Asynchronous Dispersion with Optimal Time Complexity

Anonymous authors

***Abstract*—We study the dispersion problem in anonymous port-labeled graphs: $k \leq n$ mobile agents, each with a unique ID and initially located arbitrarily on the nodes of an n-node graph with maximum degree Δ, must autonomously relocate so that each node hosts at most one agent. Dispersion serves as a fundamental task in the distributed computing of mobile agents, and its complexity stems from key challenges in local coordination under anonymity and limited memory.**

The goal is to minimize both the time to achieve dispersion and the memory required per agent. It is known that any algorithm requires $\Omega(k)$ time in the worst case in arbitrary graphs, and $\Omega(\log k)$ bits of memory per agent. A recent result gives an optimal $O(k)$-time algorithm in the synchronous setting and an $O(k \log \min\{k, \Delta\})$-time algorithm in the asynchronous setting, both using $O(\log(k + \Delta))$ bits.

In this paper, we close the time complexity gap in the asynchronous setting by presenting the first dispersion algorithm that runs in optimal $O(k)$ time using $O(\log(k + \Delta))$ bits of memory per agent. Our solution is based on a novel technique that constructs a port-one tree in anonymous graphs, which may be of independent interest.

I. Introduction

The problem of dispersion of mobile agents studied extensively in recent distributed computing literature not only takes its inspiration from biological phenomena, such as damselfish establishing non-overlapping territories on coral reefs [1], or neural crest cells migrating and distributing themselves across the developing embryo [2]; but also with practical applications such as placing a fleet of small autonomous robots (agents) under shelves (nodes) in fulfillment centers [3]. It is also closely connected to other coordination tasks, such as exploration, scattering, load balancing, and self-deployment [4], [5], [6], [7], [8], [9]. The *dispersion* problem, denoted as DISPERSION, involves $k \leq n$ mobile agents placed initially arbitrarily on the nodes of an n-node anonymous graph of maximum degree Δ. The goal for the agents is to autonomously relocate such that each agent is on a distinct node of the graph (Fig. 1). The objective is to design algorithms that simultaneously optimize time and memory complexities. Time complexity is the time required to achieve dispersion starting from any initial configuration. Memory complexity is the maximum number of bits stored in the persistent memory at each agent throughout the execution. We stress that graph nodes are memory-less and cannot store any information. Fundamental performance limits exist: certain graph topologies (e.g., line graphs) necessitate $\Omega(k)$ time for dispersion. Concurrently, $\Omega(\log k)$ memory bits per agent are required for storing unique identifiers.

DISPERSION has been studied in a series of papers, e.g., [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24] (see Table I). The state-of-the-art is the two recent results due to Kshemkalyani *et al.* [16], one in the synchronous setting and another in the asynchronous setting.

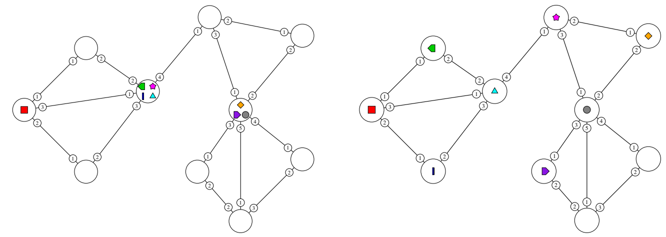


Fig. 1: DISPERSION of 8 mobile agents in a 10-node graph. On the left, 8 agents are initially located at three different nodes. On the right, agents are dispersed to occupy one node each.

Ref.	Memory/agent (bits)	Time (rounds/epochs)	Conf.
any	$\Omega(\log k)$ [14]	$\Omega(k)$ [14]	any
	Synchronous Algorithms		
[10]*	$O(k\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	any
[17]†	$O(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot \log k)$	any
[25]	$O(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot \log k)$	any
[20]	$O(D + \Delta \log k)$	$O(D\Delta(D+\Delta))$	rooted
[26]	$O(\log(k+\Delta))$	$O(k \cdot \log \min\{k, \Delta\})$	rooted
[26]	$O(\Delta)$	$O(k)$	rooted
[26]	$O(\log(k+\Delta))$	$O(k \cdot \log^2 \min\{k, \Delta\})$	any
[16]	$O(\log(k+\Delta))$	$O(k)$	any
	Asynchronous Algorithms		
[14]	$O(k\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	any
[14]	$O(\log(k+\Delta))$	$O(\min\{m, k\Delta\} \cdot k)$	any
[21]	$O(\log(k+\Delta))$	$O(\min\{m, k\Delta\})$	any
[16]	$O(\log(k+\Delta))$	$O(k \cdot \log \min\{k, \Delta\})$	any
This	$\boldsymbol{O(\log(k+\Delta))}$	$\boldsymbol{O(k)}$	**any**

TABLE I: DISPERSION of $k \leq n$ agents on an anonymous n-node m-edge graph of diameter D and maximum degree Δ. *Requires knowledge of m, n. †Requires knowledge of m, k, n, Δ. The best memory cells are highlighted in green, and optimal time cells are highlighted in blue. In a rooted initial configuration, all k agents are initially at a node.

In the *synchronous* setting ($\mathcal{SYNC}$), all agents perform their operations simultaneously in synchronized rounds (or steps), and hence time complexity (of the algorithm) is measured in rounds. However, in the *asynchronous* setting ($\mathcal{ASYNC}$), agents become active at arbitrary times and perform their operations in arbitrary duration, and hence time complexity is measured in epochs – an *epoch* represents the time interval in which each agent completes at least one cycle of execution. In $\mathcal{SYNC}$, an epoch is a round. In particular, the $\mathcal{SYNC}$ algorithm of Kshemkalyani *et al.* [16] has time complexity optimal $O(k)$ rounds and memory complexity $O(\log(k+\Delta))$ bits per agent. Their $\mathcal{ASYNC}$ algorithm [16] has time complexity $O(k \log \min\{k, \Delta\})$ epochs and memory complexity $O(\log(k+\Delta))$ bits per agent.

Contributions. In light of the state-of-the-art [16], the following question naturally arises: *Can optimal $O(k)$-epoch solution be designed for* DISPERSION *in $\mathcal{ASYNC}$?* Such a contribution would complete the picture of optimal solutions to DISPERSION. In this paper, we answer this question in the affirmative. We provide an $O(k)$-epoch solution in $\mathcal{ASYNC}$ with $O(\log(k+\Delta))$ bits per agent. Our results show that the synchrony assumption is irrelevant for time-optimal dispersion.

The result is possible from a novel construction of a special tree, which we call a *Port-One tree* (denoted as P1TREE), that we introduce in this paper. P1TREE prioritizes the edges with port-1 at either of its endpoints. This prioritization helps to find the *empty* neighbor nodes of a node (if exist) to settle agents in $O(1)$ epochs, even in $\mathcal{ASYNC}$ with $O(\log(k+\Delta))$ memory. Kshemkalyani *et al.* [16] were able to do so only in $\mathcal{SYNC}$ and their approach does not extend to $\mathcal{ASYNC}$. Since k agents can settle solving dispersion after visiting k empty nodes and visiting each such empty node takes $O(1)$ epochs, the whole algorithm finishes in optimal $O(k)$ epochs. The P1TREE concept may be useful in solving many other coordination problems in anonymous port-labeled graphs, optimizing time and memory complexities. ***Pseudocodes, many details, and several proofs are omitted due to space constraints.***

Challenges. Existing DISPERSION algorithms largely relied on breadth-first-search (BFS) and depth-first-search (DFS) techniques, with a preference for DFS due to its advantage for optimizing memory complexity along with time complexity. Given k agents in a *rooted initial configuration* (all k agents initially at a node), DFS starts in the forward phase and works alternating between forward and backtrack phases until $k-1$ empty nodes are visited to solve DISPERSION. During a forward phase, one such empty node becomes settled. To visit k different empty nodes, a DFS must perform at least $k-1$ forward phases, and at most $k-1$ backtrack phases. Therefore, the best DFS time complexity is $2(k-1)=O(k)$, which is asymptotically optimal since in graphs (such as a line graph) exactly $k-1$ forward phases are needed in the worst-case (consider the case of all k agents are on either end node of the line graph). Therefore, the challenge is to limit the traversal to exactly $k-1$ forward phases and finish each in $O(1)$ time to obtain $O(k)$ time bound. Suppose DFS is currently at a node. To finish a forward phase at a node in $O(1)$ time, an empty neighbor node (if exists) of that node needs to be found in $O(1)$ time; an empty node is the one that has no agent positioned on it yet. The state-of-the-art result of Kshemkalyani *et al.* [16] developed a technique to find an empty neighbor of a node in $O(1)$ rounds in $\mathcal{SYNC}$ and $O(\log\min\{k,\Delta\})$ epochs in $\mathcal{ASYNC}$. This technique allowed Kshemkalyani *et al.* [16] to achieve $O(k)$-round solution in $\mathcal{SYNC}$ and $O(k\log\min\{k.\Delta\})$-epoch solution in $\mathcal{ASYNC}$ in rooted initial configurations.

In *any initial configuration*, let ℓ be the number of nodes with two or more agents on them, which we call *multiplicity nodes* (for the rooted case, $\ell=1$). There will be ℓ DFSs initiated from those ℓ multiplicity nodes. It may be the case that two or more DFSs *meet*. The meeting situation needs to be handled in a way that ensures it does not increase the time required to find empty neighbor nodes. Kshemkalyani *et al.* [16] handled such meetings in additional time proportional to $O(k)$ in $\mathcal{SYNC}$ and $O(k\log\min\{k,\Delta\})$ in $\mathcal{ASYNC}$ using the size-based subsumption technique of Kshemkalyani and Sharma [21]. This allowed Kshemkalyani *et al.* [16] to achieve an $O(k)$-round solution in $\mathcal{SYNC}$ and an $O(k\log\min\{k,\Delta\})$-epoch solution in $\mathcal{ASYNC}$ for any initial configuration.

To have an $O(k)$-epoch solution in $\mathcal{ASYNC}$ (for any initial configuration), we need a technique to find an empty neighbor node (if exists) of a node in $O(1)$ epochs. A natural direction is to explore whether Kshemkalyani *et al.* [16]'s technique can be extended to $\mathcal{ASYNC}$. The major obstacle to doing so is the technique of *oscillation* used in [16]. To have sufficient agents to explore neighbors in $O(1)$ time, Kshemkalyani *et al.* [16] leave $\lceil k/3\rceil$ nodes visited by DFS empty, which we call *vacated* nodes. The $\lceil k/3\rceil$ agents that were supposed to settle at those vacated nodes are then used in neighborhood search. This approach created one problem: while probing at a node, how to differentiate an empty neighbor node from a vacated neighbor. Kshemkalyani *et al.* [16] overcame this difficulty as follows. They selected vacated nodes in such a way that there is an occupied node (with an agent on it) that makes an oscillation trip of length 6 in time 6 rounds to cover the vacated nodes assigned to it. The probing agent waits at the (possibly empty) neighbor for 6 rounds before returning. While waiting at a vacated node, it is guaranteed that an oscillating agent reaches that vacated node during those 6 rounds. For an empty node, no such agent reaches that node. It is easy to see why this approach does not extend to $\mathcal{ASYNC}$: Making decision by probing agents on how long to wait at an empty neighbor for an oscillating agent to arrive (or not arrive) is difficult. This is because, in $\mathcal{ASYNC}$, agents do not have agreement on duration and the start/end of each computational step (i.e., no agreed-upon round definition).

The novel construction of P1TREE in this paper allowed us to obviate the need of oscillation to differentiate empty neighbor nodes (if exist) from the vacated neighbor nodes. The technique also guarantees that the forward phase at a node can finish in $O(1)$ epochs. A naive implementation of P1TREE, however, needs $O(\Delta)$ bits. We improve it to $O(\log(k+\Delta))$ bits through a clever approach. We now describe our technique behind P1TREE construction and the challenges we have overcome.

Techniques. While Kshemkalyani *et al.* [16] guaranteed $\lceil k/3\rceil$ agents for probing, we guarantee at least $\lceil (k-2)/3\rceil$ agents for probing neighbor nodes (which we call *parallel probing*); and show that it is sufficient for $O(1)$ time neighborhood search. We make $\lceil (k-2)/3\rceil$ agents available by selectively vacating certain nodes of P1TREE after an agent settles. This selection typically vacates nodes for which the port-1 neighbor or the port-1 neighbor of a port-1 neighbor is not vacated. While having previously settled agents helps in $O(1)$ time parallel probing, guaranteeing their availability and not using oscillation creates five major challenges Q1–Q5 below. We need some notations. Each agent has a states *settled, unsettled, settledScout*. Initially, all k agents are *unsettled*, and they travel with the DFShead. At every new node, one agent settles. For a node v, we denote the settled agent at that node by $\psi(v)$. The settled agent $\psi(v)$ may not always remain at v.

Q1. How to run DFS? We run DFS such that it constructs a P1TREE, which primarily consists of edges containing port number 1 at one or both ends (which we call *port-1-incident* edges). Therefore, the DFS at a node prioritizes visiting empty neighbors reached via port-1-incident edges. This priority may create a cycle when a port-1-incident edge takes the DFS to a node that is already part of a P1TREE built so far. We avoid such cycles by adding an edge which is a non-port-1-incident edge. Additionally, we guarantee that the DFS will never add two consecutive non-port-1-incident edges.

