Theoretical Computer Science 1044 (2025) 115272

Contents lists available at ScienceDirect

Theoretical Computer Science

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

Dispersion of mobile robots on graphs in the asynchronous model

Ajay D. Kshemkalyani

Department of Computer Science, University of Illinois Chicago, Chicago, IL 60607, USA

ARTICLE INFO

Section Editor: Pinyan Lu

Handling Editor: Jialin Zhang

Keywords:
Distributed algorithm
Dispersion
Graph algorithm
Graph exploration
Mobile robot

ABSTRACT

The dispersion problem on graphs requires k robots placed arbitrarily at the n nodes of an anonymous graph, where $k \leq n$, to coordinate with each other to reach a final configuration in which each robot is at a distinct node of the graph. The dispersion problem is important due to its relationship to graph exploration by mobile robots, scattering on a graph, and load balancing on a graph. Prior work on solving dispersion assumed the synchronous model. We propose four algorithms to solve dispersion on graphs in the asynchronous model. The first two algorithms require $O(k \log \Delta)$ bits at each robot and $O(\min(m, k\Delta))$ steps running time, where m is the number of edges and Δ is the maximum degree of the graph. The algorithms differ in what, where, and how data structures are maintained. The third algorithm has a space usage of $O(\max(\min(D, k) \cdot \log \Delta, \log D))$ bits at each robot and uses $O(\Delta^{\min(D,k)+1})$ steps, where D is the graph diameter. The fourth algorithm has a space usage of $O(\max(\log k, \log \Delta))$ bits at each robot and uses $O(\min(m, k\Delta) \cdot k)$ steps. In contrast with existing works which all assume the synchronous model, these are the first algorithms to solve dispersion in the weaker but more realistic asynchronous model.

1. Introduction

1.1. Background and motivation

The problem of dispersion of mobile robots, which requires the robots to spread out evenly in a region, has been explored in the literature [1,2]. The dispersion problem on graphs, formulated by Augustine and Moses Jr. [3], requires k robots placed arbitrarily at the n nodes of an anonymous graph, where $k \leq n$, to coordinate with each other to reach a final configuration in which each robot is at a distinct node of the graph. This problem has various applications; for example, an intrinsic application of dispersion has been shown to be the relocation of self-driven electric cars (robots) to recharge stations (nodes) [3]. Recharging is a time-consuming process and it is better to search for a vacant recharge station than to wait. In general, the problem is applicable whenever we want to minimize the total cost of k agents sharing n resources, located at various places, subject to the constraint that the cost of moving an agent to a different resource is much smaller than the cost of multiple agents sharing a resource.

The dispersion problem is also important due to its relationship to graph exploration by mobile robots, scattering on a graph, and load balancing on a graph. These are fundamental problems that have been well-studied by varying the system model and assumptions [4,5]. Although some works consider these problems in general graphs, others consider specific graphs like grids, trees, and rings.

E-mail address: ajay@uic.edu.
URL: https://www.cs.uic.edu/~ajayk.

https://doi.org/10.1016/j.tcs.2025.115272
Received 6 September 2019; Received in revised form 16 April 2025; Accepted 20 April 2025

Available online 23 April 2025

Table 1
Comparison of the proposed algorithms for dispersion on graphs in the asynchronous model. No knowledge of graph parameters m, n, Δ, and D is required.

Algorithm	Bit Complexity at each robot	Time Complexity	Help others after docking	Terminate after docking
Helping-Async	$O(k \log \Delta)$	$O(\min(m, k\Delta))$ steps	Yes	No
Independent-Async	$O(k \log \Delta)$	$O(\min(m, k\Delta))$ steps	No	No
Independent-Bounded-Async	$O(\max(\min(D, k) \cdot \log \Delta, \log D))$	$O(\Delta^{\min(D,k)+1})$ steps	No	Yes
Tree-Switching-Async	$O(\max(\log k, \log \Delta))$	$O(\min(m, k\Delta) \cdot k)$ steps	Yes	No

1.2. Our results

Our results assume that robots have no visibility and can only communicate with other robots present at the same node as themselves. The robots are deterministic, and are distinguishable. The undirected graph, with m edges, n nodes, diameter D, and maximum degree Δ, is anonymous, i.e., nodes have no labels. Nodes also do not have any memory but the ports (leading to incident edges) at a node have locally unique labels.

We provide four efficient algorithms to solve dispersion in the asynchronous system model. These are the first algorithms to solve dispersion in the asynchronous system model. The first two algorithms require $O(k \log \Delta)$ bits at each robot, and $O(\min(m, k\Delta))$ steps running time. The third algorithm has a space usage of $O(\max(\min(D, k) \cdot \log \Delta, \log D))$ bits at each robot but uses $O(\Delta^{\min(D,k)+1})$ steps. The fourth algorithm has a space usage of $O(\max(\log k, \log \Delta))$ bits at each robot and uses $O(\min(m, k\Delta) \cdot k)$ steps. We assume that the robots do not know any of the graph parameters n, m, D, or Δ in the algorithms. The robots also do not need to know k. It is sufficient if a number of bits as specified by the memory bounds are provisioned at each robot. The motivation for providing four algorithms is that they offer different trade-offs between memory complexity, time complexity, and features, as contrasted in Table 1. No one algorithm is superior to the others in all respects. The following is an overview of our algorithms; the upper bound results are given in Table 1.

1. Algorithm *Helping-Async* has a time complexity $O(\min(m, k\Delta))$ and space complexity of $O(k \log \Delta)$ bits per robot. In this algorithm, *docked* robots, defined as robots that have reached their nodes in the final configuration, help visiting robots by maintaining data structures on their behalf. This algorithm requires each docked robot to remain active and help other visiting robots.
2. Algorithm *Independent-Async* has the same complexity ($O(\min(m, k\Delta))$ time steps and $O(k \log \Delta)$ bits per robot) as Algorithm *Helping-Async*; it differs in what, how, and where data structures are maintained. Here, each robot maintains its own data structures, as opposed to *Helping-Async* where docked robots help visiting robots by maintaining data structures on their behalf. Yet, this algorithm also requires each docked robot to remain active to relay its ID to other visiting robots.
3. Algorithm *Independent-Bounded-Async* has a bit complexity of $O(\max(\min(D, k) \cdot \log \Delta, \log D))$ at each robot and a time complexity $O(\Delta^{\min(D,k)+1})$ steps. Each robot runs its algorithm independently and there is no helping among robots. Unlike *Helping-Async* and *Independent-Async*, a robot can terminate after docking.
4. Algorithm *Tree-Switching-Async* has a bit complexity of $O(\max(\log k, \log \Delta))$ bits at each robot and a time complexity $O(\min(m, k\Delta) \cdot k)$ steps. The algorithm instance run by a robot is dependent on the algorithm instances run by other robots, and a robot switches between these algorithm instances in a structured manner. The algorithm requires a docked robot to remain active and help visiting robots.

Although the algorithms *Helping-Async*, *Independent-Async*, and *Tree-Switching-Async*, technically speaking, do not terminate because the docked robots need to be awake to relay local information to visiting robots, we state their time complexity. This is because at most the time complexity number of steps are required for each robot to perform active computations and movements until it docks at a node; after that, a docked robot merely passively helps visiting robots (until they find a node to dock).

