Efficient Dispersion of Mobile Robots on Graphs

Ajay D. Kshemkalyani* Faizan Ali †

March 17, 2018

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 show a lower bound of $\Omega(k \log \Delta)$ on the number of bits at each robot, subject to a running time of $O(m)$. Here, m is the number of edges and Δ is the degree of the graph. Then we provide efficient algorithms to solve dispersion in both the synchronous and asynchronous system models. Our algorithms meet the lower bound on bit complexity, subject to the time complexity of $O(m)$. If the diameter D of the graph is known, we give an algorithm to reduce the bit complexity at a robot to $O(\max(D \log \Delta, k))$.

Keywords: distributed algorithm; dispersion; graph algorithm; graph exploration; mobile robot

1 Introduction

1.1 Background and Motivation

The dispersion problem on graphs, formulated by Augustine and Moses Jr. [2], 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) [2]. Recharging is a time-consuming process and it is better to search for a vacant recharge station than to wait.

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 over the years by varying the system model and assumptions. Although some works consider these problems in general graphs, many other works consider specific graphs like grids, trees and rings.

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 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. In analyzing the trade-off between time complexity and space complexity, we show a lower bound of $\Omega(k \log \Delta)$ on the number of bits at each robot, assuming that a $O(m)$ time complexity has to be achieved.

We then provide efficient algorithms to solve dispersion in both the synchronous and asynchronous system models. Our algorithms meet the lower bound on bit complexity, subject to the time complexity of $O(m)$ steps. We first assume that the robots do not know any of the graph parameters n, m, D, or Δ in the

*Department of Computer Science, University of Illinois at Chicago. Email: ajay@uic.edu

†Department of Computer Science, University of Illinois at Chicago. Email: fali28@uic.edu

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

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

algorithms. It is sufficient if $O(k \log \Delta)$ bits are provisioned at each robot. 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; for this synchronous algorithm, we then assume robots know m if termination is to be achieved.

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 (that has reached its node in the final configuration) 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 how and where the data structures are maintained.

4. If the diameter D of the graph is known to the robots, we give Algorithm *Independent-Bounded-Async* to reduce the bit complexity at a robot to $O(\max(D \log \Delta, k))$, while achieving a time complexity $O(m)$ steps. Like the other asynchronous algorithms, this algorithm too can not terminate after reaching its node in the final configuration, to help visiting robots navigate the graph.

1.3 Related Work

The dispersion problem was formulated by Augustine and Moses Jr. [2]. 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, assuming 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, unfortunately, both these algorithms are incorrect. Both algorithms use variants of Depth First Search (DFS), but fail to search the graph completely, backtrack incorrectly, and can get caught in cycles while backtracking. This also renders their complexity results incorrect. Our work considers dispersion in (unrooted) graphs wherein the robots can be at arbitrary nodes in the initial configuration, for both the synchronous and the asynchronous computation models. We consider general graphs rather than restricted graphs like grids, trees and rings.

The dispersion problem 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], [10], [12], [14]. 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 [7], [20],
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 [5], [6], [9],
7. if exploration is by multiple robots, whether robots are allowed to communicate under the local communication model or the global communication model [5], [6], [9],
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 the bounds on memory are subject to time optimality solutions,
11. whether termination of the robot is required (and if so, whether at the starting node) or it is to perpetually traverse the graph

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 $(D+1)$-depth restricted DFS. They did not analyze the time complexity, which turns out to be $\sum_{i=1}^{D} O(\Delta^i) = O(\Delta^{D+1})$ which is very high. 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. Dereniowski et al. [9] study 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 consider 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 [3], [22]. By using an additional single bit at the robot, Menc et al. [15], [16] were able to speed up the Rotor-Router algorithm and achieve graph exploration in $O(m)$ time. They also proved that if the robot is not allowed any memory, then $\Omega(n^3)$ time is required for graph exploration.

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], [18], and on grids [4], 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 either use a diffusion-based approach [8], [17], [19], which is somewhat similar to our algorithms, or a dimension-exchange approach [21] 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 k-bit label from the range $[1, k]$. The robots are also endowed with a real-time synchronized clock. A robot can only communicate 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 our synchronous algorithms (Helping-Sync, and the synchronous versions of the other algorithms presented in the asynchronous model), 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, the main for-loop counting up to $4m - 2n$ could be replaced by a while-true loop. This is because even after a robot docks at a node, it needs to communicate its label (and some additional information in Algorithm Helping-Async) to visiting robots, to enable them to navigate the graph.