Q2. Which P1TREE nodes to leave vacant? We guarantee that we can leave at least $\lfloor l/3 \rfloor$ nodes of P1TREE of size l vacant. The settled agents at these $\lfloor l/3 \rfloor$ vacant nodes travel with the DFShead and help with parallel probing until DFS ends. The challenge is how to meet such a requirement. The general rule of thumb for this decision is as follows. Consider a P1TREE node v. If v has a port-1-neighbor or port-1 neighbor of port-1 neighbor that is occupied (is not vacant/empty), leave v vacant and collect the agent $\psi(v)$ as a scout (we call v a *vacated* node).

Q3. How and where to keep information about vacated nodes of the P1TREE? There are two options: (i) Store the information of vacant node at its occupied port-1 neighbor or (ii) Store information about the port-1 neighbor at the agent $\psi(v)$. Option (i) is problematic since port-1 neighbor of v may also be the port-1 neighbor of multiple nodes and hence the memory need becomes at least $O(\Delta)$ bits. We use Option (ii) such that each agent only keeps track of one port-1 neighbor, using $O(\log(k+\Delta))$ bits. $\psi(v)$ stores the ID of port-1 neighbor, and the port at port-1 neighbor, so that later in parallel probing, v can be correctly identified as vacated.

Q4. How to successfully run DFS despite some of the P1TREE nodes vacant? Suppose a node $v \in G$. If v is in P1TREE, it is either occupied or vacated. If v is not in P1TREE, it is empty. Suppose a scout agent a_s doing parallel probing from x reaches x's neighbor node y. If a_s finds y empty, it checks the port-1 neighbor z of y. If z is occupied, a_s returns to x. If z is not occupied, it visits port-1 neighbor w of z. After visiting w, a_s returns to x. Due to our strategy of choosing vacant nodes, if y is not empty, then z or w (or both) must be occupied. Even if y and z are vacant, the scout agent can always determine the settled agent at y and z by checking if such agents exist in the scout pool at x. Notice that scout a_s visits a (at most) 3-hop neighbor of w in parallel probe starting from w, and hence each parallel probe finishes in $6 = O(1)$ epochs. The probe needs to search at most $k-2$ ports (excluding parent port) at a non-root node, and the DFShead has at least one unsettled agent. Thus, $\lceil (k-2)/3 \rceil$ scouts, probing at a node, finish searching in 3 iterations, taking at most $18 = O(1)$ epochs. Furthermore, when the degree is more than $k-2$, all unsettled agents can settle by finding empty neighbors with one instance of parallel probing.

Q5. How to return scout agents to the vacated nodes of the P1TREE after DFS finishes? After having k nodes in P1TREE, the DFS finishes. Notice that each scout is associated with a node of P1TREE. We ask each scout to carry information about parent/child/sibling details (both ID and port). This information allows the scouts to re-traverse P1TREE in the reverse-order of DFS and settle at their associated node when reached. We prove that this re-traversal process finishes in $O(k)$ epochs with $O(\log(k+\Delta))$ bits at each scout.

Handling Any initial configuration. So far, we discussed techniques to achieve $O(k)$ time complexity for rooted initial configurations. In any initial configuration, there will be ℓ DFSs initiated from ℓ multiplicity nodes (ℓ not known). Each DFS follows the approach as in the rooted case. Let a node has k_1 agents running DFS D_1. We show that D_1 finishes in $O(k_1)$ epochs if D_1 does not meet any other DFS, say D_2. In case of a meeting, we develop an approach that handles the meeting of two DFSs D_1 and D_2 with overhead the size of the larger DFS between the two. In other words, k_1+k_2 agents that belong to D_1 and D_2 disperse in $O(k_1+k_2)$ epochs. If a meeting with the third DFS D_3 occurs, we show that it is handled with time complexity $O(k_1+k_2+k_3)$ epochs. Therefore, the worst-case time complexity starting from any ℓ multiplicity nodes becomes $O(k)$. Specifically, to achieve this runtime, we extend the *size-based subsumption* technique of Kshemkalyani and Sharma [21]. The subsumption technique works as follows. Suppose DFS D_1 meets DFS D_2 at node w (notice that w belongs to D_2). Let $|D_i|$ denote the number of agents settled from DFS D_i (i.e., the number of nodes in P1TREE T_{D_i} built by D_i so far). D_1 subsumes D_2 if and only if $|D_2| < |D_1|$, otherwise D_2 subsumes D_1. The agents settled from subsumed DFS (as well as scouts) are collected and given to the subsuming DFS to continue with its DFS, which essentially means that the subsumed DFS does not exist anymore. This technique guarantees that one DFS out of ℓ' met DFSs (from ℓ nodes) always remains subsuming and grows monotonically until all agents settle forming a single P1TREE.

Related work. Table I reviews the state-of-the-art time– and memory-efficient solutions for DISPERSION as well as lists our contribution. Our closely related works are [10], [27], [28], [29], [11], [12], [13], [30], [14], [17], [18], [19], [21], [22], [31], [32], [25]. The detailed discussion is omitted due to space constraints.

II. Model and Preliminaries

Graph. We consider a simple, undirected, connected graph $G=(V,E)$, where $n=|V|$ is the number of nodes and $m=|E|$ is the number of edges. For any node $v \in V$, let $N(v)$ denote the set of its neighbors and let $\delta_v = |N(v)|$ denote its degree. The maximum degree of the graph is $\Delta = \max_{v \in V} \delta_v$. Graph nodes are *anonymous* (i.e., they lack unique identifiers). However, the graph is *port-labeled*: at each node v, the incident edges are assigned distinct local labels (port numbers) from 1 to δ_v, enabling agents at v to distinguish between the outgoing edges. The port number at u for an edge $\{u,v\}$ is denoted by p_{uv}. An edge $\{u,v\}$ is associated with two port numbers: p_{uv} at node u and p_{vu} at node v. These port numbers are assigned locally and independently at each endpoint; hence, it is possible that $p_{uv} \neq p_{vu}$, and there is no inherent correlation between port assignments at different nodes. Each node $u \in V$ is memory-less.

Agents. The system comprises $k \leq n$ mobile agents, $A = \{a_1, \ldots, a_k\}$. Each agent a_i is endowed with a unique positive integer identifier, a_i.ID, drawn from the range $[1,\, k^{O(1)}]$. Since agents are assumed to be positioned arbitrarily initially, there may be the case that all $k \leq n$ agents are at the same node, which we denote as *rooted initial configuration*. Any initial configuration that is not rooted is denoted as *general*. In any general initial configuration, agents are on at least two nodes. A special case of general configuration is a *dispersion configuration* in which k agents are on k different nodes. We consider the *local* model in which an agent at a node can only communicate with other agents co-located at that node.

Time cycle. An agent a_i could be active at any time. Upon activation, a_i executes the "Communicate-Compute-Move" (CCM) cycle as follows. **Communicate:** Agent a_i positioned at node u can observe the memory of another agent a_j positioned at node u. Agent a_i can also observe its own memory. **Compute:** Agent a_i may perform an arbitrary computation using the information observed during the "Communicate" portion of that cycle. This includes the determination of a port to use to exit u and the information to store in the agent a_j located at u. **Move:** At the end of the cycle, a_i writes new information in its memory as well as in the memory of an agent a_j at u, and exits u using the computed port.

Round, epoch, time, and memory complexity. In $\mathcal{SYNC}$, all agents have a common notion of time and activate in discrete intervals called *rounds*. In $\mathcal{ASYNC}$, agents can have arbitrary activation times and can activate at arbitrary frequencies. The restriction is that every agent is active infinitely often, and each cycle finishes in finite time. An *epoch* is a minimal interval within which each agent finishes at least one CCM cycle [33]. Formally, let t_0 denote the start time. Epoch $i \geq 1$ is the time interval from $t_i - 1$ to t_i where t_i is the first time instant after $t_i - 1$ when each agent has finished at least one complete CCM cycle. Therefore, for $\mathcal{SYNC}$, a round is an epoch. We will use the term "time" to mean rounds for $\mathcal{SYNC}$ and epochs for $\mathcal{ASYNC}$. Memory complexity is the number of bits stored at any agent over one CCM cycle to the next. The temporary memory needed during the Compute phase is considered free.

Some terminologies. Throughout the paper, we encode an edge $\{u,v\}$ by the 4-tuple

$$e \;=\; [u,\, p_{uv},\, p_{vu},\, v], \qquad 1 \leq p_{uv} \leq \delta_u,\; 1 \leq p_{vu} \leq \delta_v,$$

where u, v are the (anonymous) end-nodes and p_{uv} (resp. p_{vu}) is the port number of e at u (resp. v). We define type of edge $\{u,v\}$ at u as follows:

- `t11`, if $p_{uv} = 1 = p_{vu}$
- `tp1`, if $p_{uv} \neq 1 = p_{vu}$
- `t1q`, if $p_{uv} = 1 \neq p_{vu}$
- `tpq`, if $p_{uv} \neq 1 \neq p_{vu}$.

We denote the type of an edge $\{u,v\}$ by $\texttt{type}(\{u,v\}) = \texttt{type}(p_{uv}, p_{vu}) \in \{\texttt{tp1}, \texttt{t11}, \texttt{t1q}, \texttt{tpq}\}$.

III. Port-One Tree and its Construction

In this section, we first define the port-one tree (P1Tree) and discuss its centralized and distributed construction. Then we describe a method of selecting nodes to be "vacated" on the constructed P1Tree. We finally discuss the parallel probing technique.

Port-One Tree (P1Tree)**.** Intuitively, every vertex in a P1Tree $\mathcal{T}$ is incident to at least one tree edge carrying port 1 at one of its end-points. Formally,

Definition 1 (Port-One Tree (P1Tree))**.** *Let $G = (V, E)$ be an anonymous port-labeled graph. A tree $\mathcal{T} \subseteq E$ is a* P1Tree *if each vertex $v \in \mathcal{T}$ has (at least) one incident edge leading to some node w such that edge $\{v,w\} \in \mathcal{T}$ and $\texttt{type}(\{v,w\}) \in \{\textit{tp1}, \textit{t11}, \textit{t1q}\}$.*

There may be multiple trees $\mathcal{T}$ that satisfy Definition 1 of a P1Tree. Let that set of trees $\mathcal{T}$ be denoted as $\mathbf{T}$. In this paper, we are interested in constructing a P1Tree $\mathcal{T} \in \mathbf{T}$. We will prove later that there exists at least a P1Tree $\mathcal{T} \in \mathbf{T}$ for any graph G. A P1Tree $\mathcal{T}$ is shown in Fig. 2. Notice

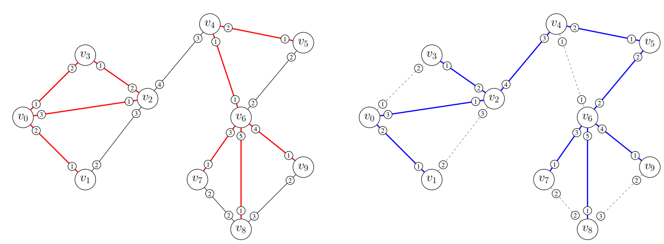


Fig. 2: An example of a Port-One Tree. Left: Edges incident to a port 1 are highlighted in red. Right: Tree edges (blue, solid) shown with non-tree edges (gray, dashed).

that a P1Tree $\mathcal{T}$ may not contain all the edges of type `tp1`, `t11`, or `t1q` because doing so may create cycles. We avoid cycles by adding edges of type `tpq`.

Since each node $v \in G$ has degree $\delta_v \geq 1$, Observation 1 follows immediately.

Observation 1. *Each node in a port-labeled graph G has an edge of type* `t11` *or* `t1q`*.*

Lemma 1. *For any port-labeled graph G, there exists at least one* P1Tree $\mathcal{T}$*.*

Centralized construction. The pseudocode is omitted due to space constraints (we name our algorithm `Centralized_P1Tree()`). Initially, $\mathcal{T}$ is empty, i.e., $\mathcal{T} \leftarrow \emptyset$. Let $\mathcal{C}_i$ be a tree component initially empty, i.e., $\mathcal{C}_i \leftarrow \emptyset$. The goal is construct $\kappa \geq 1$ (κ not known) tree components $\mathcal{C}_1, \ldots, \mathcal{C}_\kappa$ such that $\mathcal{C}_1 \cap \ldots \cap \mathcal{C}_\kappa = \emptyset$ and $\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_\kappa = V$. The κ components are then connected via $\kappa - 1$ edges to obtain $\mathcal{T}$. The algorithm starts from an arbitrary node $v \in G$. It adds in $\mathcal{C}$ (initially empty) all the incident edges of v of types `tp1`, `t11`, and `t1q` one by one in the priority order. While doing so, each neighboring node is added in a stack. If all such edges at v are exhausted, then it goes to a neighbor (top of stack) and repeats the procedure. If adding an edge $\{w, z\}$ of type `tp1`, `t11`, or `t1q` at node w creates a cycle, we set $\mathcal{C}_1 \leftarrow \mathcal{C}$. The algorithm then continues constructing $\mathcal{C}$ on $G \backslash \mathcal{C}_1$ until a cycle is detected. It then sets $\mathcal{C}_2 \leftarrow \mathcal{C}$. The component construction stops as soon as stack goes empty, which also means that $G \backslash \{\mathcal{C}_1 \cup \ldots \cup \mathcal{C}_\kappa\} = \emptyset$. We then include all these components in $\mathcal{T}$. We then connect these κ tree components adding $k - 1$ edges as follows: we sort the edges of G that are not yet considered to construct $C_1, \ldots, C_\kappa$ in lexicographical order. These edges must be of type `tpq`. We then add these edges to $\mathcal{T}$ as long as adding the edge does not create a cycle. The algorithm terminates when no more edges can be added. This also means that there is only one component containing all the nodes of G.