A preliminary version of these results was given in [6].

Organization: Section 2 describes related work. In Section 3, we give the system model. In Section 4, we present some bounds and an analysis of these bounds. Sections 5, 6, 7, and 8 give the four algorithms *Helping-Async*, *Independent-Async*, *Independent-Bounded-Async*, and *Tree-Switching-Async*, along with their correctness and complexity proofs, respectively. Section 9 gives the conclusions.

2. Related work

The dispersion problem on graphs was formulated by Augustine and Moses Jr. [3]. They showed a lower bound of $\Omega(D)$ on the time complexity, and an independent lower bound of $\Omega(\log n)$ bits per robot, to solve dispersion. They then gave several dispersion algorithms for specific types of graphs for the synchronous computation model. Besides giving dispersion algorithms for paths, rings, trees, rooted trees (a rooted tree has all the robots at the same node in the initial configuration), and rooted graphs (a rooted graph has all the robots at the same node in the initial configuration), they gave two algorithms for general graphs (in which the robots can be at arbitrary nodes in the initial configuration) assuming the synchronous system model. The first algorithm uses $O(\log n)$ bits at each robot and $O(\Delta^D)$ rounds, whereas the second algorithm uses $O(n \log n)$ bits at each robot and $O(m)$ rounds. These algorithms work in synchronous systems, and additionally require robots to know the graph parameters m and n. However, both these algorithms are incorrect. Both algorithms use variants of Depth First Search (DFS), but may backtrack incorrectly. This can lead to getting caught in

cycles while backtracking and failure in searching the graph completely. The problems arise because the algorithms fail to coordinate correctly concurrent searches of the graph by different robots, which interfere with one another. The backtracking strategy is not consistent with the forward exploration strategy. Further, while backtracking from a node, a robot uses the parent pointer of the docked robot without any coordination. Acknowledging these errors that were pointed out [7], the authors gave revised versions of these algorithms in a revised report [8]. Their revision to the first algorithm, having $O(mn + n^2)$ rounds, is based on an inefficient way to convert from one DFS tree to another. Kshemkalyani et al. [9] proposed a fast algorithm for dispersion in the synchronous model. The algorithm has $O(\min(m, k\Delta) \cdot \log k)$ steps runtime using $O(\log n)$ bits of memory at each robot. Recently, Kshemkalyani et al. [10] proposed two algorithms for dispersion on arbitrary graphs in the synchronous model assuming *global communication*. (i) The first algorithm is based on a DFS traversal and guarantees $O(\min(m, k\Delta))$ steps runtime using $\Theta(\log(\max(k, \Delta)))$ bits at each robot. (ii) The second algorithm is based on a BFS traversal and guarantees $O(\max(D, k)\Delta(D + \Delta))$ steps runtime using $O(\max(D, \Delta \log k))$ bits at each robot. We note again that all the above works on dispersion assume the synchronous model. There is no prior work on dispersion assuming the weaker but more realistic asynchronous model.

The dispersion problem on graphs is close to the problem of graph exploration by robots. In the graph exploration problem, the objective is to visit all the nodes of the graph. There are many results for this problem. Several works assume specific topologies such as trees [11,12]. Fraigniaud et al. [13] showed that using only memory at a robot, the robot can explore an anonymous graph using $\theta(D \log \Delta)$ bits based on a D-depth restricted DFS. They did not analyze the time complexity, which turns out to be $\sum_{d=1}^{D} \sum_{i=1}^{d} \Delta^i = O(\Delta^{D+1})$. Their algorithm has no mechanism to avoid getting caught in cycles other than the depth-restriction on the DFS. The robot also requires knowledge of D to terminate. Reingold [14] gave a log-space deterministic algorithm for exploring undirected graphs. The space complexity is the best possible because the exploration of undirected graphs requires $\Omega(\log n)$ space [13]. Cohen et al. [15] gave two DFS-based algorithms with $O(1)$ memory at the nodes. The first algorithm uses $O(1)$ memory at the robot and 2 bits memory at each node to traverse the graph. The 2 bits memory at each node is initialized by short labels in a pre-processing phase which takes time $O(mD)$. Thereafter, each traversal of the graph takes up to $20m$ time steps. The second algorithm uses $O(\log \Delta)$ bits at the robot and 1 bit at each node to traverse the graph. The 1 bit memory at each node is initialized by short labels in a pre-processing phase which takes time $O(mD)$. Thereafter, each traversal of the graph takes up to $O(\Delta^{10}m)$ time steps. The problem of how much knowledge a robot has to have a priori, termed as advice that is provided by an oracle, in order to explore the graph in a given time, using a deterministic algorithm was considered in [16].

Dereniowski et al. [17] studied the trade-off between graph exploration time and the number of robots. The authors considered results in both the local communication model, as well as the global communication model. The main contribution is an exploration strategy for a polynomial number of robots $Dn^{1+\epsilon} < n^{2+\epsilon}$ to explore graphs in an asymptotically optimal number of steps $O(D)$. Using the Rotor-Router algorithm allowing only $\log \Delta$ bits per node, an oblivious robot (i.e., robot that is not allowed any memory) that also has no knowledge of the entry port when it enters a node, can explore an anonymous port-labeled graph in $2mD$ time steps [18,19]. Menc et al. [20] proved a lower bound of $\Omega(mD)$ on the exploration time steps for the Rotor-Router algorithm.

The dispersion problem is similar to the problem of scattering or uniform deployment of k robots on a n node graph (and in contrast to the gathering problem [4,21,22]). The scattering problem was examined on rings [23–25], and on grids [26,27], under different system assumptions than those that we make for the dispersion problem.

The dispersion problem is also similar to the load balancing problem, wherein a given load has to be (re-)distributed among several processors. In this analogy, the robots are the load, and it is these active loads rather than the passive nodes that make decisions about movements in the graph. Load balancing in graphs has been studied extensively. Load balancing algorithms use either a diffusion-based approach [28–30], which is somewhat similar to our algorithms, or a dimension-exchange approach [31] wherein a node can balance with either a single neighbor in a round, or concurrently with all its neighbors in a round.

3. System model

We are given an undirected graph G with n nodes, m edges, and diameter D. The maximum degree of any node is Δ. The graph is anonymous, i.e., nodes do not have unique identifiers. At any node, its incident edges are uniquely identified by a label in the range $[0, \delta - 1]$, where δ is the degree of that node. We refer to this label of an edge at a node as the port number at that node. We assume no correlation between the two port numbers of an edge. There is no memory at the nodes.

In our algorithms, we consider the asynchronous model. Time complexity is measured in steps. Each robot executes its steps/iterations at an independent pace. In any step, a robot stationed at a node does some computation, perhaps after communication with local robots, and then optionally does a move along one of the incident edges to an adjacent node. We assume that each edge is a single-lane edge, in the sense that robots can move along the edge sequentially. As a result, if multiple robots make a move along an edge, they will enter the node in sequential order which can be captured by a real-time synchronized clock. When a robot determines that it will occupy a particular node in the final configuration, it *docks* at that node (by entering $state = settled$).