When multiple robots at a node contend to dock at that node, they invoke a MUTEX(node) call that guarantees that only one robot succeeds in docking. 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. 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 nodes in the 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.

3 Analysis and Bounds

A lower bound of $\Omega(D)$ on the running time was shown in [2]. However, for dispersion on general unrooted graphs, their best running time was $O(m)$. We consider designing space efficient algorithms, subject to a $O(m)$ running time. Observe that DFS based algorithms can run in $O(m)$ time.

A lower bound of $\Omega(\log n)$ bits on the memory of robots was shown in [2]. In analyzing the trade-off between memory and time requirements, we show a different lower bound, subject to the constraint that a solution meets the $O(m)$ bound on the time steps.

Theorem 3.1 *The dispersion problem requires $\Omega(k \log \Delta)$ bits at each robot assuming that a $O(m)$ time steps solution is required.*

Proof We analyze the memory bounds 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 $\Omega(k)$ 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 $\Omega(k \log \Delta)$ bits.

Thus, the overall lower bound on memory at a robot is $\Omega(k \log \Delta)$ bits. ∎

4 Dispersion Using Helping in the Synchronous Model

To achieve dispersion, 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 $visited[r]$ to true if and only if it has encountered robot r after docking. It helps a visiting robot r by communicating to it the value $visited[r]$.

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 r first visits the node, determined using $visited[r]$ of the docked node, the $entry\_port[r]$ entry of the docked robot is set to the entry port used by the visiting robot. The docked robot also communicates $entry\_port[r]$ to a visiting robot r to help it determine whether to backtrack further or resume forward exploration.

A robot uses the following variables: $port\_entered$ and $parent\_ptr$ ($\lceil \log(\Delta + 1) \rceil$ bits each); $port\_entered$ indicates the port through which the robot entered the current node on the latest visit whereas $parent\_ptr$ is a temporary variable 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.

Theorem 4.1 *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 the 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 $4(m - n) + 2n = 4m - 2n$ 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 $4m - 2n$ rounds, observe that the k robots do a collective search of the graph, using individual DFS variants. Within $4m - 2n$ 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 $4m - 2n$ rounds of the algorithm in order to help other robots which might visit this node.

5 Dispersion Using Helping in the Asynchronous Model

Algorithm Helping-Async (Algorithm 2) adapts Algorithm Helping-Sync to an asynchronous system. 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 them to dock. Another implication of an asynchronous system is that a docked robot needs to loop forever, waiting to help any other robot that might arrive at the node later.

Theorem 5.1 *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 4.1. 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. ∎

6 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. An undocked robot maintains the data following additional data structures (i) array of boolean $visited[1, k]$ to determine by checking $visited[r]$ whether it has visited the node where robot r is docked, and (ii) *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 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.

Theorem 6.1 *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 The proof that the running time is $O(m)$ steps is similar to that of Theorem 4.1. 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 (i.e., once it enters $state =$ settled) needs to loop forever, waiting to relay its label to any other robot that might arrive at the node later. Thus, termination is not possible. ∎

It is a straightforward exercise to transform the algorithm into its synchronous version, Independent-Sync. In the synchronous algorithm, a robot can terminate after $4m - 2n$ rounds, as it is guaranteed that every other robot would have found a free node by then.

7 Depth-bounded Independent Dispersion in the Asynchronous Model

Algorithm 4 (Independent-Bounded-Async) is an improvement over Algorithm 3 (Independent-Async). It leverages the idea that a d-depth-bounded DFS 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). Thus, it runs a D-bounded version of Algorithm Independent-Async. However, the algorithm assume that D is known to all the robots.

As in Independent-Async, the action of a docked robot relaying its label to a visiting robot is not explicitly shown in the Algorithm 4 pseudo-code.

In addition to the variables of Algorithm Independent-Async, the variable *depth* ($\lceil \log (D + 1) \rceil$ bits) is used to track the current depth of the robot in the graph exploration.

Theorem 7.1 *If D, the diameter of the graph, is known to all the robots, then Algorithm 4 (Independent-Bounded-Async) achieves dispersion (without termination) in an asynchronous system in $O(m)$ steps with $O(max(D \log \Delta, k))$ bits at each robot.*

Proof The proof that the running time is $O(m)$ steps is similar to that of Theorem 4.1.

From the description and analysis of the variables above, observe that *stack* requires $O(D \log \Delta)$ bits and the $visited[1, k]$ array requires k bits. Thus, it follows that the memory of each robot is bounded by $O(\max(D \log \Delta, k))$ bits.

