Efficient Dispersion of Mobile Robots on Graphs

Ajay D. Kshemkalyani
University of Illinois at Chicago
Chicago, Illinois
ajay@uic.edu

Faizan Ali
University of Illinois at Chicago
Chicago, Illinois
fali28@uic.edu

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. In addition, an intrinsic application of dispersion has been shown to be the relocation of self-driven electric cars (robots) to recharge stations (nodes). We propose five algorithms to solve dispersion on graphs. The first three algorithms require $O(k \log \Delta)$ bits at each robot and $O(m)$ steps running time, where m is the number of edges and Δ is the degree of the graph. The algorithms differ in whether they address the synchronous or the asynchronous system model, and in what, where, and how data structures are maintained. The fourth algorithm, for the asynchronous model, has a space usage of $O(D \log \Delta)$ bits at each robot and uses $O(\Delta^D)$ steps, where D is the graph diameter. The fifth algorithm, for the asynchronous model, has a space usage of $O(\max(\log k, \log \Delta))$ bits at each robot and uses $O((m - n)k)$ steps.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; • **Mathematics of computing** → *Graph algorithms*; • **Computer systems organization** → *Robotics*;

KEYWORDS

distributed algorithm, dispersion, graph algorithm, graph exploration, mobile robot, collective robot exploration

ACM Reference Format:
Ajay D. Kshemkalyani and Faizan Ali. 2019. Efficient Dispersion of Mobile Robots on Graphs. In *International Conference on Distributed Computing and Networking (ICDCN '19), January 4–7, 2019, Bangalore, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3288599.3288610

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 [15]. 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. Although some works consider these problems in general graphs, others consider specific graphs like grids, trees, and rings.

ICDCN '19, January 4–7, 2019, Bangalore, India

ACM ISBN 978-1-4503-6094-4/19/01…$15.00
https://doi.org/10.1145/3288599.3288610

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 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 five efficient algorithms to solve dispersion in both the synchronous and asynchronous system models. The following is an overview of our algorithms; the upper bound results are given in Table 1.

(1) For the synchronous model, we present algorithm *Helping-Sync* which needs $O(k \log \Delta)$ bits per robot and $O(m)$ steps time complexity; for this synchronous algorithm, we assume robots know m if termination is to be achieved. 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.
(2) Algorithm *Helping-Async* is the asynchronous version of *Helping-Sync* and has the same time complexity $O(m)$ and same space complexity of $O(k \log \Delta)$ bits per robot; however this algorithm requires each docked robot to remain active and help other visiting robots.
(3) Algorithm *Independent-Async* has the same complexity ($O(m)$ time steps and $O(k \log \Delta)$ bits per robot) and features as Algorithm *Helping-Async*; it differs in what, how, and where

Table 1: Comparison of the proposed algorithms for dispersion on graphs.

Algorithm	Model	Memory Requirement at Each Robot (in bits)	Time Complexity	Termination
Helping-Sync	Sync.	$O(k \log \Delta)$	$O(m)$ steps	need to know m for termination
Helping-Async	Async.	$O(k \log \Delta)$	$O(m)$ steps	no termination
Independent-Async	Async.	$O(k \log \Delta)$	$O(m)$ steps	no termination
Independent-Bounded-Async	Async.	$O(D \log \Delta)$	$O(\Delta^D)$ steps	termination
Tree-Switching-Async	Async.	$O(\max(\log k, \log \Delta))$	$O((m-n)k)$ steps	no termination

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.

(4) Algorithm *Independent-Bounded-Async* has a bit complexity of $O(D \log \Delta)$ at each robot and a time complexity $O(\Delta^D)$ steps. Unlike the earlier asynchronous algorithms, this algorithm is guaranteed to terminate. Each robot runs its algorithm independently and there is no helping among robots.

(5) Algorithm *Tree-Switching-Async* has a bit complexity of $O(\max(\log k, \log \Delta))$ bits at each robot and a time complexity $O((m-n)k)$ 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 asynchronous 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).

1.3 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. 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. We claim that 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 we pointed out [16], the authors gave revised versions of these algorithms in a revised report [2]. Their revision to the first algorithm, having $O(mn + n^2)$ rounds, and our *Tree-Switching-Async* algorithm use some similar ideas.

