The Arvy Distributed Directory Protocol

Pankaj Khanchandani
ETH Zurich
Zurich, Switzerland
kpankaj@ethz.ch

Roger Wattenhofer
ETH Zurich
Zurich, Switzerland
wattenhofer@ethz.ch

ABSTRACT

In this paper we consider the problem of designing a distributed directory service. The two classic directory service protocols are Arrow [5] and Ivy [11]. Arrow performs well if the network is a tree, while Ivy performs well on complete graphs. However, there are graphs for which both Arrow and Ivy yield poor performance. In this paper, we propose a new distributed directory protocol, Arvy. Arvy is a natural extension of both Arrow and Ivy, generalizing both, while keeping their simplicity and strengths. Our main contribution is to prove Arvy's correctness, in asynchronous networks with concurrent requests, for arbitrary topologies. Regarding performance, we show that Arvy achieves constant competitive ratio on rings using constant space per node.

CCS CONCEPTS

• **Theory of computation → Online algorithms**; **Graph algorithms analysis**; **Distributed algorithms**; **Concurrent algorithms**; • **Networks → Token ring networks**.

KEYWORDS

Arrow, Ivy, distributed directory, shared object

ACM Reference Format:
Pankaj Khanchandani and Roger Wattenhofer. 2019. The Arvy Distributed Directory Protocol. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19), June 22–24, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3323165.3323181

1 INTRODUCTION

The Arrow [5] and Ivy [11] protocols were originally designed to solve the *cache-coherence* problem in a multiprocessor system. In such a system, several processes may want to modify a memory location or a shared data item. To avoid having multiple and incoherent copies of the data item, the access to modify the data is granted sequentially to the requesting processes so that only one of them can modify the data at a time. Formulating in abstract terms, the core problem is how to coordinate access to a shared resource or a token. The challenge is that a node requesting the resource has to find its location using a *distributed directory* protocol. Multiple independent instances of the distributed directory protocol in parallel can be used to coordinate access to multiple data items.

Due to the abstract nature of the problem, it can be applied in various settings. For example, it can be used to coordinate access to a mobile server or a set of servers. Another example is a service that globally orders transactions that are concurrently issued by arbitrary nodes. Such a transaction ordering service is nowadays known as a "blockchain". While the focus of blockchain research is on fault-tolerance, the focus of distributed directory research is on efficiency, which in turn heavily depends on the network topology. Eventually, the two areas may merge.

In an asynchronous and reliable network, where the messages are eventually delivered and not lost, both the Arrow and Ivy protocols ensure that every request to the shared resource or the token is eventually satisfied. We briefly describe the operation of these protocols when the requests to the token are sequential, i.e., there is at most one outstanding request in the network at any time. In such a scenario, Arrow and Ivy both maintain a rooted directed tree. The root of the tree stores the token, and the edges of the tree are directed towards the root. When a node requests the token, the tree is re-rooted to the requesting node as it is the new bearer of the token.

Both Arrow and Ivy follow different rules to re-root the tree. Let r be the current root of the tree and r' be the node that issued the new request. The Arrow protocol re-roots the tree by reversing all directed edges (i.e., "arrows") on the path from r' to r. The Ivy protocol re-roots the tree by making r' the parent of *every* node on the path from r' to r. In other words, Arrow only changes the direction of the edges but not the edges themselves, which makes the Arrow protocol perfectly suited for tree topologies. Ivy on the other hand constantly changes the topology, short-cutting whenever possible, which makes the Ivy protocol more suitable for general graphs. If we have concurrent requests, understanding both protocols becomes more involved, and in particular Ivy becomes nontrivial.

In this paper, we introduce the **Arvy** protocol, which is a natural generalization of the **Ar**row and I**vy** protocols. The tree modification rule used by Arrow or Ivy are specific instances of the rule used by the Arvy protocol. For example, if $r', \ldots, v, w, \ldots, r$ is the path from r' to r in the sequential case, then in order to re-root the tree to r', Arvy may change the parent of w to v as in Arrow, or to r' as in Ivy, or to any other node between r' and v. This flexibility makes Arvy really a family of protocols, where Arrow and Ivy are just special cases.

As our main contribution, we show that Arvy is correct. Concretely, Arvy eventually satisfies every request to the shared resource in an asynchronous network. While Arvy's correctness is natural if the requests are issued sequentially, it is exciting that Arvy

SPAA '19, June 22–24, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6184-2/19/06…$15.00
https://doi.org/10.1145/3323165.3323181

continues to be correct when the requests are issued concurrently in an asynchronous network.

We can measure the performance of a distributed directory protocol using the *competitive ratio*. The idea is to compare the cost of the given protocol, which does not know the location of future requests, to the cost of the best protocol that already knows the future requests at initialization. Concretely, the *cost* of the protocol for a given sequence of requests is the total distance traversed by the messages to satisfy all the requests. The *optimal cost* for a given sequence of requests is the minimum cost incurred by any protocol that knows that sequence of requests already. The *competitive ratio* is an upper bound on the ratio of the cost of the protocol and the optimal cost for every sequence of requests.

As a second contribution, we illustrate the benefits of Arvy by analyzing its competitive ratio on rings. We show that Arvy achieves a constant competitive ratio on a ring of n nodes using constant space per node, where, interestingly, both Arrow and Ivy have an $\Omega(n)$ competitive ratio. More complex distributed directory protocols exist, e.g. [14], but they do not yield a constant competitive ratio on rings using constant space per node.

2 RELATED WORK

Raymond's tree based mutual exclusion algorithm [13] predates the similar Arrow protocol by Demmer and Herlihy [5]. The Arrow protocol has inspired lots of research, since it is not only practical but also theoretically interesting. Kuhn et al. [10] show that the competitive ratio of Arrow is $O(\log D)$, where D is the diameter of the spanning tree used by Arrow. In their work, the cost used in the competitive ratio also takes into account the arrival and waiting time of the requests. Ghodselahi et al. [7] show that running Arrow on a probabilistic distribution over the right family of trees, as obtained from an FRT embedding [6], yields an expected competitive ratio of $O(\log n)$ on a general graph of n nodes. As these results show, the performance of the Arrow protocol is limited by the quality of the tree or the distribution of trees [7] chosen initially. However, the network topology may be such that no single tree is good for every pair of points (as in a ring for example). The Arvy protocol, on the other hand, offers flexibility by allowing to change the tree during the operation of the protocol.

Li and Hudak's Ivy protocol [11] was discovered before the Arrow protocol. While the original work on Arrow already included a correctness proof, Ivy's correctness for concurrent requests is more involved and is treated by Bouabdallah et al. [3] in a separate work. However, the invariant proof about messages in transit and parent pointers misses some cases. As Ivy is a special case of Arvy, and we show that Arvy is correct in all cases, the correctness of Ivy is directly implied. Ginat et al. [8] show that Ivy has $O(\log n)$ amortized cost per request on a complete graph of n nodes with unit weight edges.

A separate line of research for solving the distributed directory problem is based on *sparse covers* [1] or similar structures. A sparse cover of a graph is a collection of connected components of the graph such that every node is in some component. The *radius* of the sparse cover is the maximum radius of a connected component in the collection. The *degree* of the sparse cover is the maximum number of connected components in which any given vertex appears. The protocols based on sparse covers typically work by building a hierarchy of sparse covers of increasing radii but low degree. A request to a shared resource is then satisfied by keeping the interaction between the nodes up to the level in the hierarchy, whose radius is just large enough. Examples of such solutions are [2, 4, 9] and [14]. These solutions are usually quite involved as each node has to simultaneously operate at $O(\log n)$ levels in the hierarchy, where n is the number of nodes in the network. For example, when these solutions are used on a ring of n nodes, then they additionally need $O(\log n)$ space per node and only achieve $O(\log n)$ competitive ratio. However, Arvy can achieve constant competitive ratio on rings by only using constant space per node and is much simpler.

3 MODEL

We consider a connected network $G = (V, E)$ where V is the set of nodes in the network and E is the set of edges connecting them. A node in the network can send a message to a neighboring node, receive a message from a neighboring node, do some computation, and store some data. Routing is considered to be solved using standard methods, i.e., a message sent from a source to a destination will find the destination on the shortest path. The messages can be delayed arbitrarily but they are never lost, i.e., every message is eventually delivered.