The k robots are distinguished from each other by a unique $\lceil \log k \rceil$-bit label from the range $[1, k]$. The robots are also endowed with a real-time synchronized clock. A robot can communicate only with other robots that are present at the same node as itself. No robot has knowledge of the graph or its parameters n, m, D, and Δ. The robots also need not know k; it suffices if a number of bits as specified by the memory bounds are provisioned at each robot. For the asynchronous algorithms *Helping-Async*, *Independent-Async*, and *Tree-Switching-Async*, the main `for` loop could be replaced by a `while-true` loop. This is because a robot breaks out of the loop once it docks at a node, and is guaranteed to dock within a finite, bounded number of steps. The `for` loop bounds are specified using m, n, k, and Δ but those are only upper bounds on the number of steps and for convenience to express the time complexity. The parameters are not known to the robots.

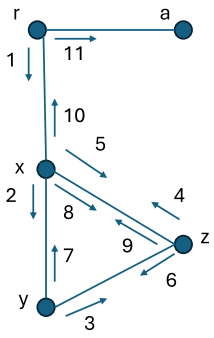

Fig. 1. An illustration of DFS. r is the root of DFS. The edges are explored in the numbered sequence. The non-tree edge (x, z) is traversed 4 times in the DFS – twice in the forward direction and twice in the backward direction.

We note that the system model in prior works that used the synchronous model [3,9,10] also assumed that robots are distinguishable and that the nodes in the graph are anonymous.

When robots contend to dock at a node, they invoke a MUTEX(node) call that guarantees that only one robot succeeds in docking. The MUTEX call returns the identifier of the robot that has docked. The MUTEX may be implemented in various ways. For example, the earliest robot (among the contending robots) that arrived at the node can win the MUTEX; if there is a tie in case of multiple robots arriving simultaneously along different ports, then the tie is broken by choosing the robot arriving along the lowest numbered port as the winner. Or, the robots can compare their labels and the robot with the smallest label wins the MUTEX. Alternatively, the MUTEX can be implemented by a hardware device to which the winner robot physically connects when it docks.

Problem Description: We are given an initial configuration of k robots, where $k \leq n$, distributed arbitrarily at the n nodes of a graph. The robots need to move around to reach a final configuration in which there is at most one robot at any node in the graph.

4. Bounds and a general analysis

For the graph dispersion problem, a lower bound of $\Omega(D)$ on the running time was shown in [3]. (Note that this prior work [3] required $k = n$ whereas we allow $k \leq n$.) We present a different lower bound.

Theorem 4.1. *The dispersion problem on graphs requires $\Omega(k)$ steps as its running time.*

Proof. Consider a line graph and all k robots colocated at one end node in the initial configuration. In order for the robots to dock at distinct nodes, some robot must travel $k - 1$ hops. □

Our algorithms use variants of the DFS. Although the standard DFS takes $2m$ edge traversals or steps, such an implementation requires Δ bits memory at each node to track the visits on each of the Δ ports. We choose to do the tracking function using only $\log \Delta$ bits. However, this savings comes at the cost of using $2(n-1) + 4(m-(n-1)) = 4m - 2n + 2$ edge traversals – 2 traversals for each of the $n - 1$ DFS tree edges, and 2 traversals each in the forward and backward directions on each of the $m - (n - 1)$ back edges of the graph. This is justified by the following example.

Example. Consider a graph $G = (V, E)$, where $V = \{r, x, y, z, a\}$, $E = \{(r, x), (x, y), (y, z), (z, x), (r, a)\}$, as illustrated in Fig. 1. Let the DFS begin from r. The sequence of edges explored are labeled in numerical order in the figure. Let forward edges of DFS tree be explored in sequence (r, x), (x, y), (y, z). From z, (z, x), which is a back edge (or non-tree) edge of the DFS tree, is explored in forward direction and then the robot backtracks along (in the direction of) (x, z). Backtracking continues on (in the direction of) (z, y) and then (y, x). From x, the robot traverses (x, z) in the forward direction, not knowing that it had previously traversed (z, x) in the forward direction and then (x, z) in backward direction. On visiting z the robot learns that it had already visited z. So it backtracks along (in the direction of) (z, x). Thus in this example, there are 4 traversals along the non-tree edge (x, z):

- (i) (z, x) in forward direction, (ii) (x, z) in backward direction, (iii) (x, z) in forward direction, (iv) (z, x) in backward direction.

In $4m - 2n + 2$ edges traversals, all the edges and hence all the nodes of the graph are visited and hence the $k \leq n$ robots can dock. We have an alternate time bound in terms of k. As each edge of the graph is traversed at most 4 times in the DFS traversal, the maximum number of visits to a node is 4Δ. Thus within $k \cdot 4\Delta$ steps, at least k unique nodes get visited and all the k robots can find a node to dock. Thus, the overall number of steps for docking all k robots is $\min(4m - 2n + 2, 4k\Delta)$, which is $O(\min(m, k\Delta))$.

A lower bound of $\Omega(\log n)$ bits on the memory of robots was shown in [3]. In the rest of this section, we analyze the memory bound of robots assuming that a $O(\min(m, k\Delta))$ time algorithm, based on independent DFS by each robot, is to be used. This analysis is useful for *Helping-Async* and *Independent-Async*. There are two challenges:

1. To determine whether a node has been visited before. Note that nodes have no memory in our system model. Although there are n nodes, we observe that a node has been visited before if and only if there is a robot docked at the node and there is a record of having encountered that robot before. As there are $k(\leq n)$ robots, it suffices to track whether or not each of the k robots has been encountered before. This imposes a bound of $O(k)$ bits.
2. If it is determined that a node has been visited before, backtracking is in order to meet the $O(\min(m, k\Delta))$ time bound. During the backtracking phase, to determine which port to use for backtracking requires identifying the parent node from which that robot first entered a particular node. Such a parent node can be identified by the local port number of the edge leading to the parent node. A port at a node can be encoded in $\log \Delta$ bits. Further, we need to track ports at at most $k-1$ nodes because only a node with a docked robot requires other visiting robots to backtrack, and up to $k-1$ nodes may be occupied by docked robots. This imposes a bound of $O(k \log \Delta)$ bits.

Thus, the overall bound on memory at a robot is $O(k \log \Delta)$ bits, assuming a $O(\min(m, k\Delta))$ time algorithm using independent DFS by each robot. The algorithms *Helping-Async* and *Independent-Async* that we propose meet these bounds.

As part of the robot memory-running time tradeoff, we also propose (i) algorithm *Independent-Bounded-Async* that uses $O(\max(\min(D, k) \cdot \log \Delta, \log D))$ bits at each robot with a running time of $O(\Delta^{\min(D,k)+1})$, and (ii) algorithm *Tree-Switching-Async* that uses $O(\max(\log k, \log \Delta))$ bits at each robot and a running time of $O(\min(m, k\Delta) \cdot k)$. In (i), we use potentially lesser memory than $k \log \Delta$ by using increasing depth-bounded search, and in (ii) we use lesser memory than $k \log \Delta$ by having the robots run DFSs that are not independent.

5. Dispersion using helping in the asynchronous model

In Algorithm *Helping-Async* (Algorithm 1) each robot begins a DFS-variant traversal of the graph, seeking to identify a node where no other robot has docked. If multiple robots arrive at about the same time at a node at which no other robot is docked, they use the MUTEX(node) function, explained in Section 3, to uniquely determine which of those robots can dock at the node. The other robots continue their search for a free node. During this search, a robot needs to determine if the node it visits has been visited before by it. (This is needed to determine whether to backtrack to avoid getting caught in cycles, or continue its forward exploration of the graph.) A node has been visited before if and only if the robot docked there has encountered the visiting robot after it docked. A robot that docks at a node helps other robots to determine whether they have visited this node before. A robot that docks initializes and maintains a boolean array $visited[1, k]$. It sets $visited[j]$ to true if and only if it has encountered robot j after docking. It helps a visiting robot j by communicating to it the value $visited[j]$.

