Near-Optimal Dispersion on Arbitrary Anonymous Graphs

Ajay D. Kshemkalyani
University of Illinois at Chicago, IL, USA

Gokarna Sharma
Kent State University, OH, USA

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 one or more nodes of the graph to exactly k different nodes of the graph, one on each node. The objective is to simultaneously minimize time to achieve dispersion and memory requirement at each robot. If all k robots are positioned initially on a single node, depth first search (DFS) traversal solves this problem in $O(\min\{m, k\Delta\})$ time with $\Theta(\log(k+\Delta))$ bits at each robot. However, if robots are positioned initially on multiple nodes, the best previously known algorithm solves this problem in $O(\min\{m, k\Delta\} \cdot \log \ell)$ time storing $\Theta(\log(k+\Delta))$ bits at each robot, where $\ell \leq k/2$ is the number of multiplicity nodes 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 $\Theta(\log(k+\Delta))$ bits at each robot, improving the time bound of the best previously known algorithm by $O(\log \ell)$ and matching asymptotically the single-source DFS traversal bounds. This is the first algorithm for dispersion that is optimal in both time and memory in arbitrary anonymous graphs of constant degree, $\Delta = O(1)$. Furthermore, the result holds in both synchronous and asynchronous settings.

2012 ACM Subject Classification Mathematics of computing → Graph algorithms; Computing methodologies → Distributed algorithms; Computer systems organization → Robotics

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

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2021.8

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 [1, 18]. 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 [1, 18, 22].

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 [12, 13]. In this paper, we study trade-off between time and memory complexities to solve Dispersion 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 (for example, compare memory and time bounds of the two algorithms from [18] given in Table 1).

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. †[19] assumes m, k, and Δ are known to the algorithm a priori. $\ell \leq k/2$ is the number of multiplicity nodes in the initial configuration; DISPERSION is already solved if there is no multiplicity node.

Algorithm	Memory/robot (in bits)	Time (in rounds/epochs)	Single-source/ Multi-source	Setting
Lower bound	$\Omega(\log(k+\Delta))$	$\Omega(k)$	any	Asynchronous
DFS	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	Single-source	Asynchronous
[18]	$O(k \log \Delta)$	$O(\min\{m, k\Delta\})$	Multi-source	Asynchronous
[18]	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot \ell)$	Multi-source	Asynchronous
[19]†	$O(\log n)$	$O(\min\{m, k\Delta\} \cdot \log \ell)^{\dagger}$	Multi-source	Synchronous
[31]	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot \log \ell)$	Multi-source	Synchronous
Th. 1	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	Multi-source	Synchronous
Th. 2	$\Theta(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	Multi-source	Asynchronous

Recent studies [19, 31] 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 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 [19, 31] were successful in keeping memory bound optimal as in [18] and reduce time bound to $O(\min\{m, k\Delta\} \cdot \log \ell)$, an improvement of $\ell/\log \ell$ factor compared to the $O(\min\{m, k\Delta\} \cdot \ell)$ time bound of [18], where $\ell \leq k/2$ is the number of multiplicity nodes in the initial configuration.

In this paper, we present a new algorithm for DISPERSION that settles the question completely, i.e., it obtains the time bound of $O(\min\{m, k\Delta\})$ keeping memory optimal $\Theta(\log(k+\Delta))$ bits at each robot, first such result for the multi-source case. The time bound is an improvement of $O(\log \ell)$ factor compared to the best previously known algorithms [19, 31]. Furthermore, the time and memory bounds match the respective bounds for the single-source case. Thus, the proposed algorithm is the first for DISPERSION that is simultaneously optimal for arbitrary anonymous graphs of constant degree $\Delta = O(1)$.

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 $[1, k]$. In contrast to graph nodes which are memory-less, the robots have memory to store information (otherwise the problem becomes unsolvable). Finally, 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 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. In this paper, we establish the following theorem in the *synchronous* setting where all robots are activated in a round, they perform their operations simultaneously in synchronized rounds, and hence the time (of the algorithm) is measured in rounds (or steps).

▶ **Theorem 1.** *Given any initial configuration of $k \leq n$ mobile robots 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 [19, 31] by a factor of $O(\log \ell)$ keeping the memory optimal, where ℓ is the number of nodes in the initial configuration with at least two robots co-located on them. 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, for constant-degree arbitrary anonymous graphs, i.e., $\Delta = O(1)$, our algorithm is asymptotically optimal w.r.t. both time and memory, first such result for Dispersion (Table 1).