The nodes in the network share a single *token*. Each node can request the token at any time. A *request* is identified by the pair (v, t), where a node $v \in V$ requests the token at a time t. A request (v, t) is considered *satisfied* as soon as v receives the token at a time $t' \geq t$. We assume that a node does not issue another request before the last request is satisfied. The task is to design an algorithm for every node so that every token request is satisfied. The nodes cannot make any assumption about future requests except that a node has at most one outstanding request as mentioned before.

The assumption regarding at most one outstanding request per node is only needed for simplicity. It can be taken care of by letting the further requests wait until the token arrives, at which point the all outstanding requests can be satisfied in one fell swoop. Note that there is no bound on message delays, and that the nodes can only assume that a message sent will eventually be delivered. The following section describes the Arvy protocol, which solves this problem.

4 THE ARVY PROTOCOL

Algorithm 1 gives the pseudocode of the Arvy protocol. The idea is that each node v keeps a pointer to a parent node $p(v)$, which points in the direction of the token Additionally, each node v may also have a next pointer $n(v)$ to a next node that is waiting for the token.

Initially, the parent pointers form a rooted tree so that the pointers are directed towards the root and the root points to itself. The token is situated at the root and the next pointers are empty when the algorithm starts. If a node v wants the the token, it sends a "find by v" message to its parent and also changes its parent pointer $p(v)$ to v, since v expects to receive the token soon (Lines 2 and 3). If a node w receives a "find by v" message from a node u, it first changes its parent pointer $p(w)$ to either the source of the message v or any other node that had received and forwarded the "find by v"

Algorithm 1 The Arvy Protocol: The parent pointer $p(v)$ of each node v is initialized so that they form a rooted tree pointing towards a root r and $p(r) = r$. The next pointer $n(v) = \perp$ for every node v initially. The initial location of the token is at the root. Node v does not issue a duplicate request if it has one outstanding.

```
1:  procedure RequestToken(v)                    ▷ v requests the token
2:      Send message "find by v" to p(v)
3:      p(v) ← v
4:  end procedure

5:  procedure ReceiveMessage(w, v, u)
                ▷ w receives a message "find by v" from u
6:      f ← p(w)
7:      p(w) ← NewParent(v)
8:      if f ≠ w then
9:          Send (forward) "find by v" message to f
10:     else
11:         n(w) = v
12:         if w has the token then
13:             SendToken(w)
14:         end if
15:     end if
16: end procedure

17: procedure NewParent(v)
18:     Return v OR any node that had received and forwarded v's
        current "find by v" message
19: end procedure

20: procedure ReceiveToken(v)                    ▷ v receives the "token"
21:     Use token to satisfy outstanding request
22:     SendToken(v)
23: end procedure

24: procedure SendToken(w)                       ▷ w sends the "token"
25:     if n(w) ≠ ⊥ then
26:         Send "token" to n(w)
27:         n(w) = ⊥
28:     end if
29: end procedure
```

message (Line 7). We do not make any assumption on the decision procedure of the new parent except that it will return some node that was visited by the "find by v" message. Depending on the specific implementation, additional parameters may be required. Our pseudocode omits these details for simplicity. Whether the "find by v" message is forwarded or not depends on the value of $p(w)$ or f when the message is received (Line 6). If $f \neq w$, then w just forwards the "find by v" message to f assuming that f is the way to the token (Line 9). If $f = w$, then we are hoping that w either has the token or will have the token soon. If w has the token, then it simply sends the token to v (Line 13). Otherwise, it sets its next pointer $n(w)$ to v so that it can forward the token to v once it is received (Line 11).

The token handling itself is easy. If a node v receives the token (Line 20), then it first uses it (Line 21), sends it to the next node $n(v)$ if one exists (Line 26) and then clears $n(v)$ (Line 27).

We show a sample execution in Figure 1 for five nodes. Initially, the token is at node a (Figure 1a). Then, a request to the token by node d is forwarded by c (Figures 1b and 1c). However, node e requests the token before "find by d" message reaches its destination (Figure 1d). The request by e passes through c and ends up as the next pointer of d (Figures 1e and 1f). Note that the new parent pointer of node d is e and the structure has changed. The "find by d" message is still stuck on way to a from c. Meanwhile, node b requests the token and is received by a, which sets its next pointer to b (Figures 1g and 1h). The "find by d" is finally received by a but now it is forwarded again and ends at b (Figure 1i and Figure 1j). The structure has changed again as the new parent of node a is d. The token is then sent around according to the next pointers, i.e., node a passes it to b, node b to d and node d to e (Figures 1k and 1l). The token could have been sent around earlier. Also, many other structures are possible depending on which node is chosen as a parent when a message is received.

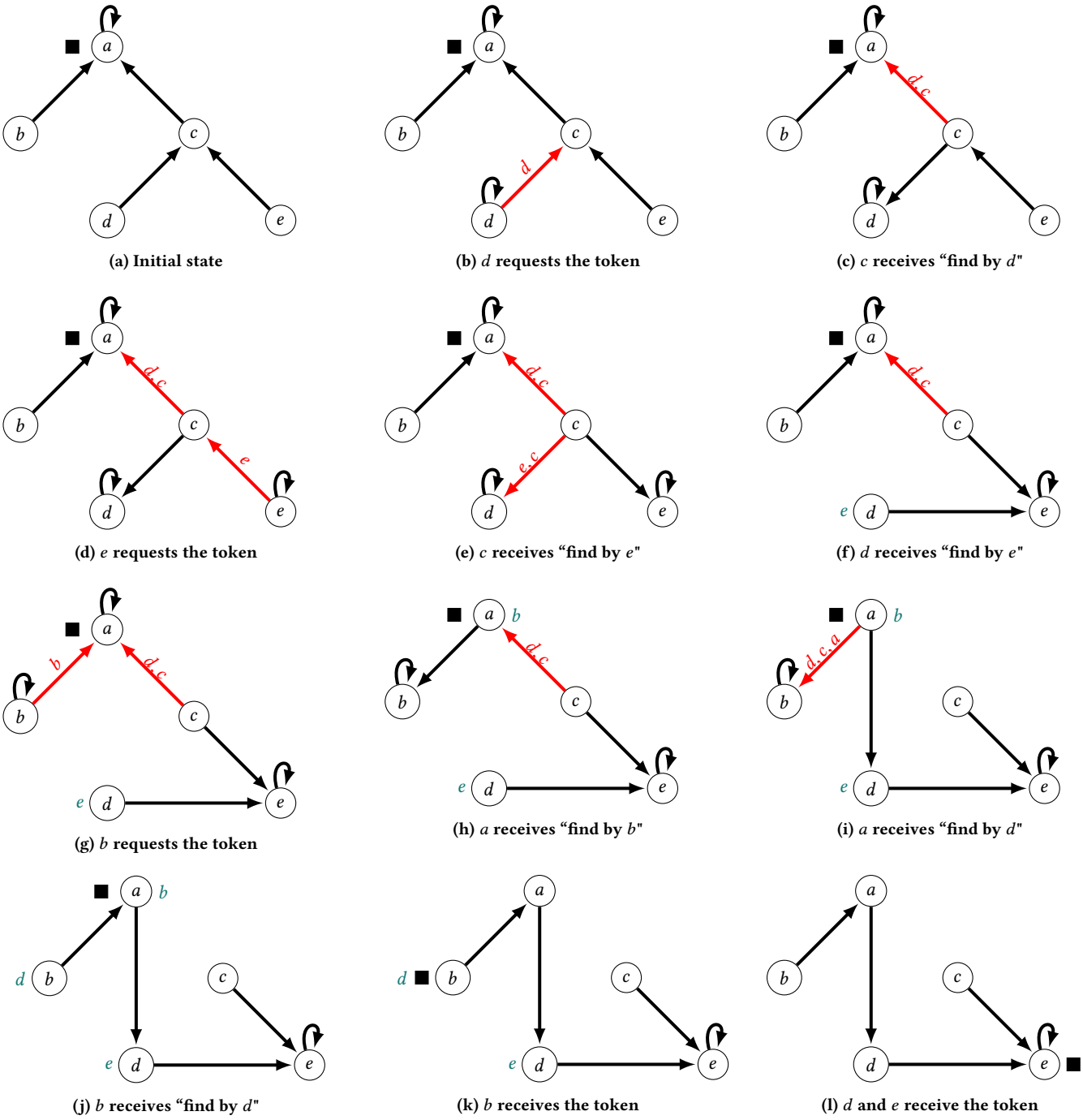
One can see from the example that the structure is highly dynamic. It is not obvious if a request message always finds the destination and is not forwarded forever. It is also not obvious if the token is passed around to every requesting node and does not miss a request or repeat one. In the following section, we prove that such situations do not occur.