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

Note that due to the nature of the asynchronous system, a docked robot (i.e., once it enters $state =$ settled) needs to loop forever, waiting to relay its label to any other robot that might arrive at the node later. Thus, termination is not possible. ∎

It is a straightforward exercise to transform the algorithm into its synchronous version, Indep-Bounded-Sync. In the synchronous algorithm, a robot can terminate after $4m - 2n$ rounds, as it is guaranteed that every other robot would have found a free node by then.

8 Conclusions

For the dispersion problem of mobile robots on general graphs, we showed a lower bound of $\Omega(k \log \Delta)$ on the memory of robots, subject to a $O(m)$ running time. We proposed algorithms to achieve this lower bound, in both the synchronous and asynchronous models.

References

[1] C. Ambühl, L. Gasieniec, A. Pelc, T. Radzik, and X. Zhang, "Tree exploration with logarithmic memory," *ACM Trans. Algorithms*, vol. 7, no. 2, 17:1–17:21, 2011. DOI: 10.1145/1921659.1921663. [Online]. Available: http://doi.acm.org/10.1145/1921659.1921663.

[2] J. Augustine and 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, 1:1–1:10. DOI: 10.1145/3154273.3154293. [Online]. Available: http://doi.acm.org/10.1145/3154273.3154293.

[3] E. Bampas, L. Gasieniec, N. Hanusse, D. Ilcinkas, R. Klasing, and 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. DOI: 10.1007/978-3-642-04355-0_44. [Online]. Available: https://doi.org/10.1007/978-3-642-04355-0_44.

[4] L. Barrière, P. Flocchini, E. M. Barrameda, and N. Santoro, "Uniform scattering of autonomous mobile robots in a grid," *Int. J. Found. Comput. Sci.*, vol. 22, no. 3, pp. 679–697, 2011. DOI: 10.1142/S0129054111008295. [Online]. Available: https://doi.org/10.1142/S0129054111008295.

[5] P. Brass, F. Cabrera-Mora, A. Gasparri, and J. Xiao, "Multirobot tree and graph exploration," *IEEE Trans. Robotics*, vol. 27, no. 4, pp. 707–717, 2011. DOI: 10.1109/TRO.2011.2121170. [Online]. Available: https://doi.org/10.1109/TRO.2011.2121170.

[6] P. Brass, I. Vigan, and N. Xu, "Improved analysis of a multirobot graph exploration strategy," in *13th International Conference on Control Automation Robotics & Vision, ICARCV 2014, Singapore, December 10-12, 2014*, 2014, pp. 1906–1910. DOI: 10.1109/ICARCV.2014.7064607. [Online]. Available: https://doi.org/10.1109/ICARCV.2014.7064607.

[7] R. Cohen, P. Fraigniaud, D. Ilcinkas, A. Korman, and D. Peleg, "Label-guided graph exploration by a finite automaton," *ACM Trans. Algorithms*, vol. 4, no. 4, 42:1–42:18, 2008. DOI: 10.1145/1383369.1383373. [Online]. Available: http://doi.acm.org/10.1145/1383369.1383373.

[8] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, no. 2, pp. 279–301, 1989. DOI: 10.1016/0743-7315(89)90021-X. [Online]. Available: https://doi.org/10.1016/0743-7315(89)90021-X.

[9] D. Dereniowski, Y. Disser, A. Kosowski, D. Pajak, and P. Uznanski, "Fast collaborative graph exploration," *Inf. Comput.*, vol. 243, pp. 37–49, 2015. DOI: 10.1016/j.ic.2014.12.005. [Online]. Available: https://doi.org/10.1016/j.ic.2014.12.005.

[10] Y. Disser, F. Mousset, A. Noever, N. Skoric, and A. Steger, "A general lower bound for collaborative tree exploration," *CoRR*, vol. abs/1610.01753, 2016. arXiv: 1610.01753. [Online]. Available: http://arxiv.org/abs/1610.01753.

[11] Y. Elor and A. M. Bruckstein, "Uniform multi-agent deployment on a ring," *Theor. Comput. Sci.*, vol. 412, no. 8-10, pp. 783–795, 2011. DOI: 10.1016/j.tcs.2010.11.023. [Online]. Available: https://doi.org/10.1016/j.tcs.2010.11.023.

[12] P. Fraigniaud, L. Gasieniec, D. R. Kowalski, and A. Pelc, "Collective tree exploration," *Networks*, vol. 48, no. 3, pp. 166–177, 2006. DOI: 10.1002/net.20127. [Online]. Available: https://doi.org/10.1002/net.20127.