While backtracking, in order for a robot to determine whether to backtrack from a (already visited) node or resume forward exploration, it needs to know the port leading to the DFS-parent node of the current node. It is helped in determining this as follows. A robot that docks initializes and maintains an array $entry\_port[1, k]$. Subsequently, when a robot j first visits the node, determined using $visited[j] = 0$ of the docked node, the $entry\_port[j]$ entry of the docked robot is set to the entry port used by the visiting robot. The docked robot also communicates $entry\_port[j]$ (in addition to $visited[j]$) to a visiting robot j to help it determine whether to backtrack further or resume forward exploration.

A robot uses the following variables:

- *port_entered* and *parent_ptr* of type port can take values from $\{-1, 0, 1, \ldots, \delta - 1\}$ ($\lceil \log(\Delta + 1) \rceil$ bits each); *port_entered* indicates the port number at the current node through which the robot entered the current node on the latest visit whereas *parent_ptr* is used to track the port through which the robot entered the current node on the first visit;
- *state* (2 bits) can take values from {*explore*, *backtrack*, and *settled*}; and
- *seen* (1 bit) is a boolean to track whether the current node has been seen/visited before.
- *round* is used as a round counter ($\log m = O(\log n)$ bits).

In addition, a robot initializes the following two arrays once it docks at a node and enters state $settled$:

- $visited[1, k]$ of type boolean (k bits), and
- $entry\_port[1, k]$ of type port ($k \lceil \log(\Delta + 1) \rceil$ bits).

The semantics of these two arrays was explained above.

In Algorithm 1, lines (30-35): a docked robot i helps visiting robot j by sending it $visited[j]$ and $entry\_port[j]$, and updating the locally maintained $visited[j]$ and $entry\_port[j]$ if this is the first visit of the robot j.

When robot i has $state = explore$ when it visits a node (line 5), there are two possibilities.

1. If some robot j is already docked (line 6), it receives $visited[i]$ and $entry\_port[i]$ from j (line 7). If the node is not already visited (line 8), i sends *port_entered* to j (line 9) which records it in $entry\_port[i]$ (lines 33-34). Whereas if the node is already visited (line 10), i backtracks through *port_entered* (line 11).
2. Robot i contends for the MUTEX (line 13) if there is no robot docked at the node. If i wins the MUTEX and docks, it (i) initializes the data structures $visited[1, k]$ and $port\_entered[1, k]$ (lines 13-15) and (ii) for other robots j concurrently at this node, it sends them $visited[j]$ and $entry\_port[j]$, receives *port_entered* from j, and fills in their entries in the newly created data structures

Algorithm 1 Helping-Async, asynchronous execution, code at robot i.

```
1: Initialize: port_entered ← −1; state ← explore; parent_ptr ← −1; seen ← 0
2: for count = 0, min(4m − 2(n − 1), 4kΔ) do
3:     if count > 0 then
4:         port_entered, parent_ptr ← entry port; seen ← 0
5:     if state = explore then                                   ▷ forward exploration mode
6:         if node has a robot j docked then
7:             seen, parent_ptr ← receive visited[i], entry_port[i] from j   ▷ receive info from docked robot
8:             if seen = 0 then                                  ▷ send info to previously unseen docked robot
9:                 parent_ptr ← port_entered; send port_entered to j
10:            if seen = 1 then
11:                state ← backtrack; move through port_entered
12:        else
13:            if i = (r ←)winner(MUTEX(node)) then              ▷ i wins MUTEX contention
14:                i docks at node; state ← settled
15:                Initialize visited[1, k] ← 0̄; entry_port[1, k] ← −1̄; break()
16:            else
17:                seen, parent_ptr ← receive visited[i], entry_port[i] from r   ▷ receive info from winner robot
18:                if seen = 0 then                              ▷ send info to previously unseen winner robot
19:                    parent_ptr ← port_entered; send port_entered to r
20:        port_entered ← (port_entered + 1) mod δ
21:        if port_entered = parent_ptr then
22:            state ← backtrack
23:        move through port_entered
24:    else if state = backtrack then                            ▷ backtrack mode
25:        seen, parent_ptr ← receive visited[i], entry_port[i] from docked robot j   ▷ receive info from docked robot
26:        port_entered ← (port_entered + 1) mod δ
27:        if port_entered ≠ parent_ptr then
28:            state ← explore
29:        move through port_entered
30: repeat                                                       ▷ state = settled
31:    for all other robot j that is/arrives at the node do
32:        send visited[j] and entry_port[j] to j                ▷ docked robot sends info to visiting/loser robot
33:        if visited[j] = 0 then                                ▷ docked robot updates info for previously unseen robot
34:            visited[j] ← 1; entry_port[j] ← receive port_entered from j
35: until true
```

(lines 31-34). Whereas if i loses the MUTEX contention to j, it receives $visited[i]$ and $entry\_port[i]$ from j, sets its $parent\_ptr$ to $port\_entered$, and sends $port\_entered$ to j (lines 17-19). (Lines (17-18) are seemingly redundant but are given so that a docked robot can interact uniformly with both newly arrived robots, and concurrently arrived robots that have lost MUTEX contention to it.)

If i has not backtracked and not docked, $state = explore$. In this case (line 20), i increases $port\_entered$ in a modulo fashion (mod δ) and moves forward to the next node, but switches $state$ to $backtrack$ if the port to move forward (new value of $port\_entered$) is the same as the entry port (in line 19, $parent\_ptr$ was set to the old value of $port\_entered$, which was set to the entry port in line 4) (lines 20-23).

If i has $state = backtrack$ when it visits a node (line 24), it implies some robot j is already docked, and i receives $visited[i]$ and $entry\_port[i]$ from j (line 25). Robot i increases $port\_entered$ in a modulo fashion (mod δ) and moves forwards to the next node while switching $state$ to $explore$, unless the port to move along (new value of $port\_entered$) is the parent pointer port (set to $entry\_port[i]$), in which case i keeps $state$ as $backtrack$ and backtracks instead of moving forward (lines 26-29).

When a robot i docks, for all robots j that are at the node (after losing contention to i) or later visit the node, i sends them their $visited$ and $entry\_port$ entries, and updates these entries if this is j's first visit to the node (lines 31-34).

Theorem 5.1. *Algorithm 1 (Helping-Async) achieves dispersion in an asynchronous system in $O(\min(m, k\Delta))$ steps with $O(k \log \Delta)$ bits at each robot.*

Proof. Observe that each robot executes a variant of a DFS in the search for a free node. Each robot may need to traverse each edge of its DFS tree two times (once forward, once backward), and each non-tree edge four times (once for exploration in each direction, and once for backtracking in each direction). So for a total of $4(m - (n-1)) + 2(n-1) = 4m - 2n + 2$ times until each edge and each node of the graph is visited. The robot executes for at most these many steps until it finds a free node to dock, so the running time is at most $4m - 2n + 2$. Also, in the DFS traversal, each edge is traversed at most 4 times. So within $4k\Delta$ steps, at least k distinct nodes are visited, a free node will be found, and the robot will dock there. Hence, dispersion is achieved.