5 CORRECTNESS ANALYSIS

To analyze the algorithm, we first define the state of the system. The *configuration* of the system is the state of each node, the find messages in transit and the location of the token. The state of a node v is the value of the variables $n(v)$ and $p(v)$. A find message in transit is a find message that was sent from a node u to a node v and is not yet received and processed by v. The location of the token is either a node u or the pair (u, v) when the token was sent by a node u to a node v but not yet received by v. The configuration changes when *events* occur. An event occurs when a node requests the token, receives a message, sends the token or receives the token. A separate event for sending a message is not required as that occurs only as a consequence of requesting the token or receiving a message. It is possible that two (or more) events occur at two (or more) different nodes concurrently. In such a case, we give the following lemma to show that the final configuration does not depend on the order of application of these events.

Lemma 1. *Let E be a set of concurrent events on a configuration C. Let π_1 and π_2 be two different permutations of E. Then, the configuration C_1 obtained after applying the events in the order π_1 is same as the configuration C_2 obtained after applying them in the order π_2.*

Proof. As the events are concurrent, there is at most one event in E per node. Let $e_v \in E$ be an event at a node v and $e_u \in E$ be an event at another node $u \neq v$. Say that e_v is a send or receive token event. Then, the event e_u must be a request token or receive message event as e_u occurs concurrently to e_v and there is only one token in the network. The change in the state caused by e_v depends on $n(v)$ or the location (w, v) of the token. Neither of these



(a) Initial state
(b) d requests the token
(c) c receives "find by d"
(d) e requests the token
(e) c receives "find by e"
(f) d receives "find by e"
(g) b requests the token
(h) a receives "find by b"
(i) a receives "find by d"
(j) b receives "find by d"
(k) b receives the token
(l) d and e receive the token

Figure 1: An example sequence of events on a graph of five nodes. The caption mentions the event and the figure shows the effect of the event. The outgoing black arrow from a node is its parent pointer. If the next pointer of a node is not empty, then its value is shown in green besides the node. The black box represents the token. A red arrow is a message in transit from the tail to the head of the arrow. The values above the red edge are the possible values of the new parent when the corresponding message is received. The network edges used for routing are not shown.

can be modified by the send or receive message event e_u. Thus, the change caused by a send or receive token event e_v is same whether a concurrent event e_u is applied before or after e_v.

Now, consider that e_v is a request token or a receive message event. Then, the change in the state caused by e_v only depends on $p(v)$ or both $p(v)$ and the message m received by v. Neither of these can be affected if e_u is a send token event or a receive token event or a request token event or a receive message event at another node. Thus, the change caused by a request token or a receive message event e_v is same whether a concurrent event e_u is applied before or after e_v.

Therefore, we can choose a fixed permutation π of events in E and the effect of applying the events according to the order π_1 or π_2 is same as if the events were applied in the order π. So, the configuration $C_1 = C_2$. □

To show the correctness, we need to show that every token request is satisfied eventually. Basically, we need to show that a find request is eventually received by a node v with $p(v) = v$ so that the request is not forwarded again, and that once the find request from a node stops being forwarded, the node eventually receives the token. We start by establishing some essential invariants about the configuration. We need to precisely define some terms to state those invariants.

We order the events into a total order as per their time of occurrence and breaking ties by node identifiers. Using Lemma 1, final configuration is not affected. We define an *execution* as an alternating sequence of events and configurations $C_0, e_1, C_1, e_2, C_2, \ldots$, where C_0 is the *initial configuration* and C_i is the configuration after the i^{th} event e_i has occurred. A *black edge* is a directed edge $(v, p(v))$ corresponding to a parent pointer of a node v. A *red edge* is a directed edge (v, w) corresponding to a "find by u" message in transit from node v to w. Thus, there is one-to-one correspondence between black edges and parent pointers, and between red edges and find messages.

Given a red edge $r = (v, w)$, we define $head(r) = w$ and $tail(r) = v$. A node v has a *self-loop* if its black edge points to itself or $p(v) = v$. We assume that a node does not request the token again if the node already requested the token and did not receive it yet. Note that a new "find by u" message is produced only when the node u requests the token. The find message is only forwarded by the other nodes until it is received by a node w with $p(w) = w$, which is the only way by which u receives the token. Thus, there is at most one "find by u" message for a given node u in any configuration. So, given a red edge r corresponding to a "find by u" message, we can define the *producer* of r, $prod(r)$, as the node u.

Let C_i be the configuration after the first $i \geq 0$ events of an execution σ have occurred. Given a red edge r in the configuration C_i, we define $visited_i(r)$ as the set of nodes including $prod(r)$ and the nodes that already received the find message corresponding to r during the first i events of σ. We say that a node w is *waiting* for a node u if $n(u) = w$ or if $n(v) = w$ and v is waiting for u. We define $waiting_i(u)$ as the set of nodes that are waiting for u in the configuration C_i. As the producer of a red edge r is unique, we also use $waiting_i(r)$ as the set of nodes that are waiting for $prod(r)$. A directed graph is a *directionless tree* if the graph is a tree when the directed edges are replaced by undirected edges. A directionless tree separates into two parts when a directed edge r from v to w is deleted. The part including v is called the *source component* of r or $src(r)$ and the part including w is called the *destination component* of r or $dst(r)$.

We show the following three properties for every reachable configuration. First, the black edges and the red edges form a directionless tree. Second, if we replace each red edge r by a *green edge* $(head(r), u)$, where u is a node visited by r or a node waiting for $prod(r)$, then the resulting graph is also a directionless tree. Third, the source component of each red edge r contains the nodes visited by r and the nodes waiting for $prod(r)$. Let B_i be the set of all the black edges *except* the self-loops and R_i be the set of red edges in C_i. Let BR_i be the graph of all the nodes V and the edges $B_i \cup R_i$. Given a red edge $r \in R_i$, the set of green edges $G_i(r)$ corresponding to the red edge r are the green edges $(head(r), u)$, where $u \in visited_i(r) \cup waiting_i(r)$. Let G_i be a set of green edges that are obtained by replacing each red edge $r \in R_i$ by a single green edge $g_r \in G_i(r)$. Let $\widetilde{G_i}$ be the set of all possible values of G_i. Let $\widetilde{BG_i}$ be the graphs $BG_i(V, B_i U G_i)$ for all $G_i \in \widetilde{G_i}$. See Figure 2 for an example. Note that all these terms are defined with respect to a configuration reached in a specific execution σ. However, we omit σ from the notation to avoid clutter and because the execution being referred to will always be clear from the context.

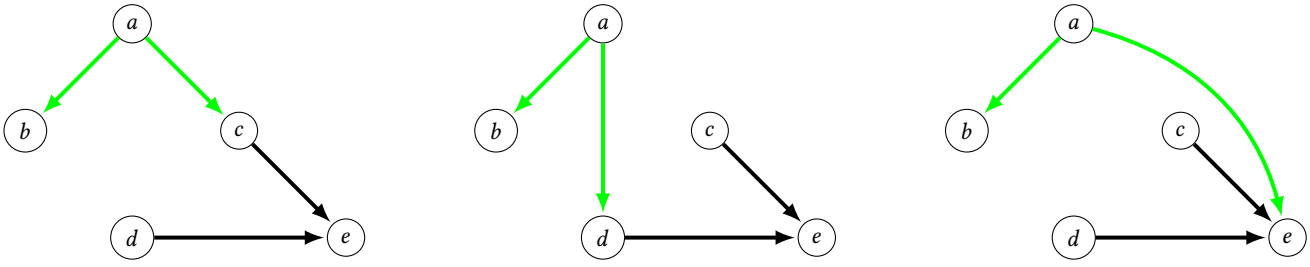
Lemma 2. *The following statements are true for any execution σ of Algorithm 1 on a set V of nodes.*

(1) The graph BR_i is a directionless tree.
(2) Each graph $BG_i \in \widetilde{BG_i}$ is a directionless tree.
(3) For each red edge r, we have $visited_i(r) \cup waiting_i(r) \subseteq src_i(r)$.

Proof. We prove the claim by induction on the events. Initially, we only have black edges in the configuration C_0, that form a directed tree. There are no red edges in the initial configuration so Part 1, Part 2 and Part 3 of the claim are true for C_0. Suppose the claim is true for C_i for $i \geq 0$. We check if the claim remains true for C_{i+1} after the event e_{i+1} occurs.