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

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 [6] storing $O(\log(k + \Delta))$ bits at each robot. The k-source Dispersion finishes in a single round, since k robots are already on k different nodes solving Dispersion. Therefore, the challenging case is k'-source Dispersion with $1 < k' < k$.

The early papers obtained better bounds on either time or memory, trading one for another. The first algorithm of [18] obtained $O(\min\{m, k\Delta\})$ time bound with memory $O(k \log \Delta)$ bits at each robot. The second algorithm of [18] kept memory optimal $O(\log(k + \Delta))$ bits at each robot and established time $O(\min\{m, k\Delta\} \cdot \ell)$, where $\ell \leq k' < k$ is the number of multiplicity nodes in the initial configuration. Their algorithm starts ℓ different single-source DFS traversals in parallel from ℓ sources with multiple robots on them. 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 of 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 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 [19, 31] reduced the $O(\ell)$ factor in the time bound to $O(\log \ell)$. Providing m, k, and Δ parameters to the algorithm beforehand, Kshemkalyani et al. [19] run ℓ-source DFS traversals in passes of interval $O(\min\{m, k\Delta\})$ rounds. After each pass, they guaranteed that the ℓ-source DFS traversal reduces to $\ell/2$-source DFS traversal. Therefore, in total $\lceil \log \ell \rceil$ passes, the ℓ-source DFS traversal reduces to a single-source DFS traversal, which then finishes in additional $O(\min\{m, k\Delta\})$ rounds, giving in the worst-case, $O(\min\{m, k\Delta\} \cdot \log \ell)$ rounds time bound. 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 $\Delta < n$. Recently,

Shintaku et al. [31] established the same time bound as in [19] avoiding the requirement for the algorithm to know m, k, Δ beforehand. Moreover, they improved the memory bound $O(\log n)$ bits in [19] to optimal $\Theta(\log(k + \Delta))$ bits at each robot.

Observing the techniques of [19, 31], 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 [18]. The implication of the pairwise subsumption is 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.

Despite these benefits, the pairwise subsumption is not matching 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 [19, 31] 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.

Techniques. We use parallel multi-source DFS traversals as in [19, 31] but devise a novel subsumption technique, leading to $O(\min\{m, k\Delta\})$ time with $O(\log(k + \Delta))$ bits at each robot, removing the $O(\log \ell)$ factor from the time bound of the best previously known algorithms [19, 31] and matching the time and memory bounds for 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 and merge of the subsumed traversal to the subsuming one can be done in time proportional to the size of the subsumed traversal, and (ii) it avoids the need of revisiting the nodes of the subsumed traversal more than once, a crucial aspect in removing the $O(\log \ell)$ factor from the time bound. Furthermore, one traversal always remains subsuming throughout the execution of the algorithm.

This is in contrast to the technique used in the best previously known algorithms [19, 31] 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 the repeating traversal of the already built DFS tree, adding a $\Theta(\log \ell)$ 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 4). 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 last robot belonging to that DFS traversal has settled. Therefore, it requires for the meeting traversal to traverse the met DFS tree to reach its head node to find its size. 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 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 number of bits per robot.

Related Work. Augustine and Moses Jr. [1] 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.

Kshemkalyani and Ali [18] provided an $\Omega(k)$ time lower bound for arbitrary graphs for $k \leq n$. They then provided three deterministic algorithms for DISPERSION in 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 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. Kshemkalyani et al. [19] provided an algorithm for arbitrary graph with $O(\min\{m, k\Delta\} \cdot \log \ell)$ time using $O(\log n)$ bits memory at each robot, with the algorithm knowing m, k, Δ beforehand. The same time bound and improved memory bound of $O(\log(k+\Delta))$ bits were obtained in [31], without the need of the algorithm knowing m, k, Δ beforehand. For grid graphs, Kshemkalyani et al. [21] provided an algorithm that runs in $O(\min\{k, \sqrt{n}\})$ time using $O(\log k)$ bits memory at each robot. Randomized algorithms were presented in [24, 8] mainly to reduce the memory requirement at each robot.

Recently, Kshemkalyani et al. [20] 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 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 model, it is not possible to extend the idea that is developed for the global model. For grid graphs, Kshemkalyani et al. [21] provided a $O(\sqrt{k})$ time algorithm with $O(\log k)$ bits at each robot in the global model.

DISPERSION in anonymous dynamic (undirected) graphs was considered in [22] where the authors provided some impossibility, lower, and upper bound results. Dispersion under crash faults was considered in [27] and under Byzantine faults was considered in [25, 26] establishing a spectrum of interesting results.

