affect the overall time complexity of the algorithm due to the larger time of $O(\min\{m, k\Delta\})$ of the growing phase. However, the sum of all subsuming times is $O(k)$ as each re-traversal of DFS component i can be done in $O(d_i)$ steps.

executions in the growing phases occur in $O(k \log k)$ epochs, and as the executions in the subsuming phases in the KS algorithm also occur in $O(k)$ epochs, the overall run-time is $O(k \log k)$.

Algorithm **RootedAsyncDisp** (Algorithm 8 in [22]) follows the same steps in conjunction with the KS algorithm for the general asynchronous case but with additional modifications. If an agent a with *treelabel* t is helping (in parallel) `Async_Probe()` procedure to search for the next DFS node of DFS t, then the agent a leaves its home node u to do probing. During that time, u might be acquired by an agent a' with *treelabel* t' in the absence of a. We call this scenario *squatting* due to agent a'. If DFS t' was terminated with the settling of a', then when a returns to u, DFS t' is said to meet DFS t and the KS algorithm executes the subsuming steps after the meeting as usual. If DFS t' was not over with the settling of a', then DFS *treelabel* t' unaware of the squatting act a', does the DFS tree traversal as per protocol. In doing so, observe that it will follow the path of DFS t either downstream away from the root, or upstream towards the root (and then possibly downstream) because agents perform DFS based on port numbering. This is because all DFSs (t and t') perform their DFSs based on port numbering beginning with port 1,2, . . ., δ_i. This approach requires the following assumption for the system model as follows. For any edge (u, v), the two ports cannot be labelled $(1, 1)$, $(1, 2)$, $(2, 1)$, or $(2, 2)$, subject to the following exceptions. (i) Port number 1 is permitted if that is the only port at a node. (ii) Port number 2 is permitted if there are only two ports at a node. This assumption is required to ensure that when the path of DFS t' meets the path of DFS t (at a node which is currently vacant due to the settled node of t helping in `Async_Probe()`), DFS t' proceeds either in the upstream or downstream direction of DFS t, irrespective of the port numbers from which the two DFSs entered that node.

There might be the case that agents of DFS t on that path followed by the agents of t' are also helping in the probing and are not present at their home nodes. Eventually, the head of t' either (i) finds an agent of t'' (may or may not equal t) or (ii) all the agents get settled along the path. For case (ii), the last agent of t' to settle (having no child) creates a meeting with t when the corresponding agent from t returns, and the KS algorithm executes the subsuming steps. For case (i), a meeting occurs with t'', and the subsequent steps of KS subsumption are executed. (This can happen recursively where t'' is distinct from t but the recursion terminates when t'' equals t or case (ii) holds.) If in the KS subsumption steps, a node is reached where no agent is present when it should be whereas an agent from another DFS is settled there (that other agent must be squatting), the agents wait there; the missing agent is helping in `Async_Probe()` and is guaranteed to return there in $O(\log k)$ epochs.

Note that when agent a of DFS t returns to its home u where a' of DFS t' is squatting, both can continue to co-exist there for a while (until one gets subsumed by or subsumes another DFS due to a meeting at possibly another node; see cases (i) and (ii) in the above paragraph.). The node u has an overlay of multiple DFSs and acts as multiple independent virtual nodes where the different DFSs' agents have settled. A third (or more) agent a'' from DFS t'' can also be at u and when it arrives, a'' is said to meet a' or a, it does not matter which. The KS algorithm steps for subsumption are executed at the meeting.

Two further modifications are required to the KS algorithm.