The dispersion problem on graphs is closest 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 [1, 12]. For general graphs, the results depend on the different system models and assumptions such as the following.

(1) what parameters of the graph are known to the robots,
(2) whether the graph is anonymous,
(3) whether memory is allowed at robots [13],
(4) whether memory is allowed at the nodes [8],
(5) whether knowledge of the incoming ports through which a robot enters nodes is allowed [13],
(6) whether exploration is by a single robot or cooperating robots [6, 7, 10],
(7) if exploration is by multiple robots, whether robots are allowed to communicate under the local communication model or the global communication model [6, 7, 10],
(8) if exploration is by multiple robots, whether robots are colocated or dispersed in the initial configuration,
(9) whether we are designing a solution that is time optimal, or space optimal,
(10) whether termination of the robot is required or it is to perpetually traverse the graph [17].

We now review a few of the closest results. 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 $\sum_{i=1}^{D} O(\Delta^i) = O(\Delta^D)$. Their algorithm has no mechanism to avoid getting caught in cycles and the only way out of cycles is the depth-restriction on the DFS. The robot also requires knowledge of D to terminate. Reingold [20] 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. [8] 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 [14].

Dereniowski et al. [10] studied the trade-off between graph exploration time and number of robots, assuming that (i) nodes have unique identifiers, (ii) when visiting a node, a list of all its neighbors is also known, (iii) all the robots are located at one node in the initial configuration, (iv) robots have unique identifiers, and (v) there is no bound on the memory of robots, which construct a map of the previously visited subgraph. 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 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 [4, 23]. Menc et al. [18] 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. The scattering problem was examined on rings [11], and on grids [5], 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 [9, 19, 21], which is somewhat similar to our algorithms, or a dimension-exchange approach [22] wherein a node can balance with either a single neighbor in a round, or concurrently with all its neighbors in a round.

2 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 both the synchronous model and the asynchronous model. In the synchronous model, there is a global clock that coordinates the processing of the robots in rounds. In any round, 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. Multiple robots can move along an edge in a round. However, 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. In the asynchronous model, there is no global mechanism that coordinates the round numbers of the robots. Thus, each robot executes its rounds/iterations at an independent pace. 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 initially has knowledge of the graph or its parameters n, m, D, and Δ. We assume each robot knows k, which is upper-bounded by n, in *Helping-Sync*, *Helping-Async*, and *Independent-Async*. In our synchronous algorithms (*Helping-Sync*, and the synchronous versions of algorithms *Independent-Async* and *Tree-Switching-Async*), we assume a robot has knowledge of the parameter m if we want to achieve local termination of the code after a robot has docked at a node in the final configuration. 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.

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, in the synchronous model, the robots can compare their labels and the robot with the smallest label wins the MUTEX. Or 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.

2.1 Bounds and their 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 2.1. *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. □

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 $O(m)$ time algorithm, based on DFS, is to be used. 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 $O(m)$ 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 $O(m)$ time algorithm. The algorithms *Helping-Sync*, *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 $O(D \log \Delta)$ bits at each robot with a running time of $O(\Delta^D)$, and (ii) algorithm *Tree-Switching-Async* that uses $O(\max(\log k, \log \Delta))$ bits at each robot and a running time of $O((m-n)k)$.

3 DISPERSION USING HELPING IN THE SYNCHRONOUS MODEL

In Algorithm 1 (*Helping-Sync*), 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 a node at which no other robot is docked in a particular round, they use the MUTEX(node) function, explained in Section 2, 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 $visited[j]$.

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 $\{-1, 0, 1, \ldots, \log \delta - 1\}$ ($\lceil \log(\Delta+1) \rceil$ bits each); $port\_entered$ indicates the port 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 (3-7): 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 visits a node where some robot j is already docked, it receives $visited[i]$ and $entry\_port[i]$ from j (line 13). If i has $state = explore$ and the node is already visited, i backtracks through $port\_entered$ (lines 16, 17). Whereas if the node is not already visited (lines 14, 15), i sends $port\_entered$ to j which records it in $entry\_port[i]$ (line 7). Robot i contends for the MUTEX (line 19) if there is no robot docked at the node. If i wins the MUTEX and docks, it initializes the data structures $visited[1, k]$ and $port\_entered[i, k]$ and for other robots j concurrently at this node in this round, it fills in their entries in the newly created data structures (lines 19-24). Whereas if i loses the MUTEX contention, it sends $port\_entered$ to the winner of MUTEX (lines 25, 26). If i has not backtracked and not docked, $state = explore$. In this case (line 27), 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 15, $parent\_ptr$ was set to the old value of $port\_entered$, which was set to the entry port in line 10) (lines 28-31).