The related problem of exploration has been quite heavily studied in the literature for specific as well as arbitrary graphs, e.g., [2, 4, 9, 15, 14, 17, 23]. It was shown that a robot can explore an anonymous graph using $\Theta(D \log \Delta)$-bits memory; the runtime of the algorithm is $O(\Delta^{D+1})$ [15]. In the model where graph nodes also have memory, Cohen et al. [4] 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 [23]. The collective exploration by a team of robots is studied in [14] for trees. The dual of the DISPERSION problem is gathering, which has been extensively studied, e.g., [10, 16]. Another problem related to DISPERSION is the scattering of k robots on graphs. This problem has been mainly studied for rings [11, 30] and grids [3]. Recently, Poudel and Sharma [28, 29] 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 [7]. We refer readers to [12, 13] for other recent developments in these topics.

Roadmap. We discuss model details in Section 2. We discuss the single-source DFS traversal in Section 3. We then present our (synchronous) multi-source DFS traversal algorithm in Section 4. We prove the correctness, time, and memory complexity of our algorithm in Section 5 (i.e., Theorem 1). We then discuss the extensions to the asynchronous setting, proving 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.

Each robot has a unique $\lceil \log k \rceil$-bit ID taken from the range $[1, k]$. 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 time duration of 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 robots do not experience fault.

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 even 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. 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 to 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 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 settings 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 [5]. 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. Another important parameter is memory – the number of bits stored in persistent memory at each robot.

3 DFS traversal of a Graph (Algorithm *DFS*(k))

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

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 r_i 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* (initally 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* [6]; (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** ([19]). *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.*

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
9              else
10                 child ← (child + 1) mod δ, r.child ← child
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
```

4 The Algorithm

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. 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. 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. 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 $head(j)$ 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 have yet to return to $head(j)$). 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 $\min\{d_i, d_j\}$.

Algorithm 2 Algorithm *Exploration* to explore $parent(i)$ component on reaching junction $head(i)$ 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))
10     traverse parent(i) informing each node (a) that parent(i) is locked and will be
         collapsing, and also (b) value of d_parent(i), and return to head(parent(i))
11     wait until parent(i)'s explorers return from parent(parent(i))
12     follow action (Collapse_Into_Child(parent(i), i)) which will be determined on their
         return (if head(parent(i)) is not junction, execute Collapse_Into_Child(parent(i), i))
13 else if exploring robots find parent(i) is collapsing or learn that parent(i) is locked and will
     be collapsing then
14     Parent_Is_Collapsing
15 else if explorers E's path meets another explorers F's path then
16     wait until F return
17     if parent(i) is collapsing then
18         Parent_Is_Collapsing
19     else if parent(i) is not collapsing then
20         continue E's exploration
```

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 the DFS l it 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 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 4** (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.*

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

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

Collapse_Into_Child(i,j)
explorers of i go from $head(i)$ 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 $head(j)$
collapsed robots change ID *treelabel* to 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 $head(i)$
robot at $head(i)$ decrements $collapsing\_children$
collapsed robots change ID *treelabel* to $parent(i)$
explorers and collapsed robots go to $head(parent(i))$ by following *child* pointers
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)
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**
    $Collapse\_Into\_Child(i, j)$

focus on a single connected component of G' by default; other connected components are dealt with similarly. The algorithm implicitly 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. In this process, the at most one cycle in any connected component of G' is also broken. 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*).

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

```
1  master(i)
2  if d_parent(i) > d_i then
3  |  t1 ← time when explorers of i return to head(i) from parent(i)
4  |  t2 (initialized to ∞) ← the time, if any, when first child j locks head(i)
5  |  if t1 < t2 then  w ← parent(i)
6  |  else if t1 > t2 then  w ← j
7  |  return(master(w))
8  else
9  |  if ∃ a first child j to lock head(i) then  return(master(j))
10 |  else  return(i)
```

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.

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 component $C(M) = X(M) \cup Y(M) \cup \{M\}$.

A component $C(M)$ 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 a increasing path to a Y_trunk node – such nodes are termed Y_branch nodes. 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 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 along Y_trunk and then into the master node by executing $Collapse\_Into\_Child$.

After nodes in $C(M)$ get subsumed in M, the master node grows 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.

