Journal of Computer and System Sciences 152 (2025) 103656

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

journal homepage: www.elsevier.com/locate/jcss

Near-optimal dispersion on arbitrary anonymous graphs ☆

Ajay D. Kshemkalyani [a], Gokarna Sharma [b,*]

[a] Department of Computer Science, University of Illinois at Chicago, Chicago, IL, USA
[b] Department of Computer Science, Kent State University, Kent, OH, USA

ARTICLE INFO

Article history:
Received 2 November 2022
Received in revised form 13 February 2025
Accepted 6 March 2025
Available online 22 March 2025

Keywords:
Distributed algorithms
Multi-agent systems
Mobile robots
Local communication
Dispersion
Exploration
Time and memory complexity

ABSTRACT

Given an undirected, anonymous, port-labeled graph of n memory-less nodes, m edges, and degree Δ, we consider the problem of dispersing $k \leq n$ robots (or tokens) positioned initially arbitrarily on the nodes of the graph to exactly k different nodes, one on each node. The objective is to simultaneously minimize time and memory requirement at each robot. The best previously known algorithm solves this problem in $O(\min\{m, k\Delta\} \cdot \log \ell)$ time storing $O(\log(k+\Delta))$ bits at each robot, where $\ell \leq k/2$ is the number of nodes with multiple robots positioned on them in the initial configuration. In this paper, we present a novel multi-source DFS traversal algorithm solving this problem in $O(\min\{m, k\Delta\})$ time with $O(\log(k+\Delta))$ bits at each robot. The memory complexity of our algorithm is already asymptotically optimal and the time complexity is asymptotically optimal for the graphs of constant degree $\Delta = O(1)$. The result holds in both synchronous and asynchronous settings.

1. Introduction

Given an undirected, anonymous, port-labeled graph of n memory-less nodes, m edges, and (maximum) degree Δ, we consider the problem of dispersing $k \leq n$ robots (or tokens) positioned initially arbitrarily on one or more nodes of the graph to exactly k different nodes of the graph, one on each node (which we call the DISPERSION problem). This problem has many practical applications, for example, in relocating self-driven electric cars (robots) to recharge stations (nodes), assuming that the cars have smart devices to communicate with each other to find a free/empty charging station [2,3]. This problem is also important because it has the flavor of many other well-studied robot coordination problems, such as exploration, scattering, load balancing, covering, and self-deployment [2–4].

One of the key aspects of mobile-robot research is to understand how to use the resource-limited robots to accomplish some large task in a distributed manner [5,6]. Along these lines, in this paper, we study the trade-off between time and memory complexities to solve DISPERSION by the resource-limited robots on arbitrary anonymous graphs. Time complexity is measured as the time duration to achieve dispersion and memory complexity is measured as the number of bits stored in persistent memory at each robot. The literature typically traded memory (or time) to obtain better time (or memory) bounds in arbitrary anonymous graphs (for example, compare memory and time bounds of the two algorithms from [3] given in Table 1).

☆ A preliminary version of this article appears in the Proceedings of OPODIS'21 [1].

* Corresponding author.
E-mail addresses: ajay@uic.edu (A.D. Kshemkalyani), gsharma2@kent.edu (G. Sharma).

https://doi.org/10.1016/j.jcss.2025.103656

Table 1
Algorithms solving DISPERSION for $k \leq n$ robots on undirected, anonymous, port-labeled graphs of n memory-less nodes, m edges, and (maximum) degree Δ. $\ell \leq k/2$ is the number of nodes in the graph with multiple robots positioned on them in the initial configuration (we call such nodes as multiplicity nodes); DISPERSION is already solved if there is no multiplicity node. †This time bound was obtained assuming m, k, and Δ are known to the algorithm a priori. ‡This time bound was obtained assuming only k and Δ are known to the algorithm a priori (but not m), which provides a different time-memory trade-off, i.e., with known m, the memory becomes sub-optimal $O(\log n)$ bits and time becomes $O(\min\{m, k\Delta\} \cdot \log \ell)$ but without known m, memory becomes optimal $\Theta(\log(k+\Delta))$ bits but time becomes $O(k\Delta \cdot \log \ell)$ which is worse compared to $O(\min\{m, k\Delta\} \cdot \log \ell)$ when $m < k\Delta$.

Algorithm	Memory/robot (in bits)	Time (in rounds/epochs)	Single-source/ Multi-source	Setting
Lower bound	$\Omega(\log(k+\Delta))$	$\Omega(k)$	both	Asynchronous
DFS	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	Single-source	Asynchronous
[3]	$O(k \log \Delta)$	$O(\min\{m, k\Delta\})$	Multi-source	Asynchronous
[3]	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot \ell)$	Multi-source	Asynchronous
[7]	$O(\log n)$	$O(\min\{m, k\Delta\} \cdot \log \ell)^{\dagger}$	Multi-source	Synchronous
[7]	$\Theta(\log(k+\Delta))$	$O(k\Delta \cdot \log \ell)^{\ddagger}$	Multi-source	Synchronous
[8]	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot \log \ell)$	Multi-source	Synchronous
Theorem 1	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	Multi-source	Synchronous
Theorem 2	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	Multi-source	Asynchronous

Recent studies [7,8] focused on minimizing time and memory complexities simultaneously. More precisely, they tried to answer the following question: *Can the time bound of* $O(\min\{m, k\Delta\})$ *be obtained keeping memory optimal* $\Theta(\log(k+\Delta))$ *bits at each robot?* This question can be easily answered in the single-source case of all $k \leq n$ robots initially co-located on a node. The idea is to run a *depth first search* (DFS) traversal starting from the node where all k robots are initially positioned, leaving a robot on each new node visited by the traversal. It can be shown that this procedure finishes in $O(\min\{m, k\Delta\})$ time with $O(\log(k+\Delta))$ bits at each robot. The challenge is how to answer it in the multi-source case of the robots initially on two or more nodes of the graph. For the multi-source case, the algorithms in [7,8] were successful in obtaining memory bounds either optimal or very close to optimal and time bounds at $O(\log \ell)$ factor away from $O(\min\{m, k\Delta\})$, where $\ell \leq k/2$ is the number of nodes in the graph with multiple robots positioned on them in the initial configuration (we call them *multiplicity nodes* throughout the paper). Specifically, Kshemkalyani et al. [7] obtained two results. In the first result, they obtained a time bound of $O(\min\{m, k\Delta\} \cdot \log \ell)$ and a memory bound of $O(\log n)$ bits, and in the second result, they obtained a time bound of $O(k\Delta \cdot \log \ell)$ and a memory bound of $\Theta(\log(k+\Delta))$ bits. In the first result, they assumed parameters m, k, and Δ are known to the robots a priori and in the second result, only k and Δ are assumed to be known to the robots. The first result is an improvement of $\ell / \log \ell$ factor compared to the $O(\min\{m, k\Delta\} \cdot \ell)$ time bound of [3] with sub-optimal memory. The second result is optimal in memory but the time bound has a $O(k\Delta)$ factor instead of a $O(\min\{m, k\Delta\})$ factor in the first result, which is worse when $m < k\Delta$. Shintaku et al. [8] obtained the optimal memory bound of $\Theta(\log(k+\Delta))$ bits and the time bound $O(\min\{m, k\Delta\} \cdot \log \ell)$ without robots knowing m, k, and Δ a priori.

In this paper, we present a new deterministic algorithm for DISPERSION that settles the question completely, i.e., it obtains the time bound of $O(\min\{m, k\Delta\})$ keeping memory optimal at $\Theta(\log(k+\Delta))$ bits at each robot, which is the first such result for the multi-source case. The time bound is an improvement of a $O(\log \ell)$ factor compared to the best previously known algorithms [7,8]. In fact, the result is obtained without knowing any of the parameters m, k, and Δ by robots a priori. Additionally, the result shows that the time bound is independent of the number of multiplicity nodes ℓ. Furthermore, the time and memory bounds match the respective bounds for the single-source case. Additionally, the memory complexity of our algorithm is already asymptotically optimal and time complexity is within $O(\Delta)$ factor from the asymptotically optimal time bound of $\Omega(k)$ for any arbitrary graph of arbitrary degree $\Delta < n$ [3]. If the graph happens to be of constant degree $\Delta = O(1)$ (which of course does not need to be known beforehand), then the time complexity also becomes asymptotically optimal, i.e., our algorithm becomes simultaneously optimal in time and memory requirement, the first such result.

1.1. Overview of the model and results

We consider $k \leq n$ robots operating on an undirected, anonymous (no node IDs), port-labeled graph G of n memory-less nodes, m edges, and degree Δ. The ports (leading to incident edges) at each node have unique labels from $[0, \delta - 1]$, where δ is the degree of that node. (Δ is the maximum over δ's of all n nodes.) The robots have unique IDs in the range $[1, k^{O(1)}]$. In contrast to graph nodes which are memory-less, the robots have memory to store information (otherwise the problem becomes unsolvable because the co-located robots cannot distinguish each other and cannot decide on which would leave the node and which would settle at the node). Finally, we consider the *local communication* model where, at any time, the robots co-located at the same node of G can communicate and exchange information, if needed, but they cannot communicate and exchange any information when located on different nodes. We call an initial configuration *single-source* if all k robots are initially positioned on a single node of G, otherwise we call it *multi-source*. Even in the multi-source initial configurations, the robots can only be on $1 < k' < k$ nodes, since for the case of $k' = k$, the initial configuration is

already a configuration that solves Dispersion. Furthermore, there can be only at most $\ell \leq k/2$ multiplicity nodes in any initial configuration, since for a node to be designated multiplicity node, it must have at least 2 robots positioned on it, and there are altogether k robots.

In this article, we establish the following theorem in the *synchronous* setting, in which all robots are activated in a round and they perform their operations simultaneously in synchronized rounds. In the synchronous setting, the time complexity (of the algorithm) is measured in rounds (or steps).

Theorem 1. *Given any initial configuration of $k \leq n$ mobile robots positioned on the nodes of an undirected, anonymous, port-labeled graph G of n memory-less nodes, m edges, and degree Δ,* Dispersion *can be solved deterministically in $O(\min\{m, k\Delta\})$ rounds in the synchronous setting storing $O(\log(k+\Delta))$ bits at each robot.*

Theorem 1 improves the time bound $O(\min\{m, k\Delta\} \cdot \log \ell)$ of the best previously known algorithms [7,8] by a factor of $O(\log \ell)$ keeping the memory optimal. Interestingly, both time and memory bounds of Theorem 1 match asymptotically the $O(\min\{m, k\Delta\})$ time and $O(\log(k+\Delta))$ memory bounds for the single-source case, which is inherent for any DFS traversal based algorithm for Dispersion. Finally, if the considered graph is of degree constant (of course the degree does not need to be known beforehand), i.e., $\Delta = O(1)$, then our algorithm becomes asymptotically optimal with respect to time in addition to memory, which is the first such result for Dispersion (Theorem 1). Notice that the memory is asymptotically optimal irrespective of the graph degree and time is asymptotically optimal within $O(\Delta)$ for any graph with any degree since there is a time lower bound of $\Omega(k)$ [3].