Send or receive token: If the event e_{i+1} is a send token or receive token, then the claim remains true as the black and red edges remain unchanged.

Request token: Consider that e_{i+1} is a request token event by a node v for which $p(v) = w$. Then, the node v sends find message to w and sets $p(v) = v$. As a result, the graph BR_{i+1} is obtained from BR_i by changing the color of the black edge (v, w) to red. As BR_i is a directionless tree by induction hypothesis, so is BR_{i+1} and Part 1 remains true. Note that for a red edge $r \in R_i$, we have $visited_i(r) = visited_{i+1}(r)$ and $waiting_i(r) = waiting_{i+1}(r)$. Thus, the set $\widetilde{G_i}$ is the set of all possible ways to replace all the red edges in BR_{i+1}, except the red edge (v, w), by green edges. Also, the red edge (v, w) can only be replaced by the green edge (w, v) as the node v just requested the token and $visited_{i+1}((v, w)) \cup waiting_{i+1}((v, w)) = \{v\}$. So, the graphs $\widetilde{BG_{i+1}}$ can be generated by enumerating through all $BG_i \in \widetilde{BG_i}$ and replacing the black edge (v, w) by the green edge (w, v). As the graphs $\widetilde{BG_i}$ are directionless trees by induction hypothesis, the graphs $\widetilde{BG_{i+1}}$ are also directionless trees and Part 2 remains true. For Part 3, we note that BR_{i+1} is obtained from BR_i by changing the color of the single edge (v, w) to red. Thus, for every red edge $r \in R_i$, we have $src_i(r) = src_{i+1}(r)$. Moreover, we have $visited_i(r) = visited_{i+1}(r)$ and $waiting_i(r) = waiting_{i+1}(r)$ for



(a) $(b, a) \to (a, b), (c, a) \to (a, c)$ **(b)** $(b, a) \to (a, b), (c, a) \to (a, d)$ **(c)** $(b, a) \to (a, b), (c, a) \to (a, e)$

Figure 2: The graphs $\widetilde{BG}_6$ obtained from Figure 1g by replacing the red edges $r_1 = (b, a)$ and $r_2 = (c, a)$ with the green edges in $G_6(r_1)$ and $G_6(r_2)$ respectively.

all $r \in R_i$. Thus, for all $r \in R_{i+1} \backslash \{(v, w)\}$, Part 3 holds by induction hypothesis. For $(v, w) \in R_{i+1}$, we have $visited_{i+1}((v, w)) \cup waiting_{i+1}((v, w)) = \{v\} \subseteq src_{i+1}((v, w))$ and Part 3 is true for all $r \in R_{i+1}$.

Receive message by a node with a self-loop: Consider that e_{i+1} is a find message received by a node w with a self-loop and a node v sent that find message. Let r be the red edge (v, w) in C_i.

For Part 1, we first note that parent $p(w)$ is updated to a node $u \in visited_i(r)$. So, deleting the red edge r from BR_i and adding the edge (w, u) yields the graph BR_{i+1}. By Part 1 of the induction hypothesis, the graph BR_i is a directionless tree and therefore, removing the red edge r from BR_i breaks it into two parts, say A and B. Using Part 3 of the induction hypothesis, $visited_i(r) \subseteq src_i(r)$ and so $u \in src_i(r)$. Thus, adding the edge (w, u) after removing the red edge r from BR_i connects the parts A and B again. So, the graph BR_{i+1} is a directionless tree and Part 1 remains true.

For Part 2, we consider two cases. In the first case, suppose that there is no red edge r' such that $w \in waiting_i(r')$. So, we have $visited_i(r'') = visited_{i+1}(r'')$ and $waiting_i(r'') = waiting_{i+1}(r'')$ for each red edge $r'' \in R_i \backslash \{r\}$. Consider the set of graphs $BG'_i \subseteq \widetilde{BG}_i$ for which the red edge r was replaced by the green edge (w, u). Therefore, the set of graphs $\widetilde{BG}_{i+1}$ can be obtained from $BG'_i \subseteq \widetilde{BG}_i$ by taking the ones with the green edge (w, u) and changing its color to black. As the graphs $\widetilde{BG}_i$ are directionless trees by induction hypothesis, the graphs $\widetilde{BG}_{i+1}$ are directionless trees as well and Part 2 remains true when there is no red edge r' so that $w \in waiting_i(r')$.

Next, consider the second case when there is a red edge r' so that $w \in waiting_i(r')$. Observe that such an edge r' is unique. Indeed, if there are red edges r'_1 and $r'_2 \neq r'_1$ so that $w \in waiting_i(r'_1)$ and $w \in waiting_i(r'_2)$, then there are nodes w_1 and w_2 for whom $n(w_1) = n(w_2) := q$, but the node q does not issue another find request until the previous one is satisfied, so either $n(w_1)$ or $n(w_2)$ is $\perp$, a contradiction. Consider the graph $\widetilde{G} \subseteq \widetilde{BG}_i$ so that the red edge r was replaced with the green edge (w, u) and the red edge r' was reversed into a green edge, i.e.,

$$\widetilde{G} = \{BG \mid BG \in \widetilde{BG}_i, (w, u) \in E(BG), g \in E(BG)\},$$

where $E(X)$ denotes the set of edges of a graph X and g' is the green edge $(head(r'), tail(r'))$. Let $\widetilde{H}$ be the graphs obtained from $\widetilde{G}$ by changing the color of the green edge (w, u) to black. Note that $visited_i(r'') = visited_{i+1}(r'')$ and $waiting_i(r'') = waiting_{i+1}(r'')$ for each $r'' \in R_i \backslash \{r, r'\}$. Therefore, we have

$$\widetilde{BG}_{i+1} = \{H' \mid V(H') = V, E(H') = E(H) \backslash \{g'\} \cup \{g_{r'}\}, g_{r'} \in G_{i+1}(r'), H \in \widetilde{H}\}.$$

The graphs $\widetilde{H}$ are directionless trees as they differ from the graphs $\widetilde{G} \subseteq \widetilde{BG}_i$ in the color of a single edge and the graphs $\widetilde{BG}_i$ are directionless trees by the induction hypothesis. Thus, we only need to show that replacing the edge g' in each graph $\widetilde{H}$ with any green edge $g_{r'} \in G_{i+1}(r')$ yields a directionless tree. Assume for contradiction that there exists $H \in \widetilde{H}$ so that replacing g' with a green edge from $G_{i+1}(r')$ does not yield a directionless tree. The graphs $\widetilde{H}$ are directionless trees as they are obtained from the graphs $\widetilde{G} \subseteq \widetilde{BG}_i$ by only changing the color of an edge and $\widetilde{BG}_i$ are directionless trees by induction hypothesis. So, there is a graph $H \in \widetilde{H}$ that has a node $q \in src_H(g')$ such that $q \in visited_{i+1}(r') \cup waiting_{i+1}(r')$ because replacing g' with a green edge from $G_{i+1}(r')$ does not yield a directionless tree. Therefore, there is also a graph $G \in \widetilde{G}$ that has a node $q \in src_G(g')$ such that $q \in visited_{i+1}(r') \cup waiting_{i+1}(r')$ as H and G only differ in the color of a single edge. Consider such a graph G with the smallest number of *changed* red edges, where a red edge $r'' \in BR_i$ is considered *unchanged* if it is replaced with the reversed green edge $(head(r''), tail(r''))$. Let $R' \subseteq R_i$ be the set of red edges that were *changed* in BR_i to obtain the graph G, i.e., not counting a red edge if it was reversed into a green edge. Let GR' be the set of green edges that replaced the edges R' and G' be the graph obtained from BR_i by replacing the red edges R' with the green edges GR'. Pick some $r'' \in R'$ and let gr'' be the corresponding replacement green edge in GR'. Let G'' be the graph obtained by replacing the red edges in $R' \backslash \{r''\}$.