▶ **Definition 5** (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 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 6** (Component $C(M,h)$)**.**

1. $X(M,h)$ *is the set of child nodes in the meeting tree that directly and transitively participate only in* $Collapse\_Into\_Parent$ *until collapsing into* (M,h).
2. $Y(M,h)$ *is the set of child nodes in the meeting tree that directly and transitively participate in at least one* $Collapse\_Into\_Child$ *until collapsing into* (M,h).
3. $C(M,h) = X(M,h) \cup Y(M,h) \cup \{(M, prev(h))\}$, *where for any* $z \in C(M,h)$, $z = (a, prev(h))$ *and* $prev(h)$ *is defined as the highest value less than* h *for which node* $(a, prev(h))$ *has been created.*

For any node (i,h), we also define $next(h)$ as the value h' such that $(i,h) \in C(M,h')$ for some M. If such a h' does not exist, we define it 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*. This can be achieved by going to $root(i)$ and doing a (new) DFS traversal of only those nodes (using a duplicate set of variables *state* and *parent* for DFS); if you reach a node which has no settled robot or a settled robot having a different *treelabel*, one simply backtracks along that edge. Such a DFS traversal occurs in (i) Algorithm *Exploration* when $d_i > d_{parent(i)}$ and i locks $head(parent(i))$ junction, (ii) procedure $Collapse\_Into\_Child$, and (iii) procedure $Collapse\_Into\_Parent$, and can be executed in $4\Delta d_i$ steps. In (ii) and (iii), a settled robot not on the collect path gets unsettled and gets collected in the DFS traversal to the collect path when the DFS backtracks from the node where the robot was settled.

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.)

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 $Y\_trunk \cup \{M\}$. 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 F return. If they learn that the y is collapsing, they retrace to their *head* nodes else if they learn y is not collapsing, they continue their exploration towards $head(y)$ but may be blocked again if their path meets another explorers' path. 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 who locks y does so before such $L*$-type children try to explore y and try to lock y. Such $L*$-type children learn that y is collapsing.
3. Each Sl-type child node b contributes delay $4d_b\Delta + 3d_b$. The sum of such delays at y is denoted $t_{y(M,h)}$. 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 $4d_z\Delta + 3d_z$, whose sum for all z is denoted by $t_{(M,prev(h))}$. Later, we show how to bound the sum of such delays across multiple M and h.

Note that for any $x \in X$, (1) each type Su child belongs to X and the delay is already accounted for in *Collapse_Into_Parent* executed by x. (2) each type Sl child and type $L*$ child does not contribute any delay beyond that of *Collapse_Into_Parent* executed by x and already accounted for. (The type $L*$ child does not succeed in locking $head(x)$ and learns that x is collapsing into its parent.)

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 above. The time for dispersion/settling after collection and until the $next(h)$ meeting is $s(M,h)$. These are defined as follows.

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

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

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

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,

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

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 7.** *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 $next(j) \not< k'$.
   - **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

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

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

Proof is deferred to Appendix due to space constraints.

▶ **Theorem 9.** *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 7 and 8. So the time till termination of the Algorithms 1 (DFS), 2 ($Exploration$), and Algorithm 3 (*various procedues invoked*) is $O(k\Delta)$. As $k \leq n$, this is $O(n\Delta)$. Now observe that in our derivations (Lemmas 7 and 8), 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) 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). 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 9. ◀

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 $O(\log(k + \Delta))$ bits at each robot. This is the first algorithm that is simultaneously optimal w.r.t. both time and memory in arbitrary anonymous graphs of constant degree, i.e., $\Delta = O(1)$. This algorithm improves the time bound established in the best previously known results [19, 31] 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.