Furthermore, we extend Theorem 1 to the *asynchronous* setting where robots become active and perform their operations at an arbitrary speed, keeping the same time and memory bounds. Here we measure time in epochs (instead of rounds), which represents the time interval of each robot becoming active at least once.

Theorem 2. *Given the setting as in Theorem 1,* Dispersion *can be solved deterministically in $O(\min\{m, k\Delta\})$ epochs in the asynchronous setting storing $O(\log(k+\Delta))$ bits per robot.*

1.2. Challenges

The single-source Dispersion can be solved in $\min\{4m - 2n + 2, 4k\Delta\}$ rounds in any anonymous graph G having n memory-less nodes using the well-known DFS traversal [9] storing $O(\log(k+\Delta))$ bits at each robot. The multi-source Dispersion with the number of sources equal to k finishes in a single round, since k robots are already on k different nodes, a solution configuration for Dispersion. Therefore, the challenging case is the multi-source Dispersion with the number of sources k' satisfying $1 < k' < k$. Note here that $\ell \leq k'$ since ℓ only counts the number of sources with at least two robots positioned on them and k' considers all the nodes with at least a robot positioned on them.

The early papers obtained better bounds on either time or memory, trading one for another. The first algorithm of [3] obtained $O(\min\{m, k\Delta\})$ time bound with memory $O(k \log \Delta)$ bits at each robot. The second algorithm of [3] kept memory optimal at $O(\log(k+\Delta))$ bits at each robot and established time $O(\min\{m, k\Delta\} \cdot \ell)$, where $\ell \leq k/2$ is the number of multiplicity nodes in the initial configuration. Their algorithm starts ℓ different single-source DFS traversals in parallel from ℓ multiplicity sources. Each DFS traversal is given a unique ID, which is the smallest robot ID present on that source. Each DFS traversal leaves a robot on each new node it visits. If no DFS traversals meet, then k robots are on k different nodes and Dispersion is solved in time and memory bounds akin to the single-source DFS bounds. In case that two (or more) DFS traversals meet, the higher ID DFS traversal subsumes the lower ID DFS traversal. The problem here is that if the lower ID DFS traversal meets the higher ID DFS traversal, in the subsumption process, the higher ID DFS traversal may again visit all the nodes that the lower ID DFS traversal has already visited. Therefore, in the worst-case, the time becomes the multiplication of $O(\min\{m, k\Delta\})$ rounds for the single-source DFS traversal times ℓ parallel traversals, i.e., $O(\min\{m, k\Delta\} \cdot \ell)$ rounds.

Recent studies [7,8] reduced the $O(\ell)$ factor in the time bound to $O(\log \ell)$. Providing the m, k, and Δ parameters to the algorithm beforehand, Kshemkalyani et al. [7] ran ℓ-source DFS traversals in passes of $O(\min\{m, k\Delta\})$ rounds. After each pass, they guaranteed that the ℓ-source DFS traversal reduces to a $\ell/2$-source DFS traversal. Therefore, in a total of $\lceil \log \ell \rceil$ passes, the ℓ-source DFS traversal reduces to a single-source DFS traversal, which then finishes in an additional $O(\min\{m, k\Delta\})$ rounds, giving in the worst-case, $O(\min\{m, k\Delta\} \cdot \log \ell)$ rounds. The memory requirement is $O(\log n)$ bits at each robot, due to the memory to store $m (\leq n^2)$ which dominates the memory to store $k (\leq n)$ and $\Delta (< n)$. With only knowing the k and Δ parameters beforehand (and not m), the algorithm of Kshemkalyani et al. [7] can run the ℓ-source DFS traversals in passes of $O(k\Delta)$ rounds, solving Dispersion in $O(k\Delta \cdot \log \ell)$ rounds. The memory requirement becomes the optimal $\Theta(\log(k+\Delta))$ bits at each robot, improving on the $O(\log n)$ bits. The problem is that the time bound is worse when $m < k\Delta$. Recently, Shintaku et al. [8] established the time bound of $O(\min\{m, k\Delta\} \cdot \log \ell)$ rounds and memory bound of optimal $\Theta(\log(k+\Delta))$ bits at each robot, without the algorithm knowing m, k, and Δ beforehand.

Observing the techniques of [7,8], the algorithms developed there subsume different DFS traversals pairwise which helps in improving the sequential subsumption of the different DFS traversals in the algorithm of [3]. The implication of the pairwise subsumption is that only a $O(\log \ell)$ factor more cost is needed to subsume all ℓ parallel DFS traversals to obtain a

single DFS traversal. This $O(\log \ell)$ factor is significantly better compared to the $O(\ell)$ factor obtained due to the sequential subsumption. Recall that since ℓ can be $O(k)$, $O(\log \ell)$ becomes $O(\log k)$ which is a significant improvement compared to an $O(\ell) = O(k)$ factor.

Despite these benefits, the time bound due to the pairwise subsumption does not match the single-source DFS traversal time bound and, more importantly, it is not clear whether the $O(\log \ell)$ factor arising in the pairwise subsumption technique in [7,8] can be removed from the time bound. Therefore, a new set of ideas are needed, which we develop in this paper and they constitute our main contribution.

1.3. Techniques

We use parallel multi-source DFS traversals as in [7,8] but devise a novel subsumption technique, leading to $O(\min\{m, k\Delta\})$ time with optimal $O(\log(k+\Delta))$ bits at each robot, removing the $O(\log \ell)$ factor from the time bound of the best previously known algorithms [7,8] and matching the time and memory bounds for the single-source DFS traversal. Each DFS traversal constructs a *DFS tree*. Our technique executes subsumption on the two DFS traversals that meet based on the size of the DFS traversal measured as the number of settled robots with the same DFS tree ID. In fact, the larger size DFS traversal subsumes the smaller size DFS traversal. The subsumed DFS traversal is collapsed to a single node, collecting all the robots on that traversal at that node, and those robots are given to the subsuming DFS traversal allowing it to extend its DFS traversal. The benefit is two-fold: (i) the size of the subsumed traversal is smaller than the size of the subsuming traversal and hence the collapse of the subsumed traversal can be done in time proportional to the size of the subsumed traversal, and (ii) it avoids the need by the subsuming traversal of revisiting the nodes of the subsumed traversal more than once after the subsumption, a crucial aspect in removing the $O(\log \ell)$ factor from the time bound. The subsuming traversal may visit the nodes of the subsumed traversal at most once more. Thus the subsuming traversal visits all nodes at most once. Furthermore, one traversal always remains subsuming and continues to grow throughout the execution of the algorithm. The strength of our technique is that it does not use well-separated passes and the algorithm does a more careful bookkeeping of the time.

This is in contrast to the technique used in the best previously known algorithms [7,8] that uses IDs of the DFS traversals (larger ID DFS traversal subsumes smaller ID DFS traversal). The drawback of the subsumption based on DFS ID is that the algorithm cannot limit/avoid the repeating traversal of the already built DFS tree, adding a $O(\log \ell)$ factor in the subsumption process, and hence leading to a $O(\min\{m, k\Delta\} \cdot \log \ell)$ time bound.

We particularly tackle two major challenges: (i) how to execute the size-based subsumption, and (ii) what to do when more than two DFS traversals meet at different nodes forming a transitive chain, or more generally, what we define as a *meeting graph* (Definition 1). The first challenge is due to the fact that the exact size of the DFS traversal is only known by its *head node* which is either the current node that has all not-yet-settled robots (if any) belonging to that DFS traversal, or else the node on which the last robot belonging to that DFS traversal has settled. Therefore, it requires for the meeting DFS traversal to traverse the met DFS tree to reach its head node to find its size. We show that the DFS tree can be correctly traversed, the size can be correctly determined, and the DFS tree can be correctly collapsed. Our technique of collapsing the subsumed traversal successfully fulfills this requirement in time proportional to the size of the smaller size DFS traversal.

The second challenge is due to the fact that if not synchronized carefully, different DFS traversals in the transitive chain or meeting graph might run into a deadlock situation. We devise a technique that partitions the DFS traversals in the meeting graph such that in each partition, one DFS traversal subsumes the others without introducing any deadlock and in time proportional to the size of the DFS traversals (or the DFS trees) that were subsumed and collapsed.

Through these techniques, we finally show that one DFS traversal (among those that meet in the meeting graph) always grows bigger and the total cost remains proportional to the total size of the DFS traversals that are subsumed by the DFS traversal, giving our claimed time bound. Interestingly, the process is executed keeping the memory at an (asymptotically) optimal $\Theta(\log(k+\Delta))$ number of bits per robot.

1.4. Related work

Augustine and Moses Jr. [2] proved a memory lower bound of $\Omega(\log n)$ bits at each robot and a time lower bound of $\Omega(D)$ ($\Omega(n)$ in arbitrary graphs) for any deterministic algorithm for Dispersion on graphs. They then provided deterministic algorithms using $O(\log n)$ bits at each robot to solve Dispersion on lines, rings, and trees in $O(n)$ time. For arbitrary graphs, they gave one algorithm using $O(\log n)$ bits at each robot with $O(mn)$ time and another using $O(n \log n)$ bits at each robot with $O(m)$ time. All the upper and lower bound results of Augustine and Moses Jr. [2] assume $k = n$.

Kshemkalyani and Ali [3] and subsequent papers [7,8] considered the cases of $k \leq n$. Kshemkalyani and Ali [3] provided an $\Omega(k)$ time lower bound for arbitrary graphs and a memory lower bound of $\Theta(\log(k+\Delta))$ bits at each robot. Kshemkalyani and Ali [3] then provided three deterministic algorithms for Dispersion on arbitrary graphs: (i) The first algorithm using $O(k \log \Delta)$ bits at each robot with $O(\min\{m, k\Delta\})$ time, (ii) The second algorithm using $O(D \log \Delta)$ bits at each robot with $O(\Delta^D)$ time (D is the diameter of graph), and (iii) The third algorithm using $O(\log(k+\Delta))$ bits at each robot with $O(\min\{m, k\Delta\} \cdot \ell)$ time. As outlined in Table 1, the ℓ factor in [3] is improved to $\log \ell$ in Kshemkalyani et al. [7] assuming m, k, and Δ are known beforehand. The knowledge on m, k, and Δ in [7] is removed in Shintaku et al. [8]. For grid graphs, Kshemkalyani et al. [10] provided an algorithm that runs in $O(\min\{k, \sqrt{n}\})$ time using $\Theta(\log k)$ bits memory at each robot,

which is optimal for $k = n/c$, for some constant $c \geq 1$. Randomized algorithms were presented in [11,12] mainly to reduce the memory requirement at each robot. These algorithms typically considered the case of single-source Dispersion.

Recently, Kshemkalyani et al. [13] provided an algorithm for arbitrary graphs with time $O(\min\{m, k\Delta\})$ when all robots can communicate and exchange information in every round (that is, even the non-co-located robots can communicate and exchange information, which is called the *global communication* model). The global model comes handy while dealing with subsuming the multiple DFS traversals that meet in the transient chain or meeting graph. The information each robot can have allows the head node of the highest ID DFS traversal (satisfying a certain property) in the transient chain/meeting graph to ask the head nodes of the rest of the DFS traversals to stop growing their DFS tree. This makes sure that one DFS traversal always grows and others stop as soon as they find that they were met by the DFS traversal that is of higher ID then theirs. The result presented in this paper is different since only the co-located robots can communicate and it is called the *local communication* model. In the local communication model, it is not possible to extend the idea that is developed for the global communication model. For grid graphs in the global communication model, Kshemkalyani et al. [10] provided a time and memory optimal $\Theta(\sqrt{k})$ time algorithm with $\Theta(\log k)$ bits at each robot.