Note that $r' \notin R'$ as the green edge g' in G is just the reversed red edge r' and this does not count as a changed edge. So, the red edge r' is contained in both G' and G''. Also, both G' and G'' are directionless trees as they are obtained by replacing a subset of red edges R_i with the green edges, and reversing the remaining red edges into green edges yields a member of $\widetilde{BG}_i$, which are directionless trees by induction hypothesis. Let $q \in src_G(g')$ be a node such that $q \in visited_{i+1}(r') \cup waiting_{i+1}(r')$. As G and G' are the same trees if we ignore the direction and color of the edges and r' is the reversed edge g', we also have

$q \in dst_{G'}(r')$. So, we have $q \in src_{G''}(r')$, otherwise, we can remove r'' from R', contradicting its minimal size. As per our assumption, we have $q \in visited_{i+1}(r') \cup waiting_{i+1}(r')$. We have $visited_{i+1}(r') = visited_i(r')$ as the event e_{i+1} does not involve r' and $waiting_{i+1}(r') = waiting_i(r') \cup waiting_i(r)$ as $w \in waiting_i(r')$ by assumption. If the node $q \in visited_i(r') \cup waiting_i(r')$, then replacing g' in the graph G with a green edge from $G_{i+1}(r')$ yields a member of $\widetilde{BG}_i$, which are directionless trees. So, it must be that $q \in waiting_i(r) \subseteq src_i(r)$ using Part 3 of the induction hypothesis. Now, we consider different configurations of G'' depending on the location of q and r' relative to r''.

First, suppose that $\{q, r'\} \subseteq src_{G''}(r'')$, i.e., both q and r' lie in the source component of r'' in G''. Then, replacing r'' by a green edge cannot make $q \in dst_{G'}(r')$. Indeed, if we replace r'' in G'' with a green edge, we get G', which is a directionless tree. So, the replaced green edge is $(head(r''), u'')$ for some $u'' \in src_{G''}(r'')$. As $q \in src_{G''}(r'')$ by assumption, the location of q with respect to r' does not change, i.e., we have $q \in src_{G'}(r')$ because $q \in src_{G''}(r')$. Thus, it is not possible that $\{q, r'\} \subseteq src_{G''}(r'')$. Second, we also cannot have $\{q, r'\} \subseteq dst_{G''}(r'')$ due to the same argument as in the previous case. Third, suppose that $q \in src_{G''}(r'')$ and $r' \in dst_{G''}(r'')$. As $q \in src_{G''}(r')$, we have $head(r'') \in src_{G''}(r')$. So, after replacing the red edge r'' with green edge $(head(r''), u'')$ for some $u'' \in src_{G''}(r'')$, we get the directionless tree G' and $q \in src_{G'}(r')$, a contradiction.

Fourth, suppose that $q \in dst_{G''}(r'')$ and $r' \in src_{G''}(r'')$. Say that r'' was replaced with a green edge $(head(r''), u'')$ to yield G', where $q \in dst_{G'}(r')$. Then, the node $u'' \in dst_{G''}(r')$. Also, the node $w = head(r) \in dst_{G''}(r'')$, otherwise, we have $w \in src_{G''}(r'')$ and we can replace the black edge (w, u) in G'' with the green edge (w, q) to obtain a graph T, where $q \in waiting_i(r)$. The graph T is not a directionless tree as there is an edge (w, q) is between $(src_T(r''), dst_T(r''))$ apart from the edge r'' itself. However, the graph T must be a direction tree as each of the remaining red edge in T can be replaced with a reversed green edge to yield a member of $\widetilde{BG}_i$, which are directionless trees. So, it must be that $w \in dst_{G''}(r'')$. However, if $w \in dst_{G''}(r'')$, then the edges r' and r'' in G'' can be respectively replaced with the green edges $(head(r'), w)$ and $(head(r''), u'')$ to yield a graph T, which is not a directionless tree as there are two paths, namely $(head(r'), w)$ and $(head(r''), u'')$, between $dst_T(r'')$ and $dst_T(r')$. Recall that $w \in waiting_i(r)$ by assumption. So, the graph T must be a directionless tree as replacing each of the remaining red edges in T with a reversed green edge yields a graph in $\widetilde{BG}_i$, which are directionless trees by induction hypothesis. So, it is also not possible that $q \in dst_{G''}(r'')$ and $r' \in src_{G''}(r'')$ and Part 2 of the claim remains true.

For Part 3, we check if the claim remains true for each red edge $r' \in R_i \backslash \{r\}$ as the red edge r is replaced with a black edge in BR_{i+1}. As the event e_{i+1} involves $r \neq r'$, we have $visited_{i+1}(r') = visited_i(r')$ for $r' \in R_i \backslash \{r\}$. We have two cases depending on whether $r' \in dst_i(r)$ or $r' \in src_i(r)$. First, suppose that $r' \in dst_i(r)$. Then, we have $src_{i+1}(r') = src_i(r')$ whether $w \in src_i(r')$ or $w \notin src_i(r')$. If $w \notin src_i(r')$, then $w \notin waiting_i(r')$ by Part 3 of the induction hypothesis and $waiting_i(r') = waiting_{i+1}(r')$. As each term in Part 3 of the claim remains unchanged, it remains true. If $w \in src_i(r')$, then we have $src_{i+1}(r') = src_i(r') \supseteq src_i(r)$. So, we have $src_{i+1}(r') = src_i(r') \cup src_i(r) \supseteq waiting_i(r') \cup waiting_i(r)$ using induction hypothesis. We also have $waiting_{i+1}(r') \subseteq waiting_i(r') \cup waiting_i(r)$ as it is possible that $w \in waiting_i(r')$. Thus, we get $waiting_{i+1}(r') \subseteq src_{i+1}(r')$. Moreover, we already have $src_{i+1}(r') = src_i(r')$ and $visited_{i+1}(r') = visited_i(r')$. So, we have $visited_{i+1}(r') \subseteq src_{i+1}(r')$ using induction hypothesis and Part 3 remains true when $r' \in dst_i(r)$.

Second, suppose that $r' \in src_i(r)$. Then, we have four possibilities depending on the location of the nodes v and u with respect to r'.

(1) Suppose that $\{v, u\} \subseteq dst_i(r')$. Then, we have $src_{i+1}(r') = src_i(r')$ as the red edge (v, w) in BR_i is replaced with the black edge (w, u) and neither $v \in src_i(r')$ nor $u \in src_i(r')$. Also, we have $waiting_{i+1}(r') = waiting_i(r')$ as $\{v, w\} \subseteq dst_i(r')$ and thus, $w \notin src_i(r') \supseteq waiting_i(r')$ by induction hypothesis. Moreover, we already have $visited_{i+1}(r') = visited_i(r')$. As all the terms involved in Part 3 remain unchanged, it remains true.

(2) Suppose that $\{v, u\} \subseteq src_i(r')$. Then, $src_{i+1}(r') = src_i(r')$ as (v, w) is replaced with (w, u) to obtain BR_{i+1} and $\{v, u\} \subseteq src_i(r')$. If $w \notin waiting_i(r')$, then we have $waiting_{i+1}(r') = waiting_i(r')$. Moreover, we already have $visited_{i+1}(r') = visited_i(r')$ and thus, Part 3 remains true as all the three terms remain unchanged. If $w \in waiting_i(r')$, then $waiting_{i+1}(r') = waiting_i(r') \cup waiting_i(r)$. If $waiting_i(r) \subseteq src_i(r')$, then $waiting_{i+1}(r') \subseteq src_i(r') = src_{i+1}(r')$ using induction hypothesis. Moreover, we have $visited_{i+1}(r') = visited_i(r') \subseteq src_i(r') = src_{i+1}(r')$ already and thus, Part 3 remains true. If $waiting_i(r) \nsubseteq src_i(r')$, then there is a node $q \in dst_i(r')$ so that $q \in waiting_i(r)$. But, then we can replace the edges r and r' in BR_i with the green edges (w, q) and $(head(r'), w)$ respectively to get a graph T, which is not a directionless tree as there are two paths, namely (w, q) and $(head(r'), w)$, between $dst_i(r')$ and $dst_i(r)$, a contradiction to Part 2 of the claim for $i + 1$. Thus, Part 3 remains true in this case.

(3) Suppose that $v \in dst_i(r')$ and $u \in src_i(r')$. Then, we have have $src_i(r') \subseteq src_{i+1}(r')$ as the red edge (v, w) is replaced with the black edge (w, u) to obtain BR_{i+1}. Since $w \notin src_i(r') \supseteq waiting_i(r')$, we have $waiting_{i+1}(r') = waiting_i(r') \subseteq src_i(r')$ and $src_i(r') \subseteq src_{i+1}(r')$. Also, we know that $visited_{i+1}(r') = visited_i(r') \subseteq src_i(r')$. So, we have $visited_{i+1}(r') \subseteq src_{i+1}(r')$ as well.