To show that dispersion is achieved in $4m - 2n + 2$ steps, observe that the k robots do a collective search of the graph, using individual DFS variants. Within $4m - 2n + 2$ steps, if a robot is not yet docked, it will visit each node at least once, and since $k \leq n$, each robot will find a free node and dock there. Also within $4k\Delta$ steps, at least k unique nodes will be visited, and the robot will find a free node to dock. So the running time is $\min(4m - 2n + 2, 4k\Delta)$.

From the description and analysis of the variables above, it follows that the memory of each robot is bounded by $O(k \log \Delta)$ bits. □

Note that a docked robot needs to loop forever, waiting to help any other robot that might arrive at the node later. Thus, termination is not possible.

It is possible to transform the algorithm into its synchronous version, *Helping-Sync*. In the synchronous algorithm, a robot can terminate after $\min(4m - 2(n-1), 4k\Delta)$ steps, as it is guaranteed that every other robot would have found a free node by then. However, robots would need to know m, n, k, and Δ.

Algorithm 2 Independent-Async, asynchronous execution, code at robot i.

```
1: Initialize: port_entered ← −1; state ← explore; visited[1, k] ← 0̄; stack ←⊥
2: for count = 0, min(4m − 2(n − 1), 4kΔ) do
3:     if count > 0 then
4:         port_entered ← entry port
5:     if state = explore then
6:         if node is free then
7:             if i = winner(MUTEX(node)) then
8:                 i docks at node; state ← settled; break()
9:         if j is docked at node AND visited[j] = 0 then
10:            visited[j] ← 1
11:            push(stack, port_entered)
12:            port_entered ← (port_entered + 1) mod δ
13:            if port_entered = top(stack) then
14:                state ← backtrack; pop(stack)
15:            move through port_entered
16:        else if j is docked at node AND visited[j] = 1 then
17:            state ← backtrack; move through port_entered
18:    else if state = backtrack then
19:        port_entered ← (port_entered + 1) mod δ
20:        if port_entered ≠ top(stack) then
21:            state ← explore
22:        else
23:            pop(stack)
24:        move through port_entered
```

6. Independent dispersion in the asynchronous model

In Algorithm 2 (*Independent-Async*) for the asynchronous model, the DFS-like traversal of the graph by each robot is the same as in algorithm *Helping-Async*. However, there is no helping of undocked robots by docked robots. In addition to *port_entered* and *state*, an undocked robot maintains the following additional data structures:

- array of boolean $visited[1, k]$ to determine by checking $visited[r]$ whether it has visited the node where robot r is docked.
- *stack* of type port number, to determine the parent pointer of the nodes it has visited. Specifically, the port numbers in the stack (from top to bottom) help the robot to backtrack from the current node all the way to its origin node in the initial configuration. When a robot explores the graph in a step, the entry port number into the current node get pushed onto the stack, and as a robot backtracks in a step, the port number gets popped from the stack. In addition, the top of the stack entry is used for determining whether a robot should switch from backtracking state to explore state, or switch from explore state to backtracking state.

Thus, undocked robots are largely independent of docked robots. However, even in this algorithm, a docked robot cannot terminate; it needs to stay up so that it can relay its label r to a visiting undocked robot, which can then look up $visited[r]$, and if necessary, manipulate its *stack*, in order to take further actions for exploring the graph. This action of docked robots (once they enter *settled* state) is not explicitly shown in the Algorithm 2 pseudo-code.

In addition to the *port_entered* ($\lceil \log(\Delta + 1) \rceil$ bits) and *state* (two bits) variables used by the previous algorithm, the boolean array $visited[1, k]$ takes $O(k)$ bits and the *stack* takes $O(k \log \Delta)$ bits, because the maximum depth of the stack is $k - 1$, the maximum number of nodes at which there is a docked robot encountered.

In Algorithm 2, when robot i visits a node and $state = explore$ (line 5):

1. (lines 6-8): if the node is free, i contends for the MUTEX to dock. If i wins, it docks and breaks from the loop.
2. (lines 9-15): if (possibly after having lost MUTEX contention) i finds that robot j is docked at the node but the node has not been visited before, robot i marks $visited[j]$ as true and increments $port\_entered$ in a modulo fashion (mod δ). If the new value of $port\_entered$ equals its old value (which is the case when $\delta = 1$), i changes $state$ to $backtrack$ and moves through $port\_entered$; else the old value of $port\_entered$ is pushed onto the $stack$ and i moves through $port\_entered$ to continue the forward exploration of the graph.
3. (lines 16-17): if a robot j is docked and the node has been visited before, robot i backtracks.

When robot i visits a node and $state = backtrack$ (line 18), robot i increments $port\_entered$ in a modulo fashion (mod δ) and moves forward to the next node while switching $state$ to $explore$, unless the port it is going to move along is the parent pointer port (the top of the $stack$), in which case i keeps $state$ as $backtrack$ and pops the top of the $stack$ before moving along (lines 19-24).

Theorem 6.1. *Algorithm 2 (Independent-Async) achieves dispersion in an asynchronous system in $O(\min(m, k\Delta))$ steps with $O(k \log \Delta)$ bits at each robot.*

Proof. Dispersion is achieved because each robot traverses an independently built DFS tree; the correctness follows from using an argument similar to that in the proof of Theorem 5.1. The proof that the running time is $O(\min(m, k\Delta))$, or more specifically $\min(4m - 2n + 2, 4k\Delta)$ steps, is similar to that in the proof of Theorem 5.1. From the description and analysis of the variables above, it follows that the memory of each robot is bounded by $O(k \log \Delta)$ bits. □

Note that due to the nature of the asynchronous system, a docked robot needs to loop forever, waiting to relay its label to any other robot that might arrive at the node later. (This action is not explicitly shown in Algorithm 2.) Thus, termination is not possible.

It is possible to transform the algorithm into its synchronous version, *Independent-Sync*. In the synchronous algorithm, a robot can terminate after $\min(4m - 2(n-1), 4k\Delta)$ steps, as it is guaranteed that every other robot would have found a free node by then. However, robots would need to know m, n, k, and Δ.