Dispersion in anonymous dynamic (undirected) graphs was considered in [4] where the authors provided some impossibility, lower, and upper bound results. Dispersion under crash faults was considered in [14] and under Byzantine faults was considered in [15,16] establishing a spectrum of interesting results. Recently, Dispersion was considered in directed anonymous graphs, where the authors [17] provided some impossibility, time and memory lower and upper bound results.

The related problem of exploration has been quite heavily studied in the literature for specific graphs such as grids and rings as well as arbitrary graphs, e.g., [18–24]. It was shown that a robot can explore an arbitrary anonymous graph using $\Theta(D \log \Delta)$-bits memory; the runtime of the algorithm is $O(\Delta^{D+1})$ [21]. In the model where graph nodes also have memory, Cohen et al. [19] gave two algorithms: The first algorithm uses $O(1)$-bits at the robot and 2 bits at each node, and the second algorithm uses $O(\log \Delta)$ bits at the robot and 1 bit at each node. The runtime of both algorithms is $O(m)$ with preprocessing time of $O(mD)$. The trade-off between exploration time and number of robots is studied in [24]. The collective exploration by a team of robots is studied in [22] for trees. The dual of the Dispersion problem is gathering, which has been extensively studied, e.g., [25,26]. Another problem related to Dispersion is the scattering of k robots on graphs. This problem has been mainly studied for rings [27,28] and grids [29]. Recently, Poudel and Sharma [30,31] provided improved time algorithms for uniform scattering on grids. Furthermore, Dispersion is related to the load balancing problem, where a given load at the nodes has to be (re-)distributed among several processors (nodes). This problem has been studied quite heavily in graphs, e.g., see [32,33]. We refer readers to these two excellent books [5,6] for many other recent developments in these topics.

1.5. Roadmap

We discuss model details in Section 2. The single-source DFS traversal is reviewed in Section 3. We then present our (synchronous) multi-source DFS traversal algorithm in Section 4, and prove its correctness, time, and memory complexities in Section 5, which establishes Theorem 1. We then discuss the extensions to the asynchronous setting, which establishes Theorem 2. Finally, we conclude in Section 6 with a short discussion.

2. Model

Graph. Let $G = (V, E)$ be a connected, unweighted, and undirected graph of n nodes, m edges, and maximum degree Δ. G is *anonymous* – nodes do not have identifiers but, at any node, its incident edges are uniquely identified by a port number in the range $[0, \delta - 1]$, where δ is the *degree* of that node. (Δ is the maximum among the degree δ of the nodes in G.) We assume that there is no correlation between two port numbers of an edge. Any number of robots are allowed to move along an edge at any time (i.e., unlimited edge bandwidth). The graph nodes are memory-less (do not have memory).

Robots. Let $\mathcal{R} = \{r_1, r_2, \ldots, r_k\}$ be the set of $k \leq n$ robots residing on the nodes of G. No robot can reside on the edges of G, but one or more robots can occupy the same node of G, which we call co-located robots. In the initial configuration, we assume that all k robots in $\mathcal{R}$ can be in one or more nodes of G but in the final configuration there must be exactly one robot on k different nodes of G. Suppose robots are on $k' \leq k$ nodes of G in the initial configuration. We denote by $\ell \leq k'$ the number of nodes in the initial configuration which have at least two robots co-located on them, i.e., there are ℓ multiplicity nodes.

Each robot has a unique $\lceil \log k \rceil$-bit ID taken from the range $[1, k^{O(1)}]$. When a robot moves from node u to node v in G, it is aware of the port of u it used to leave u and the port of v it used to enter v. We do not restrict the time duration of the local computation of the robots. The only guarantee is that all this happens in a finite cycle of "Communicate-Compute-Move" (defined below) and we measure time with respect to the number of cycles until Dispersion is achieved. Furthermore, it is assumed that each robot is equipped with memory (the goal is to use as less memory as possible). The robots work correctly at all times, i.e., they do not experience faults (neither crash nor Byzantine).

Communication Model. This paper considers the local communication model where only co-located robots at a graph node can communicate and exchange information. This model is in contrast to the global communication model where co-located as well as non-co-located robots (i.e., at different graph nodes) can communicate and exchange information.

Time Cycle. An active robot r_i performs the "Communicate-Compute-Move" (CCM) cycle as follows.

- *Communicate:* Let r_i be on node v_i. Since we consider the local communication model, for each robot $r_j \in \mathcal{R}$ that is co-located at v_i, r_i can observe the memory of r_j, including its own memory;
- *Compute:* r_i may perform an arbitrary computation using the information observed during the "communicate" portion of that cycle. This includes determination of a (possibly) port to use to exit v_i, the information to carry while exiting, and the information to store in the robot(s) r_j that stays at v_i;
- *Move:* r_i writes new information (if any) in the memory of a robot r_j at v_i, and exits v_i using the computed port to reach a neighbor node of v_i.

Robot Activation. In the *synchronous* setting, every robot is active in every CCM cycle. In the *asynchronous* setting, there is no common notion of time and no assumption is made on the number, duration, and frequency of CCM cycles in which a robot can be active. The only guarantee is that each robot is active infinitely often.

Time and Memory Complexity. For the synchronous setting, time is measured in *rounds*. Since a robot in the asynchronous setting could stay inactive for an indeterminate but finite time, we bound a robot's inactivity introducing the idea of an epoch. An *epoch* is the smallest interval of time within which each robot is guaranteed to be active at least once [34]. Let t_i be the time at which a robot $r_i \in \mathcal{R}$ starts its CCM cycle. Let t_j be the time at which the last robot finishes its CCM cycle. The time interval $t_j - t_i$ is an epoch. Memory complexity is measured as the number of bits stored in persistent memory at each robot. The algorithm we present does not require more memory in the Compute phase of the CCM cycle.

Our goal is to solve Dispersion minimizing both time (measured in rounds/epochs) and memory complexities (measured in bits stored per robot).

3. Single-source DFS traversal algorithm

We describe here a single-source DFS traversal algorithm, $DFS(k)$, that disperses all k robots in the set $R(v)$ situated initially at a node v to exactly k nodes of G, solving Dispersion. $DFS(k)$ will be heavily used in Section 4 as a basic building block for the multi-source DFS traversal algorithm.

Each robot r_i stores in its memory five variables.

(i) *parent* (initially assigned $\perp$), for a settled robot, denotes the port through which it first entered the node it is settled at;
(ii) *child* (initially assigned -1), for an unsettled robot, stores the port that it has last taken (while entering/exiting the node). For a settled robot, it indicates the port through which the other robots last left the node except when they entered the node in forward mode for the second or subsequent time;
(iii) *treelabel* (initially assigned $\min\{R(v)\}$) stores the ID of the smallest ID robot the tree is associated with;
(iv) $state \in \{forward, backtrack, settled\}$ (initially assigned $forward$). $DFS(k)$ executes in two phases, $forward$ and *backtrack* [9];
(v) *rank* (initialized to 0), for a settled robot indicates the serial number of the order in which it settled in its DFS tree.

The algorithm pseudo-code is shown in Algorithm 1. The robots in $R(v)$ move together in a DFS, leaving behind the highest ID robot at each newly discovered node. They all adopt the ID of the lowest ID robot in $R(v)$ which is the last to settle, as their *treelabel*. The algorithm executes in forward and backtrack modes.

Theorem 3 *([7]). The single-source DFS traversal algorithm $DFS(k)$ solves* Dispersion *for $k \leq n$ robots initially positioned on a single node of an arbitrary anonymous graph G of n memory-less nodes, m edges, and degree Δ in $\min\{4m - 2n + 2, 4k\Delta\}$ rounds using $O(\log(k + \Delta))$ bits at each robot.*

Proof. We include the proof to make the article self-contained. We first show that Dispersion is achieved by $DFS(k)$. Because every robot starts at the same node and follows the same path as other not-yet-settled robots until it is assigned to a node, $DFS(k)$ resembles the DFS traversal of an anonymous port-numbered graph [2] with all robots starting from the same node. Therefore, $DFS(k)$ visits k different nodes, where each robot is settled.

We now prove the time and memory bounds. The DFS traversal may take up to $4m - 2n + 2$ rounds, which is derived as follows. First consider the worst case where $k = n$. Define a forward edge as an edge which when traversed in the forward phase of the DFS traversal leads to the placement of a robot and define a backward edge as an edge which when traversed in the forward phase leads to a node where a robot has already been placed. There are $n - 1$ forward edges and each is traversed twice, once in the forward phase and once in the backward phase. There are $m - (n - 1)$ backward edges and each is traversed four times, once in both directions in the forward phase and once in both directions in the backward phase. This leads to $4m - 2n + 2$ edge traversals and as many rounds. Second, observe that in $4k\Delta$ rounds, $DFS(k)$ is guaranteed to visit at least k different nodes of G because in the DFS, each edge can be traversed at most 4 times and hence at most 4Δ traversals can visit a particular node [3]. If $4m - 2n + 2 < 4k\Delta$, $DFS(k)$ visits all n nodes of G. Therefore, it is clear that the

Algorithm 1: Algorithm *DFS(k)* for DFS traversal of a graph by k robots from a rooted initial configuration. Code for robot i. r is robot settled at the current node.

```
1  Initialize: child ← −1, parent ←⊥, state ← forward, treelabel ← min{R(v)}, rank ← 0
2  for round = 1 to min{4m − 2n + 2, 4kΔ} do
3      child ← port through which node is entered
4      if state = forward then
5          if node is free then
6              rank ← rank + 1
7              if i is the highest ID robot on the node then
8                  state ← settled, i settles at the node (does not move henceforth), parent ← child, treelabel ← lowest ID robot at the node,
                     child ← (child + 1) mod δ
9              else
10                 child ← (child + 1) mod δ
11                 if child = parent of robot settled at node then
12                     state ← backtrack
13         else
14             state ← backtrack
15     else if state=backtrack then
16         child ← (child + 1) mod δ, r.child ← child
17         if child ≠ parent of robot settled at node then
18             state ← forward
19     move out through child
```

runtime of $DFS(k)$ is $\min\{4m - 2n + 2, 4k\Delta\}$ rounds. Regarding memory, variable *treelabel* takes $O(\log k)$ bits, *state* takes $O(1)$ bits, and *parent* and *child* take $O(\log \Delta)$ bits. The k robots can be distinguished through $O(\log k)$ bits since their IDs are in the range $[1, k^{O(1)}]$. Thus, each robot requires $O(\log(k + \Delta))$ bits. □

4. Multi-source DFS traversal algorithm

4.1. Idea and algorithm

We now discuss the multi-source DFS traversal algorithm, which is the main contribution of this article.

The *root* of a DFS i (which equals the identifier (*treelabel*)) is the node where the first robot settles. This is the settled robot having $rank = 1$. The *head* of a DFS i is the node where the unsettled robots (if any) of that DFS are currently located at, or else it is the node where the last robot of that DFS settled. Node $root(i)$ is reachable by following *parent* pointers; node $head(i)$ is reachable by following *child* pointers.