1. When a DFS t' is collapsing, if a node has agents from t' and other DFSs (such as t), only the tree t' collapses, and only agents of t' are collected/participate in the collapse.
2. In the KS algorithm, d_i denotes the number of settled agents in DFS i, and $parent(i)$ denotes the DFS that DFS i has met. $head(i)$ denotes the last node in DFS i at which agents of DFS i are currently present. In *Parent_Is_Collapsing* (Algorithm 3 of KS, lines 28-29), we require the change given below following the boldface "**after**".
   "If ($d_i > d_{parent(i)}$) and $head(i)$ junction is not locked and remains unlocked until $parent(i)$'s collapse reaches $head(i)$, then unsettled robots get absorbed in $parent(i)$ during its collapse **after** the agent a'' of DFS i at $head(i)$ informs its parent node w, i.e., $parent(a'')$ in DFS i, to create a meeting with t, so that (if and when if not already) the corresponding agent from t returns to w after helping with parallel probing, the KS algorithm executes the subsequent subsuming steps."

The sum of all the subsuming phases of KS is $O(k)$ and, from the analysis of Algorithm **RootedAsyncDisp**, the sum of all the growing phases is $O(k \log k)$. All the above changes do not add asymptotically at this time. The waiting for the agent of DFS t to return to its home node after helping with parallel probing increases the subsuming time of $O(k)$ by a factor of at most $O(\log k)$ as the wait time of $O(\log k)$ can be incurred $O(k)$ times — as $O(k)$ number of times `Async_Probe()` is invoked when nodes may be at the head of some DFS in the growing phase. Thus, the general initial configuration can be solved in $O(k \log k)$ epochs in $\mathcal{ASYNC}$. This gives the following result.

Theorem 7.2. *Starting from general initial configurations, dispersion can be solved in $O(k \log k)$ epochs using $O(\log(k+\Delta))$ bits per agent in $\mathcal{ASYNC}$.*

8 Concluding Remarks

In this paper, we have developed two novel techniques, one for $\mathcal{SYNC}$ and another for $\mathcal{ASYNC}$. For $\mathcal{SYNC}$, we have introduced a technique of leaving some DFS tree nodes empty during the DFS traversal which are covered through an oscillation trip of (at most) 6 rounds by a settled agent at a node. This technique allowed to solve dispersion with optimal time complexity $O(k)$ in $\mathcal{SYNC}$ with memory only $O(\log(k+\Delta))$ bits per agent, improving the best previously known time bound by a factor of $O(\log^2 k)$. For $\mathcal{ASYNC}$, we developed a technique to extend the previous idea in $\mathcal{SYNC}$ due to Sudo *et al.* [40], solving dispersion in $O(k \log k)$ time complexity with memory only $O(\log(k + \Delta))$ bits per agent. This is almost a quadratic improvement compared to the best previously known time bound of $O(\min\{m, k\Delta\})$ in $\mathcal{ASYNC}$ with $O(\log(k + \Delta))$ bits per agent. Closing the $O(\log k)$ factor gap in time complexity in $\mathcal{ASYNC}$ remains an intriguing open question for future work. The starting point toward addressing this open question to see whether our $\mathcal{SYNC}$ technique can be extended to $\mathcal{ASYNC}$ with no overhead on time complexity. Another direction is to see where dispersion could be solved in optimal time/memory complexities under faults (crash or Byzantine).

References

[1] John Augustine and William K. Moses Jr. 2018. Dispersion of Mobile Robots: A Study of Memory-Time Trade-offs. In *ICDCN*. 1:1–1:10.

[2] Evangelos Bampas, Leszek Gasieniec, Nicolas Hanusse, David Ilcinkas, Ralf Klasing, and Adrian Kosowski. 2009. Euler Tour Lock-in Problem in the Rotor-router Model: I Choose Pointers and You Choose Port Numbers. In *DISC* (Elche, Spain). 423–435.

[3] Rik Banerjee, Manish Kumar, and Anisur Rahaman Molla. 2024. Optimizing Robot Dispersion on Unoriented Grids: With and Without Fault Tolerance. In *ALGOWIN*, Quentin Bramas, Arnaud Casteigts, and Kitty Meeks (Eds.). Springer, 31–45.