7. Depth-bounded independent dispersion in the asynchronous model

Algorithm 3 Independent-Bounded-Async, asynchronous execution, code at robot i.

```
1:  Initialize: port_entered ← −1; state ← explore; depth ← −1; depth_bound ← 1; stack ←⊥
2:  while true do
3:      if depth > −1 then
4:          port_entered ← entry port
5:      if state = explore then
6:          depth ← depth + 1
7:          if node is free then
8:              if i = winner(MUTEX(node)) then
9:                  i docks at node; state ← settled; break()
10:         if depth < depth_bound then
11:             push(stack, port_entered)
12:             port_entered ← (port_entered + 1) mod δ
13:             if port_entered = top(stack) then
14:                 state ← backtrack; pop(stack)
15:             move through port_entered
16:         else if depth = depth_bound then
17:             state ← backtrack; move through port_entered
18:     else if state = backtrack then
19:         depth ← depth − 1
20:         port_entered ← (port_entered + 1) mod δ
21:         if top(stack) = −1 AND port_entered = 0 then
22:             depth_bound = depth_bound + 1
23:         if port_entered ≠ top(stack) then
24:             state ← explore
25:         else
26:             pop(stack)
27:         move through port_entered
```

Algorithm 3 (*Independent-Bounded-Async*) improves on the memory requirement of Algorithm 2 (*Independent-Async*), assuming $D < k$. It leverages the idea that a d-depth-bounded search of the graph can reduce the size of the stack from a maximum of k entries to a maximum of d entries, while being able to explore all the nodes in the graph as long as $d \geq D$ (the diameter of the graph). Since

D is not known, the algorithm at each robot runs increasing-depth-bounded searches. The algorithms run by the different robots are independent. Note that we cannot use the idea of curtailing the search if a robot visits a node that it has already visited. If we curtailed the search using that idea, we may not be able to discover shorter paths through already visited nodes, and we will be unable to reach all the nodes of the graph. Thus, this algorithm cannot use the $visited$ array and is fundamentally different from Algorithms *Helping-Async* and *Independent-Async*. Since we cannot curtail the search if a node has been visited before and we do an exhaustive search along every path rooted at the start node, there is redundancy in the algorithm and the time complexity is higher than the $O(\min(m, k\Delta))$ steps of the prior algorithms. The algorithm can be seen as a modification of the algorithm by Fraigniaud et al. [13] and incurs the same space and time complexity.

In addition to the variables $port\_entered$, $state$, and $stack$ of Algorithm *Independent-Async*, the following variables are used.

- $depth$ ($\lceil \log(D+1) \rceil$ bits) is used to track the current depth of the robot in the graph exploration.
- $depth\_bound$ ($\lceil \log(D+1) \rceil$ bits) is used to track the current depth bound in the graph exploration.

Theorem 7.1. *Algorithm 3 (Independent-Bounded-Async) achieves dispersion in an asynchronous system in* $O(\Delta^{\min(D,k)+1})$ *steps with* $O(\max(\min(D,k) \cdot \log\Delta, \log D))$ *bits at each robot.*

Proof. The algorithm uses an increasing depth-bounded search of the graph. When the depth becomes $\min(D,k)$, it is guaranteed that either all nodes of the graph can be visited (if D is lower) or at least k distinct nodes are visited (if k is lower), and since $k \leq n$, each robot will find a free node and successfully dock there. Thus the algorithm terminates and dispersion is achieved. The running time is $\sum_{d=1}^{\min(D,k)} \sum_{i=1}^{d} (\Delta-1)^i = O(\Delta^{\min(D,k)+1})$.

From the description and analysis of the variables above, observe that $stack$ requires $O(\min(D,k) \cdot \log\Delta)$ bits and $depth$ and $depth\_bound$ require $\lceil \log(D+1) \rceil$ bits. $port\_entered$ and $state$ require $\lceil \log(\Delta+1) \rceil$ and two bits, respectively. Thus, it follows that the memory of each robot is bounded by $O(\max(\min(D,k) \cdot \log\Delta, \log D))$ bits. □

It is possible to transform the algorithm into its synchronous version, *Independent-Bounded-Sync*. In the synchronous algorithm, a robot can terminate within $O(\Delta^{\min(D,k)+1})$ steps, as soon as it docks.

8. Prioritized tree-switching based dispersion in the asynchronous model

In the previous algorithms, each robot performed a separate DFS and the $parent_ptr$s for up to $k-1$ DFSs had to be stored at a docked robot (in *Helping-Async*), or a traversing robot had to track up to the $k-1$ $parent_ptr$s (in *Independent-Async*) or up to the $\min(D,k)-1$ $parent_ptr$s (in *Independent-Bounded-Async*) for its own DFS. Algorithm 4 (*Tree-Switching-Async*) uses $O(\max(\log k, \log\Delta))$ bits at each robot. With such limited memory, $O(1)$ $parent_ptr$s can be stored. As multiple robots pass through a docked robot's node, which DFS tree's $parent\_ptr$ should be stored at the docked robot? As a traversing robot encounters different docked robots, each associated with a possibly different DFS, with which tree and its local $parent\_ptr$ should it associate? It is critical to ensure that the robots coordinate in associating with a DFS tree and its local $parent_ptr$s. We solve this challenge as follows.

In addition to $port\_entered$, $state$, $parent\_ptr$ (set by a docked robot), and $depth$ used by previous algorithms, the following variables are used.

- $virtual\_id$, taken from the domain of robot identifiers ($\lceil \log k \rceil$ bits) is used to track the DFS tree instance the robot is associated with currently. The $virtual\_id$ is initialized to the robot identifier.
- $port\_fwd$: used by a docked robot to point to the port along which the DFS, for the tree instance with which it is currently associated, should continue out from that node. $port\_fwd$ is initialized to $(parent\_ptr + 1) \bmod \delta$ on docking at a node.

To achieve dispersion with limited memory $O(\max(\log k, \log\Delta))$, robots perform DFS like before; however, they do not perform independent DFSs. Rather, a strict priority order (a total order) is defined on the robot identifiers, and hence on the DFS tree instances which are tracked by the $virtual_id$s. As a robot traverses the graph, it induces a DFS tree identified by its $virtual\_id$. Whenever two robots (a docked robot and a traversing robot) meet, their DFS trees intersect. The lower priority robot abandons its partially computed DFS tree and switches to the higher priority DFS tree. (If the two priorities, i.e., $virtual_id$s, are the same the robots share the same tree; no switch is needed.)

In doing a switch, the lower priority robot:

1. updates its $virtual\_id$ to the higher priority,
2. updates its $depth$ variable to the new depth in the higher priority tree,
3. (if docked) updates its $parent\_ptr$ to $port\_entered$ of the traversing robot, and
4. (if docked) updates its $port\_fwd$ to $(port\_entered + 1) \bmod \delta$, where the higher priority robot entered the node through $port\_entered$.

If the traversing robot (whether in *explore* or *backtrack* state) does the switch, it then continues the DFS in the newly-switched-to tree along the port $r.port\_fwd$. Note that multiple robots may be executing the same tree instance possibly in different parts of the graph if they share the same $virtual\_id$.