In the initial configuration, if robots are at $k' < k$ nodes ($k' = k$ solves Dispersion in the first round without any robot moving), k' DFS traversals are initiated in parallel. If there is a single robot on a node, then it settles at that node and does not do anything. A DFS i *meets* DFS j if the robots of DFS i arrive at a node x where a robot from DFS j is settled. Node x is called a *junction* node of $head(i)$. If robots from multiple DFSs/nodes arrive at a node where there is no settled robot, a robot from the DFS with the highest ID settles in that round and the other DFSs are said to meet this DFS. If DFS i has met DFS j, we define $head(i)$ to be *blocked*, else we define $head(i)$ to be *free*.

The *size* d_i of a DFS i is the number of settled robots in that DFS. We first consider the isolated case where one DFS i meets one DFS j. When DFS i meets DFS j, the first task is to determine whether $d_i > d_j$ or $d_j > d_i$, where we define a total order ($>$) by using the DFS IDs as tiebreakers if the number of settled robots is the same. Size d_i is known to robots of DFS i at $head(i)$ by reading *rank* of DFS tree i. The unsettled robots at $head(i)$ traverse DFS j to $head(j)$ in an exploration to determine d_j. If they reach $head(j)$ without encountering a node with *rank* greater than d_i, then $d_i > d_j$. The junction of $head(j)$ is defined to be *locked* by i if DFS i's robots are the first to reach $head(j)$ in such an exploration (and at this time, j's exploratory robots that went on exploration because $head(j)$ was blocked have yet to return to $head(j)$). If $head(j)$ is locked by i then $head(j)$ has not been locked yet by any other k. However, if the exploratory robots of DFS i encounter a node with *rank* greater than d_i before reaching $head(j)$, they return to $head(i)$ as $d_j > d_i$. A key advantage of this mechanism is that $d_i > d_j$ can be determined in time proportional to $\min\{d_i, d_j\}$.

Knowing the sizes, the general idea is that if d_i is greater, DFS j is *subsumed* by DFS i and DFS j *collapses* by having all its robots collected to the $head(i)$ to continue DFS i. This collapse however cannot begin immediately because j's robots may be exploring some DFS l that DFS j has met and they must return to $head(j)$ before j starts its collapse. (The algorithm ensures there are no such cyclic waits to prevent deadlocks.) However, if d_j is greater, DFS i gets *subsumed*, i.e., DFS j subsumes DFS i. The free robots of i exploring j return to $head(i)$, DFS i *collapses* by having all its robots collected to $head(i)$, and then they all move to $head(j)$ to continue DFS j. Now, these above policies regarding which DFS collapses and gets subsumed by which other have to be adapted to the following fact – due to concurrent actions in different parts of G, a DFS j may

Algorithm 2: Algorithm *Exploration* to explore $parent(i)$ component on reaching junction of $head(i)$ by DFS of component i.

```
1  explorers move to root(parent(i)) leaving retrace pointers for return path. Then they follow child pointers from root(parent(i)) to head(parent(i)).
   There are 4 possibilities.
2  if d_parent(i) > d_i, i.e., rank > d_i is encountered, implying explorers do not reach head(parent(i)) (possibly the next junction) then
3      return to head(i) junction
4      if head(i) is not locked then
5          Collapse_Into_Parent(i)
6      else if head(i) is locked by j then
7          Collapse_Into_Child(i, j)
8  else if d_parent(i) < d_i, implying head(parent(i)) is reached (possibly next junction) then
9      lock head(parent(i)) if it is a junction node
10     traverse parent(i) informing each node (a) that parent(i) will be collapsing (and whether it is locked), and also (b) value of d_parent(i), and
         return to head(parent(i))
11     wait until parent(i)'s explorers (if head(parent(i)) is junction) return from parent(parent(i))
12     if head(parent(i)) is junction node then
13         follow action (Collapse_Into_Child(parent(i), i)) which will be determined on their return
14     else if head(parent(i)) is not junction node then
15         execute Collapse_Into_Child(parent(i), i)
16 else if exploring robots find parent(i) is collapsing or learn that parent(i) will be collapsing and is possibly locked then
17     Parent_Is_Collapsing
18 else if explorers E's path meets another explorers F's path then
19     wait until F return
20     if parent(i) is collapsing then
21         Parent_Is_Collapsing
22     else if parent(i) is not collapsing then
23         continue E's exploration
```

be met by different other DFSs, and DFS j may in turn meet another DFS concurrently. Further, transitive chains of such meetings can occur concurrently. This leads us to formalize the notion of a *meeting graph*.

Definition 1. (Meeting graph.) The directed meeting graph $G' = (V', E')$ is defined as follows. V' is the set of concurrently existing DFS IDs. There is a (directed) edge in E' from i to j if DFS i meets DFS j.

If multiple DFSs meet DFS j and $head(j)$ is locked by i among them, then j will collapse and get subsumed by i.

For an edge (i, j) in the meeting graph, DFS j is defined to be $parent(i)$ and DFS i is defined to be $child(j)$. The size of a node in the meeting graph is defined to be the size of the DFS for that node. Nodes in V' have an arbitrary in-degree ($< k'$) but out-degree at most 1. There may also be a cycle in each connected component of G'. Henceforth, we focus on a single connected component of G' by default; other connected components are dealt with similarly. The algorithm partitions a connected component of G' into (connected) sub-components such that each sub-component is defined to have a master node M into which all other nodes of that sub-component are subsumed, directly or transitively. At most one sub-component may have a cycle. However the algorithm ensures that there is no cyclic wait for subsuming in that cycle, as will follow from Property 1. In each sub-component, the master node M has the highest value of d and the other smaller (or equal sized) nodes, i.e., DFSs, get subsumed. The pseudo-code is given in Algorithm 2 and in Algorithm 3. In Algorithm 2, j is explored by robots from i to determine if $d_i > d_j$ (therefore, we sometimes call Algorithm 2 *Exploration*), and the appropriate procedures for collapsing and collecting are given in Algorithm 3 (therefore, we sometimes call Algorithm 3 *various procedures invoked*).

For any given node $i \in V'$, its master node is given as per Algorithm 4. Note that this algorithm is not actually executed and the master node of a node need not be known – it is given only to aid our understanding and in the complexity proof. If $master(j)$ gets invoked directly or transitively in the invocation of $master(i)$ for any i, then i must be subsumed and its robots collected completely before j gets subsumed and its robots are collected completely.

As the actions of collapsing occur concurrently (but acyclically, as we will show in Theorem 4), there should not arise a situation where a node j is collapsing (to be subsumed into its parent or some child) while some (other) child i is concurrently collapsing in order to be subsumed into its parent j only to find after collapsing that its parent j has already collapsed and disappeared/been subsumed. To prevent this, when it is determined that i should collapse into its parent, but before the collapse actually begins, a mark is left at j's robot stationed at the junction node $head(i)$ where DFS i has met DFS j. While j concurrently collapses (in which it has to visit the nodes of DFS j to collect the settled robots), if its explorers visit a node of j where there is a mark left, they deduce that they should pause j's collapse until its child node (i) that has met j at that junction node completes its collapse and assembles all i's robots at that junction node. The mark is reset by i at the time, signaling to j that it is safe to collect the j DFS robots along with i's collapsed robots from that

Algorithm 3: Algorithms *Collapse_Into_Child*, *Collapse_Into_Parent*, and *Parent_Is_Collapsing*.

Collapse_Into_Child(i,j)
explorers of i go from $head(i)$, which may be locked by j, to $root(i)$
explorers of i do i's DFS tree traversal collecting all robots to collapse path ($root(i)$ to $head(j)$) marked by *retrace* pointers, waiting until $collapsing\_children = 0$ at each node
from $root(i)$ collect all robots accumulated on collapse path to j's junction of $head(j)$
collapsed robots change ID *treelabel* to j, and *state*, *parent*, *child*, *rank* to that of the explorers of j
if $head(j)$ *is locked by* l **then**
    $Collapse\_Into\_Child(j, l)$
else if $head(j)$ *is not locked* **then**
    continue j's DFS

Collapse_Into_Parent(i)
robot at $head(i)$ increments $collapsing\_children$
explorers of i go from $head(i)$ to $root(i)$ leaving *collapse* pointers
explorers of i do i's DFS tree traversal collecting all robots to collapse path ($root(i)$ to $head(i)$) marked by *collapse* pointers, waiting until $collapsing\_children = 0$ at each node
from $root(i)$ collect all robots accumulated on collapse path to i's junction of $head(i)$
robot at $head(i)$ decrements $collapsing\_children$
explorers of i and collapsed robots change ID *treelabel* to $parent(i)$
explorers of i and collapsed robots go to $root(parent(i))$ and then to $head(parent(i))$ by following *child* pointers
explorers of i and collapsed robots change *state*, *parent*, *child*, *rank* to that of the settled robot at $head(parent(i))$ (if free) or of explorers of $parent(i)$ at $head(parent(i))$ (otherwise)
if $parent(i)$ *along the way is found to be collapsing* **then**
    collapse with it; break()
if $head(parent(i))$ *is free* **then**
    continue $parent(i)$'s DFS
else if $head(parent(i))$ *is blocked and possibly also locked* **then**
    wait until $parent(i)$ collapses (and collapse with it) or becomes unblocked (and continue $parent(i)$'s action acting as its unsettled robots)

Parent_Is_Collapsing
retrace path to $head(i)$ junction
if $d_i < d_{parent(i)}$ *and* $head(i)$ *junction is not locked* **then**
    $Collapse\_Into\_Parent(i)$
else 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
else if $head(i)$ *junction of* i *(is locked by* j*) or (gets locked by* j *before* $parent(i)$*'s collapse reaches* $head(i)$ *and* $d_i > d_{parent(i)}$*)* **then**
    $Collapse\_Into\_Child(i, j)$

Algorithm 4: Algorithm *Determine_Master(i)* to identify master component in which component i will collapse.

master(i)
if $d_{parent(i)} > d_i$ **then**
    $t1 \leftarrow$ time when explorers of i return to $head(i)$ from $parent(i)$
    $t2$ (initialized to ∞) $\leftarrow$ the time, if any, when first child j locks $head(i)$
    if $t1 < t2$ **then** $w \leftarrow parent(i)$
    else if $t1 > t2$ **then** $w \leftarrow j$
    return($master(w)$)
else
    if $\exists$ a first child j to lock $head(i)$ **then** return($master(j)$)
    else return(i)

junction node and continue its collapse. As multiple child DFSs can meet j at the same junction node and if they are all collapsing into their parent j, they must all collapse before j collects that junction node robot; thus they must all leave a mark at that junction node. This setting and resetting the mark takes the form of incrementing and decrementing an integer counter, which the algorithm names *collapsing_children*. When explorers of j visit each node in j when it is collapsing (to be subsumed by either its parent or some child), they wait for this counter to have value 0 before they collect the robot stationed there (along with possibly other robots of collpased children). Observe from the algorithm code that there will be no cyclic waits among DFSs' explorers.