[4] Rik Banerjee, Manish Kumar, and Anisur Rahaman Molla. 2025. Optimal Fault-Tolerant Dispersion on Oriented Grids. In *ICDCN*, Amos Korman, Sandip Chakraborty, Sathya Peri, Chiara Boldrini, and Peter Robinson (Eds.). ACM, 254–258.

[5] L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro. 2009. Uniform scattering of autonomous mobile robots in a grid. In *IPDPS*. 1–8.

[6] Prabhat Kumar Chand, Manish Kumar, Anisur Rahaman Molla, and Sumathi Sivasubramaniam. 2023. Fault-Tolerant Dispersion of Mobile Robots. In *CALDAM*, Amitabha Bagchi and Rahul Muthu (Eds.). 28–40.

[7] Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. 2008. Label-guided Graph Exploration by a Finite Automaton. *ACM Trans. Algorithms* 4, 4 (Aug. 2008), 42:1–42:18.

[8] Andreas Cord-Landwehr, Bastian Degener, Matthias Fischer, Martina Hüllmann, Barbara Kempkes, Alexander Klaas, Peter Kling, Sven Kurras, Marcus Märtens, Friedhelm Meyer auf der Heide, Christoph Raupach, Kamil Swierkot, Daniel Warner, Christoph Weddemann, and Daniel Wonisch. 2011. A New Approach for Analyzing Convergence Algorithms for Mobile Robots. In *ICALP*. 650–661.

[9] G. Cybenko. 1989. Dynamic Load Balancing for Distributed Memory Multiprocessors. *J. Parallel Distrib. Comput.* 7, 2 (Oct. 1989), 279–301.

[10] Archak Das, Kaustav Bose, and Buddhadeb Sau. 2021. Memory Optimal Dispersion by Anonymous Mobile Robots. In *CALDAM*, Apurva Mudgal and C. R. Subramanian (Eds.), Vol. 12601. Springer, 426–439.

[11] Shantanu Das, Paola Flocchini, Giuseppe Prencipe, Nicola Santoro, and Masafumi Yamashita. 2016. Autonomous mobile robots with lights. *Theor. Comput. Sci.* 609 (2016), 171–184.

[12] Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pajak, and Przemyslaw Uznański. 2015. Fast Collaborative Graph Exploration. *Inf. Comput.* 243, C (Aug. 2015), 37–49.

[13] Yotam Elor and Alfred M. Bruckstein. 2011. Uniform multi-agent deployment on a ring. *Theor. Comput. Sci.* 412, 8-10 (2011), 783–795.

[14] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. 2005. Graph Exploration by a Finite Automaton. *Theor. Comput. Sci.* 345, 2-3 (Nov. 2005), 331–344.

[15] Barun Gorain, Partha Sarathi Mandal, Kaushik Mondal, and Supantha Pandit. 2024. Collaborative dispersion by silent robots. *J. Parallel Distributed Comput.* 188 (2024), 104852.

[16] Giuseppe F. Italiano, Debasish Pattanayak, and Gokarna Sharma. 2022. Dispersion of Mobile Robots on Directed Anonymous Graphs. In *SIROCCO*. 11:1–11:21.

[17] Tanvir Kaur and Kaushik Mondal. 2023. Distance-2-Dispersion: Dispersion with Further Constraints. In *NETYS*, David Mohaisen and Thomas Wies (Eds.). 157–173.

[18] Ajay D. Kshemkalyani. 2025. Dispersion of mobile robots on graphs in the asynchronous model. *Theor. Comput. Sci.* 1044 (2025), 115272.

[19] Ajay D. Kshemkalyani and Faizan Ali. 2019. Efficient Dispersion of Mobile Robots on Graphs. In *ICDCN*. 218–227.

[20] Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. 2024. Brief Announcement: Agent-Based Leader Election, MST, and Beyond. In *DISC (LIPIcs, Vol. 319)*, Dan Alistarh (Ed.). 50:1–50:7.