Algorithm 4 Tree-Switching-Async, asynchronous execution, code at robot i. At any node, the docked robot, if any, is denoted r.

```
1: Initialize: port_entered ← −1; state ← explore; parent_ptr ← −1; virtual_id ← i; depth ← −1; port_fwd ← −1
2: for count = 0, min(4m − 2n + 2, 4kΔ) ∗ (k − 1) do
3:     if count > 0 then
4:         port_entered ← entry port
5:     if state = explore then                                              ▷ graph exploration mode
6:         depth ← depth + 1
7:         if i = (r ←)winner(MUTEX(node)) then
8:             i docks at node; parent_ptr ← port_entered; state ← settled
9:             port_fwd ← (parent_ptr + 1) mod δ; break()
10:        if virtual_id > r.virtual_id then                                ▷ i switches to tree of r
11:            virtual_id ← r.virtual_id; depth ← r.depth
12:        else if virtual_id < r.virtual_id then                           ▷ r switches to tree of i
13:            r.parent_ptr ← port_entered; r.virtual_id ← virtual_id; r.depth ← depth
14:            r.port_fwd ← (r.parent_ptr + 1) mod δ
15:        if depth = r.depth then   ▷ i and r share same tree (possibly after switch); arrived on tree edge
16:            if r.port_fwd = r.parent_ptr then
17:                state ← backtrack
18:            move through r.port_fwd
19:        else if depth ≠ r.depth then   ▷ i and r share same tree (no switch); arrived on back edge
20:            state ← backtrack; move through port_entered
21:    else if state = backtrack then                                       ▷ backtracking mode
22:        depth ← depth − 1
23:        if virtual_id > r.virtual_id then   ▷ i switches to tree of r; virtual_id < r.virtual_id not possible
24:            virtual_id ← r.virtual_id; depth ← r.depth
25:        else if virtual_id = r.virtual_id then   ▷ i and r share same tree and same depth
26:            r.port_fwd ← max(r.port_fwd, (port_entered + 1) mod δ) in the ordered sequence ⟨r.parent_ptr + 1, ..., δ − 1, 0, 1, ..., r.parent_ptr⟩
27:        if r.port_fwd ≠ r.parent_ptr then
28:            state ← explore
29:        move through r.port_fwd
30: repeat                                                                ▷ state = settled
31:    if any other robot arrives at the node then
32:        participate in the algorithm assuming the role of the docked robot r
33: until true
```

A $virtual\_id$ of a robot is the highest priority $virtual\_id$ of any robot (including itself) encountered until now in its traversal and docked durations. The $virtual\_id$ of a robot may be transitively inherited from other robots. We define a higher priority to be a lower valued $virtual\_id$. The total order on the $virtual_id$s bounds the number of times a robot is forced to switch trees, to $k-1$.

In addition to tracking only the highest-seen priority $virtual\_id$, a robot also tracks its current depth $depth$ in the corresponding tree, and a docked robot also tracks its $parent\_ptr$ and $port\_fwd$ in the corresponding tree. This $parent\_ptr$ and $port\_fwd$ together store the information for backtracking on the tree corresponding to the local $virtual\_id$. $virtual\_id = r.virtual\_id$ after line 14 (after or without a switch). The $depth$ and $r.depth$ after line 14 are used to determine whether the visiting robot should backtrack.

- If the $depth$s are the same (this happens certainly if there was a switch or possibly if there was no switch) (line 15), the visiting robot is deemed to have arrived on a tree edge in exploration mode and should continue as usual (lines 16-18).
- Otherwise (the $depth$s are unequal implying) no tree switch happened and the visiting robot arrived on a back edge in exploration mode, and therefore it should backtrack through $port\_entered$ (lines 19-20).

When a robot arrives in backtracking mode, the $depth$s will always be the same after line 24 (after or without a switch). The visiting robot is deemed to have arrived on a tree edge in exploration mode (if a switch happened) in lines 23-34, or on a tree edge or back edge in backtracking mode (if no switch happened) (line 25). In this latter case, $r.port\_fwd$ is updated as shown in line 26. Then in either case (switch or no switch), if $r.port\_fwd = r.parent\_ptr$ then i moves out on $r.port\_fwd$ in backtrack mode, but if $r.port\_fwd \neq r.parent\_ptr$ then i changes state to explore mode before it moves out on $r.port\_fwd$ (lines 27-29).

In the asynchronous algorithm, we assume for simplicity that if there is more than one visiting (undocked) robot at a node, they execute their code serially. This can be implemented by the docked robot using a token to communicate with each visiting robot. Thus, a docked robot interacts with one visiting robot at a time.

Lemma 8.1. *For any value of $virtual\_id$, an undocked robot docks or switches to a higher priority $virtual\_id$ within* $\min(4m-2n+2, 4k\Delta)$ *steps.*

Proof. The main steps of the proof are as follows.

1. Consider an undocked robot i with $virtual\_id$ vid. Until it docks or switches to a higher priority $virtual\_id$, it visits nodes with a docked robot r having virtual id vid (if lower priority than vid, then $r.virtual\_id \leftarrow vid$).
2. $r.depth$ is set correctly for all docked robots r with $r.virtual\_id = vid$.
3. The way that $depth$ is updated, if $depth = r.depth$ after line 6 or 22, then the robot i has traversed a DFS tree edge (in forward or backward direction), or has backtracked along a back edge. And if $depth \neq r.depth$, then the robot has traversed a back edge in explore mode. (In the algorithm, a back edge gets traversed twice in opposite directions in explore mode.)
4. Correct identification of tree edges and back edges leads to correct decisions about exploration and backtracking (acyclically) on the tree associated with vid.
5. For the tree associated with vid, the DFS is continued correctly by forwarding along $r.port\_fwd$, and updating $r.port\_fwd$ correctly in line 14 (when visiting in explore mode) and in line 26 (when visiting in backtrack mode).
6. When a robot switches to $virtual\_id$ vid at node v, there is no free node from the root node of the tree associated with vid up until the DFS search exits(ed) node v through $r.port\_fwd$ (here, r is docked at v). So right after the switch, the search continues from $r.port\_fwd$.
7. Hence, the DFS tree with $virtual\_id = vid$ is built/traversed correctly. A robot traverses each tree edge 2 times and each back edge 4 times. Thus, leading to $4(m-(n-1))+2(n-1) = 4m-2n+2$ steps. Within these many steps, the robot will find a free node and dock, or encounter a docked robot associated with a higher priority tree and switch its $virtual\_id$ to that higher priority. Further, as each tree edge is traversed at most 4 times, within $4k\Delta$ steps, the robot is bound to find a free node and dock. Hence, the lemma follows. □

Theorem 8.2. *Algorithm 4 (Tree-Switching-Async) achieves dispersion in an asynchronous system in* $O(\min(m, k\Delta)\cdot k)$ *steps with* $O(\max(\log k, \log\Delta))$ *bits at each robot.*

Proof. From Lemma 8.1, for any value of $virtual\_id$, a robot docks or switches to a higher priority $virtual\_id$ tree within $\min(4m-2n+2, 4k\Delta)$ steps. After each switch, it takes at most $\min(4m-2n+2, 4k\Delta)$ steps in the newly joined DFS tree before a robot finds a free node and docks, or makes another switch to an even higher priority $virtual\_id$ tree. Such a switch can occur to a robot at most $k-1$ times due to the total order on the bounded set of k identifiers. In the DFS traversal after the last such switch, the robot will necessarily dock and hence dispersion is achieved.

From the above reasoning, it also follows that the running time is $O(\min(m, k\Delta)\cdot k)$ steps until a robot docks.

Besides the $port\_entered$ ($O(\log\Delta)$ bits), $state$ (2 bits), $parent\_ptr$ ($O(\log\Delta)$ bits), and $depth$ ($O(\log k)$ bits) variables used in the earlier algorithms, this algorithm also uses $virtual\_id$ ($\lceil\log k\rceil$ bits) and $port\_fwd$ ($O(\log\Delta)$ bits). Thus, the memory at each robot is $O(\max(\log k, \log\Delta))$ bits. □

Observe that a docked robot needs to loop forever to cooperate with visiting robots.

It is possible to transform the algorithm into its synchronous version, *Tree-Switching-Sync*. In the synchronous algorithm, a robot can terminate within $O(\min(m, k\Delta)\cdot k)$ rounds, as it is guaranteed that every other robot would have found a free node by then. However, the robots would need to know m, n, k, and Δ.