Due to concurrent operations by the DFSs, multiple DFSs j may meet DFS i concurrently at possibly different junction nodes and start exploration of i from $head(j)$ to go to $head(i)$ via $root(i)$. DFS j explorers have to leave *retrace* pointers going from $head(j)$ to $root(i)$ to enable their way back. Different DFSs' exploratons' exploration paths in i may overlap. To bound the memory required to store multiple (up to $k' - 1$) *retrace* pointers (along with as many *explorer_ID* variables) at any settled robot in i to the memory for a single *retrace* pointer, the algorithm does as follows. When an explorer set E's path reaches a node in i where explorer set F has already passed in the forward exploration, explorer set E's robots wait

until explorer set F's robots return to that node on their way back and then E's robots resume their forward exploration, reusing the *retrace* pointer at the robots settled in i (and resetting it on the way back). Whereas if explorers from different DFSs arrive at a node of i in the same round, priority such as based on DFS ID can be used to select which explorer set proceeds while the others wait (acyclically).

4.2. Properties and correctness of the algorithm

The algorithm satisfies the following property.

Property 1. *For any node i in the meeting graph:*

1. *i collapses into one larger child j if j has locked $head(i)$[1] or explorers of j are the first to reach unblocked $head(i)$. Otherwise i collapses into $parent(i)$ if $parent(i)$ is larger and explorers of i return to $head(i)$ from $parent(i)$ and $head(i)$ is not locked until then. If neither of the above occurs, i does not collapse.*
2. *A smaller $child(i)$ collapses into i if $head(child(i))$ has not been locked by some $child(child(i))$ by the time that $child(i)$'s explorers return from exploring i to $head(child(i))$.*
3. *A larger $child(i)$ does not collapse into i.*
4. *A larger $child(i)$ that has not locked $head(i)$ and such that $head(child(i))$ is not locked until the time the collapse of i reaches $head(child(i))$ transfers its unsettled robots, which are at $head(child(i))$, to i and those robots get transitively absorbed by $master(i)$.*
5. *$parent(i)$ collapses into i if i is larger and has locked $head(parent(i))$[2] or if i is larger and its explorers are the first to reach unblocked $head(parent(i))$.*
6. *i neither collapses nor subsumes if (i is unblocked or is blocked on a smaller $parent(i)$ that collapses into its some other child or its parent), and (there is no $child(i)$ or each $child(i)$ is smaller and locked by one of its children $child(child(i))$) into which that $child(i)$ collapses.*

Theorem 4. ***(Deadlock-freedom/Safety:)*** *In any connected component of the meeting graph of size more than one, there will be no cyclic waits and no concurrent cyclic collapse/subsumption.*

Proof. If there is a cycle in a connected component of G' spanning more than one sub-component, such a cycle is temporary until exploration and subsumption completes; any collapses will be into more than one master and there cannot arise any permanent cyclic waits or concurrent cyclic collapse/subsumption.

So consider the case that there is a cycle in any connected sub-component of G'. We show that this cycle is temporary. Let i be any node (DFS) such that d_i is a local minima in the cycle – its size being less than that of its child and parent in the cycle. Only one of the following two must hold. (a) i has a larger child j that has locked $head(i)$ (before explorers of i return to $head(i)$), in which case i collapses into j. (b) Explorers of i return to $head(i)$ from the larger $parent(i)$ and $head(i)$ is not locked until then, in which case i collapses into $parent(i)$. In either case, the cycle which existed temporarily (until the explorations took place) is broken. This also follows from Property 1. 1; i will collapse and be subsumed either by its parent or that child in the cycle, or by a (larger) child not in the cycle. In any case, the cycle will be broken and neither will there be a concurrent cyclic collapse/subsumption as subsumption will not occur on both the incident edges in the cycle.

Further, the temporary cycle that might have existed was for the duration of the explorations. Observe from the code that there will not be indefinite waits as part of exploration and possible collapse along any of the cycle edges or any other edge of G'. □

Theorem 5. ***(Liveness:)*** *In any connected component of the meeting graph of size greater than one, at least one node gets subsumed and at least one master node of that connected component progresses with its DFS after it subsumes the other nodes in its sub-component.*

Proof. It follows from Property 1 and Theorem 4 that at least one node gets subsumed. In particular, at least the smallest size DFS in that connected component must get subsumed; as its parent and all its children are larger it must collapse into one of them.

The master node M of that node collects the robots of the subsumed DFS(s) at $head(M)$ from the algorithm. Further, M must now be unblocked because

1. if it were still blocked on a smaller parent $parent(i) = j$, that parent would have collapsed into (a) another of its children that had managed to lock $head(parent(i))$, otherwise (b) into $parent(parent(i))$ if larger than $parent(i)$ or (c) into M which would have succeeded in locking $parent(i)$, and

[1] Recall from the definition that j locks $head(i)$ if $head(i)$ is blocked and explorers of i do not return to $head(i)$ from $parent(i)$ before explorers of j reach $head(i)$.

[2] This happens if explorers of $parent(i)$ do not return to $head(parent(i))$ before it gets locked by i.

2. if it were still blocked on a larger parent, it would have collapsed into that parent and not be a master node.

Case (2) is an impossibility and so cannot happen. In Case 1(c), M would be unblocked and able to continue its DFS with the unsettled (and subsumed) robots. In Cases 1(a) and 1(b), M might transfer out unsettled (including subsumed DFSs') robots to j and transitively those would be collected at $head(master(j))$. The same logic applies to $master(j)$ – if cases 1(a) or 1(b) applied to it, it would transfer out the unsettled (including subsumed DFSs') robots which would be transitively collected at $head(master(parent(master(j))))$. This argument can continue inductively at most $k' - 1$ times and for the last time, the master node identified must not only be unblocked after subsuming nodes in its sub-component but must also be able to continue its DFS. To see this further, even if there were a cycle in the meeting graph, once M transfers out its unsettled (and collected) robots to $parent(M)$ and becomes unblocked, $parent(M)$ has collapsed and those transferred out robots even if traversing the cycle via transfers by the master nodes along the cycle were to come back to M, they would not be transferred out a second time along the same cycle as that $parent(M)$ no longer exists. At this time, M is either unblocked and possibly continuing its DFS, or has blocked after continuing its DFS and is part of a new connected component. □

A path in G' is an *increasing* (*decreasing*) path if the node sizes along the path are increasing (decreasing). For a master node M, the nodes x in its sub-component of G' that directly and transitively participate in only *Collapse_Into_Parent* and no *Collapse_Into_Child* until collapsing into M form the set $X(M)$. Whereas the (other) nodes y in the sub-component that directly and transitively invoke at least one *Collapse_Into_Child* until they collapse into M belong to the set $Y(M)$. The sub-component $C(M) = X(M) \cup Y(M) \cup \{M\}$.

Even though there may be a cycle in sub-component $C(M)$, when excluding the E' edges along which no subsumption occurs, the component is acyclic. For an edge (i, j), i is the child and j is the parent. Nodes in the set X have an increasing path to the master node. They collapse into and get subsumed by the master node (possibly transitively) by executing *Collapse_Into_Parent*. Nodes in the set Y are reachable from the master node on a decreasing path – such nodes are termed *Y_trunk* nodes, or have an increasing path to a *Y_trunk* node – such nodes are termed *Y_branch* nodes. Thus there is just one decreasing path from the master node and the increasing paths are the other paths that end in some nodes of the decreasing path that includes the master. Nodes in Y (i.e., in *Y_trunk* and *Y_branch*) collapse into and get subsumed by the master node, possibly transitively. First, the *Y_branch* nodes collapse into and get subsumed by their ancestors transitively on the increasing path ending in a *Y_trunk* node by executing *Collapse_Into_Parent*; then the *Y_trunk* nodes collapse and get subsumed into their child nodes transitively along *Y_trunk* and then into the master node by executing *Collapse_Into_Child*.

4.3. Dealing with dynamism

After nodes in $C(M)$ get subsumed in M, the DFS of the master node is unblocked and continues again until involved in more meetings and new meeting graphs are formed. Thus the meeting graph is *dynamic*. We define a related notion of a *meeting tree* that represents which nodes (DFSs) have met and been subsumed by which master node, in which meeting sequence number of meetings for each such node over time. A node in the meeting graph G' as well as in the meeting tree is formally identified by a tuple (a, h) where a is the DFS ID of the master node and h indicates that it has participated in at most h meetings until now. More specifically, h denotes the maximum of the number of times that the robots of some initial DFS component a' have participated in subsumption (subsumed by or subsumed) progressively over time until now they are a part of component having DFS ID a. h is thus the height of the node (a, h) in the meeting tree and is upper-bounded by $k' - 1$.

Definition 2. (**Meeting tree.**) The k' initial DFSs i form the k' leaf nodes $(i, 0)$ at level 0. When α nodes (a_i, h_i) for $i \in [1, \alpha]$ meet in a component and get subsumed by one of them — the master node with DFS identifier M of the meeting graph, — a node (M, h), where $h = 1 + \max_{i \in [1,\alpha]} h_i$, is created in the meeting tree as the parent of the child nodes (a_i, h_i), for $i \in [1, \alpha]$.

For a node (M, h), h is the length of the longest path from some leaf node to that node. We now formally define $X(M, h)$, $Y(M, h)$, and $C(M, h)$.

Definition 3. (Component $C(M, h)$.) For node (M, h) in the meeting tree we define $C(M, h)$ as follows.
For $h = 0$, $C(M, h) = \{(M, 0)\}$.
For $h > 0$, $C(M, h) = \{(M, z)\} \cup X(M, h) \cup Y(M, h)$ where:

1. (M, z) is the child node of (M, h) in the meeting tree having the same DFS identifier M.
2. $X(M, h)$ is the set of child nodes of (M, h) in the meeting tree that directly and transitively participate only in *Collapse_Into_Parent* until collapsing into (M, z) as part of forming (M, h).
3. $Y(M, h)$ is the set of child nodes of (M, h) in the meeting tree that directly and transitively participate in at least one *Collapse_Into_Child* until collapsing into (M, z) as part of forming (M, h).

Definition 4. ($prev(h)$ **and** $next(h)$.)

- For a node (a, z) in the meeting tree having parent (b, h), define $prev(h)$[3] to be z. More simply, a node $(a, prev(h))$ in the meeting tree has its parent (b, h).
- For node (i, h) in the meeting tree, define $next(h)$ to be the value h' such that (M, h') is the parent of (i, h), if a parent exists, otherwise define $next(h)$ to be k'.

We omit h in (i, h) and $C(M, h)$ in places where it is understood or not required.