References

1. John Augustine and William K. Moses Jr. Dispersion of mobile robots: A study of memory-time trade-offs. In *ICDCN*, pages 1:1–1:10, 2018.
2. Evangelos Bampas, Leszek Gasieniec, Nicolas Hanusse, David Ilcinkas, Ralf Klasing, and Adrian Kosowski. Euler tour lock-in problem in the rotor-router model: I choose pointers and you choose port numbers. In *DISC*, pages 423–435, 2009.
3. L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro. Uniform scattering of autonomous mobile robots in a grid. In *IPDPS*, pages 1–8, 2009.
4. Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms*, 4(4):42:1–42:18, August 2008.
5. Andreas Cord-Landwehr, Bastian Degener, Matthias Fischer, Martina Hüllmann, Barbara Kempkes, Alexander Klaas, Peter Kling, Sven Kurras, Marcus Märtens, Friedhelm Meyer auf der Heide, Christoph Raupach, Kamil Swierkot, Daniel Warner, Christoph Weddemann, and Daniel Wonisch. A new approach for analyzing convergence algorithms for mobile robots. In *ICALP*, pages 650–661, 2011.
6. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
7. G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7(2):279–301, October 1989.
8. Archak Das, Kaustav Bose, and Buddhadeb Sau. Memory optimal dispersion by anonymous mobile robots. In *CALDAM*, pages 426–439, 2021.
9. Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pajak, and Przemyslaw Uznański. Fast collaborative graph exploration. *Inf. Comput.*, 243(C):37–49, August 2015.
10. Anders Dessmark, Pierre Fraigniaud, Dariusz R. Kowalski, and Andrzej Pelc. Deterministic rendezvous in graphs. *Algorithmica*, 46(1):69–96, 2006. doi:10.1007/s00453-006-0074-2.
11. Yotam Elor and Alfred M. Bruckstein. Uniform multi-agent deployment on a ring. *Theor. Comput. Sci.*, 412(8-10):783–795, 2011.
12. Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Oblivious Mobile Robots*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. doi:10.2200/S00440ED1V01Y201208DCT010.
13. Paola Flocchini, Giuseppe Prencipe, and Nicola Santoro. *Distributed Computing by Mobile Entities*, volume 1 of *Theoretical Computer Science and General Issues*. Springer International Publishing, 2019.
14. Pierre Fraigniaud, Leszek Gasieniec, Dariusz R. Kowalski, and Andrzej Pelc. Collective tree exploration. *Networks*, 48(3):166–177, 2006.
15. Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. Graph exploration by a finite automaton. *Theor. Comput. Sci.*, 345(2-3):331–344, November 2005.
16. Dariusz R. Kowalski and Adam Malinowski. How to meet in anonymous network. *Theor. Comput. Sci.*, 399(1-2):141–156, 2008. doi:10.1016/j.tcs.2008.02.010.
17. Ajay D. Kshemkalyani and Faizan Ali. Fast graph exploration by a mobile robot. In *First IEEE International Conference on Artificial Intelligence and Knowledge Engineering, AIKE*, pages 115–118, 2018. doi:10.1109/AIKE.2018.00025.
18. Ajay D. Kshemkalyani and Faizan Ali. Efficient dispersion of mobile robots on graphs. In *ICDCN*, pages 218–227, 2019.
19. Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Fast dispersion of mobile robots on arbitrary graphs. In *ALGOSENSORS*, pages 23–40, 2019.

20 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots in the global communication model. In *ICDCN*, pages 12:1–12:10, 2020.

21 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Dispersion of mobile robots on grids. In *WALCOM*, pages 183–197, 2020.

22 Ajay D. Kshemkalyani, Anisur Rahaman Molla, and Gokarna Sharma. Efficient dispersion of mobile robots on dynamic graphs. In *ICDCS*, pages 732–742, 2020.

23 Artur Menc, Dominik Pajak, and Przemyslaw Uznanski. Time and space optimality of rotor-router graph exploration. *Inf. Process. Lett.*, 127:17–20, 2017.

24 Anisur Rahaman Molla and William K. Moses Jr. Dispersion of mobile robots: The power of randomness. In *TAMC*, pages 481–500, 2019.

25 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Efficient dispersion on an anonymous ring in the presence of weak byzantine robots. In *ALGOSENSORS*, pages 154–169, 2020.

26 Anisur Rahaman Molla, Kaushik Mondal, and William K. Moses Jr. Byzantine dispersion on graphs. In *IPDPS*, pages 1–10, 2021.

27 Debasish Pattanayak, Gokarna Sharma, and Partha Sarathi Mandal. Dispersion of mobile robots tolerating faults. In *ICDCN*, pages 133–138, 2021.

28 Pavan Poudel and Gokarna Sharma. Time-optimal uniform scattering in a grid. In *ICDCN*, pages 228–237, 2019.

29 Pavan Poudel and Gokarna Sharma. Fast uniform scattering on a grid for asynchronous oblivious robots. In *SSS*, pages 211–228, 2020.

30 Masahiro Shibata, Toshiya Mega, Fukuhito Ooshita, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Uniform deployment of mobile agents in asynchronous rings. In *PODC*, pages 415–424, 2016.

31 Takahiro Shintaku, Yuichi Sudo, Hirotsugu Kakugawa, and Toshimitsu Masuzawa. Efficient dispersion of mobile agents without global knowledge. In *SSS*, pages 280–294, 2020.

A Appendix

Proof of Lemma 8. 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. ◀