(4) Suppose that $v \in src_i(r')$ and $u \in dst_i(r')$. If $dst_i(r) \cap (visited_{i+1}(r') \cup waiting_{i+1}(r')) = \phi$, then $visited_{i+1}(r') \cup waiting_{i+1}(r') \subseteq src_{i+1}(r')$ again as the edge (v, w) is replaced with black edge (w, u) to obtain BR_{i+1}. Suppose there is a $q \in dst_i(r) \cap (visited_{i+1}(r') \cup waiting_{i+1}(r')) \neq \phi$. Then, we can replace the red edges r and r' with the green edges (w, u) and $(head(r'), q)$ respectively to yield a graph T, which is not a directionless tree as there are two paths, namely (w, u) and $(head(r'), q)$, between $dst_i(r)$ and $dst_i(r')$, contradicting Part 2 for $i + 1$. Therefore, we have $visited_{i+1}(r') \cup waiting_{i+1}(r') \subseteq src_{i+1}(r')$ when $r' \in src_i(r)$ as well and Part 3 of the claim is true for $i + 1$.

Receive message by a node without a self-loop: Lastly, consider that the event e_{i+1} is a receive message event by a node w from

a node v, where $p(w) = p$. Let r be the red edge (v, w) in BR_i and s be the edge (w, p). Using Part 3 of the induction hypothesis, we have $u \in visited_i(r) \subseteq src_i(r)$. The graph BR_{i+1} is obtained from BR_i by replacing the red edge (v, w) with the black edge (w, u) and changing the color of the black edge (w, p) to red. As BR_i is a directionless tree by Part 1 of the induction hypothesis, so is BR_{i+1}.

For Part 2, consider the graphs $\widetilde{G} \subseteq \widetilde{BG}_i$ in which r was replaced with the edge (w, u), i.e.,

$$\widetilde{G} = \{BG \mid BG \in \widetilde{BG}_i, (w, u) \in E(BG)\}.$$

Let $\widetilde{H}$ be the graphs obtained from $\widetilde{G}$ by changing the color of the green edge (w, u) to black. As $visited_i(r') \cup waiting_i(r') = visited_{i+1}(r') \cup waiting_{i+1}(r')$ for each $r' \in R_i \backslash \{r\}$, we have

$$\begin{aligned}\widetilde{BG}_{i+1} = \{H' \mid\ & V(H') = V,\\ & E(H') = E(H) \backslash \{(w, p)\} \cup \{(p, q)\},\\ & (p, q) \in G_{i+1}(r),\ H \in \widetilde{H}\}.\end{aligned}$$

Assume for contradiction that there is a graph $BG \in \widetilde{BG}_{i+1}$ that is not a directionless tree. The graphs $\widetilde{H}$ are directionless trees as they differ from $\widetilde{G} \subseteq \widetilde{BG}_i$ in only the color a single edge (w, u) and $\widetilde{BG}_i$ are directionless trees by induction hypothesis. So, there is a graph $H \in \widetilde{H}$ in which there is a node $q \in dst_H(s)$ such that $q \in visited_{i+1}(r) \cup waiting_{i+1}(r) = visited_i(r) \cup waiting_i(r) \cup \{w\}$ because replacing (w, p) with $(p, q) \in G_{i+1}r$ does not yield a directionless tree as per the assumption for contradiction. Then, there is also a graph $G \in \widetilde{G}$ having a node q with the same property. Consider such a graph G that can be obtained from BR_i by *changing* the smallest number of red edges into green edges, where a red edge is considered to be *changed* if the replaced green edge is not the reversed one. Let $R' \subseteq R_i$ be the set of red edges that were *changed* to get G from BR_i and let GR' be the green edges that replaced R'. Let G' be the graph obtained from BR_i by replacing the red edges in R' with the green edges GR' and keeping the remaining red edges $R_i \backslash R'$ unchanged, i.e., the graph G' is same as G except that the edges $R_i \backslash R'$ have different color and opposite direction. Fix a red edge $r' \in R'$ and let $gr' \in GR'$ be the corresponding replacement green edge. Let G'' be the graph obtained by replacing the red edges $R' \backslash r'$ by the corresponding green edges in $GR' \backslash gr'$.

Note that both G' and G'' are directionless trees as they are obtained by replacing a subset of the edges R_i, and the each of the remaining red edges can be reversed into a green edge to yield a member of $\widetilde{BG}_i$, which are directionless trees by induction hypothesis. Let q be the node with the property that $q \in dst_G(s)$ and $q \in visited_i(r) \cup waiting_i(r) \cup \{w\}$. Using the definitions of G and G', they are the same trees ignoring the directions and colors of the edges. So, the node $q \in visited_i(r) \cup waiting_i(r) \cup \{w\}$ also satisfies $q \in dst_{G'}(s)$. As R' is the smallest number of red edges that must be changed in BR_i to get the graph G that has a node $q \in dst_G(s)$ such that $q \in visited_i(r) \cup waiting_i(r) \cup \{w\}$, it cannot be that $q \in dst_{G''}(s)$, otherwise, the edge r' can be removed from R', contradicting its minimum size. So, we have $q \in src_{G''}(s)$ and we look for the location of r' in G'' that can make $q \in dst_{G'}(s)$ after r' is replaced with a green edge $g_{r'} \in G_i(r')$, i.e., an edge $(head(r'), p')$ for some $p' \in visited_i(r') \cup waiting_i(r') \subseteq src_i(r')$. We define the set of nodes $A'' = dst_{G''}((w, u))$, $C'' = dst_{G''}((w, p))$ and $B'' = src_{G''}((w, u)) \cap src_{G''}((w, p))$. So, we have $q \in A''$ or $q \in B''$. We consider three cases depending on the location of r'.

First, consider that $r' \in C''$. Then, either $p \in dst_{G''}(r')$ or $p \in src_{G''}(r')$. In either case, replacing r' with a green edge $g_{r'} \in G_i(r')$ does not change the position of $q \in A'' \cup B''$ with respect to the edge s or (w, p), i.e., $q \in src_{G'}(s)$. As $q \in dst_{G'}(s)$, it cannot be that $r' \in C''$.

Second, consider that $r' \in B''$. Then, either $w \in dst_{G''}(r')$ or $w \in src_{G''}(r')$. If $w \in dst_{G''}(r')$, then either $q \in dst_{G''}(r')$ or $q \in src_{G''}(r')$. In either case, the positions of q with respect to the edge s remains unchanged after replacing r' with a green edge $g'_r \in G_i(r')$, i.e., $q \in src_{G'}(s)$, contradicting the assumption that $q \in dst_{G'}(s)$. So, it is not possible that $r' \in B''$ and $w \in dst_{G''}(r')$. Next, suppose that $w \in src_{G''}(r')$ and $r' \in B''$ as before. Again, either $q \in src_{G''}(r')$ or $q \in dst_{G''}(r')$. If $q \in src_{G''}(r')$, then replacing r' with a green edge $g_{r'} \in G_i(r')$ implies $q \in src_{G'}(s)$, a contradiction. If $q \in dst_{G''}(r')$, then we replace the green edge (w, u) with (w, q) and reverse the remaining red edges in G'' into green edges to obtain a graph T. Note that $T \in \widetilde{BG}_i$, as $q \in visited_i(r) \cup waiting_i(r) \cup \{w\}$ by definition of q, and $q \neq w$ since $q \in dst_{G''}(r')$ and $w \in src_{G''}(r')$ by assumption, implying that $q \in visited_i(r) \cup waiting_i(r)$ and $(w, q) \in G_i r$. But, the graph T is not a direction tree as there is an edge (w, q) between $src_{G''}(r') \cap B''$ and $dst_{G''}(r')$ apart from the reversed edge r', a contradiction since $T \in \widetilde{BG}_i$, which are directionless trees by induction hypothesis.

Third, consider that $r' \in A''$. We have two cases, either $u \in dst_{G''}(r')$ or $u \in src_{G''}(r')$. If $u \in dst_{G''}(r')$, then irrespective of $q \in dst_{G''}(r')$ or $q \in src_{G''}(r')$, after replacing r' we have $q \in src_{G'}(s)$, a contradiction. If $u \in src_{G''}(r')$ and $q \in src_{G''}(r')$, then after replacing r' with a green edge from $G_i(r')$, we again have $q \in src_{G'}(s)$, a contradiction. If $u \in src_{G''}(r')$ and $q \in dst_{G''}(r')$, then replacing r' by the edge $(head(r'), q') \in G_i(r')$ can make $q \in dst_{G'}(s)$ if $q' \in dst_{G''}(s)$. However, in that case we can replace the edges (w, u) and r' by the edges (w, q) and $(head(r'), q')$ respectively, and reverse the remaining red edges into green edges, to get a graph $T \in \widetilde{BG}_i$, which are directionless trees. However, the graph T is not a direction tree as there are two paths, namely (q, w, p) and $(head(r'), q')$, between $dst_{G''}(r')$ and C'', a contradiction. Therefore, there is no $G \in \widetilde{G}$ in which there is a $q \in dst_G(s)$ such that $q \in visited_i(r) \cup waiting_i(r) \cup \{w\}$ and $\widetilde{BG}_{i+1}$ are directionless trees, establishing Part 2 of the claim for $i + 1$.