If i has $state = backtrack$ when it visits a node (line 32), it implies some robot j is already docked, and i receives $visited[i]$ and $entry\_port[i]$ from j (line 33). 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 34-37).

Algorithm 1 Helping-Sync, synchronous execution, code at robot i

```
Initialize: port_entered ← −1; state ← explore; parent_ptr ← −1; seen ← 0
for round = 0, 4m − 2(n − 1) do
    if state = settled then
        for all other robot j on the node do
            send visited[j] and entry_port[j] to j          ▷ docked robot sends info to visiting robots
            if visited[j] = 0 then                          ▷ docked robot updates info for previously unseen robots
                visited[j] ← 1; entry_port[j] ← receive port_entered from j
    else
        if round > 0 then
            port_entered, parent_ptr ← entry port; seen ← 0
    if state = explore then                                 ▷ forward exploration mode
        if node has a robot j docked in an earlier round then
            seen, parent_ptr ← receive visited[i], entry_port[i] from j    ▷ receive info from docked robot
            if seen = 0 then                                ▷ send info to previously unseen docked robot
                parent_ptr ← port_entered; send port_entered to j
            if seen = 1 then
                state ← backtrack; move through port_entered
        else
            if i = (r ←)winner(MUTEX(node)) then            ▷ i wins MUTEX contention
                i docks at node; state ← settled
                Initialize visited[1, k] ← 0̄; entry_port[1, k] ← −1 (overlined)
                for all robot j on the node do              ▷ winner i updates info for loser robots
                    entry_port[j] ← receive port_entered from j
                    visited[j] ← 1
            else                                            ▷ i loses MUTEX contention
                send port_entered to r                      ▷ loser sends info to winner of MUTEX
        if state = explore then
            port_entered ← (port_entered + 1) mod δ
            if port_entered = parent_ptr then
                state ← backtrack
            move through port_entered
    else if state = backtrack then                          ▷ backtrack mode
        seen, parent_ptr ← receive visited[i], entry_port[i] from docked robot j    ▷ receive info from docked robot
        port_entered ← (port_entered + 1) mod δ
        if port_entered ≠ parent_ptr then
            state ← explore
        move through port_entered
```

Theorem 3.1. *Algorithm 1 (Helping-Sync) achieves dispersion in a synchronous system in $O(m)$ rounds 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. The robot executes for these many rounds, so the running time is $O(m)$.

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.

To show that dispersion is achieved in $4m-2n+2$ rounds, observe that the k robots do a collective search of the graph, using individual DFS variants. Within $4m-2n+2$ rounds, 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. □

Note that although a robot may dock at a node, it needs to be active for the rest of the $4m - 2n + 2$ rounds of the algorithm in order to help other robots which might visit this node.

4 DISPERSION USING HELPING IN THE ASYNCHRONOUS MODEL

Algorithm *Helping-Async* (Algorithm 2) adapts Algorithm *Helping-Sync* to an asynchronous system but uses the same variables. When a robot arrives at a node, either another robot is docked or not docked at that node; in the latter case, if multiple robots arrive at about the same time, then function MUTEX(node) selects one of

Algorithm 2 Helping-Async, asynchronous execution, code at robot i

```
1:  Initialize: port_entered ← −1; state ← explore; parent_ptr ← −1; seen ← 0
2:  for count = 0, 4m − 2(n − 1) 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 docked winner robot
18:                 if seen = 0 then               ▷ send info to previously unseen docked 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
34:             visited[j] ← 1; entry_port[j] ← receive port_entered from j   ▷ docked robot updates info for previously unseen robot
35: until true
```

them to dock. Lines (17-18) are seemingly redundant but are given so that a docked robot can interact uniformly with both newly arrived and concurrently arrived robots.