[13] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg, "Graph exploration by a finite automaton," *Theor. Comput. Sci.*, vol. 345, no. 2-3, pp. 331–344, 2005. DOI: 10.1016/j.tcs.2005.07.014. [Online]. Available: https://doi.org/10.1016/j.tcs.2005.07.014.

[14] Y. Higashikawa, N. Katoh, S. Langerman, and S. Tanigawa, "Online graph exploration algorithms for cycles and trees by multiple searchers," *J. Comb. Optim.*, vol. 28, no. 2, pp. 480–495, 2014. DOI: 10.1007/s10878-012-9571-y. [Online]. Available: https://doi.org/10.1007/s10878-012-9571-y.

[15] A. Menc, D. Pajak, and P. Uznanski, "On the power of one bit: How to explore a graph when you cannot backtrack?" *CoRR*, vol. abs/1502.05545, 2015. arXiv: 1502.05545. [Online]. Available: http://arxiv.org/abs/1502.05545.

[16] ——, "Time and space optimality of rotor-router graph exploration," *Inf. Process. Lett.*, vol. 127, pp. 17–20, 2017. DOI: 10.1016/j.ipl.2017.06.010. [Online]. Available: https://doi.org/10.1016/j.ipl.2017.06.010.

[17] S. Muthukrishnan, B. Ghosh, and M. H. Schultz, "First- and second-order diffusive methods for rapid, coarse, distributed load balancing," *Theory Comput. Syst.*, vol. 31, no. 4, pp. 331–354, 1998. DOI: 10.1007/s002240000092. [Online]. Available: https://doi.org/10.1007/s002240000092.

[18] M. Shibata, T. Mega, F. Ooshita, H. Kakugawa, and 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. DOI: 10.1145/2933057.2933093. [Online]. Available: http://doi.acm.org/10.1145/2933057.2933093.

[19] R. Subramanian and I. D. Scherson, "An analysis of diffusive load-balancing," in *SPAA*, 1994, pp. 220–225. DOI: 10.1145/181014.181361. [Online]. Available: http://doi.acm.org/10.1145/181014.181361.

[20] Y. Sudo, D. Baba, J. Nakamura, F. Ooshita, H. Kakugawa, and T. Masuzawa, "An agent exploration in unknown undirected graphs with whiteboards," in *Proceedings of the Third International Workshop on Reliability, Availability, and Security, WRAS 2010, conjunction with PODC 2010 Zurich, Switzerland, July 29, 2010*, 2010, p. 8. DOI: 10.1145/1953563.1953570. [Online]. Available: http://doi.acm.org/10.1145/1953563.1953570.

[21] C. Xu and F. C. M. Lau, "Anlaysis of the generalized dimension exchange method for dynamic load balancing," *J. Parallel Distrib. Comput.*, vol. 16, no. 4, pp. 385–393, 1992. DOI: 10.1016/0743-7315(92)90021-E. [Online]. Available: https://doi.org/10.1016/0743-7315(92)90021-E.

[22] V. Yanovski, I. A. Wagner, and A. M. Bruckstein, "A distributed ant algorithm for efficiently patrolling a network," *Algorithmica*, vol. 37, no. 3, pp. 165–186, 2003. DOI: 10.1007/s00453-003-1030-9. [Online]. Available: https://doi.org/10.1007/s00453-003-1030-9.

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 − 2n do
    if state = settled then
        for all other robot j on the node do
            send visited[j] and entry_port[j] to j
            if visited[j] = 0 then
                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 node has a robot j docked in an earlier round then
            seen, parent_ptr ← receive visited[i], entry_port[i] from j
            if seen = 0 then
                parent_ptr ← port_entered; send port_entered to j
    if state = explore then
        if node has a robot j docked in an earlier round then
            if seen = 1 then
                state ← backtrack; move through port_entered
        else
            if i = (r ←)winner(MUTEX(node)) then
                i docks at node; state ← settled
                Initialize visited[1, k] ← \overline{0}; entry_port[1, k] ← \overline{−1}
                for all  robot j on the node do
                    entry_port[j] ← receive port_entered from j
                    visited[j] ← 1
            else
                send port_entered to r
        if state = explore then
            port_entered ← (port_entered + 1) mod degree of node
            if port_entered = parent_ptr then
                state ← backtrack
            move through port_entered
    else if state = backtrack then
        port_entered ← (port_entered + 1) mod degree of node
        if port_entered ≠ parent_ptr then
            state ← explore
        move through port_entered
```

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