Lemma 2. *Given a port-labeled graph G, Algorithm 1 correctly constructs a* P1TREE $\mathcal{T}$.

DFS-based construction. DFS explores G through forward and backtrack phases, switching between them as needed. The forward phase takes it as deeply as possible along each branch, and the backtrack phase helps in finding nodes from which the forward phase can continue again. Let *DFShead* be the node where DFS is currently performing the forward or backtrack phase. Initially, the starting node of DFS acts as the DFShead.

Each node $v \in G$ has the following two states:

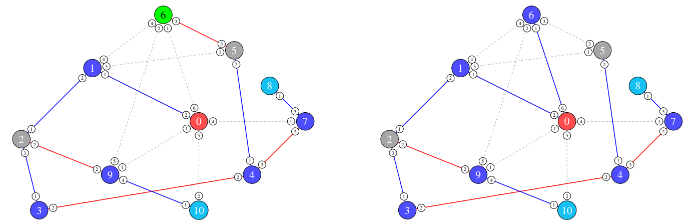
- EMPTY: v has not been visited by the DFShead yet.
- OCCUPIED: v has been visited by the DFShead already.

We denote by *parent edge* of $v \in \mathcal{T}$ the edge in $\mathcal{T}$ from which DFS first visited v during the forward phase. We call the associated type of the edge as *parent edge type*. Moreover, we categorize each node $v \in G$ into the following four types:

- **unvisited**: v is not yet visited by the DFShead.
- **fullyVisited**: v is visited by the DFShead and v has no empty neighbors.
- **partiallyVisited**: v has a parent edge of type `tpq` and each empty neighbor is reached by an edge of type `tpq`.
- **visited**: v is visited by the DFShead and v is not **partiallyVisited** or **fullyVisited**.

We overload the `type()` function to indicate the type of a node u in $\{$**unvisited**, **partiallyVisited**, **fullyVisited**, **visited**$\}$.

Algorithm. We now describe the algorithm to construct a P1TREE $\mathcal{T} \in \mathbf{T}$ (we call our algorithm `DFS_P1Tree()`; the pseudocode is omitted due to space constraints). `DFS_P1Tree()` prioritizes the edge types at any node $v \in V$ in the following order: `tp1` $\succ$ `t11` $\sim$ `t1q` $\succ$ `tpq` (for multiple edges of same type at v, they are prioritized in the increasing order of the port number at v).



(a) DFS backtracks at 6 marking it **partiallyVisited** since parent edge $\{6, 5\}$ is of type `tpq` and its only **unvisited** neighbor 9 is connected via a `tpq` edge $\{6, 9\}$.

(b) Reconfiguration of the P1Tree $\mathcal{T}$ for a **partiallyVisited** node removing the `tpq` edge $\{6, 5\}$ and adding the edge $\{0, 6\}$ of type `tp1`.

Fig. 3: An illustration of reconfiguration on a P1TREE $\mathcal{T}$ constructed so far, swapping a `tpq` edge by an edge of type `tp1` or `t11`.

Suppose DFShead reaches node u. We determine the type of node u by checking $N(u)$. Based on the type of u, the DFShead does the following:

(D0) all nodes are initially **unvisited**.
(D1) if u is **fullyVisited**, then DFShead backtracks to parent of u.
(D2) if u is **visited**, then DFShead continues along the highest priority edge to an empty neighbor of u.

`DFS_P1Tree()` terminates when all nodes of G become **fullyVisited**. The rules (D0), (D1), and (D2) are analogous to the standard DFS traversal with a major change that introduces **partiallyVisited** node types to ensure that each node of $\mathcal{T}$ satisfy the P1TREE property. In particular, `DFS_P1Tree()` has the following additional rules:

(D3) if u is **partiallyVisited**, then DFShead backtracks to parent of u.
(D4) a **partiallyVisited** node u has state EMPTY when DFShead is at a node w such that $\texttt{type}(\{w, u\} \in \{\texttt{tp1}, \texttt{t11}\}$; otherwise, OCCUPIED.

`DFS_P1Tree()` converts a **partiallyVisited** node u to a **visited** node when DFShead visits u from w such that $p_{uw} = 1$ (i.e., w is the port-1 neighbor of u). Additionally, the parent edge of u now *swapped* to make w the parent of u in $\mathcal{T}$. We prove later that this parent swap does not create a cycle. We call this process the *reconfiguration* of a **partiallyVisited** node.

Fig. 3 illustrates these ideas. As shown in Fig. 3a, the DFS backtracks at 6 marking 6 **partiallyVisited**, since the parent edge $\{6, 5\}$ is of type `tpq` and its only **unvisited** neighbor 9 is connected via a `tpq` edge $\{6, 9\}$. As shown in Fig. 3b, when DFS reaches again to 6 in the forward phase via edge $\{0, 6\}$, parent of 6 is now swapped to 0 (i.e., edge $\{6, 5\}$ is removed and edge $\{0, 6\}$ is added), which makes 6 **visited**.

Theorem 1. *`DFS_P1Tree()` produces a* P1TREE $\mathcal{T}$ *of a port-labeled graph G.*

IV. Constructing P1Tree with Agents

Now, we describe how Algorithm 2 (`DFS_P1Tree()`) can be executed by agents. We first describe a straightforward (non-optimal) construction of the P1Tree, and then show the utilization of certain structural properties of P1Tree to construct it in optimal time.

High-level overview. Suppose agents are initially located at a node v_0. The highest ID agent settles at v_0, which keeps track of the state and type of node v_0. Now the agents perform a neighborhood search to determine the edge types and choose the highest priority edge leading to an empty node. This edge is chosen as the next edge to be included in the DFS tree. The neighborhood search means that the neighbors of the current node are visited one by one, and the settled agents in the neighborhood nodes indicate the state and type of the nodes. On reaching an unvisited node, a new agent (the highest ID among the unsettled agents) settles. It sets its parent to the node where the DFShead arrived from. The result of the neighborhood search determines the type and state of the node. When no empty neighbor is available, the DFShead travels back to the parent node of the current node. Analogous to Algorithm 2 (`DFS_P1Tree()`), DFShead at x moves in the forward phase when the neighbor y is **partiallyVisited** and $p_{yx} = 1$. The settled agent at y, changes its type and parent.

Preliminaries. Consider there are n agents initially located at a node v_0 of the n-node graph G. We call v_0 the root node. We draw analogies from the node states and types to describe the agent states and variables. Then we show how we maintain the same information using agents. Initially, all agents are in state *unsettled*. We say an agent a is settled at node x, by setting the state of agent a to *settled*. Recall that the nodes are anonymous and hence there is no identifier associated with them. We associate the ID of the agent settled at a node as a proxy for the ID of the node. We represent the settled agent at node x as $\psi(x)$. Similarly, an agent a at x can identify the type of the edge e_{xy} after traversing it, since it can only know the port number p_{yx} after reaching y. When agent a gets settled at node x, it also stores the type of the node x in variable $a.\mathsf{nodeType}$. Initially, $a.\mathsf{nodeType}$ is **unvisited**, but the agent must obtain its correct node type by doing a neighborhood search.

Initialization. The DFShead is initially located at v_0. The highest ID agent among all the agents present settles at v_0. Let a_0 be that agent. We say $\psi(v_0) = a_0$. The agent a_0 changes its state $a_0.\mathsf{state}$ to *settled*. The agents construct the tree by keeping track of their parent node. Initially, $a_0.\mathsf{parentID} = \perp$, $a_0.\mathsf{parentPort} = \perp$, and $a_0.\mathsf{portAtParent} = \perp$.

Neighborhood search. Similar to Algorithm 2 (`DFS_P1Tree()`), the DFShead must find the next edge to travel according to the priority order. To determine the next edge in priority order, the agents at x traverse the ports in increasing order. They set their phase to *probe*, and traverse an edge to determine the state and type. On traversing an edge corresponding to port p_{xy}, the agents determine the result as a 4-tuple $\langle p_{xy}, \mathtt{type}(\{x,y\}), \mathtt{type}(y), \psi(y)\rangle$. Notice that all of the constituents of the 4-tuple can be determined locally, since when y is **unvisited**, then $\psi(y) = \perp$. After determining the results for all the ports incident at x, the agents choose the next edge to traverse accordingly.

Handling partiallyVisited nodes. In Algorithm 2 (`DFS_P1Tree()`), a node is **partiallyVisited** when its parent edge type is `tpq`, and there are empty neighbors connected by `tpq` edges. The agents can determine this locally after performing a neighborhood search, and then a node w is marked **partiallyVisited** by retaining this information at $\psi(w).\mathsf{nodeType}$. Then the unsettled agents leave via $\psi(w).\mathsf{parentPort}$ to reach the parent of w.

On the other hand, when the highest priority result during neighborhood search is the tuple $\langle p_{xw}, \mathtt{tp1/t11}, \textbf{partiallyVisited}, \psi(w)\rangle$, the agents move to w via $p_{wx} = 1$, and thus $\psi(w)$ updates its parameters to $\psi(w).\mathsf{parent} = (\psi(x).\mathsf{ID}, p_{xw})$, $\psi(w).\mathsf{nodeType} =$ **visited** and $\psi(w).\mathsf{parentPort} = 1$.

Termination. Similar to Algorithm 2 (`DFS_P1Tree()`), the termination happens when all nodes are of type **fullyVisited**. The last agent that settles has to perform the neighborhood search by itself and determine that it has no empty neighbors left, marking it **fullyVisited**. Then the DFShead traverses the tree back to the root, performing neighborhood search at each node to ensure that it becomes **fullyVisited**, and reconfigures **partiallyVisited** neighbors. Note that the port-one neighbor of a **partiallyVisited** node is marked **visited**. It can only be marked **fullyVisited** when the **partiallyVisited** node is reconfigured to become a **visited** node. Hence, on termination, no **partiallyVisited** nodes remain.

Time complexity optimization. This straight-forward algorithm takes $2\delta_x$ epochs to perform a neighborhood search at x. However, when multiple agents are present at x, they can always visit the ports at x in parallel to determine the result corresponding to a port. In Sections IV-A and IV-B, we present two methods that go hand-in-hand for performing $O(1)$-epoch neighborhood search at a node x. First, some of the nodes are chosen to be VACATED such that the settled agents at those nodes could travel with the DFShead to perform the neighborhood search. Second, even in the absence of settled agents at VACATED nodes, the parallel probe can correctly determine that it is VACATED.

A. Selecting Vacant Nodes

Here, we describe how vacant nodes are chosen. Once a vacant node w is chosen, the state of w becomes VACATED, and the settled agent $\psi(w)$ travels with the DFShead instead of remaining at w. However, the agent $\psi(w)$ needs to collect certain information before it can travel with the DFShead. Informally, the condition for designating a **visited** node x to be VACATED is that, the port-1 neighbor w must be OCCUPIED in P1Tree $\mathcal{T}$ constructed so far. The **fullyVisited** and **partiallyVisited** nodes are always VACATED.

Detailed description. Consider an execution of Algorithm 2 (`DFS_P1Tree()`). Suppose the DFShead is located at x. Let z be the parent of x. On arrival at x, the DFShead determines the state of node x. Let w be the port-1 neighbor of x, i.e, $p_{xw} = 1$. The settled agent $\psi(x)$ stores the information about its port-1 neighbor using the variables: $\psi(x)$.P1Neighbor $= \psi(w)$ ($\psi(w)$ is $\perp$ when w is EMPTY) and $\psi(x)$.portAtP1Neighbor $= p_{wx}$. Formally,

(V1) the root node is always OCCUPIED.
(V2) x is VACATED, if w is OCCUPIED and x is **visited**.
(V3) x is VACATED, if it is **fullyVisited** and $\psi(x)$.vacatedNeighbor $= false$.
(V4) x is VACATED, if it is **partiallyVisited**.
(V5) z is VACATED, if $\psi(z)$.vacatedNeighbor $= false$, and $p_{zx} = 1$.

When x is VACATED, the DFShead moves to w to assign $\psi(w)$.vacatedNeighbor $= true$. Notice that, x is vacated when $p_{xz} = 1$ (by (V2)), and thus making $\psi(z)$.vacatedNeighbor $= true$. Then, even if $p_{zx} = 1$, rule (V5) is not applicable anymore. This shows the priority order among the rules, and they are applicable in that priority order. The pseudocode is provided in Algorithm 3 `Can_Vacate()`, which returns the state *settledScout* for the agent if the node has state VACATED.

Fig. 3a illustrates these ideas. The blue/green nodes are vacant, and the gray nodes are occupied. Node ⓪ is OCCUPIED since it is the root (by (V1)). Nodes ①, ③, and ⑨ are vacant since their port-1 neighbor ⓪ is OCCUPIED (by (V2)). Nodes ⑧, and ⑩ are VACATED since they are **fullyVisited** and do not have any dependent VACATED neighbors (by (V3)). Node ⑥ is VACATED since it is **partiallyVisited** (by (V4)). Node ④ is VACATED only after DFShead reaches ⑤, and ⑤ cannot be vacant since port-1 neighbor of ⑤ (node ①) is VACATED already (by (V5)).