THEOREM 4.1. *Algorithm 2 (Helping-Async) achieves dispersion (without termination) in an asynchronous system in $O(m)$ steps with $O(k \log \Delta)$ bits at each robot.*

PROOF. The proof is similar to that of Theorem 3.1. The difference is that due to the nature of the asynchronous system, 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. □

5 INDEPENDENT DISPERSION IN THE ASYNCHRONOUS MODEL

In Algorithm 3 (*Independent-Async*) for the asynchronous model, the traversal of the graph by each robot is the same as in the previous two algorithms. 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.

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

```
1: Initialize: port_entered ← −1; state ← explore; visited[1, k] ← 0̄; stack ←⊥
2: for count = 0, 4m − 2(n − 1) 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
```

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 3 pseudo-code.

In addition to the $port\_entered$ ($\lceil \log(\Delta + 1) \rceil$ bits) and $state$ (two bits) variables used by the previous algorithms, the boolean $visited[1, k]$ array 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 3, 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, 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 5.1. *Algorithm 3 (Independent-Async) achieves dispersion (without termination) in an asynchronous system in $O(m)$ steps with $O(k \log \Delta)$ bits at each robot.*

Proof. Dispersion is achieved because each robot traverses an independently built DFS tree. The proof that the running time is $O(m)$, or more specifically $4m - 2n + 2$ steps, is similar to that of Theorem 3.1. 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 3.) 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 $4m - 2(n - 1)$ rounds, as it is guaranteed that every other robot would have found a free node by then.

6 DEPTH-BOUNDED INDEPENDENT DISPERSION IN THE ASYNCHRONOUS MODEL

Algorithm 4 (*Independent-Bounded-Async*) improves on the memory requirement of Algorithm 3 (*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 Algorithm *Independent-Async* and the previous algorithms. 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 $O(m)$ 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 variables $depth$ and $depth\_bound$ ($\lceil \log (D + 1) \rceil$ bits) are used to track the current depth of the robot in the graph exploration, and the current depth bound, respectively.

Algorithm 4 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
```

Theorem 6.1. *Algorithm 4 (Independent-Bounded-Async) achieves dispersion in an asynchronous system in $O(\Delta^D)$ steps with $O(D \log \Delta)$ bits at each robot.*

Proof. The algorithm uses an increasing depth-bounded search of the graph. When the depth becomes D (the diameter), it is guaranteed that all nodes of the graph will be visited, 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 $\sum_{i=1}^{D} \Delta^i$ which is bounded by $O(\Delta^D)$.

From the description and analysis of the variables above, observe that *stack* requires $O(D \log \Delta)$ 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 $O(D \log \Delta)$ bits. □

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

7 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, or a traversing robot had to track up to the $k-1$ *parent_ptr*s for its own DFS. Algorithm 5 (*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 variable *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.

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 (i) updates its *virtual_id* to the higher priority, (ii) updates its *depth* variable to the new depth in the higher priority tree, and (iii) updates its *parent_ptr* (if docked) to *port_entered* of the traversing robot or its *port_entered* (if traversing) to *parent_ptr* of the docked robot. If the traversing robot (whether in *explore* or *backtrack* state) does the switch, it then continues the DFS in the newly-switched-to tree as if it had just entered that node where the switch occurs in *explore* state for the first time. 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*.

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* in the corresponding tree. This *parent_ptr* stores the information for backtracking on the tree corresponding to the local *virtual_id*. $virtual\_id = r.virtual\_id$ after line 12 (after or without a switch). The *depth* and $r.depth$ after line 12 are used to determine whether the visiting robot should backtrack.

Algorithm 5 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
2:  for count = 0, (4m − 2n + 2) ∗ (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; break()
9:          if virtual_id > r.virtual_id then                                ▷ i switches to tree of r
10:             virtual_id ← r.virtual_id; depth ← r.depth; port_entered ← r.parent_ptr
11:         else if virtual_id < r.virtual_id then                           ▷ r switches to tree of i
12:             r.parent_ptr ← port_entered; r.virtual_id ← virtual_id; r.depth ← depth
13:         if depth = r.depth then     ▷ i and r share same tree (possibly after switch); arrived on tree edge
14:             port_entered ← (port_entered + 1) mod δ
15:             if port_entered = r.parent_ptr then
16:                 state ← backtrack
17:         else if depth ≠ r.depth then       ▷ i and r share same tree (no switch); arrived on back edge
18:             state ← backtrack
19:     else if state = backtrack then                                        ▷ backtracking mode
20:         depth ← depth − 1
21:         if virtual_id > r.virtual_id then   ▷ i switches to tree of r; virtual_id < r.virtual_id not possible
22:             virtual_id ← r.virtual_id; depth ← r.depth; port_entered ← r.parent_ptr
23:         port_entered ← (port_entered + 1) mod δ   ▷ i and r share same tree (possibly after switch) and same depth
24:         if port_entered ≠ r.parent_ptr then
25:             state ← explore
26:     move through port_entered
27: repeat                                                                        ▷ state = settled
28:     if any other robot arrives at the node then
29:         participate in the algorithm assuming the role of the docked robot r
30: until true
```