We check Part 3 of the claim for the red edges r and $r' \neq r$ depending on whether $r' \in A$, $r' \in B$ or $r' \in C$, where $A = src_i(r)$, $B = dst_i(r) \cap src_i(s)$ and $C = dst_i(s)$. As the event e_{i+1} is a message receive event by a node without a self-loop, we have $waiting_{i+1}(r'') = waiting_i(r'')$ for all $r'' \in R_i$. Also, the event e_{i+1} involves r so $visited_{i+1}(r') = visited_i(r')$ for $r' \neq r$. First, we check the claim for r. We have $visited_{i+1}(r) = visited_i(r) \cup \{w\} \subseteq A \cup B = src_{i+1}(r)$ using $visited_i(r) \subseteq src_i(r)$ from the induction hypothesis. Also, we have $waiting_{i+1}(r) = waiting_i(r) \subseteq A \subseteq src_{i+1}(r)$ using $waiting_i(r) \subseteq src_i(r) = A$ from Part 3 of the induction hypothesis. So, Part 3 remains true for r. Second, consider $r' \in C$. It could be that $p \in src_i(r')$ or $p \in dst_i(r')$. In either case, we have $src_{i+1}(r') = src_i(r')$. Moreover, we already have $visited_{i+1}(r') = visited_i(r')$ and $waiting_{i+1}(r') = waiting_i(r')$. Therefore, Part 3 remains true as all three terms in claim remain remain unchanged. Third, consider

$r' \in B$. Irrespective of whether $w \in src_i(r')$ or $w \in dst_i(r')$, we have $src_{i+1}(r') = src_i(r')$ and the claim remains true as in the previous case.

Fourth, consider $r' \in A$. Suppose that $\{u, v\} \subseteq dst_i(r')$ or $\{u, v\} \subseteq src_i(r')$. Then, we have $src_{i+1}(r') = src_i(r')$ and the claim remains true as in the previous case. Next, suppose that $u \in src_i(r')$ and $v \in dst_i(r')$. As the edge (v, w) was replaced with (w, u) to obtain BR_{i+1}, we have $src_{i+1}(r') = src_i(r') \cup B \cup C$ or $src_i(r') \subseteq src_{i+1}(r')$. Moreover, we already have $visited_{i+1}(r') \cup waiting_{i+1}(r') = visited_i(r') \cup waiting_i(r')$ and so, the claim remains true using induction hypothesis. Lastly, suppose that $u \in dst_i(r')$ and $v \in src_i(r')$. We have the following three cases.

(1) Consider that $(visited_i(r') \cup waiting_i(r')) \cap (B \cup C) = \phi$. Then, we have $visited_i(r') \cup waiting_i(r') \subseteq (A \cap src_i(r'))$ using induction hypothesis. As we already have $visited_{i+1}(r') \cup waiting_{i+1}(r') = visited_i(r') \cup waiting_i(r')$, the claim remain true.

(2) Consider a node $q \in ((visited_i(r') \cup waiting_i(r')) \cap B) \neq \phi$. Then, we can replace r and r' by the green edges (w, u) and $(head(r'), q)$ respectively, and reverse the remaining red edges into green edges, to yield a graph $T \in \widetilde{BG}_i$, which are directionless trees. However, the graph T is not a direction tree as there are two paths, namely (w, u) and $(head(r'), q)$, between A and B, a contradiction.

(3) Consider a node $q \in ((visited_i(r') \cup waiting_i(r')) \cap C) \neq \phi$. Then, we can replace r and r' by the green edges (w, u) and $(head(r'), q)$ respectively, and reverse the remaining red edges into green edges, to yield a graph $T \in \widetilde{BG}_i$, which are directionless trees. However, the graph T is not a direction tree as there are two paths, namely (u, w, p) and $(head(r'), q)$, between A and C, a contradiction.

Therefore, Part 3 remains true for each $r'' \in R_i$ when e_{i+1} is receive message event by a node without a self-loop and the claim remains true for any event e_{i+1}. □

Using the properties of the configuration established above, we can now show that a find request is always satisfied with the token. Given a configuration C_i for $i \geq 0$, we define $previous_i(w)$ of a node w as a node u such that $n(u) = w$ if such a u exists. Otherwise, we define $previous_i(w) = \perp$. Observe that $previous_i(w)$ is unique for $i \geq 0$. Indeed, a node w does not issue another find request until the current one is satisfied with the token, which is sent by another node, say w'. When w' sends the token, then $n(w')$ is set to $\perp$. So, there is at most a single node u such that $n(u) = w$. Let $p_0, p_1, \ldots, p_k$ be a sequence of nodes such that $p_0 = v$ and $p_j = previous_i(j-1)$ for $1 \leq j \leq k$. Observe that the values p_j for $0 \leq j \leq k$ are all distinct. Indeed, if they are not, then we have a smallest sequence of distinct nodes $q_0, q_1, \ldots, q_{k'}$ such that $q_j = previous_i(q_{(j-1) \bmod k'})$ for $0 \leq j \leq k'$. Let e_h, $h \geq 1$ be the latest event prior to e_i that resulted in such a configuration of the nodes q_j, $0 \leq j \leq k'$. Then, the event e_h was a find message received by a node q_b and produced by q_a, where $b = (a-1) \bmod k$ and $0 \leq a \leq k'$. Also, the previous configuration C_{h-1} is such that $q_j = previous_{h-1}(q_{(j-1) \bmod k'})$ for $0 \leq j \leq k'$, $j \neq a$. Using definition of $waiting_{h-1}(q_a)$ and Lemma 2, we have $q_b \in waiting_{h-1}(q_a) \subseteq src_{h-1}(r)$, where r is the red edge (q_a, q_b). However, we have $q_b \in dst_{h-1}(r)$ by definition of r, a contradiction. So, for every node v and $i \geq 0$, we can consider the sequence $v = p_0, p_1, \ldots, p_k$, where $p_{j+1} = previous_i(p_j)$, $previous_i(p_k) = \perp$ and define $top_i(v) = p_k$. The following lemma is an important consequence of the definition.

Lemma 3. *Let w be a node with a self-loop and $w' = top_i(w)$ for $i \geq 0$. Then, either w' has the token, or the token was already sent to w', or there is a red edge $r \in R_i$ such that $prod(r) = w'$.*

Proof. We describe the state $S(v)$ of a node v as a subset of $\{L, T, N\}$, where L means that v has a self-loop, T means that v has the token, and N means that $n(v) \neq \perp$. Initially, the state of every node is either $\{L, T\}$ or $\{\}$. Each state can undergo four possible transitions, which are send token, receive token, request token and receive message, with the restriction that a node does not request the token if it already has the token or an outstanding token request. Thus, one can conclude from the transition diagram that there are only five possible reachable states, which are $\{L, T\}$, $\{\}$, $\{T, N\}$, $\{L\}$ and $\{N\}$. Also, if $S(v) = \{L\}$ or $S(v) = \{N\}$, then the node v requested the token but has not received it yet.

First, suppose that $w' = w$. By assumption, node w has a self-loop. So, we have $S(w) = \{L, T\}$ or $S(w) = \{L\}$ as w has a self-loop. If $S(w) = \{L, T\}$, then w has the token and the claim is true. If $S(w) = \{L\}$, then node w requested the token. Thus, node w sent a find message. If the message was not received by a node u with a self-loop, then we have a $r \in R_i$ such that $prod(r) = w$ and the claim holds. If the message was received by a node u, then $n(u)$ was set to w. Since $w = top_i(w')$, we have $previous_i(w) = \perp$. So $n(u)$ was reset to $\perp$ after it was set to w and the token was sent to w, establishing the claim.