Lemma 3. *Consider three consecutive nodes v_1, v_2, and v_3 visited by the DFShead. Suppose v_1 was visited for the first time. Then at least one of v_1, v_2, and v_3 is* VACATED.

Proof. We prove this by contradiction. For the sake of contradiction, assume that v_1, v_2, and v_3 are OCCUPIED. The edges traversed by the DFShead are $\{v_1, v_2\}$ and $\{v_2, v_3\}$. We have the following cases.

- If v_2 is the parent of v_1, then DFShead is backtracking from v_1, which implies, v_1 is either **fullyVisited** or **partiallyVisited**. Then v_1 is VACATED by rules (V3) or (V4).
- If $v_1 = v_3$, then when the DFShead is at v_2, it backtracks, which implies that v_2 is either **fullyVisited** or **partiallyVisited**. Then v_2 is VACATED by rules (V3) or (V4).
- Otherwise, there is a parent-child path $v_1 \to v_2 \to v_3$, such that v_2 is a child of v_1. Now, v_2 must be **visited**. Let u_2 be the port-1 neighbor of v_2.
  - If $u_2 = v_1$, then v_2 is VACATED by rule (V2) when DFShead reaches v_2.
  - If $u_2 = v_3$, then u_2 is EMPTY when DFShead reaches v_2 and `type`$(\{v_2, v_3\})$ is either `t11` or `t1q`. Since v_2 was EMPTY before the first visit of DFShead, thus $\psi(v_2)$.vacatedNeighbor must be $false$. Thus, v_2 is VACATED by rule (V5) once DFShead reaches v_3 if `type`$(\{v_2, v_3\}) =$ `t1q` or v_3 is VACATED by rule (V2) if `type`$(\{v_2, v_3\}) =$ `t11`.
  - If $u_2 (\neq v_3)$ is EMPTY, and since the DFShead moves according to the edge priority, `type`$(\{v_2, v_3\})$ must be `tp1`, i.e., $p_{v_3 v_2} = 1$. Then v_3 would be VACATED by rule (V2).
  - If v_1 is the root, then v_1 remains OCCUPIED by rule (V1), however, u_2 must be EMPTY since $u_2 \neq v_1$. This falls in the case of u_2 is EMPTY.
  - Consider $u_2 \neq v_1$ and $u_2 \neq v_3$. If u_2 is VACATED, we have the following cases.
    * If `type`$(\{v_1, v_2\})$ is `t1q`, then once DFShead reaches v_2, rule (V5) is applicable to v_1 and is VACATED. This is possible since v_1 was EMPTY before the first visit of DFShead and thus $\psi(v_1)$.vacatedNeighbor is $false$.
    * If `type`$(\{v_1, v_2\})$ is `tpq`, then `type`$(\{v_2, v_3\})$ cannot be `tpq`, because then `type`(v_2) would be **partiallyVisited**. When `type`$(\{v_2, v_3\})$ is either `t11` or `t1q`, it is the case $u_2 = v_3$. If `type`$(\{v_2, v_3\})$ is `tp1`, then v_3 is VACATED once DFShead reaches v_3 by rule (V2).

In each of the cases, we obtain a contradiction by showing at least one of v_1, v_2 or v_3 is VACATED. Hence proved. □

Lemma 4. *Let $\mathcal{T}$ be the partial tree constructed by Algorithm 2 (`DFS_P1Tree()`) at a given moment, and let $k = |\mathcal{T}|$ be the number of vertices in $\mathcal{T}$. Then at least $\lfloor k/3 \rfloor$ of these k vertices are in state* VACATED.

Proof. We define $v_i \prec v_j$, if v_i is visited before v_j by the DFShead for the first time. Consider the vertices in this order as $D_v = (v_1, v_2, \ldots, v_k)$, where $v_i \prec v_{i+1}$ for $1 \le i < k$. Let v_i' and v_i'' be the two subsequent nodes visited by DFShead immediately after v_i. Now, we define $B(v_i)$ be the corresponding block of v_i comprising of nodes in (v_i, v_i', v_i'') if they appear after v_i in the DFS order. Specifically, if $v_i' \prec v_i$, then v_i' does not belong to $B(v_i)$; likewise for v_i''. We say $B(v_i)$ is degenerate if $|B(v_i)| < 3$. By default, $|B(v_i)| \ge 1$ for all $1 \le i \le k$, since $v_i \in B(v_i)$. If $B(v_i)$ is not degenerate, then it contains v_i, v_{i+1}, and v_{i+2}.

We determine a subsequence $S_v = (\bar{v}_0, \ldots, \bar{v}_l)$ of D_v iteratively, (i) $v_0 = \bar{v}_0$; (ii) $\bar{v}_j = v_i$ where v_i is the first element in $D_v \setminus \bigcup_{\alpha=0}^{j-1} B(\bar{v}_\alpha)$. We have $l \ge \lceil k/3 \rceil$, since $B(v_i) \le 3$. When $B(\bar{v}_j) < 3$ for $j < l$, then the DFShead backtracks from $\bar{v}_j$, i.e., $\bar{v}_j$ is VACATED. Also, when $B(\bar{v}_j) = 3$, at least one node in $B(\bar{v}_j)$ is VACATED by Lemma 3.

Finally, consider $B(\bar{v}_l)$. If $B(\bar{v}_l) = 3$, then we have a VACATED node in $B(\bar{v}_l) = 3$. If $B(\bar{v}_l) < 3$, then there may not be VACATED nodes in $B(\bar{v}_l)$. Overall, we have at least $l-1$ VACATED nodes, where $l - 1 = \lfloor k/3 \rfloor$. Hence proved. □

B. Parallel Probing

Now, we present the core technique on which our DISPERSION algorithm hinges. We have selected the nodes that are designated as VACATED. The agents settled at those nodes travel with the DFShead to help with the neighborhood search. Here, we show to use the agents at DFShead to perform a neighborhood search, called *Parallel Probing*, to determine the state of neighbors from EMPTY, VACATED and OCCUPIED; and obtain the scout result corresponding to a port p_{xy} as the 4-tuple $\langle p_{xy}, \texttt{type}(\{x,y\}), \texttt{type}(y), \psi(y)\rangle$. The pseudocode of the algorithm is given in Algorithm 4 which we call `Parallel_Probe()`.

Highlevel overview. The core idea of parallel probing stems from the fundamental observation that each node has a port-1 neighbor. As we have seen in Section IV-A, a **visited** node is VACATED when its port-1 neighbor is occupied or when a node is **partiallyVisited** or **fullyVisited**; we call these agents settled scouts. In OCCUPIED nodes the settled agent remains, but in VACATED nodes the settled agent travels with the DFShead.

Consider a node x where the DFShead is doing a neighborhood search. DFShead first settles an agent at x. The unsettled agents and settled scouts perform the neighborhood search. The ports at x are assigned to agents in the increasing order of their IDs until all ports are probed. An agent visiting a neighbor y (i.e., probing port p_{xy}), determines whether y is EMPTY, OCCUPIED or VACATED. By default, if y is OCCUPIED, then the probing agent returns directly. The challenge arises when the probing agent must distinguish between a node that is EMPTY vs. VACATED. When a **visited** node is VACATED, its port-1 neighbor is OCCUPIED in the tree. Exploiting this fact, the probing agent visits the port-1 neighbor z of y. If z is occupied, then it returns to x. At x, it checks among the other agents present, if there exists an agent b such that port-1 neighbor of b is $\psi(z)$. However, the VACATED node y can be **partiallyVisited** or **fullyVisited**. In that case, z may be a **visited** node that is VACATED. Then the probing agent visits port-1 neighbor w of z, and returns to x if $\psi(w)$ is present. The probing agent can then transitively check for an agent c such that $\psi(w)$ is c's port-1 neighbor; and b such that c is b's port-1 neighbor.

Detailed description. Let x be the DFShead with settled agent $\psi(x)$. Let its parent port be $\psi(x)$.parentPort ($\perp$ at the root). For every $p_{xy} \in \{1, \dots, \delta_x\} \setminus \{\psi(x).\text{parentPort}\}$ the head chooses a scout agent $a \in A_{scout}$ in the increasing order of their ID to scout the ports leading to $N(v)$. To describe the rules in a simple manner, we consider the neighbor y to be a variable. Scout a learns the edge type $\texttt{type}(\{x,y\}) \in \{\texttt{tpq, tp1, t11, t1q}\}$ when it reaches the neighbor $y \in N(v)$, based on the arrival port p_{yx}. Upon arrival the scout sets a.scoutEdgeType $\leftarrow \texttt{type}(\{x,y\})$. The agent a can also determine $\xi(y)$ as the settled node present at y. The node y can be either EMPTY, VACATED or OCCUPIED. Note that, when $\psi(y)$ is VACATED or EMPTY, $\xi(y)$ is $\perp$. The node type of y is stored at $\psi(y)$.nodeType (invalid when y is **unvisited**, and assigned $\perp$). At node x, the probe rules are as follows to determine $\psi(y)$. Once $\psi(y)$ is determined, the scout result is stored as a.scoutResult $\leftarrow \langle p_{xy}, a.\text{scoutEdgeType}, \psi(y).\text{nodeType}, \psi(y)\rangle$.

- **(R1)** If $\xi(y) \neq \perp$, then $\psi(y) = \xi(y)$ and return to x.
- **(R2)** If $\xi(y) = \perp$ and $p_{yx} = 1$, then $\psi(y) = \perp$ and return to x.
- **(R3)** If $\xi(y) = \perp$ and $p_{yx} \neq 1$, let z be the port-1 neighbor of y. Go to z. Store a.scoutP1Neighbor $= \xi(z)$ and a.scoutPortAtP1Neighbor $= p_{zy}$.
  - **(R3a)** If $\xi(z) \neq \perp$, return to x and check whether $\exists\, b \in A_{scout}$ with b.P1Neighbor $= \xi(z)$ and b.portAtP1Neighbor $= p_{zy}$.
    - **(R3a-i)** If such b exists, then $\psi(y) = b$.
    - **(R3a-ii)** Otherwise $\psi(y) = \perp$.
  - **(R3b)** If $\xi(z) = \perp$ and $p_{zy} = 1$, then $\psi(y) = \perp$ and return to x.
  - **(R3c)** If $\xi(z) = \perp$ and $p_{zy} \neq 1$, visit the port-1 neighbor w of z. Store a.scoutP1P1Neighbor $= \xi(w)$ and a.scoutPortAtP1P1Neighbor $= p_{wz}$.
    - **(R3c-i)** If $\xi(w) = \perp$, then $\psi(y) = \perp$.
    - **(R3c-ii)** If $\xi(w) \neq \perp$, return to x and check (i) $\exists\, c \in A_{scout}$ with c.port1Neighbor $= \xi(w)$ and c.portAtP1Neighbor $= p_{wz}$, and (ii) $\exists\, b \in A_{scout}$ with b.P1Neighbor $= c$ and b.portAtP1Neighbor $= p_{zy}$.
      - (α) If both c and b exist, then $\psi(y) = b$.
      - (β) Otherwise $\psi(y) = \perp$.

Note that, the checking for presence of another agent that was originally VACATED happens only after all the agents in A_{scout} return to x.

Fig. 4 shows one instance for every rule from **(R1)** to **(R3c-ii)**. The central red node x is the position of DFShead performing the parallel probe. Small white circles carry local port numbers.

Lemma 5. *Algorithm 4 (`Parallel_Probe()`) at a node $x \in V$ correctly determines the state of a neighbor node in $O(1)$ epochs.*

Proof. On running Algorithm 4 (`Parallel_Probe()`) at x, the state of a neighbor y is clearly determined when $\xi(y) \neq \perp$. The settled agent at y remains at y, and hence it results in OCCUPIED. The main challenge of correctly determining state is identifying the VACATED neighbors, since by default the result assumes y to be EMPTY. When y is VACATED, it can be either **visited**, **fullyVisited** or **partiallyVisited**. We handle each of the cases separately.

- When y is **visited** and VACATED, then port-1 neighbor of y must be OCCUPIED (rule (V2)). Hence by visiting the port-1 neighbor z of y, the agent will find $\xi(z) \neq \perp$. When y was VACATED, $\psi(y)$ must have stored $\xi(z)$ as $\psi(y)$.P1Neighbor. Thus, after the agent returns to x, the

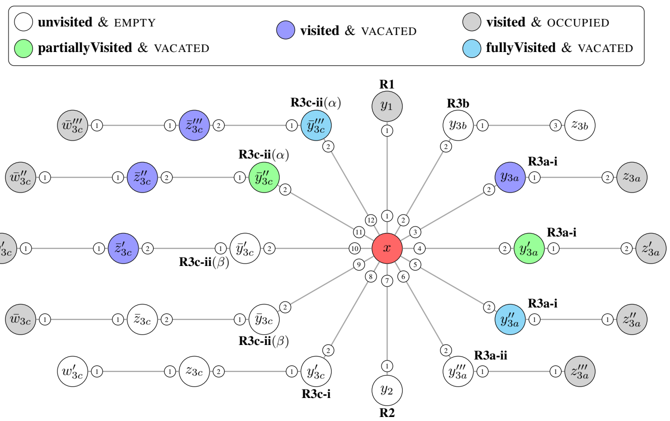


Fig. 4: Examples for rules **(R1)–(R3c-ii)**. Neighbors of x are labeled with the rule that determines whether they are EMPTY or OCCUPIED.

agent $\psi(y)$ must be at x since it is travelling with the DFShead.