- If the *depth*s are the same (this happens certainly if there was a switch or possibly if there was no switch), the visiting robot is deemed to have arrived on a tree edge in exploration mode and should continue as usual (lines 14-16,26).
- 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.

The *depth*s will always be the same after line 22 (after or without a switch) and the visiting robot is deemed to have arrived on a tree edge in exploration mode (if a switch happened), or on a tree edge or back edge in backtracking mode (if no switch happened).

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 7.1. *For any value of $virtual\_id$, an undocked robot docks or switches to a higher priority $virtual\_id$ within $4m - 2n + 2$ steps.*

Proof. We summarize the main steps of the proof.

(1) Consider an undocked robot with $virtual\_id$ vid. Until it docks or switches to a higher priority $virtual\_id$, it visits nodes with a docked robot 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 with $r.virtual\_id = vid$.
(3) The way that $depth$ is updated, if $depth = r.depth$ after line 6 or 20, then the robot 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) 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 enters(ed) node v for the first time. So right after the switch, the search continues from $(port\_entered (= r.parent\_ptr) + 1) \mod \delta$ at node v.
(6) 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.

□

Theorem 7.2. *Algorithm 5 (Tree-Switching-Async) achieves dispersion (without termination) in an asynchronous system in $O((m-n)k)$ steps with $O(\max(\log k, \log \Delta))$ bits at each robot.*

Proof. From Lemma 7.1, for any value of *virtual_id*, a robot docks or switches to a higher priority *virtual_id* tree within $4m - 2n + 2$ steps. After each switch, it takes at most $4m - 2n + 2$ steps in the newly joined DFS tree before a robot finds a free node and docks, or makes another switch. 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. Thus, the runnning time is $O((m - n)k)$ steps until a robot docks. (However, a docked robot needs to loop forever to cooperate with visiting robots. Thus, termination is not possible.)

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). Thus, the memory at each robot is $O(\max(\log k, \log \Delta))$ bits. □

We can reduce the number of steps traversed by robots by using the following optimization. A docked robot maintains a variable *port_fwd*, initialized to $(parent\_ptr+1) \mod \delta$ when it docks (line 8) or changes its *parent_ptr* (line 12), to indicate the outgoing port on which the next robot should traverse in the forward direction. This port $r.port\_fwd$ is used for moving out of the node (line 26) except if lines 17-18 are executed in which case the robot moves out of *port_entered*. This port is used (instead of *port_entered*) to compare with $r.parent\_ptr$ to determine the state (line 15, 24). The code block (lines 23-25) is split into two cases: after line 20, (i) $virtual\_id > r.virtual\_id$ and (ii) $virtual\_id = r.virtual\_id$. For the latter case (ii), the line 23 equivalent is replaced by: $r.port\_fwd \leftarrow \max(r.port\_fwd, (port\_entered+1) \mod \delta)$ in the ordered sequence $\langle r.parent\_ptr+1, \ldots, \delta-1, 0, \ldots, r.parent\_ptr \rangle$. Lines 14 and the equivalent of line 23 for case (i) are deleted.

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

Unlike our algorithm, the revised algorithm [2] for synchronous systems uses an extra (fourth) state, *backtrack_to_root*. When robots decide to change their tree, those robots (excluding the docked robot) backtrack to the root of the new tree. They then begin a DFS from that root. The backtracking to the root, and restarting the DFS from the root, adds overhead and complexity. Also, a different condition is used to decide when to backtrack.

8 CONCLUSIONS

For the dispersion problem on graphs, we proposed five algorithms for the synchronous and the asynchronous system models. It is a challenge to design more space and time efficient algorithms.

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 an unoccupied 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.