5. Analysis of the algorithm

In our algorithm, a common module is to traverse an already identified DFS component with nodes having the same *treelabel*. Such a DFS traversal occurs in (i) Algorithm *Exploration* (traversal of $parent(i)$ to notify robots at nodes in $parent(i)$ of its impending collapse) when $d_i > d_{parent(i)}$ and i locks $head(parent(i))$ junction, (ii) procedure *Collapse_Into_Child* (traversal of i to collect the settled robots in i for collapse), and (iii) procedure *Collapse_Into_Parent* (traversal of i to collect the settled robots in i for collapse). In (ii) and (iii), a settled robot not on the collapse path gets unsettled and gets collected in the DFS traversal to the collapse path when the DFS backtracks from the node where the robot was settled. Note that there can be at most two such DFS re-traversals of any i despite concurrent explorations of i – either in *Collapse_Into_Child* or else in *Collapse_Into_Parent*, and once in Algorithm *Exploration* if *Collapse_Into_Child* is executed – so two duplicate sets of variables *treelabel*, *state*, *child*, *rank*, and *parent* suffice. Such a retraversal of component i can be achieved by going to $root(i)$ and doing a (new) DFS traversal of only those nodes that are in i (using a duplicate set of variables *treelabel*, *state*, *rank*, *child*, and *parent*). This new DFS traversal uses the same initial values of the duplicate variables as used in the initial traversal which happened in spurts each time component i subsumed other components and then resumed dispersion (extending the DFS traversal until then) via the next spurt in its DFS traversal. Note that the robots of the other subsumed components set their *treelabel*, *state*, *rank*, *parent*, and *child* variables to that of the last-settled robot of the then-current DFS i component into which they collapsed. Hence the new DFS traversal will be identical to the initial (in-spurts) traversal in terms of sequence of nodes visited; if one reaches a node which has no settled robot or a settled robot having a different *treelabel*, one backtracks along that edge and ends the traversal. Such a DFS retraversal of i can be executed in $4\Delta d_i$ steps.[4]

The time complexity of Algorithms 2 and 3 is as follows.

1. Algorithm 2 takes time bounded by $8d_i\Delta + 3d_i$. The derivation is as follows.
   (a) $\min\{d_i, d_{parent(i)}\}$ to go from $head(i)$ to $root(parent(i))$.
   (b) $4\min\{d_i, d_{parent(i)}\}\Delta$ to go then to $head(parent(i))$.
   (c) if $d_{parent(i)} > d_i$, then $2d_i$ to return to $head(i)$ via $root(parent(i))$.
   (d) if $d_{parent(i)} < d_i$ and i locks $head(parent(i))$, then $4d_{parent(i)}\Delta + 2d_{parent(i)}$ for DFS traversal of $parent(i)$ component from $root(parent(i))$ plus to $root(parent(i))$ from $head(parent(i))$ and back.

   If explorers E's path meets explorers F's path, the explorers E wait until F's return. This delay is analyzed later.
2. In Algorithm 3,
   (a) *Collapse_Into_Child* takes $4d_i\Delta + 2d_i$.
   Time d_i to go from $head(i)$ to $root(i)$; $4\Delta d_i$ for a DFS traversal of i component from $root(i)$; and d_i to collect the accumulated robots from $root(i)$ to $head(j)$ along the collapse path.
   (b) *Collapse_Into_Parent* takes $4d_i\Delta + 2d_i + 4d_{parent(i)}\Delta$.
   Time d_i to go from $head(i)$ to $root(i)$; $4\Delta d_i$ for a DFS traversal of i component from $root(i)$; d_i to collect the accumulated robots from $root(i)$ to $head(i)$; and $4d_{parent(i)}\Delta$ to then go to $head(parent(i))$.
   (c) The cost of *Parent_Is_Collapsing* is $\min\{d_i, d_{parent(i)}\}$ but is subsumed in the cost of Algorithm 2.
   This cost is to return to $head(i)$ from the exploration point in $parent(i)$ component where it is invoked.

The contributions to this time complexity by the various nodes in $C(M)$ are as follows. (The cost is the sum of Algorithm *Exploration* plus appropriate invoked procedure costs.)

[3] We avoid the more technically accurate expression $prev(a, h)$ because we never use $prev(h)$ stand-alone but it is always used in conjunction with a in a tuple or a is implicit from context.

[4] A retraversal can be done in $8d_i$ steps by maintaining a *next_sibling* variable that gives that port number at the parent, following which leads to the next sibling node in the initial DFS tree. Updating this variable would complicate the presentation and using this does not reduce the asymptotic complexity because the initial DFS traversal in Algorithm 1 requires $4\Delta d_i$ steps (Theorem 3).

1. Each $x \in X$ executes *Collapse_Into_Parent* after *Exploration*, as it is part of an increasing path. So it contributes the sum of the two contributions, giving $12d_x\Delta + 5d_x + 4d_{parent(x)}\Delta$.

   The $4d_{parent(x)}\Delta$ is for traversing to $head(parent(x))$ after x collapses to $head(x)$, and this can be done concurrently by multiple x that are children of the same parent. As each x can be thought of as the *parent* of another element in X, so the cost of subsuming the X set is $\sum_{x \in X} 16d_x\Delta + 5d_x$ + (if $X \neq \emptyset$, $4d_M\Delta$).
2. Each $y \in Y\_branch$ executes *Collapse_Into_Parent* after *Exploration*, as it is part of an increasing path. So it contributes the sum of the two contributions, giving $16d_y\Delta + 5d_y$.

   Each $y \in Y\_trunk$ executes *Collapse_Into_Child* after *Exploration*, as it is part of a decreasing path. So it contributes the sum of the two contributions, giving $12d_y\Delta + 5d_y$, plus it potentially acts as a parent of a node on a *Y_branch* that executed *Collapse_Into_Parent* so it contributes an added $4d_y\Delta$, giving a total of $16d_y\Delta + 5d_y$.
3. Node M will contribute in Algorithm *Exploration* $4\min\{d_M, d_{parent(M)}\}\Delta + \min\{d_M, d_{parent(M)}\}$, plus $4d_{parent(M)}\Delta + 2d_{parent(M)}$ as $parent(M)$ is smaller. Thus, a total of $8d_{parent(M)}\Delta + 3d_{parent(M)}$. This can be counted towards a contribution by $parent(M) = y \in Y$, thus the contribution of each $y \in Y$ can be bounded by $24d_y\Delta + 8d_y$ with M contributing nil.

There is another source of time overhead contributed by nodes in $Y\_trunk \cup \{M\}$, which belong to a decreasing path. Nodes y, i.e., $head(y) \in G$, for $y \in Y\_trunk$, are locked by their child. Before this can happen, other children of y may be exploring y by leaving *retrace* pointers. However, due to the $O(\log(k + \Delta))$ bits bound on memory at each robot, a *retrace* pointer at a node in y can be left by only $O(1)$ children, not by $O(k')$ children. Therefore in Algorithm 2, if explorers E path meets another explorers F path, they wait at the meeting node until explorers F return. If they learn that y is collapsing, they retrace to their *head* nodes, else if they learn that y is not collapsing, they continue their exploration towards $head(y)$ but may have to wait again if their path meets the path already taken by another explorers F'. This waiting due to concurrently exploring children introduces delays.

A child of y outside Y_trunk may be either locked (l) or unlocked (u) and is also smaller (S) or larger (L) than y. Thus, there are 4 classes of such children.

1. *Su*-type children belong to *Y_branch* and their introduced delays are already accounted for above.
2. Each *Lu*-type and *Ll*-type child does not contribute any delay. This is because even though these children are larger than y, they are not the child in Y who succeeds in locking y; the child in Y (in *Y_trunk*) or M who locks y does so before such $L*$-type children try (and fail) to lock y in their exploration of y. Such $L*$-type children learn that y is collapsing. A *Lu*-type child transfers its unsettled robots to y as part of y's collapse, without any additional time cost.
3. Each *Sl*-type child node b contributes delay $4d_b\Delta + 3d_b$; this follows from steps 1(a)-1(c) of the timing analysis of Algorithm 2. The sum of such delays at y ($\in Y\_trunk$) is *denoted* t_y. Later, we show how to bound the sum of such delays across multiple M, h and y.

Similar reasoning can be used for M delaying its children in X due to explorations of other children $z \notin X$. Specifically, (1) type *Su* child z of M: $\nexists$ child $z \notin X$. (2) type $L*$ child z of M: $\nexists$ such a child z. If it existed, it would have succeeded in locking M and M would not be master. (3) Each type *Sl* child z contributes delay $4d_z\Delta + 3d_z$; the sum for all z is *denoted* $t_{(M,prev(h))}$ (for $h > 0$). Later, we show how to bound the sum of such delays across multiple M and h.

Let $A = X \cup Y\_branch$. Any $a \in X$ lies on an increasing path to M and any $a \in Y\_branch$ lies on an increasing path to the first encountered node in *Y_trunk*. We analyze the delays introduced by $a \in A$ delaying their children in A due to explorations by their children not in A. A child of a outside A may be of type LL, Lu, Sl, Su. For any $a \in A$, (1) each type *Su* child belongs to A and the delay is already accounted for in *Collapse_Into_Parent* executed by a. (2) each type *Sl* child and type $L*$ child (is not in A and) does not contribute any delay beyond that of *Collapse_Into_Parent* executed by a and already accounted for therein. The type $L*$ child does not succeed in locking $head(a)$ and learns that a is collapsing into its parent. A *Lu* child transfers its unsettled robots to x with no extra time cost. Thus nodes in X (or *Y_branch*) do not contribute additional delays to their children in X (or *Y_branch*) due to explorations of other children not in X (or not in *Y_branch*).

Define a node DFS z to become *passive* if all its robots are settled or it is a *Lu*-child of some i and transfers out its unsettled robots from $head(z)$. Note that a node $i \in C(M, h)$ receives a transfer of unsettled robots of a (blocked) child a of type *Lu* not in $C(M, h)$ (Property 1.4) and these unsettled robots are transitively absorbed/transferred to $(M, prev(h))$ and then to (M, h), while node a goes passive. Such a transfer of unsettled robots from one DFS to another does not cost extra time as it happens in the time it takes for the collapse of i and its subsumption into $(M, prev(h))$. Further, the resulting configuration would have occurred if the initial distribution of robots had been different and the algorithm time bound analysis is valid for any initial configuration.

A passive DFS node i *reactivates* again if unsettled robots arrive at $head(i)$ and resume DFS traversal of i. This (getting reactivated again) is possible only if some *Su*-child DFS j now blocks on i and gets subsumed. If $(i, prev(h))$ reactivates, its parent in the meeting tree must be (i, h) and it does not block nor get subsumed until (i, h) forms.

The algorithm *terminates* when all concurrently existing DFS nodes become *passive*. When this happens, all robots in such DFSs are settled.

Define *idle time* with respect to a node (i, h) in the meeting tree as the period in which all the concurrently existing DFSs that are eventually transitively subsumed by the ancestor node DFS (i, h) are concurrently passive. For a $(child, parent)$ edge

in the meeting tree, idle time is the sub-interval of the idle time of (i, h) within the window (*child* forms, it gets subsumed by *parent*).

Theorem 6. *For any (M, h) in the meeting tree, there is a path from some leaf node to it going through edges along each of which there is no idle time, and subsumption and dispersion that immediately begin after blocking incurs necessary delays.*

Proof. For any (M, h) in the meeting tree, observe from the structure of the meeting graph of nodes in $C(M, h)$ that at most one child $(i, prev(h))$ of (M, h) could have become passive after $(i, prev(h))$ formed and all other children $(j, prev(h))$ are blocked. From the algorithm, observe that:

- Since the time $(j, prev(h))$ formed until it blocked, its DFS was constantly dispersing/growing, and
- since the time such a $(j, prev(h))$ blocked until $C(M, h)$ formed including the time until the collapses and subsumptions occurred, all time was spent in traversals, collapses, subsumptions, and necessary waits as dictated by the Algorithms 2 and 3 and accounted for in the timing analysis of the various procedures. Recall that the necessary waits are bounded as the algorithm was argued to be deadlock-free due to acyclic waits.