- When y is **partiallyVisited** and VACATED, that means, the port-1 neighbor z of y can either be OCCUPIED or VACATED. When $\xi(z) \neq \perp$, we can determine the presence $\psi(y)$ in A_{scout} by checking if $\xi(z)$ is the port-1 neighbor of $\psi(y)$. Furthermore, when z is also VACATED, then z must be of type **visited**. Since y can be **partiallyVisited** if only if there are no empty neighbors of y via `tp1`, `t1q` or `t11` type edges when DFShead first reached y. Since z is connected to y via either `tp1` or `t11` edge, z must be of type **visited** since it had an empty neighbor (y) when DFShead was at z. Thus the port-1 neighbor w of z must be OCCUPIED when z is VACATED. Thus, when y is **partiallyVisited**, either z or w must be OCCUPIED. The agent after returning to x can check for existence of $\psi(z)$ and $\psi(y)$ in A_{scout} and determine $\psi(y)$.
- When y is **fullyVisited**, it is chosen to be VACATED only if $\psi(y)$.vacatedNeighbor is $false$ (by (V3)). When DFShead is at y, it must have no EMPTY neighbors. The port-1 neighbor z of y must be **visited** since when DFShead was at z, it had an edge of type `tp1` or `t11` leading to y (which was EMPTY). Analogous to the previous case, since z is a **visited** node, if it is VACATED, then the port-1 neighbor of z must be OCCUPIED. Hence the agent can transitively determine $\psi(y)$ at x from the agents in A_{scout}.

Certainly, the agent from a VACATED node maybe probing another port. We can say that all probing agents spend at most 6 epochs before returning to x. Hence in $O(1)$ epochs, all probing agents are at x. At that time, all agents can determine the existence of other agents in A_{scout}. □

Lemma 6. *Algorithm 4 (`Parallel_Probe()`) at a node $x \in V$ determines the state of $O(|A_{scout}|)$ neighbor nodes in $O(1)$ epochs.*

Proof. As we observe in Lemma 5, it takes $O(1)$ edge traversals (thus epochs) to determine the state of a neighbor. Each agent in A_{scout} can probe in parallel to determine the state of $|A_{scout}|$ neighbors in $O(1)$ epochs. Hence $O(|A_{scout}|)$ neighbor states can be identified in $O(1)$ epochs. □

V. ROOTED DISPERSION IN $\mathcal{ASYNC}$

The rooted dispersion of $k \leq n$ agents unfolds in two phases. In the first phase, the agents follow Algorithm 2 (`DFS_P1tree()`) to construct a P1TREE, only until there are unsettled agents. Once the tree size is k, the construction is complete. The agents that were settled at VACATED nodes have traveled with the DFShead helping with the construction of the P1TREE. In the second phase, the agents retrace their path along the tree edges to return the settled agents from VACATED nodes to their originally settled node. We will utilize the techniques described in Sections IV-A and IV-B to construct the P1TREE. Further, we also use Algorithm 6 `Retrace()`; to return the *settledScout* agents to the original node they settled at. Now, we describe the Algorithm 5 `RootedAsync()`.

Description. Initially, the agents are located at v_0 and have state *unsettled*. On visiting a node v, if there is no settled agent in v, the agent with the highest ID among the unsettled agents at v settles and becomes $\psi(v)$. The settled agent sets its parentPort to $\perp$ and its arrivalPort to the port it used to reach v. It sets portAtParent to the port at the parent node that was used to reach v. All other agents at v are part of the scout pool, denoted as A_{scout}. The settled agent conducts a neighborhood search using `Parallel_Probe()`. Then it checks if it can vacate its position by calling Algorithm 3 `Can_Vacate()` ~~(pseudocode in Appendix C)~~. If the settled agent can vacate, it becomes part of $A_{vacated}$ and is added to A_{scout} with state *settledScout*. All agents in A_{scout} move through the next port returned by `Parallel_Probe()`. If no port is available, all agents in A_{scout} move through $\psi(v)$.parentPort. The settled agent updates its recentPort to the port it used to move. The process continues until no unsettled agents remain. To keep track of the tree, a settled agent at v also keeps track of its sibling w (if exists) that was visited immediately before at its parent u, and the settled agent at u only keeps track of the last child (i.e., v).

A. Retrace Phase

Once the P1TREE contains exactly k vertices, each VACATED node has a *settledScout* that travelled alongside the DFShead during the construction. These agents, collected in the set $A_{vacated}$ and currently co-located at the last vertex added to the tree, now have to walk backwards through the tree so that every VACATED vertex regains its original settled agent. We call this controlled backward traversal *retrace*.

Local variables maintained by every settled agent $\psi(v)$.

- $\psi(v)$.parent: ID of agent at parent node u $\psi(u)$.ID(is $\perp$ for the root), and the port at u that leads to v;
- $\psi(v)$.parentPort: the port of v that leads to u;

- $\psi(v)$.recentChild: the port at v that leads to the most recently visited child;
- $\psi(v)$.sibling: the ID, and the port of a child at u which was visited immediately before v. It is $\perp$ when v is the first child.

These pointers are sufficient for retrace: *recentChild* tells us where the DFS went last, and *sibling* lets us jump sideways to the child that was explored immediately before the current one.

Description. Retrace performs a depth-first *post-order* walk of the tree: whenever it backtracks to an internal vertex u from node v via the port $\psi(u)$.recentChild, it first tries to move sideways to the previous sibling of v (if any, stored in $\psi(v)$.sibling); only when no such sibling exists, it ascends further to the parent of u. Each sideways jump also updates $\psi(u)$.recentChild so that the just-visited leaf is *logically deleted* from the tree. An agent from $A_{vacated}$ realizes that it has reached its original node from the ID stored at a settled agent that we keep track as the next agent ID. Then the agent changes its state from *settledScout* to *settled*. Once all agents become *settled*, `Retrace()` ends. DISPERSION is achieved. An example run of Algorithm 5 is omitted due to space constraints.

B. Correctness and Complexity

Lemma 7. *For any parallel probe in Algorithm 5 (*`RootedAsync()`*), $|A_{scout}| \geq \lceil (k-2)/3 \rceil$.*

Proof. Parallel Probe happens only when there are unsettled agents at the current position of DFShead. Initially, the size of $\mathcal{T}$ is 1, and there are $k-1$ unsettled agents in A_{scout}; thus $k-1 > \lceil (k-2)/3 \rceil$. Consider a tree $\mathcal{T}$ of size j constructed, and the DFShead is at the $j+1$th vertex. From Lemma 4, there are at least $\lfloor j/3 \rfloor$ vacated nodes, thus $\lfloor j/3 \rfloor$ agents in $A_{vacated}$. The current position of DFShead has one settled agent and $k-(j+1)$ unsettled agents. Since $k-(j+1) \geq 1$, we have $j \leq k-2$. So, $|A_{scout}| = |A_{vacated}| + |A_{unsettled}| = \lfloor j/3 \rfloor + k - (j+1)$. For $j < k-1$, $\lfloor j/3 \rfloor + k - 1 - j \geq \lceil (k-2)/3 \rceil$. □

We have the following remark due to Lemmas 6 and 7.

Remark 1. *At each node where DFShead performs* `Parallel_Probe()`*, it takes at most $18 = O(1)$ epochs.*

Since DFShead performs `Parallel_Probe()` only when there are unsettled agents, and it needs to check at most $k-1$ ports at the root node and $k-2$ ports (excluding the parent port) at a non-root node. Since there are $k-1$ scouts at the root node and at least $\lceil (k-2)/3 \rceil$ scouts at other nodes, `Parallel_Probe()` runs for at most three iterations, each of which is at most 6 epochs.

Lemma 8. *Algorithm 6 (*`Retrace()`*) takes at most $O(k)$ epochs.*

Theorem 2. *Algorithm 5 (*`RootedAsync()`*) takes at most $O(k)$ epochs to achieve* DISPERSION *with $O(\log k + \Delta)$ bits of memory at each agent.*

VI. General Dispersion

The idea of general dispersion, i.e., when there are multiple nodes in the initial configuration with more than one agent, broadly follows from the merger strategy of Kshemkalyani and Sharma [21]. Each multiplicity starts its own `RootedAsync()` with a treelabel that consists of $\langle a_{highest}.\mathsf{ID}, a_{highest}.\mathsf{level}, a_{highest}.\mathsf{weight} \rangle$ to create its P1TREE. The level parameter starts at 0. The level increases for every merger between two P1TREE of the same size. Otherwise, the level remains the same and the higher weight P1TREE wins. The weight of the traversal is updated to reflect the total number of agents in the merged P1TREE.

We use the following modifications to make the `RootedAsync()` compatible with the merger. When doing `Parallel_Probe()`, a node y is considered to be EMPTY if the treelabel of $\xi(y)$ is different from the treelabel of the scouting agent. Analogously for `Can_Vacate()` when going to the port-1 neighbor.

Every time a lower priority traversal meets a higher priority traversal, it revisits all the nodes of its own traversal to collect all the settled agents and joins the higher priority traversal, by chasing the DFShead. If a higher priority traversal finds a lower priority traversal head, then it subsumes it completely, otherwise, if it finds settled agents, then it treats those nodes as empty nodes and absorbs the settled agent. To facilitate the collection of agents by a lower priority DFShead, the agents essentially perform `Retrace()` to traverse the tree, and unlike retrace, all settled agents move with the DFShead. Once all agents are collected at the root, they follow recentPort to reach the previous position of DFShead where higher priority DFS tree exists.

With these primitives added to `RootedAsync()`, it can utilize the merger strategy of Kshemkalyani and Sharma [21] with overhead proportional to size of the tree, which is $O(k)$ in the worst case. Thus we achieve a $O(k)$ time solution for any arbitrary distribution of agents on the graph.

VII. Concluding Remarks

We have considered in this paper a fundamental problem of DISPERSION which asks $k \leq n$ mobile agents with limited memory positioned initially arbitrarily on the nodes (memory-less) of an n-node port-labeled anonymous graph of maximum degree Δ to autonomously relocate to the nodes of the graph such that each node hosts no more than one agent, and closed the complexity gap in the literature by providing an $O(k)$-time solution in the asynchronous setting. Our solution is obtained through a novel technique of port-one tree we develop in this paper that prioritizes visiting edges with port-1 at (at least) one end-point. For future work, it would be interesting to explore whether our port-one tree technique could be useful in solving other fundamental problems in port-labeled anonymous graphs.

References

[1] S. H. Brawley and W. H. Adey, "Territorial behavior of threespot damselfish (eupomacentrus planifrons) increases reef algal biomass and productivity," *Environmental Biology of Fishes*, vol. 2, pp. 45–51, 1977.

[2] N. Le Douarin and C. Kalcheim, *The neural crest*. No. 36, Cambridge university press, 1999.

[3] P. R. Wurman, R. D'Andrea, and M. Mountz, "Coordinating hundreds of cooperative, autonomous vehicles in warehouses," *AI magazine*, vol. 29, no. 1, pp. 9–9, 2008.

[4] L. Barriere, P. Flocchini, E. Mesa-Barrameda, and N. Santoro, "Uniform scattering of autonomous mobile robots in a grid," in *IPDPS*, pp. 1–8, 2009.

[5] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, pp. 279–301, Oct. 1989.

[6] S. Das, D. Dereniowski, and C. Karousatou, "Collaborative exploration of trees by energy-constrained mobile robots," *Theory Comput. Syst.*, vol. 62, no. 5, pp. 1223–1240, 2018.

[7] D. Dereniowski, Y. Disser, A. Kosowski, D. Pajak, and P. Uznański, "Fast collaborative graph exploration," *Inf. Comput.*, vol. 243, pp. 37–49, Aug. 2015.

[8] P. Fraigniaud, D. Ilcinkas, G. Peer, A. Pelc, and D. Peleg, "Graph exploration by a finite automaton," *Theor. Comput. Sci.*, vol. 345, pp. 331–344, Nov. 2005.

[9] R. Subramanian and I. D. Scherson, "An analysis of diffusive load-balancing," in *SPAA*, pp. 220–225, 1994.

[10] J. Augustine and W. K. Moses Jr., "Dispersion of mobile robots: A study of memory-time trade-offs," in *ICDCN*, pp. 1:1–1:10, 2018.

[11] A. Das, K. Bose, and B. Sau, "Memory optimal dispersion by anonymous mobile robots," in *CALDAM* (A. Mudgal and C. R. Subramanian, eds.), vol. 12601, pp. 426–439, Springer, 2021.

[12] B. Gorain, P. S. Mandal, K. Mondal, and S. Pandit, "Collaborative dispersion by silent robots," in *SSS*, vol. 13751, pp. 254–269, Springer, 2022.

[13] G. F. Italiano, D. Pattanayak, and G. Sharma, "Dispersion of mobile robots on directed anonymous graphs," in *SIROCCO* (M. Parter, ed.), pp. 191–211, Springer, 2022.