9. Conclusions

For the dispersion problem on graphs, we proposed four algorithms for the asynchronous system model. These are the first algorithms to solve dispersion in the asynchronous model; until now, prior work considered dispersion only in the stronger synchronous model which is not realistic and does not model the real world. It is a challenge to design more space and time efficient algorithms and to prove lower bounds on the time and space complexity. Another challenge is to design dispersion algorithms for dynamically changing graphs.

We introduce the problem of *ongoing dispersion* on graphs. Rather than a one-shot dispersion, a robot, after docking and recharging, moves again on the graph (for an unspecified number of hops) and after some time, finds itself at some node from where it wants to search for a free node to dock again. Every time a docked robot moves, it creates a free node. This cycle repeats. It would be interesting to analyze our proposed algorithms and design new algorithms for ongoing dispersion.

CRediT authorship contribution statement

Ajay D. Kshemkalyani: Conceptualization, Methodology, Validation, Formal analysis, Writing - original draft, Writing - review and editing.

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] T. Hsiang, E.M. Arkin, M.A. Bender, S.P. Fekete, J.S.B. Mitchell, Algorithms for rapidly dispersing robot swarms in unknown environments, in: Algorithmic Foundations of Robotics V, Selected Contributions of the Fifth International Workshop on the Algorithmic Foundations of Robotics, WAFR 2002, Nice, France, December 15-17, 2002, 2002, pp. 77–94.

[2] J. Mclurkin, J. Smith, Distributed algorithms for dispersion in indoor environments using a swarm of autonomous mobile robots, in: 7th International Symposium on Distributed Autonomous Robotic Systems (DARS), 2004.

[3] J. Augustine, W.K. Moses-Jr., Dispersion of mobile robots: a study of memory-time trade-offs, in: Proceedings of the 19th International Conference on Distributed Computing and Networking, ICDCN 2018, Varanasi, India, January 4-7, 2018, 2018, pp. 1:1–1:10.

[4] P. Flocchini, G. Prencipe, N. Santoro, Distributed Computing by Oblivious Mobile Robots, Synthesis Lectures on Distributed Computing Theory, Morgan & Claypool Publishers, 2012.

[5] P. Flocchini, G. Prencipe, N. Santoro (Eds.), Distributed Computing by Mobile Entities, Current Research in Moving and Computing, Lecture Notes in Computer Science, vol. 11340, Springer, 2019.

[6] A.D. Kshemkalyani, F. Ali, Efficient dispersion of mobile robots on graphs, in: Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019, 2019, pp. 218–227.

[7] A.D. Kshemkalyani, F. Ali, Efficient dispersion of mobile robots on graphs, CoRR, arXiv:1805.12242, 2018.

[8] J. Augustine, W.K. Moses-Jr., Dispersion of mobile robots: a study of memory-time trade-offs, CoRR, arXiv:1707.05629v4, 2018.

[9] A.D. Kshemkalyani, A.R. Molla, G. Sharma, Fast dispersion of mobile robots on arbitrary graphs, in: F. Dressler, C. Scheideler (Eds.), 15th International Symposium on Algorithms and Experiments for Wireless Sensor Networks, ALGOSENSORS 2019, in: Lecture Notes in Computer Science, vol. 11931, Springer, 2019, pp. 23–40.

[10] A.D. Kshemkalyani, A.R. Molla, G. Sharma, Dispersion of mobile robots in the global communication model, in: 21st International Conference on Distributed Computing and Networking (ICDCN), ACM, 2020, pp. 12:1–12:10.

[11] C. Ambühl, L. Gasieniec, A. Pelc, T. Radzik, X. Zhang, Tree exploration with logarithmic memory, ACM Trans. Algorithms 7 (2) (2011) 17:1–17:21.

[12] P. Fraigniaud, L. Gasieniec, D.R. Kowalski, A. Pelc, Collective tree exploration, Networks 48 (3) (2006) 166–177.

[13] 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.

[14] O. Reingold, Undirected connectivity in log-space, J. ACM 55 (4) (2008) 17:1–17:24.

[15] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, D. Peleg, Label-guided graph exploration by a finite automaton, ACM Trans. Algorithms 4 (4) (2008) 42:1–42:18.

[16] B. Gorain, A. Pelc, Deterministic graph exploration with advice, in: 44th International Colloquium on Automata, Languages, and Programming (ICALP 2017), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 80, 2017, pp. 132:1–132:14.

[17] D. Dereniowski, Y. Disser, A. Kosowski, D. Pajak, P. Uznanski, Fast collaborative graph exploration, Inf. Comput. 243 (2015) 37–49.

[18] V. Yanovski, I.A. Wagner, A.M. Bruckstein, A distributed ant algorithm for efficiently patrolling a network, Algorithmica 37 (3) (2003) 165–186.

[19] E. Bampas, L. Gasieniec, N. Hanusse, D. Ilcinkas, R. Klasing, A. Kosowski, Euler tour lock-in problem in the rotor-router model, in: Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings, 2009, pp. 423–435.

[20] A. Menc, D. Pajak, P. Uznanski, Time and space optimality of rotor-router graph exploration, Inf. Process. Lett. 127 (2017) 17–20.

[21] S. Bhagat, S.G. Chaudhuri, K. Mukhopadhyaya, Fault-tolerant gathering of asynchronous oblivious mobile robots under one-axis agreement, J. Discret. Algorithms 36 (2016) 50–62.

[22] D. Pattanayak, K. Mondal, H. Ramesh, P.S. Mandal, Gathering of mobile robots with weak multiplicity detection in presence of crash-faults, J. Parallel Distrib. Comput. 123 (2019) 145–155.

[23] Y. Elor, A.M. Bruckstein, Uniform multi-agent deployment on a ring, Theor. Comput. Sci. 412 (8–10) (2011) 783–795.

[24] M. Shibata, T. Mega, F. Ooshita, H. Kakugawa, T. Masuzawa, Uniform deployment of mobile agents in asynchronous rings, in: Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016, 2016, pp. 415–424.

[25] M. Shibata, T. Mega, F. Ooshita, H. Kakugawa, T. Masuzawa, Uniform deployment of mobile agents in asynchronous rings, J. Parallel Distrib. Comput. 119 (2018) 92–106.

[26] L. Barrière, P. Flocchini, E.M. Barrameda, N. Santoro, Uniform scattering of autonomous mobile robots in a grid, Int. J. Found. Comput. Sci. 22 (3) (2011) 679–697.

[27] P. Poudel, G. Sharma, Time-optimal uniform scattering in a grid, in: Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019, 2019, pp. 228–237.

[28] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors, J. Parallel Distrib. Comput. 7 (2) (1989) 279–301.

[29] S. Muthukrishnan, B. Ghosh, M.H. Schultz, First- and second-order diffusive methods for rapid, coarse, distributed load balancing, Theory Comput. Syst. 31 (4) (1998) 331–354.

[30] R. Subramanian, I.D. Scherson, An analysis of diffusive load-balancing, in: SPAA, 1994, pp. 220–225.

[31] C. Xu, F.C.M. Lau, Analysis of the generalized dimension exchange method for dynamic load balancing, J. Parallel Distrib. Comput. 16 (4) (1992) 385–393.