From the 2 observations above about the algorithm, it follows that there is no idle time from the time any such $(j, prev(h))$ formed until (M, h) formed. As this is true for any (M, h), the theorem follows. □

Corollary 1. *For any (M, h) in the meeting tree, there is no idle time interval until its formation since the start of the algorithm.*

Observe that the time for any (M, h) to form, i.e., to collapse settled robots and collect such collapsed robots and unsettled robots of the subsumed children $(z, prev(h))$, from the time the first of the $(j, prev(h))$ children blocked, is upper-bounded (due to possibly concurrent executions) by the sum of the times identified in the analysis of the various procedures executed by each $x \in X, y \in Y$, and $(M, prev(h))$. The time by which the first $(j, prev(h))$ child blocked is upper bounded by the time the last $a \in X \cup \{(M, prev(h))\}$ blocked. Note that only those nodes that satisfy the definition of $y \in Y(M, h)$ by the time $(M, prev(h))$ blocks will be in $Y(M, h)$. We now proceed with the detailed time complexity derivation of our algorithm.

Thus far, the size d_i of node i referred to the number of settled robots in it, and is henceforth referred to as d_i^s. More specifically, $d_{i,h}^s$ will refer to the number of settled robots up until just before the $next(h)$ meeting of i. The number of unsettled robots in i up until just before the $next(h)$ meeting of i is referred to as $d_{i,h}^u$. Let $T(M, h)$ denote the time to settle DFS M up until meeting at depth h of the meeting tree, and from then on until the next meeting $(next(h))$ for M. The collapse and collection time to $head(M)$ has components $c(M, h)$ and $g(M, h)$. $c(M, h)$ has a upper bound factor of $(24\Delta + 8)$ for $x \in X$ and $y \in Y$ as derived earlier in this section. $g(M, h)$ represents the sum of wait times introduced by *Sl*-type child nodes of *Y_trunk* nodes $y \in Y(M, h)$ and of $(M, prev(h))$. The time for dispersion/settling after collection and until the $next(h)$ meeting is $s(M, h)$. Thus $s(M, h)$ is the time for the growth phase of the DFS tree. These are defined as follows.

$$c(M,h) = \begin{cases} 0 & \text{if } h = 0 \\ (24\Delta+8)(\sum_{x\in X(M,h)} d_x^s + \sum_{y\in Y(M,h)} d_y^s) & \text{if } h > 0 \\ (+4\Delta(d_{M,prev(h)}^s) \text{ if } X(M,h) \neq \emptyset) & \end{cases} \tag{1}$$

$$s(M,h) = \begin{cases} 4\Delta(d_{M,h}^s - d_{M,prev(h)}^s) & \text{if } next(h) < k' \\ 4\Delta(\sum_{x\in X(M,h)} d_x^s + \sum_{y\in Y(M,h)} d_y^s & \text{otherwise} \\ + \sum_{x\in X(M,h)} d_x^u + \sum_{y\in Y(M,h)} d_y^u & \\ + d_{M,prev(h)}^u) & \end{cases} \tag{2}$$

$$g(M,h) = \begin{cases} 0 & \text{if } h = 0 \\ \sum_{y\in Y(M,h)} t_y + t_{(M,prev(h))} & \text{if } h > 0 \end{cases} \tag{3}$$

This process of collapsing and collecting for instance (M, h) began at the very latest (since the start of the algorithm) at the time at which the latest of the x nodes, x', got blocked. Thus,

$$\begin{aligned} T(M,h) &\leq \overbrace{c(M,h) + s(M,h)}^{f(M,h)} + g(M,h) + T(x', prev(h)), \\ x' &= argmax_{x \mid (x,prev(h)) \in X(M,h) \cup \{(M,prev(h))\}} T(x, prev(h)), \\ & c(*,0) = 0, g(*,0) = 0, s(*,0) = d_{*,0}^s. \end{aligned} \tag{4}$$

We break $T(M,h)$ into two series, and bound them separately. The two series are:

$$\begin{aligned} S1 = \quad & f(M,h) + f(x'(M,h), prev(h)) \\ & + f(x'(x'(M,h), prev(h)), prev(prev(h))) + \cdots + f(*,0) \\ S2 = \quad & g_{(M,h)} + g_{(x'(M,h),prev(h))} + \cdots + (g(*,0) = 0) \\ = \quad & \sum_{y \in Y(M,h)} t_y + \sum_{y \in Y(x'(M,h),prev(h))} t_y + \cdots + (\sum_{y \in Y(*,0)} t_y = 0) \\ & + t_{(M,prev(h))} + t_{(x'(M,h),prev(prev(h)))} + \cdots + (t_{(*,prev(0))} = 0) \end{aligned} \tag{5}$$

Lemma 1. *The sum in the series $S1$ is $O(k\Delta)$.*

Proof. We consider levels of the meeting tree from level 1 upwards to h ($\leq k'-1$). Let η DFS components collapse and merge into one of them, and let the size (i.e., number of settled robots) of each component be d. We consider two extreme cases and show for each that the lemma holds.

1. Case 1: At each level when components collapse and collect in a master component, immediately afterwards (before the collected unsettled robots can settle) the master component meets another component at the next level, and the collapse and collection happen at the next level. Again, immediately afterwards, the (new) master component meets another component at the yet next higher level, and so on till level h. This case assumes $s(i,*) = 0$.

   (a) At level 1, η components of size d each merge into one of size d in $O(\eta d\Delta)$ time, leading to a total of ηd robots in the master component.
   (b) At level 2, η components of size d each merge into one of size d in $O(\eta d\Delta)$ time, leading to a total of $\eta^2 d$ robots in the master component.
   (c) At level h, η components of size d each merge into one of size d in $O(\eta d\Delta)$ time, leading to a total of $\eta^h d$ robots in the master component.

   $\eta^h d$ is at most the maximum number of robots k. Solving $k = \eta^h d$, $h = \log_\eta \frac{k}{d}$. Therefore the maximum total elapsed time until the h-th level meeting and collapse takes place is

   $$\text{Max. elapsed time is } O(h(\eta d\Delta)) = O(\eta d\Delta \log_\eta \frac{k}{d})$$

   This maximum elapsed time is $O(k\Delta)$, considering both extreme cases (a) $\eta d = O(1)$ and (b) $\eta d = O(k)$.
2. Case 2: At each level when components collapse and collect in a master component, the collected robots (almost) fully disperse after which the master component meets another component at the next level, and the collapse and collection happen at the next level. Again, the robots collected by the (new) master component (almost) fully disperse after which the master component meets another component at the yet next higher level, and so on till level h. This case assumes $\forall j, s(i,j)$ satisfies $next(j) \neq k'$.

   (a) At level 1, η components of size d each merge into one of size ηd in $O(\eta d\Delta)$ time, leading to a total of ηd robots in the master component.
   (b) At level 2, η components of size ηd each merge into one of size $\eta^2 d$ in $O(\eta^2 d\Delta)$ time, leading to a total of $\eta^2 d$ robots in the master component.
   (c) At level h, η components of size $\eta^{h-1} d$ each merge into one of size $\eta^h d$ in $O(\eta^h d\Delta)$ time, leading to a total of $\eta^h d$ robots in the master component.

   $\eta^h d$ is at most the maximum number of robots k. Solving $k = \eta^h d$, $h = \log_\eta \frac{k}{d}$. Therefore the maximum total elapsed time until the h-th level meeting and collapse/dispersion takes place is

$$\begin{aligned} O(\Delta(\eta d + \eta^2 d + \eta^3 d + \ldots + \eta^h d)) = \quad & O(\Delta \eta d \frac{\eta^h - 1}{\eta - 1}) \\ = \quad & O(\frac{\Delta \eta d}{\eta - 1}(\eta^{\log_\eta \frac{k}{d}} - 1)) \\ = \quad & O(\frac{\Delta \eta d}{\eta - 1}(\frac{k}{d} - 1)) \\ = \quad & O(k\Delta) \end{aligned}$$

There is also a special case in which a single component M, each time ($\forall h'$), grows and meets other fully dispersed component(s) that collapse (transitively) in to it and no component meets M. Here, $\forall h'$, $X(M,h') = \emptyset$ as all subsumed components belong to $Y(M,h')$ sets. Observe that $\sum_{h'} c(M,h') = \sum_{h'} s(M,h') = O(k\Delta)$.

The lemma follows. □

Lemma 2. *The sum in the series S2 is $O(k\Delta)$.*

Proof. The series $S2$ is the sum of all the waits introduced by children a of a Y_trunk node y and of M, that are of type Sl. Such a Sl child contributes delay up to $4d_a\Delta + d_a$ ($\leq 4d_y\Delta + 3d_y$ or $\leq 4d_M\Delta + 3d_M$, respectively) and then collapses and gets subsumed by the node b that has locked it. Thus Sl type children can occur at most $k' - 1$ times in the lifetime of the execution. Note also that $d_b \geq d_a$ as b to a is a decreasing path.

If all the Sl children were never involved in any meeting until now, then $\sum d_a \leq k$ and the lemma follows. However we need to also analyze the case where a Sl node gets subsumed by another node b, and then the node b becomes a Sl node later. In this case, the robots subsumed from a may be double-counted in the size of b when b later becomes a type Sl node. This can happen at most $k' - 1$ times.

Let η DFS components, including the Sl component, collapse and merge into one of them, and let the size (i.e., number of settled robots) of each component be d. We consider two extreme cases and show for each that the lemma holds.

1. Case 1: When components collapse and are collected, immediately afterwards (before the collected unsettled robots can settle) the master component becomes a Sl-type node, and the collapse and collection happen again. Again, immediately afterwards, the new master component becomes a type Sl node, and so on.

   (a) The first time, η components of size d each merge into one of size d in $O(\eta d\Delta)$ time, leading to a total of ηd robots in the master component.
   (b) The second time, η components of size d each merge into one of size d in $O(\eta d\Delta)$ time, leading to a total of $\eta^2 d$ robots in the new master component.
   (c) The j-th time, η components of size d each merge into one of size d in $O(\eta d\Delta)$ time, leading to a total of $\eta^j d$ robots in the master component.

   $\eta^j d$ is at most the maximum number of robots k. Solving $k = \eta^j d$, $j = \log_\eta \frac{k}{d}$. Therefore the total delay introduced in series $S2$ which is linearly proportional to Δ times the sum of sizes of the type Sl components, is $O(\eta\Delta dj)$.

   $$\text{Sum of delays is } O(\eta\Delta dj) = O(\eta\Delta d \log_\eta \frac{k}{d})$$

   This maximum elapsed time is $O(k\Delta)$, considering both extreme cases (a) $\eta d = O(1)$ and (b) $\eta d = O(k)$.
2. Case 2: When components collapse and are collected, the collected robots (almost) fully disperse after which the master component becomes a type Sl node, and the collapse and collection happen again. Again, the collected robots in the new master component (almost) fully disperse after which the (new) master component becomes a type Sl node and collapses and gets collected, and so on.