[14] A. D. Kshemkalyani and F. Ali, "Efficient dispersion of mobile robots on graphs," in *ICDCN*, pp. 218–227, 2019.

[15] A. D. Kshemkalyani, "Dispersion of mobile robots on graphs in the asynchronous model," *Theor. Comput. Sci.*, vol. 1044, p. 115272, 2025.

[16] A. D. Kshemkalyani, M. Kumar, A. R. Molla, D. Pattanayak, and G. Sharma, "Dispersion is (almost) optimal under (a) synchrony," in *SPAA*, 2025.

[17] A. D. Kshemkalyani, A. R. Molla, and G. Sharma, "Fast dispersion of mobile robots on arbitrary graphs," in *ALGOSENSORS*, pp. 23–40, 2019.

[18] A. D. Kshemkalyani, A. R. Molla, and G. Sharma, "Dispersion of mobile robots on grids," in *WALCOM*, pp. 183–197, 2020.

[19] A. D. Kshemkalyani, A. R. Molla, and G. Sharma, "Efficient dispersion of mobile robots on dynamic graphs," in *ICDCS*, pp. 732–742, 2020.

[20] A. D. Kshemkalyani, A. R. Molla, and G. Sharma, "Dispersion of mobile robots using global communication," *J. Parallel Distributed Comput.*, vol. 161, pp. 100–117, 2022.

[21] A. D. Kshemkalyani and G. Sharma, "Near-optimal dispersion on arbitrary anonymous graphs," in *OPODIS* (Q. Bramas, V. Gramoli, and A. Milani, eds.), vol. 217 of *LIPIcs*, pp. 8:1–8:19, 2021.

[22] A. R. Molla and W. K. M. Jr., "Dispersion of mobile robots: The power of randomness," in *TAMC*, pp. 481–500, 2019.

[23] A. R. Molla, K. Mondal, and W. K. M. Jr., "Byzantine dispersion on graphs," in *IPDPS*, pp. 942–951, IEEE, 2021.

[24] A. R. Molla, K. Mondal, and W. K. M. Jr., "Optimal dispersion on an anonymous ring in the presence of weak byzantine robots," *Theor. Comput. Sci.*, vol. 887, pp. 111–121, 2021.

[25] T. Shintaku, Y. Sudo, H. Kakugawa, and T. Masuzawa, "Efficient dispersion of mobile agents without global knowledge," in *SSS*, vol. 12514, pp. 280–294, 2020.

[26] Y. Sudo, M. Shibata, J. Nakamura, Y. Kim, and T. Masuzawa, "Near-linear time dispersion of mobile agents," in *DISC* (D. Alistarh, ed.), pp. 38:1–38:22.

[27] R. Banerjee, M. Kumar, and A. R. Molla, "Optimizing robot dispersion on unoriented grids: With and without fault tolerance," in *ALGOWIN* (Q. Bramas, A. Casteigts, and K. Meeks, eds.), pp. 31–45, Springer, 2024.

[28] R. Banerjee, M. Kumar, and A. R. Molla, "Optimal fault-tolerant dispersion on oriented grids," in *ICDCN* (A. Korman, S. Chakraborty, S. Peri, C. Boldrini, and P. Robinson, eds.), pp. 254–258, ACM, 2025.

[29] P. K. Chand, M. Kumar, A. R. Molla, and S. Sivasubramaniam, "Fault-tolerant dispersion of mobile robots," in *CALDAM* (A. Bagchi and R. Muthu, eds.), pp. 28–40, 2023.

[30] T. Kaur and K. Mondal, "Distance-2-dispersion: Dispersion with further constraints," in *Networked Systems* (D. Mohaisen and T. Wies, eds.), pp. 157–173, 2023.

[31] A. Saxena, T. Kaur, and K. Mondal, "Dispersion on time varying graphs," in *ICDCN* (A. Korman, S. Chakraborty, S. Peri, C. Boldrini, and P. Robinson, eds.), pp. 269–273, ACM, 2025.

[32] A. Saxena and K. Mondal, "Path connected dynamic graphs with a study of efficient dispersion," in *ICDCN* (A. Korman, S. Chakraborty, S. Peri, C. Boldrini, and P. Robinson, eds.), pp. 171–180, ACM, 2025.

[33] A. Cord-Landwehr, B. Degener, M. Fischer, M. Hüllmann, B. Kempkes, A. Klaas, P. Kling, S. Kurras, M. Märtens, F. Meyer auf der Heide, C. Raupach, K. Swierkot, D. Warner, C. Weddemann, and D. Wonisch, "A new approach for analyzing convergence algorithms for mobile robots," in *ICALP*, pp. 650–661, 2011.

APPENDIX

AN EXAMPLE RUN OF ALGORITHM ROOTEDASYNC()

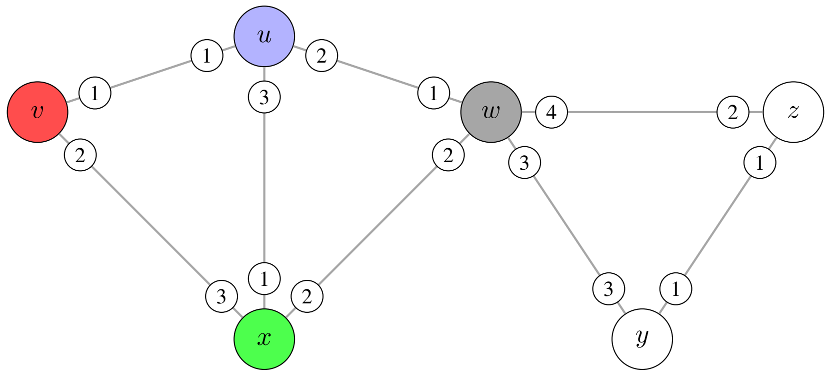


Fig. 5: Graph for the example run. Node colors are illustrative from the original image and may not reflect dynamic states in this trace.

Consider $k = 6$ agents, $a_1, a_2, a_3, a_4, a_5, a_6$, with IDs sorted in ascending order (i.e., a_1.ID $<$ a_2.ID $< \cdots <$ a_6.ID). Initially, all agents are at node v (the root) and are in state *unsettled*. The DFS-based P1Tree construction begins. $\psi(N)$ denotes the agent settled at node N. $A_{unsettled}$ is the set of unsettled agents. $A_{vacated}$ is the set of agents whose nodes were vacated. $A_{scout} = A_{unsettled} \cup A_{vacated}$. The agent a_{min} is the agent in A_{scout} with the lowest ID.

1) **Settle at v (Root):** Agents $\{a_1, \ldots, a_6\}$ are at v. a_6 (highest ID) settles: $\psi(v) = a_6$. $a_6.state \leftarrow$ *settled*. $A_{unsettled} = \{a_1, \ldots, a_5\}$. $A_{scout} = \{a_1, \ldots, a_5\}$. $a_{min} = a_1$. $\psi(v)$.parentPort $= \perp$.
   `Parallel_Probe()` at v: Ports are 1 (to u) and 2 (to x). a_1 scouts port 1 (to u); a_2 scouts port 2 (to x).
   - a_1 (to u): Edge $\{v, u\}$ is type `t11`. Node u is **unvisited**. Scout a_1 finds $\xi(u) = \perp$ and $p_{uv} = 1$ (Rule R2). Reports u as EMPTY.
   - a_2 (to x): Edge $\{v, x\}$ is type `tpq`. Node x is **unvisited**. Scout a_2 finds $\xi(x) = \perp, p_{xv} \neq 1$. (Rule R3). port-1 neighbor (P1N) of x is u. a_2 visits u. $\xi(u) = \perp, p_{ux} \neq 1$. (Rule R3c). P1N of u is v. a_2 visits v. $\xi(v) = a_6$. Reports x as EMPTY.

   Probe results processed: u (`t11`, EMPTY), x (`tpq`, EMPTY). Based on probe results, node v is **visited** (it has unvisited neighbors).
   `Can_Vacate`($\psi(v) = a_6$): $\psi(v)$.parentPort $= \perp$ (root). a_6 remains *settled*. Node v is OCCUPIED. Next edge is to u (priority). A_{scout} moves to u. a_6.recentChild $\leftarrow$ (port 1 to u). a_{min}.childPort (i.e. a_1.childPort) $\leftarrow$ (port 1 to u).
2) **Settle at u:** a_5 (highest ID in $A_{unsettled} = \{a_1, \ldots, a_5\}$) settles: $\psi(u) = a_5$. $a_5.state \leftarrow$ *settled*. a_5.parent $\leftarrow$ (a_6.ID, port 1 at v). a_5.parentPort $\leftarrow$ (port 1 at u). $A_{unsettled} = \{a_1, \ldots, a_4\}$.
   `Parallel_Probe()` at u: Ports (excl. parent port 1) are 2 (to w), 3 (to x). a_1 scouts port 2 (to w); a_2 scouts port 3 (to x).
   - a_1 (to w): Edge $\{u, w\}$ is type `tp1`. Node w is **unvisited**. Rule R2. Reports w as EMPTY.
   - a_2 (to x): Edge $\{u, x\}$ is type `tp1`. Node x is **unvisited**. Rule R2. Reports x as EMPTY.

   Probe results: w (`tp1`, EMPTY), x (`tp1`, EMPTY). Node u is **visited**.
   `Can_Vacate`($\psi(u) = a_5$): $\psi(u)$.nodeType $=$ **visited**. P1N of u is v. $\xi(v) = a_6 \neq \perp$ (and v is OCCUPIED). Rule (V2) applies. a_5 becomes *settledScout*. Node u is VACATED. $A_{vacated} = \{a_5\}$. $A_{scout} = \{a_1, \ldots, a_4, a_5\}$. $a_{min} = a_1$. $\psi(u) = a_5$ stores a_5.P1Neighbor $\leftarrow$ a_6.ID, a_5.portAtP1Neighbor $\leftarrow p_{vu} = 1$. Next edge to w. A_{scout} moves to w. $\psi(u) = a_5$ (now scout) updates a_5.recentChild $\leftarrow$ (port 2 to w). a_1.childPort $\leftarrow$ (port 2 to w).
3) **Settle at w:** a_4 settles: $\psi(w) = a_4$. $a_4.state \leftarrow$ *settled*. a_4.parent $\leftarrow$ (a_5.ID, port 2 at u). a_4.parentPort $\leftarrow$ (port 1 at w). $A_{unsettled} = \{a_1, a_2, a_3\}$.
   `Parallel_Probe()` at w: Ports (excl. parent port 1) are 2 (to x), 3 (to y), 4 (to z). a_1 scouts port 2 (to x); a_2 scouts port 3 (to y); a_3 scouts port 4 (to z).
   - a_1 (to x): Reports x as EMPTY (as per step 2b logic, after 2-hop check, finds no scout $\psi(x)$ or $\psi(u)$ in A_{scout} that are settled at those nodes for P1N purposes, as $a_5 = \psi(u)$ is a scout now, but its P1N info is about v).
   - a_2 (to y): Reports y as EMPTY (Rule R3b, P1N z is empty, $p_{zy} = 1$).
   - a_3 (to z): Reports z as EMPTY (Rule R3b, P1N y is empty, $p_{yz} = 1$).

   Probe results: x, y, z all (`tpq`, EMPTY). Node w is **visited**.

`Can_Vacate`$(\psi(w) = a_4)$: $\psi(w)$.nodeType = **visited**. P1N of w is u. $\xi(u) = \perp$ (since $a_5 = \psi(u)$ is a scout). Condition "P1N is OCCUPIED" is false. a_4 remains *settled*. Node w is OCCUPIED. $A_{vacated} = \{a_5\}$. $A_{scout} = \{a_1, a_2, a_3, a_5\}$. $a_{min} = a_1$. Next edge to x. A_{scout} moves to x. a_4.recentChild $\leftarrow$ (port 2 to x). a_1.childPort $\leftarrow$ (port 2 to x).

4) **Settle at x:** a_3 settles: $\psi(x) = a_3$. $a_3.state \leftarrow$ *settled*. a_3.parent $\leftarrow$ (a_4.ID, port 2 at w). a_3.parentPort $\leftarrow$ (port 2 at x). $A_{unsettled} = \{a_1, a_2\}$. $A_{scout} = \{a_1, a_2, a_5\}$. $a_{min} = a_1$.
   `Parallel_Probe()` at x: Ports (excl. parent port 2) are 1 (to u), 3 (to v).
   - a_1 (to u): Edge $\{x, u\}$ is `t1p`. $\xi(u) = \perp$. P1N of u is v. $\xi(v) = a_6$. At x, find $b = \psi(u)$, which is $a_5 \in A_{scout}$, with a_5.P1Neighbor = a_6.ID. Yes. Reports u as VACATED (**visited**, Rule R3a-ii).
   - a_2 (to v): Edge $\{x, v\}$ is `tpq`. $\xi(v) = a_6$. Rule R1. Reports v as OCCUPIED (**visited**).