```
Initialize: port_entered ← −1; state ← explore; parent_ptr ← −1; seen ← 0
for round = 0, 4m − 2n do
    if round > 0 then
        port_entered, parent_ptr ← entry port; seen ← 0
    if node has a robot j docked then
        seen, parent_ptr ← receive visited[i], entry_port[i] from j
        if seen = 0 then
            parent_ptr ← port_entered; send port_entered to j
    if state = explore then
        if node has a robot j docked then
            if seen = 1 then
                state ← backtrack; move through port_entered
        else
            if i = (r ←)winner(MUTEX(node)) then
                i docks at node; state ← settled
                Initialize visited[1, k] ← 0̄; entry_port[1, k] ← −1̄; break()
            else
                seen, parent_ptr ← receive visited[i], entry_port[i] from r
                if seen = 0 then
                    parent_ptr ← port_entered; send port_entered to r
        port_entered ← (port_entered + 1) mod degree of node
        if port_entered = parent_ptr then
            state ← backtrack
        move through port_entered
    else if state = backtrack then
        port_entered ← (port_entered + 1) mod degree of node
        if port_entered ≠ parent_ptr then
            state ← explore
        move through port_entered
repeat                                                        ▷ state = settled
    for all other robot j that is/arrives at the node do
        send visited[j] and entry_port[j] to j
        if visited[j] = 0 then
            visited[j] ← 1; entry_port[j] ← receive port_entered from j
until true
```

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

Initialize: $port\_entered \leftarrow -1; state \leftarrow explore$; $visited[1,k] \leftarrow \overline{0}$; $stack \leftarrow \perp$
for $round = 0, 4m - 2n$ **do**
    if $round > 0$ **then**
        $port\_entered \leftarrow$ entry port
    if $state = explore$ **then**
        if robot j is docked at node AND $visited[j] = 1$ **then**
            $state \leftarrow backtrack$; move through $port\_entered$
        else if robot j is docked at node AND $visited[j] = 0$ **then**
            $visited[j] \leftarrow 1$
            $push(stack, port\_entered)$
            $port\_entered \leftarrow port\_entered + 1$ mod degree of node
            if $port\_entered = top(stack)$ **then**
                $state \leftarrow backtrack; pop(stack)$
            move through $port\_entered$
        else if node is free **then**
            if $i = (r \leftarrow) winner(MUTEX(node))$ **then**
                i docks at node; $state \leftarrow settled$; break()
            else
                $visited[r] \leftarrow 1$
                $push(stack, port\_entered)$
                $port\_entered \leftarrow port\_entered + 1$ mod degree of node
                if $port\_entered = top(stack)$ **then**
                    $state \leftarrow backtrack; pop(stack)$
                move through $port\_entered$
    else if $state = backtrack$ **then**
        $port\_entered \leftarrow port\_entered + 1$ mode degree of node
        if $port\_entered \neq top(stack)$ **then**
            $state \leftarrow explore$
        else
            $pop(stack)$
        move through $port\_entered$

Algorithm 4 Independent-Bounded-Async, asynchronous execution, code at robot i

```
Initialize: depth ← −1; port_entered ← −1; state ← explore; visited[1, k] ← 0̄; stack ←⊥
for round = 0, 4m − 2n do
    if round > 0 then
        port_entered ← entry port
    if state = explore then
        depth ← depth + 1
        if robot j is docked at node AND visited[j] = 1 then
            state ← backtrack; move through port_entered
        else if robot j is docked at node AND visited[j] = 0 AND depth < D then
            visited[j] ← 1
            push(stack, port_entered)
            port_entered ← port_entered + 1 mod degree of node
            if port_entered = top(stack) then
                state ← backtrack; pop(stack)
            move through port_entered
        else if robot j is docked at node AND visited[j] = 0 AND depth = D then
            visited[j] ← 1; state ← backtrack; move through port_entered
        else if node is free AND depth < D then
            if i = (r ←)winner(MUTEX(node)) then
                i docks at node; state ← settled; break()
            else
                visited[r] ← 1
                push(stack, port_entered)
                port_entered ← port_entered + 1 mod degree of node
                if port_entered = top(stack) then
                    state ← backtrack; pop(stack)
                move through port_entered
        else if node is free AND depth = D then
            if i = (r ←)winner(MUTEX(node)) then
                i docks at node; state ← settled; break()
            else
                visited[r] ← 1; state ← backtrack; move through port_entered
    else if state = backtrack then
        depth ← depth − 1
        port_entered ← port_entered + 1 mode degree of node
        if port_entered ≠ top(stack) then
            state ← explore
        else
            pop(stack)
        move through port_entered
```