   (a) The first time, η components of size d each merge and settle into one of size ηd in $O(\eta d\Delta)$ time, leading to a total of ηd robots in the master component.
   (b) The second time, η components of size ηd each merge and settle into one of size $\eta^2 d$ in $O(\eta^2 d\Delta)$ time, leading to a total of $\eta^2 d$ robots in the master component.
   (c) The j-th time, η components of size $\eta^{j-1} d$ each merge and settle into one of size $\eta^j d$ in $O(\eta^j d\Delta)$ time, leading to a total of $\eta^j d$ robots in the master component.

   $\eta^j d$ is at most the maximum number of robots k. Solving $k = \eta^j d$, $j = \log_\eta \frac{k}{d}$. Therefore the total delay introduced in series $S2$ which is linearly proportional to Δ times the sum of sizes of the type Sl components, is

   $$\begin{aligned} O(\Delta(\eta d + \eta^2 d + \eta^3 d + \ldots + \eta^j d)) &= O(\Delta\eta d \frac{\eta^h - 1}{\eta - 1}) \\ &= O(\frac{\Delta\eta d}{\eta - 1}(\eta^{\log_\eta \frac{k}{d}} - 1)) \\ &= O(\frac{\Delta\eta d}{\eta - 1}(\frac{k}{d} - 1)) \\ &= O(k\Delta) \end{aligned}$$

The lemma follows. □

Theorem 7. *Algorithm Exploration (Algorithm 2) in conjunction with Algorithm $DFS(k)$ correctly solves* Dispersion *for $k \leq n$ robots initially positioned arbitrarily on the nodes of an arbitrary anonymous graph G of n memory-less nodes, m edges, and degree Δ in $O(\min\{m, k\Delta\})$ rounds using $O(\log(k+\Delta))$ bits at each robot.*

Proof. $T(M,h)$ is the sum of the series $S1$ and $S2$ which are both $O(k\Delta)$ by Lemmas 1 and 2. So the time till termination of the Algorithms 1 ($DFS(k)$), 2 (*Exploration*), and Algorithm 3 (*various procedures invoked*) is $O(k\Delta)$. As $k \leq n$, this is $O(n\Delta)$. Now observe that in our derivations (Lemmas 1 and 2), the Δ factor is an overestimate. The actual upper bound is $O(\sum_{i=1}^{n} \delta_i)$ which is $O(m)$, the number of edges in the graph. This upper bound is better when $m < k\Delta$ and hence the time complexity is $O(\min\{m, k\Delta\})$.

The highest level node (i,h) in each tree in the final forest of the meeting graph represents a master node that has never been subsumed and always alternated between growing and subsuming other components, and growing again. The growth happens as per Algorithm 1 ($DFS(k)$) which correctly solves Dispersion by Theorem 3. Whereas the subsuming of other components merely collects the robots of the other components to the head node $head(i)$ (Algorithm *Exploration*) which subsequently get dispersed by the growing phases (Algorithm $DFS(k)$). Hence, Dispersion is achieved.

The *retrace* and *collapse* variable at each robot used in Algorithm 2 and 3 are $O(\log \Delta)$. *collapsing_children* takes $O(\log k)$ bits and a single bit each is required to track whether the component is locked and whether it is collapsing. The space requirement of Algorithm 1 was shown in Theorem 3 to be $(\log(k+\Delta))$ bits. The theorem follows. □

Proof of Theorem 1. Follows from Theorem 7. □

Proof of Theorem 2. In the asynchronous setting, in every CCM cycle, each robot at a node u determines x, the number of co-located robots, if any, that should be moving with it to node v. It then moves as per its own schedule. On arriving at v, it does not start its next CCM cycle until x robots have arrived from u. This essentially constitutes one epoch and ensures that the robots that move together in a round in a synchronous setting move together in one epoch in the asynchronous setting. With this simple modification, the algorithm given for the synchronous setting works for the asynchronous setting. The space and time complexities, as given in Theorem 1, carry over to the asynchronous setting. □

6. Concluding remarks

In this paper, we have presented a deterministic algorithm that solves Dispersion, starting from any initial configuration of $k \leq n$ robots positioned on the nodes of an arbitrary anonymous graph G having n memory-less nodes, m edges, and degree Δ, in time $O(\min\{m, k\Delta\})$ with $\Theta(\log(k+\Delta))$ bits at each robot. Memory is optimal for any degree Δ and the time also becomes optimal if the graph has constant degree, i.e., $\Delta = O(1)$, the first simultaneously optimal result for arbitrary graphs. For any degree Δ, time is optimal within a $O(\Delta)$ factor since there exists a time lower bound of $\Omega(k)$ [3]. This algorithm improves the time bound established in the best previously known results [7,8] by an $O(\log \ell)$ factor and matches asymptotically the time and memory bound of the single-source DFS traversal. This algorithm uses a non-trivial approach of subsuming parallel DFS traversals into single one based on their DFS tree sizes, limiting the subsumption process overhead to the time proportional to the time needed in the single-source DFS traversal. This approach might be of independent interest.

For future work, it will be interesting to improve the existing time lower bound of $\Omega(k)$ to $\Omega(\min\{m, k\Delta\})$ or improve the time bound to $O(k)$ removing the $O(\Delta)$ factor. The second interesting direction will be to consider faulty (crash and/or Byzantine) robots.

CRediT authorship contribution statement

Ajay D. Kshemkalyani: Writing – review & editing, Writing – original draft, Methodology, Formal analysis. **Gokarna Sharma:** Writing – review & editing, Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

No data was used for the research described in the article.

References

[1] A.D. Kshemkalyani, G. Sharma, Near-optimal dispersion on arbitrary anonymous graphs, in: 25th International Conference on Principles of Distributed Systems, OPODIS, 2021, 8.
[2] J. Augustine, W.K.M. Jr., Dispersion of mobile robots: a study of memory-time trade-offs, in: ICDCN, 2018, pp. 1–10.
[3] A.D. Kshemkalyani, F. Ali, Efficient dispersion of mobile robots on graphs, in: ICDCN, 2019, pp. 218–227.
[4] A.D. Kshemkalyani, A.R. Molla, G. Sharma, Efficient dispersion of mobile robots on dynamic graphs, in: ICDCS, 2020, pp. 732–742.
[5] P. Flocchini, G. Prencipe, N. Santoro, Distributed Computing by Oblivious Mobile Robots, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2012.
[6] P. Flocchini, G. Prencipe, N. Santoro, Distributed Computing by Mobile Entities, Theoretical Computer Science and General Issues, vol. 1, Springer International Publishing, 2019.
[7] A.D. Kshemkalyani, A.R. Molla, G. Sharma, Fast dispersion of mobile robots on arbitrary graphs, in: ALGOSENSORS, 2019, pp. 23–40.
[8] T. Shintaku, Y. Sudo, H. Kakugawa, T. Masuzawa, Efficient dispersion of mobile agents without global knowledge, in: SSS, 2020, pp. 280–294.
[9] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, The MIT Press, 2009.
[10] A.D. Kshemkalyani, A.R. Molla, G. Sharma, Dispersion of mobile robots on grids, in: WALCOM, 2020, pp. 183–197.
[11] A.R. Molla, W.K.M. Jr., Dispersion of mobile robots: the power of randomness, in: TAMC, 2019, pp. 481–500.
[12] A. Das, K. Bose, B. Sau, Memory optimal dispersion by anonymous mobile robots, in: CALDAM, 2021, pp. 426–439.
[13] A.D. Kshemkalyani, A.R. Molla, G. Sharma, Dispersion of mobile robots in the global communication model, in: ICDCN, 2020, 12.
[14] D. Pattanayak, G. Sharma, P.S. Mandal, Dispersion of mobile robots tolerating faults, in: ICDCN, 2021, pp. 133–138.
[15] A.R. Molla, K. Mondal, W.K. Moses Jr., Efficient dispersion on an anonymous ring in the presence of weak Byzantine robots, in: ALGOSENSORS, 2020, pp. 154–169.
[16] A.R. Molla, K. Mondal, W.K. Moses Jr., Byzantine dispersion on graphs, in: IPDPS, 2021, pp. 1–10.
[17] G.F. Italiano, D. Pattanayak, G. Sharma, Dispersion of mobile robots on directed anonymous graphs, in: 29th International Colloquium on Structural Information and Communication Complexity SIROCCO, 2022, 11.
[18] E. Bampas, L. Gasieniec, N. Hanusse, D. Ilcinkas, R. Klasing, A. Kosowski, Euler tour lock-in problem in the rotor-router model: I choose pointers and you choose port numbers, in: DISC, 2009, pp. 423–435.
[19] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, D. Peleg, Label-guided graph exploration by a finite automaton, ACM Trans. Algorithms 4 (4) (2008) 42.
[20] D. Dereniowski, Y. Disser, A. Kosowski, D. Pajak, P. Uznański, Fast collaborative graph exploration, Inf. Comput. 243 (C) (2015) 37–49.
[21] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, D. Peleg, Graph exploration by a finite automaton, Theor. Comput. Sci. 345 (2–3) (2005) 331–344.
[22] P. Fraigniaud, L. Gasieniec, D.R. Kowalski, A. Pelc, Collective tree exploration, Networks 48 (3) (2006) 166–177.
[23] A.D. Kshemkalyani, F. Ali, Fast graph exploration by a mobile robot, in: First IEEE International Conference on Artificial Intelligence and Knowledge Engineering, AIKE, 2018, pp. 115–118.
[24] A. Menc, D. Pajak, P. Uznanski, Time and space optimality of rotor-router graph exploration, Inf. Process. Lett. 127 (2017) 17–20.
[25] A. Dessmark, P. Fraigniaud, D.R. Kowalski, A. Pelc, Deterministic rendezvous in graphs, Algorithmica 46 (1) (2006) 69–96.
[26] D.R. Kowalski, A. Malinowski, How to meet in anonymous network, Theor. Comput. Sci. 399 (1–2) (2008) 141–156.
[27] Y. Elor, A.M. Bruckstein, Uniform multi-agent deployment on a ring, Theor. Comput. Sci. 412 (8–10) (2011) 783–795.
[28] M. Shibata, T. Mega, F. Ooshita, H. Kakugawa, T. Masuzawa, Uniform deployment of mobile agents in asynchronous rings, in: PODC, 2016, pp. 415–424.
[29] L. Barriere, P. Flocchini, E. Mesa-Barrameda, N. Santoro, Uniform scattering of autonomous mobile robots in a grid, in: IPDPS, 2009, pp. 1–8.
[30] P. Poudel, G. Sharma, Time-optimal uniform scattering in a grid, in: ICDCN, 2019, pp. 228–237.
[31] P. Poudel, G. Sharma, Fast uniform scattering on a grid for asynchronous oblivious robots, in: SSS, 2020, pp. 211–228.
[32] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, J. Parallel Distrib. Comput. 7 (2) (1989) 279–301.
[33] R. Subramanian, I.D. Scherson, An analysis of diffusive load-balancing, in: SPAA, 1994, pp. 220–225.
[34] A. Cord-Landwehr, B. Degener, M. Fischer, M. Hüllmann, B. Kempkes, A. Klaas, P. Kling, S. Kurras, M. Märtens, F. Meyer auf der Heide, C. Raupach, K. Swierkot, D. Warner, C. Weddemann, D. Wonisch, A new approach for analyzing convergence algorithms for mobile robots, in: ICALP, 2011, pp. 650–661.