   Probe results: All explorable non-parent neighbors (u, v) are not EMPTY. Parent edge $\{w, x\}$ is `tpq`. Assume no other EMPTY neighbors via non-`tpq` edges. Node x becomes **partiallyVisited**.
   `Can_Vacate`$(\psi(x) = a_3)$: $\psi(x)$.nodeType = **partiallyVisited**. Rule (V4). a_3 becomes *settledScout*. Node x is VACATED. $A_{vacated} = \{a_5, a_3\}$. $A_{scout} = \{a_1, a_2, a_5, a_3\}$. DFS backtracks (nextPort is $\perp$). A_{scout} moves to w. a_1.childDetails $\leftarrow$ (a_3.ID, port 2 at w).
5) **At w (after backtrack from x):** $\psi(w) = a_4$. **`Parallel_Probe()` at w:** Next ports to probe: 3 (to y), 4 (to z).
   - a_1 scouts port 3 (to y): reports EMPTY. Same as before.
   - a_2 scouts port 4 (to z): reports EMPTY. Same as before.

   Probe results update: y and z are EMPTY. Node w remains **visited**.
   `Can_Vacate`$(\psi(w) = a_4)$: No change, a_4 remains *settled*, w is OCCUPIED. Priority to y (port 3). A_{scout} moves to y. a_4.recentChild $\leftarrow$ (port 3 to y). a_1.childPort $\leftarrow$ (port 3 to y). a_1.siblingDetails $\leftarrow$ (a_3.ID, port 2 at w) (sibling of y is x).
6) **Settle at y:** a_2 settles: $\psi(y) = a_2$. $a_2.state \leftarrow$ *settled*. a_2.parent $\leftarrow$ (a_4.ID, port 3 at w). a_2.parentPort $\leftarrow$ (port 3 at y). a_2.sibling $\leftarrow$ (a_3.ID, port 2 at w). $A_{unsettled} = \{a_1\}$. $A_{scout} = \{a_1, a_5, a_3\}$. $a_{min} = a_1$.
   `Parallel_Probe()` at y: Ports (excl. parent port 3) are 1 (to z), 2 (to x).
   - a_1 (to z): Edge $\{y, z\}$ is `t11`. Node z is **unvisited**. Rule R2. Reports z as EMPTY.
   - a_5 (to x): Edge $\{y, x\}$ is `tpq`. $\xi(x) = \perp$. P1N u, P1N of u is $v(\xi(v) = a_6)$. At y: find $c = \psi(u) = a_5 \in A_{scout}$ (yes). Find $b = \psi(x) = a_3 \in A_{scout}$ (yes). Rule R3c-ii(α). Reports x as VACATED (**partiallyVisited**).

   Probe results: z (`t11`, EMPTY), x (`tpq`, **partiallyVisited**). Node y is **visited**.
   `Can_Vacate`$(\psi(y) = a_2)$: $\psi(y)$.nodeType = **visited**. P1N of y is z. $\xi(z) = \perp$ (it's EMPTY). Returns *settled*. Node y is OCCUPIED. Next edge to z. A_{scout} moves to z. a_2.recentChild $\leftarrow$ (port 1 to z). a_1.childPort $\leftarrow$ (port 1 to z). a_1.siblingDetails $\leftarrow \perp$.
7) **Settle at z:** a_1 settles: $\psi(z) = a_1$. $a_1.state \leftarrow$ *settled*. a_1.parent $\leftarrow$ (a_2.ID, port 1 at y). a_1.parentPort $\leftarrow$ (port 1 at z). a_1.sibling $\leftarrow \perp$. $A_{unsettled} = \{\}$. $k = 6$ agents are settled.

Construction phase ends as $A_{unsettled}$ is empty. Current agent states and logical locations (physical location of scouts is z): $v : \psi(v) = a_6(\textit{settled})$ $u : \psi(u) = a_5(\textit{settledScout})$ $w : \psi(w) = a_4(\textit{settled})$ $x : \psi(x) = a_3(\textit{settledScout})$ $y : \psi(y) = a_2(\textit{settled})$ $z : \psi(z) = a_1(\textit{settled})$ $A_{vacated} = \{a_3, a_5\}$ (sorted by ID). They are physically co-located at z.

Retrace Phase: $A_{vacated} = \{a_3, a_5\}$. Current location of $A_{vacated}$ group: z. The lead retrace agent $a_{min\_retrace}$ is a_3. Goal: a_3 to x, a_5 to u.

a. **At z (settled agent $\psi(z) = a_1$):** $A_{vacated} = \{a_3, a_5\}$ is present. $a_{min} = a_3$. Node z is occupied by $a_1(\xi(z) = a_1)$. $\psi(z) = a_1$ has a_1.recentChild = $\perp$ (leaf in construction). The group moves to parent of z. a_3.nextAgentID $\leftarrow \psi(z)$.parent.ID (i.e., a_2.ID). a_3.nextPort $\leftarrow \psi(z)$.parentPort (port 1 at z). a_3.siblingDetails $\leftarrow \psi(z)$.sibling ($\perp$). $A_{vacated} = \{a_3, a_5\}$ moves to y via z's port 1. a_3.arrivalPort at y becomes port 1.
b. **At y (settled agent $\psi(y) = a_2$):** $A_{vacated} = \{a_3, a_5\}$ arrives. $a_{min} = a_3$. Node y is occupied by $a_2(\xi(y) = a_2)$. $\psi(y) = a_2$ has a_2.recentChild = (port 1 to z). a_3.arrivalPort (port 1 at y) matches $\psi(y)$.recentChild. a_3.siblingDetails is $\perp$. $\psi(y)$.recentChild $\leftarrow \perp$. The group moves to parent of y. a_3.nextAgentID $\leftarrow \psi(y)$.parent.ID (i.e., a_4.ID). a_3.nextPort $\leftarrow$ $\psi(y)$.parentPort (port 3 at y). a_3.siblingDetails $\leftarrow \psi(y)$.sibling ((a_3.ID, port 2 at w)). $A_{vacated} = \{a_3, a_5\}$ moves to w via y's port 3. a_3.arrivalPort at w becomes port 3.
c. **At w (settled agent $\psi(w) = a_4$):** $A_{vacated} = \{a_3, a_5\}$ arrives. $a_{min} = a_3$. Node w is occupied by $a_4(\xi(w) = a_4)$. $\psi(w) = a_4$ has a_4.recentChild = (port 3 to y). a_3.arrivalPort (port 3 at w) matches $\psi(w)$.recentChild. a_3.siblingDetails is (a_3.ID, port 2 at w), which is not $\perp$. The group moves to the sibling node x. a_3.nextAgentID $\leftarrow$ a_3.ID (from siblingDetails). a_3.nextPort $\leftarrow$ (port 2 at w) (from siblingDetails). $\psi(w)$.recentChild $\leftarrow$ (port 2 at w) (updates to current traversal direction). a_3.siblingDetails $\leftarrow \perp$. $A_{vacated} = \{a_3, a_5\}$ moves to x via w's port 2. a_3.arrivalPort at x becomes port 2.
d. **At x (original node of a_3, currently VACATED):** $A_{vacated} = \{a_3, a_5\}$ arrives. $a_{min} = a_3$. Node x is VACATED

$(\xi(x) = \perp)$. a_3.nextAgentID is a_3.ID. Agent $a_3 \in A_{vacated}$ matches. $a_3.state \leftarrow$ *settled*. a_3 occupies x. $\psi(x) \leftarrow a_3$. $A_{vacated}$ becomes $\{a_5\}$. a_{min} (for the remaining $A_{vacated}$) is now a_5. $\psi(x) = a_3$ has a_3.recentChild $= \perp$. The group (now just $\{a_5\}$) moves to parent of x. a_5.nextAgentID $\leftarrow \psi(x)$.parent.ID (i.e., a_4.ID). a_5.nextPort $\leftarrow \psi(x)$.parentPort (port 2 at x). a_5.siblingDetails $\leftarrow \psi(x)$.sibling$(\perp)$. $A_{vacated} = \{a_5\}$ moves to w via x's port 2. a_5.arrivalPort at w becomes port 2.

e. **At w (settled agent $\psi(w) = a_4$):** $A_{vacated} = \{a_5\}$ arrives. $a_{min} = a_5$. Node w is occupied by $a_4(\xi(w) = a_4)$. $\psi(w) = a_4$ has a_4.recentChild $=$ (port 2 at w) (updated in step c). a_5.arrivalPort (port 2 at w) matches $\psi(w)$.recentChild. a_5.siblingDetails is $\perp$. $\psi(w)$.recentChild $\leftarrow \perp$. The group moves to parent of w. a_5.nextAgentID $\leftarrow \psi(w)$.parent.ID (i.e., a_5.ID). a_5.nextPort $\leftarrow \psi(w)$.parentPort (port 1 at w). a_5.siblingDetails $\leftarrow \psi(w)$.sibling$(\perp)$. $A_{vacated} = \{a_5\}$ moves to u via w's port 1. a_5.arrivalPort at u becomes port 2.

f. **At u (original node of a_5, currently VACATED):** $A_{vacated} = \{a_5\}$ arrives. $a_{min} = a_5$. Node u is VACATED $(\xi(u) = \perp)$. a_5.nextAgentID is a_5.ID. Agent $a_5 \in A_{vacated}$ matches. $a_5.state \leftarrow$ *settled*. a_5 occupies u. $\psi(u) \leftarrow a_5$. $A_{vacated}$ becomes $\emptyset$. Retrace phase ends as $A_{vacated}$ is empty.

Final agent settlement: $\psi(v) = a_6, \psi(u) = a_5, \psi(w) = a_4, \psi(x) = a_3, \psi(y) = a_2, \psi(z) = a_1$. Dispersion is achieved.

TABLE OF VARIABLES

TABLE II: Variables Used by Agents

Variable Name	**Description**
Generic Agent Properties (applicable to any agent a)	
a.ID	Unique identifier of the agent.
a.state	Current operational state of the agent (e.g., *unsettled*, *settled*, *settledScout*).
a.arrivalPort	The port number through which agent a arrived at its current node.
a.treeLabel	For general dispersion: a tuple $\langle \texttt{leaderID}, \texttt{level}, \texttt{weight} \rangle$ identifying the P1Tree exploration the agent is part of. Contains the ID of the tree's root agent, the tree's merger level, and its current weight (number of agents).
Variables for Settled Agents (e.g., agent $a = \psi(v)$ at node v)	
a.nodeType	The type of node v where the agent is settled (e.g., **unvisited**, **partiallyVisited**, **fullyVisited**, **visited**).
a.parent	A tuple: (ID of the agent settled at v's parent node in the P1Tree, port number at the parent node leading to v). Is $\perp$ for the root agent.
a.parentPort	The port number at node v that leads to its parent in the P1Tree. Is $\perp$ for the root agent.
a.P1Neighbor	ID of the agent settled at the port-1 neighbor of node v. Stores $\perp$ if the port-1 neighbor is EMPTY or unvisited.
a.portAtP1Neighbor	The port number at v's port-1 neighbor (say w) that leads back to v.
a.vacatedNeighbor	Boolean flag. True if a neighbor of v (for which v is a port-1 neighbor and would need to be OCCUPIED for that neighbor to be VACATED) has itself become VACATED. Used in Algorithm `Can_Vacate`.
a.recentChild	The port number at node v that leads to the child most recently visited by the DFS traversal originating from v.
a.sibling	A tuple: (ID of the agent at the previous sibling node in the DFS tree, port number at v's parent leading to that sibling). Is $\perp$ if v is the first child.
a.recentPort	The port number most recently used by the scout agents to depart from node v (either towards a child or back to the parent).
a.probeResult	Stores the overall highest priority result (e.g., next edge to traverse) obtained from the `Parallel_Probe` procedure executed at node v.
a.checked	The count of incident ports at node v that have already been explored during the `Parallel_Probe` procedure.
Temporary Variables for Scouting Agents (agent $a \in A_{\text{SCOUT}}$ during `Parallel_Probe`)	
a.scoutPort	The port number at the current DFS head node that this scout agent a is assigned to explore.
a.scoutEdgeType	The type of the edge (e.g., `tp1`, `t11`) discovered by scout a along its `scoutPort`.

Continued on next page

TABLE II – continued from previous page

`Variable Name`	**Description**
a.scoutP1Neighbor	(During probe of neighbor y) Stores ID of the agent at port-1 neighbor of y (say z), or $\perp$.
a.scoutPortAtP1Neighbor	(During probe of y) Stores port at z leading to y.
a.scoutP1P1Neighbor	(During probe of y's P1N z) Stores ID of agent at port-1 neighbor of z (say w), or $\perp$.
a.scoutPortAtP1P1Neighbor	(During probe of z) Stores port at w leading to z.
a.scoutResult	A tuple $\langle p_{xy}, \texttt{edgeType}, \texttt{nodeType}_y, a' \rangle$ storing the individual result found by scout a for its assigned port.
Context Variables for the Lead Scout Agent (e.g., $a = a_{\min}$**)**	
a.prevID	ID of the agent settled at the node from which the DFS head (and scout group) just departed.
a.childPort	Port at the current DFS head that will be taken to visit the next child. This info is used to set up the child's parent information.
a.siblingDetails	A tuple carrying information about the current child's previous sibling, to be passed to the agent settling at the next child. Format: (Sibling Agent ID, Port at Parent to Sibling).
a.childDetails	A tuple carrying information about the child node just exited during a backtrack operation, to be used by the parent. Format: (Child Agent ID, Port at Parent to Child).
a.nextAgentID	(During Retrace phase) The ID of the agent whose original settled node the A_{VACATED} group is currently moving towards.
a.nextPort	(During Retrace phase) The port number the A_{VACATED} group will take to reach the node associated with `nextAgentID`.