[21] Ajay D Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. 2024. Faster Leader Election and Its Applications for Mobile Agents with Parameter Advice. In *ICDCIT*. Springer, 108–123.

[22] Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. 2025. Dispersion is (Almost) Optimal under (A)synchrony. *CoRR* abs/2503.16216 (2025). https://doi.org/10.48550/ARXIV.2503.16216 arXiv:2503.16216

[23] Ajay D. Kshemkalyani, Manish Kumar, Anisur Rahaman Molla, and Gokarna Sharma. 2025. Near-Linear Time Leader Election in Multiagent Networks. In *AAMAS* (Detroit, MI, USA). 1218–1226.

[24] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. 2019. Fast Dispersion of Mobile Robots on Arbitrary Graphs. In *ALGOSENSORS*. 23–40.

[25] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. 2020. Dispersion of Mobile Robots on Grids. In *WALCOM*. 183–197.

[26] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. 2020. Efficient Dispersion of Mobile Robots on Dynamic Graphs. In *ICDCS*. 732–742.

[27] Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. 2022. Dispersion of mobile robots using global communication. *J. Parallel Distributed Comput.* 161 (2022), 100–117. https://doi.org/10.1016/j.jpdc.2021.11.007

[28] Ajay D. Kshemkalyani and Gokarna Sharma. 2021. Near-Optimal Dispersion on Arbitrary Anonymous Graphs. In *OPODIS*. 8:1–8:19.

[29] Artur Menc, Dominik Pajak, and Przemyslaw Uznanski. 2017. Time and space optimality of rotor-router graph exploration. *Inf. Process. Lett.* 127 (2017), 17–20.

[30] Anisur Rahaman Molla and William K. Moses Jr. 2019. Dispersion of Mobile Robots: The Power of Randomness. In *TAMC*. 481–500.

[31] Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. 2021. Byzantine Dispersion on Graphs. In *IPDPS*. IEEE, 942–951.

[32] Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. 2021. Optimal dispersion on an anonymous ring in the presence of weak Byzantine robots. *Theor. Comput. Sci.* 887 (2021), 111–121.

[33] Debasish Pattanayak, Gokarna Sharma, and Partha Sarathi Mandal. 2021. Dispersion of Mobile Robots Tolerating Faults. In *WDALFR*. 17:1–17:6.

[34] Pavan Poudel and Gokarna Sharma. 2019. Time-Optimal Uniform Scattering in a Grid. In *ICDCN*. 228–237.

[35] Ashish Saxena, Tanvir Kaur, and Kaushik Mondal. 2025. Dispersion on Time Varying Graphs. In *ICDCN*, Amos Korman, Sandip Chakraborty, Sathya Peri, Chiara Boldrini, and Peter Robinson (Eds.). ACM, 269–273.

[36] Ashish Saxena and Kaushik Mondal. 2025. Path Connected Dynamic Graphs with a Study of Efficient Dispersion. In *ICDCN*, Amos Korman, Sandip Chakraborty, Sathya Peri, Chiara Boldrini, and Peter Robinson (Eds.). ACM, 171–180.

[37] Masahiro Shibata, Toshiya Mega, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. 2016. Uniform Deployment of Mobile Agents in Asynchronous Rings. In *PODC* (Chicago, Illinois, USA). 415–424.

[38] Takahiro Shintaku, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. 2020. Efficient Dispersion of Mobile Agents without Global Knowledge. In *SSS*. 280–294.

[39] Raghu Subramanian and Isaac D. Scherson. 1994. An Analysis of Diffusive Load-balancing. In *SPAA* (Cape May, New Jersey, USA). 220–225.

[40] Yuichi Sudo, Masahiro Shibata, Junya Nakamura, Yonghwan Kim, and Toshimitsu Masuzawa. 2024. Near-linear Time Dispersion of Mobile Agents. In *DISC*. LIPIcs, 38:1–38:22.