REFERENCES

[1] Christoph Ambühl, Leszek Gasieniec, Andrzej Pelc, Tomasz Radzik, and Xiaohui Zhang. 2011. Tree exploration with logarithmic memory. *ACM Trans. Algorithms* 7, 2 (2011), 17:1–17:21.

[2] John Augustine and William K. Moses Jr. 2017. Dispersion of Mobile Robots: A Study of Memory-Time Trade-offs. *CoRR* abs/1707.05629 (2017). arXiv:1707.05629 http://arxiv.org/abs/1707.05629

[3] John Augustine and William K. Moses-Jr. 2018. 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.* 1:1–1:10.

[4] Evangelos Bampas, Leszek Gasieniec, Nicolas Hanusse, David Ilcinkas, Ralf Klasing, and Adrian Kosowski. 2009. 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.* 423–435.

[5] Lali Barrière, Paola Flocchini, Eduardo Mesa Barrameda, and Nicola Santoro. 2011. Uniform Scattering of Autonomous Mobile Robots in a Grid. *Int. J. Found. Comput. Sci.* 22, 3 (2011), 679–697.

[6] Peter Brass, Flavio Cabrera-Mora, Andrea Gasparri, and Jizhong Xiao. 2011. Multirobot Tree and Graph Exploration. *IEEE Trans. Robotics* 27, 4 (2011), 707–717.

[7] Peter Brass, Ivo Vigan, and Ning Xu. 2014. Improved analysis of a multirobot graph exploration strategy. In *13th International Conference on Control Automation Robotics & Vision, ICARCV 2014, Singapore, December 10-12, 2014.* 1906–1910.

[8] Reuven Cohen, Pierre Fraigniaud, David Ilcinkas, Amos Korman, and David Peleg. 2008. Label-guided graph exploration by a finite automaton. *ACM Trans. Algorithms* 4, 4 (2008), 42:1–42:18.

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

[10] Dariusz Dereniowski, Yann Disser, Adrian Kosowski, Dominik Pajak, and Przemyslaw Uznanski. 2015. Fast collaborative graph exploration. *Inf. Comput.* 243 (2015), 37–49.

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

[12] Pierre Fraigniaud, Leszek Gasieniec, Dariusz R. Kowalski, and Andrzej Pelc. 2006. Collective tree exploration. *Networks* 48, 3 (2006), 166–177.

[13] Pierre Fraigniaud, David Ilcinkas, Guy Peer, Andrzej Pelc, and David Peleg. 2005. Graph exploration by a finite automaton. *Theor. Comput. Sci.* 345, 2-3 (2005), 331–344.

[14] Barun Gorain and Andrzej Pelc. 2017. Deterministic Graph Exploration with Advice. In *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 80. 132:1–132:14.

[15] Tien-Ruey Hsiang, Esther M. Arkin, Michael A. Bender, Sándor P. Fekete, and Joseph S. B. Mitchell. 2002. 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.* 77–94.

[16] Ajay D. Kshemkalyani and Faizan Ali. 2018. Efficient Dispersion of Mobile Robots on Graphs. *CoRR* abs/1805.12242 (2018). arXiv:1805.12242 http://arxiv.org/abs/1805.12242

[17] Ajay D. Kshemkalyani and Faizan Ali. 2018. Fast Graph Exploration by a Mobile Robot. In *1st IEEE International Conference on Artificial Intelligence and Knowledge Engineering, AIKE 2018, Laguna Hills, CA, USA, September 26-28, 2018.* 115–118.

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

[19] S. Muthukrishnan, Bhaskar Ghosh, and Martin H. Schultz. 1998. First- and Second-Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing. *Theory Comput. Syst.* 31, 4 (1998), 331–354.

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

[21] Raghu Subramanian and Isaac D. Scherson. 1994. An Analysis of Diffusive Load-Balancing. In *SPAA*. 220–225.

[22] Cheng-Zhong Xu and Francis C. M. Lau. 1992. Anlaysis of the Generalized Dimension Exchange Method for Dynamic Load Balancing. *J. Parallel Distrib. Comput.* 16, 4 (1992), 385–393.

[23] Vladimir Yanovski, Israel A. Wagner, and Alfred M. Bruckstein. 2003. A Distributed Ant Algorithm for Efficiently Patrolling a Network. *Algorithmica* 37, 3 (2003), 165–186.