Pseudocodes of Algorithms

A. *Pseudocode of Algorithm 1* `Centralized_P1Tree()`

Algorithm 1: Centralized_P1Tree(G)

Input: connected, port-labelled graph $G = (V, E)$
Output: a Port-One tree $\mathcal{T}$ of G

$\mathcal{T} \leftarrow \emptyset$;
mark all vertices **unvisited**;
while *there exists an* **unvisited** *vertex* **do**
  pick any **unvisited** vertex v, push v on a stack S;
  while $S \neq \emptyset$ **do**
    $u \leftarrow$ pop S;
    foreach *incident edge* $e = [u, p_{uv}, p_{vu}, v]$ *of type* `tp1`, `t11` *or* `t1q` **do**
      if v *is* **unvisited then**
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{e\}$;
        push v on S;
        mark v **visited**;
sort all edges of type `tpq` in lexicographical order;
foreach *edge* e *in sorted order* **do**
  if $\mathcal{T} \cup \{e\}$ *is acyclic* **then**
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{e\}$;
    if $\mathcal{T}$ *forms a single connected component* **then**
      break;
return $\mathcal{T}$;

B. *Pseudocode of Algorithm 2 `DFS_P1Tree()`*

Algorithm 2: `DFS_P1Tree()`

Input: Root vertex v_0, port-labelled graph $G = (V, E)$
Output: P1TREE $\mathcal{T}$

edge priority: $\texttt{tp1} \succ \texttt{t11} \sim \texttt{t1q} \succ \texttt{tpq}$, smallest incident port number under each type;
initialize: $\mathcal{T} \leftarrow \emptyset$, mark all vertices **unvisited**, stack $S \leftarrow \emptyset$;
$S.push(v_0)$;
$\texttt{type}(v_0) \leftarrow$ **visited**;
while $S \neq \emptyset$ **do**
    $u \leftarrow S.top()$;
    $e_{next} \leftarrow \varnothing$;
    $\mathcal{E} \leftarrow$ sorted list of edges incident to u in order of edge-priority;
    for $e \in \mathcal{E}$ **do**
        let $e = [u, p_{uv}, p_{vu}, v]$ be the edge;
        if $\texttt{type}(v) =$ **unvisited then**
            $e_{next} \leftarrow e$;
            break;
        else
            if $\texttt{type}(v)$ **= partiallyVisited** and $\texttt{type}(\{u, v\}) \in \{\texttt{tp1}, \texttt{t11}\}$ **then**
                $e_{next} \leftarrow e$;
                break;
    if $e_{next} \neq \varnothing$ **then**
        let $e = [u, p_{uv}, p_{vu}, v]$ be the edge;
        let $e_\uparrow = [w, p_{wu}, p_{uw}, u]$ be the parent edge of u;
        if *$e, e_\uparrow$ are $\texttt{tpq}$ and no incident edge at u in $\mathcal{T}$ is of type $\langle \texttt{tp1, t11, t1q} \rangle$* **then**
            $\texttt{type}(u) \leftarrow$ **partiallyVisited**;
            $S.pop()$;
        else
            $parent(v) \leftarrow u$;
            $S.push(v)$;
            $\texttt{type}(v) \leftarrow$ **visited**;
    else
        $\texttt{type}(u) \leftarrow$**fullyVisited**;
        $S.pop()$;

C. Pseudocode of Algorithm 3 `Can_Vacate`*()*

Algorithm 3: `Can_Vacate()`

Input: Agent $\psi(x)$ at node x
Output: State of $\psi(x)$

if $\psi(x).\textit{parentPort} = \perp$ **then**
  return *settled*;
else if $\psi(x).\textit{nodeType} =$ ***visited*** **then**
  visit port 1 neighbor w;
  if $\xi(w) \neq \perp$ **then**
    $\psi(w) \leftarrow \xi(w)$;
    set $\psi(w)$.vacatedNeighbor $= true$;
    return to x;
    return *settledScout*;
  return *settled*;
else if $\psi(x).\textit{nodeType} =$ ***fullyVisited*** *and* $\psi(x).\textit{vacatedNeighbor} = false$ **then**
  return *settledScout*;
else if $\psi(x).\textit{nodeType} =$ ***partiallyVisited*** **then**
  return *settledScout*;
else if $\psi(x).\textit{portAtParent} = 1$ **then**
  Visit parent z of x;
  if $\psi(z).\textit{vacatedNeighbor} = false$ **then**
    $\psi(z)$.state $\leftarrow$*settledScout*;
    $\psi(z)$ joins $A_{vacated}$;
    return to x;
    set $\psi(x)$.vacatedNeighbor $= true$;
    return *settled*;
  else
    return to x;
    return *settled*;

D. *Pseudocode of Algorithm 4* `Parallel_Probe`*()*

Algorithm 4: `Parallel_Probe()`

Input: Current DFS-head x with settled agent $\psi(x)$, and A_{scout}

Output: Next port p_{xy}

```
\psi(x).probeResult \leftarrow \perp; \psi(x).checked \leftarrow 0;
while \psi(x).checked < \delta_x do
    A_{scout} = \{a_1, \dots, a_s\} in the increasing order ID;
    \Delta' \leftarrow \min(s, \delta_x - \psi(x).checked);
    for j \leftarrow 1 to \Delta' do
        a \leftarrow next agent in A_{scout};
        if \psi(x).parent.Port = j + \psi(x).checked then
            j \leftarrow j + 1; \Delta' \leftarrow \min(s + 1, \delta_x - \psi(x).checked);
        a.scoutPort \leftarrow j + \psi(x).checked;
        move via a.scoutPort to reach y;
        a.scoutEdgeType \leftarrow type(\{x, y\});
        if \xi(y) \neq \perp then
            \psi(y) \leftarrow \xi(y);
            a returns to x;
        else
            if \xi(y) = \perp \wedge p_{yx} = 1 then
                \psi(y) \leftarrow \perp;
                a returns to x;
            else
                z \leftarrow port-1 neighbor of y;
                a.scoutP1Neighbor \leftarrow \xi(z);
                a.scoutPortAtP1Neighbor \leftarrow p_{zy};
                if \xi(z) \neq \perp then
                    a returns to x;
                    check \exists b \in A_{scout} : b.scoutP1Neighbor = \xi(z) \wedge b.scoutPortAtP1Neighbor = p_{zy};
                    if b found then
                        \psi(y) \leftarrow b
                    else
                        \psi(y) \leftarrow \perp
                else
                    if \xi(z) = \perp \wedge p_{zy} = 1 then
                        \psi(y) \leftarrow \perp;
                        a returns to x;
                    else
                        w \leftarrow port-1 neighbor of z;
                        a.scoutP1P1Neighbor \leftarrow \xi(w);
                        a.scoutPortAtP1P1Neighbor \leftarrow p_{wz};
                        if \xi(w) = \perp then
                            \psi(y) \leftarrow \perp
                        else
                            a returns to x;
                            check \exists c \in A_{scout} : c.scoutP1Neighbor = \xi(w) \wedge c.scoutPortAtP1Neighbor = p_{wz};
                            if c found then
                                check \exists b \in A_{scout} : b.scoutP1Neighbor = c \wedge b.scoutPortAtP1Neighbor = p_{zy};
                                if b found then
                                    \psi(y) \leftarrow b
                                else
                                    \psi(y) \leftarrow \perp
                            else
                                \psi(y) \leftarrow \perp
        a.scoutResult \leftarrow \langle p_{xy}, a.scoutEdgeType, \psi(y).nodeType, \psi(y) \rangle;
    \psi(x).checked \leftarrow \psi(x).checked + \Delta';
    \psi(x).probeResult \leftarrow highest priority edge from a \in A_{scout} based on \psi(y).nodeType;
return p_{xy} from \psi(x).probeResult;
```

E. Pseudocode of Algorithm 5 `RootedAsync()`

Algorithm 5: `RootedAsync()`

Input: A set of k agents at root node v_0 in G
$A \leftarrow$ set of agents;
For each $a \in A$, initialize all variables to $\perp$;
for $a \in A$ **do**
    a.state $\leftarrow$ *unsettled*;
$A_{unsettled} \leftarrow A$;
$A_{vacated} \leftarrow \emptyset$;
while $A_{unsettled} \neq \emptyset$ **do**
    $v \leftarrow$ current node;
    $A_{scout} \leftarrow A_{unsettled} \cup A_{vacated}$;
    $a_{min} \leftarrow$ Lowest ID agent in A_{scout} ;
    if *there is no settled agent in* v **then**
        $\psi(v) \leftarrow$ agent with highest ID in $A_{unsettled}$;
        $\psi(v)$.state $\leftarrow$ *settled*;
        $\psi(v)$.parent $\leftarrow$ (a_{min}.prevID, a_{min}.childPort);
        a_{min}.childPort $\leftarrow \perp$;
        $\psi(v)$.parentPort $\leftarrow a_{min}$.arrivalPort;
        $A_{unsettled} \leftarrow A_{unsettled} - \{\psi(v)\}$;
        if $A_{unsettled} = \emptyset$ **then**
            break;
    a_{min}.prevID $\leftarrow \psi(v)$.ID;
    if $\delta_v \geq k-1$ **then**
        run `Parallel_Probe`($\psi(v), A_{scout}$) for $k-1$ ports;
        send unsettled agents to empty neighbors;
        break;
    $\psi(v)$.sibling $\leftarrow a_{min}$.siblingDetails;
    a_{min}.siblingDetails $\leftarrow \perp$;
    nextPort $\leftarrow$ `Parallel_Probe`($\psi(x), A_{scout}$);
    $\psi(v)$.state $\leftarrow$ `Can_Vacate()`;
    if $\psi(v)$.*state* =*settledScout* **then**
        $A_{vacated} \leftarrow A_{vacated} \cup \{\psi(v)\}$;
        $A_{scout} \leftarrow A_{unsettled} \cup A_{vacated}$;
    if *nextPort* $\neq \perp$ **then**
        $\psi(v)$.recentPort $\leftarrow$ nextPort;
        a_{min}.childPort $\leftarrow$ nextPort;
        if $\psi(v)$.*recentChild* $= \perp$ **then**
            $\psi(v)$.recentChild $\leftarrow$ nextPort;
        else
            a_{min}.siblingDetails $\leftarrow a_{min}$.childDetails;
            a_{min}.childDetails $\leftarrow \perp$;
            $\psi(v)$.recentChild $\leftarrow$ nextPort;
        All agents in A_{scout} move through nextPort;
    else
        a_{min}.childDetails $\leftarrow$ ($\psi(v)$.ID, $\psi(v)$.portAtParent);
        a_{min}.childPort $\leftarrow \perp$;
        $\psi(v)$.recentPort $\leftarrow \psi(v)$.parentPort;
        All agents in A_{scout} move though $\psi(v)$.parentPort;
`Retrace`($A_{vacated}$);

F. Pseudocode of Algorithm 6 `Retrace()`

Algorithm 6: `Retrace()`

Input: $A_{vacated}$ - set of agents with state *settledScout*

while $A_{vacated} \neq \emptyset$ **do**
  $v \leftarrow$ current node;
  $a_{min} \leftarrow$ Lowest ID agent in $A_{vacated}$;
  if $\xi(v) = \perp$ **then**
    // no settled agent present at v, it must be in $A_{vacated}$
    find $a \in A_{vacated}$ with a.ID $= a_{min}$.nextAgentID at v;
    a.state $\leftarrow$ *settled*;
    $A_{vacated} \leftarrow A_{vacated} - \{a\}$;
    $a_{min} \leftarrow$ Lowest ID agent in $A_{vacated}$;
    $\psi(v) \leftarrow a$;
  // If all agents are settled, retrace is complete
  if $A_{vacated} = \emptyset$ **then**
    break;
  // Determine next move in post-order traversal
  if $\psi(v)$.*recentChild* $\neq \perp$ **then**
    if $\psi(v)$.*recentChild* $= a_{min}$.*arrivalPort* **then**
      if a_{min}.*siblingDetails* $= \perp$ **then**
        $\psi(v)$.recentChild $\leftarrow \perp$;
        $(a_{min}$.nextAgentID$, a_{min}$.nextPort$) \leftarrow \psi(v).parent$;
        a_{min}.siblingDetails $\leftarrow \psi(v)$.sibling;
      else
        $(a_{min}$.nextAgentID$, a_{min}$.nextPort$) \leftarrow a_{min}$.siblingDetails;
        a_{min}.siblingDetails $\leftarrow \perp$;
        $\psi(v)$.recentChild $\leftarrow a_{min}$.nextPort;
    else
      a_{min}.nextPort $\leftarrow \psi(v)$.recentChild;
      Check if $\exists a \in A_{vacated}$: a.parent $= (\psi(v)$.ID$, \psi(v)$.recentChild$)$;
      if a *found* **then**
        a_{min}.nextAgentID $\leftarrow a$.ID;
        a_{min}.nextPort $\leftarrow \psi(v)$.recentChild;
  else
    $(parentID, portAtParent) \leftarrow \psi(v)$.parent;
    a_{min}.nextAgentID $\leftarrow parentID$;
    a_{min}.nextPort $\leftarrow \psi(v)$.parentPort;
    a_{min}.siblingDetails $\leftarrow \psi(v)$.sibling;
  All agents in $A_{vacated}$ move through a_{min}.nextPort;