Second, suppose that $w' \neq w$. Using definition of $top_i(w)$, we have $n(w') \neq \perp$. Thus, out of the five reachable states, either $S(w') = \{N, T\}$ or $S(w') = \{N\}$. If $S(w') = \{N, T\}$, then w' has the token. If $S(w') = \{N\}$, then we already have that w' requested the token. As in the previous case, we have that either there is an edge $r \in R_i$ such that $prod(r) = w'$ or the token was already sent to w'. □

Theorem 4. *Every find request is received by a node with a self-loop.*

Proof. Using Lemma 2, a find message corresponding to a red edge r is received by a given node v at most once as after v receives the message, say event e_k, we have $v \in visited_i(r) \subseteq src_i(r)$ for all C_i, $i \geq k$. As the number of nodes is limited by n, the find message is eventually received by a node v that does not forward it. So, the node v must have a self-loop. □

Theorem 5. *Every token request is satisfied.*

Proof. Suppose that node v requests the token. Then, node v sends a find message, which is received by a node w with a self-loop, say event e_i, by Theorem 4. If $w' := top_i(w)$ has the token or the token was sent to it, then v receives the token once the nodes $w = q_0, q_1, \ldots, q_k = w'$ have used it, where $q_j = previous_i(q_{j-1})$ for $1 \leq j \leq k$. Otherwise, we conclude using Lemma 3 that w' has a find message in the network. Using Theorem 4, the find message is received by a node w'' with a self-loop, say event e_j, and $|waiting_j(top_j(v))| > |waiting_i(top_i(v))|$. So, whenever $top_i(v)$ does not have the token, an event e_j occurs later, and $|waiting_j(top_j(v))| > |waiting_i(top_i(v))|$. As soon

as $|waiting_j(top_j(v))| = n$, the token is at $top_j(v)$. Otherwise, there is a red edge $r \in R_j$ such that $prod(r) = top_j(v)$ by Lemma 3. As $|waiting_j(top_j(v))| = n$, every node and specifically $head(r) \in waiting_j(r)$. This is a contradiction because $head(r) \in dst_j(r)$ and $waiting_j(r) \subseteq src_j(r)$ by Lemma 2. Thus, node $top_{j'}(v)$ must have received the token for $j' < j$ and consequently, node v receives the token as well. □

6 NEWPARENT FOR RINGS

In this section, we apply Algorithm 1 to ring networks by designing an appropriate NewParent function. For simplicity, we consider a ring with n nodes and unit weight edges, where n is even. The idea is that we split the ring into two semi-circular parts and connect them via a "bridge". The semi-circular parts consist of parent pointers that coincide with the ring edges where as the bridge is a parent pointer into the other semi-circular part and may not necessarily coincide with a ring edge. If the find message does not cross the bridge, then the new parent is just the node that forwarded or sent the find message. If it does, then the new parent is the producer of the find message. So, the new bridge now connects the node that received the find message and the producer of that find message. A node v needs to maintain whether the edge $(v, p(v))$ is a bridge and include this information when it sends a find message. We omit this from the pseudocode for simplicity.

Algorithm 2 NewParent function for rings with even number of nodes n and edges of unit weight. Let the nodes be $v_1, v_2, \ldots v_n$ starting from a node v_1 in clockwise direction. Initially, we have $p(v_i) = v_{i+1}$ for $1 \le i < \frac{n}{2}$, $p(v_i) = v_{i-1}$ for $\frac{n}{2} < i \le n$ and $p(v_{n/2}) = v_{n/2}$. We initialize $n(v_i) = \perp$ for $1 \le i \le n$. Initially, the "bridge" is the edge $(v_{n/2+1}, v_{n/2})$.

```
1: procedure NewParent(v, u)
       ▷ Called when w receives "find by v" message from u
2:     if (u, w) is the "bridge" then
3:         new "bridge" is (w, v).
4:         return v
5:     else
6:         return u
7:     end if
8: end procedure
```

We analyze the algorithm on a *sequence* of nodes σ that request the token, by comparing against an optimal algorithm that can freely access an oracle to ask the current position of the token. Thus, the cost $OPT(\sigma)$ of the optimal algorithm is at least the sum of the shortest paths between the consecutive requests. If $ARVY(\sigma)$ is the cost of the Arvy protocol with the NewParent function above, then we define *competitive ratio* r such that $ARVY(\sigma) \le rOPT(\sigma) + c$ for every request sequence σ and a constant c. Obviously, lower competitive ratio implies the performance is close to the optimal case when each request takes the shortest path to the current location of the object. Observe that on a sequence of requests, Algorithm 2 ensures that there is a black edge between every pair (v_i, v_{i+1}) of nodes for $i \ne n/2$. The bridge, which is the black edge between a node in $A = \{v_i \mid 1 \le i \le \frac{n}{2}\}$ and a node in $B = \{v_j \mid \frac{n}{2}+1 \le j \le n\}$, changes whenever a find request crosses the bridge. However, out of the two *ends* of the bridge, one end is always in set A and the other is always in set B. If the set A contains the token, then we often refer to A as the *side of the token* and B as the *other side of the token*. Similarly, we define these terms when B contains the token, and A does not. We use uv or vu to denote the shortest distance between the nodes u and v. The *length* of the bridge between the nodes u and v is uv.

Theorem 6. *The Arvy protocol along with the bridge heuristic has a competitive ratio of 5 on rings of unit weight edges.*

Proof. We prove the claim by managing coins. We assume that each request gives us $5t$ coins, where t is the optimal cost of the request, and use these coins to pay the cost of the Arvy protocol. Initially, the bridge $v_{n/2+1}v_{n/2}$ has length 1, the token is at $v_{n/2}$ and we keep 2 coins on the bridge. After every request, we maintain the following *coin invariant*: the bridge has $2l$ coins if its length is l, and each (unit) edge on the path from the token position to the end of the bridge on the token side has 4 coins. Let r be the current position of the token. Let f be the end of the bridge on the token side.

First, consider that the next request is at a node r' on the token side. So, the cost of Arvy is rr', which is also the optimal cost. Thus, we get $5 \cdot rr'$ coins, out of which we use rr' to pay for Arvy and put $4 \cdot rr'$ coins on the path from r to r', 4 coins on each edge. As we had 4 coins on each edge from f to r from the invariant, we have 4 coins on each edge from f to r', which is the new token position. Also, neither the bridge, nor the coins on it changed. So, the *coin invariant* holds.

Second, consider that the next request is at a node r' on the other side of the token. Let f' be the end of the bridge on the other side of the token. Thus, the bridge $f'f$ is replaced by $r'f$ as the request originated at r'. The Arvy cost is $r'f' + f'f + fr$ as the current token position is r and the bridge of length $f'f$ has to be crossed. Additionally, we need $2 \cdot r'f$ coins to keep on the new bridge. The optimal cost is rr' and thus, we get $5 \cdot rr'$ coins. Using the invariant, we have $4 \cdot rf$ coins on the path from r to f and $2 \cdot f'f$ coins on the bridge, making a total of $5 \cdot rr' + 4 \cdot rf + 2 \cdot f'f$ available coins. As $rr' + rf \ge r'f$ using triangle inequality, we have $5 \cdot rr' + 4 \cdot rf \ge 3 \cdot (rr' + rf) + rf \ge 3 \cdot r'f + rf$. Therefore, we have $3 \cdot r'f + rf + 2 \cdot f'f$ available coins. Then, we use $2 \cdot r'f$ coins to keep on the new bridge. As r' is also the end of the new bridge on the token side, the coin invariant holds. To pay for the Arvy cost, which is $r'f' + f'f + fr$, we have $r'f + rf + 2 \cdot f'f$ coins left. We use $rf + f'f$ coins to pay for that part of the Arvy cost. Now, we only need to pay $r'f'$ towards the Arvy cost using $r'f + f'f$ coins. Using triangle inequality, we have $r'f + f'f \ge r'f'$ and we can also pay $r'f'$ towards the Arvy cost. □

The above result can be easily extended to weighted rings. We initialize the parent pointers as follows. We start with a set of $n-1$ edges of the ring. These edges form a tree. We choose a tree edge as the bridge so that the total weight of the tree edges on either side of the bridge is less than $W/2$, where W is the total weight of all the edges that form the ring. The proof of the previous theorem still holds if the number of coins used on an edge are increased

proportionally to its weight. So, we can also state the following result.

Theorem 7. *The Arvy protocol along with the bridge heuristic has a competitive ratio of 5 on rings.*

But, what about the competitive ratio of Arrow or Ivy on rings? In the framework of Algorithm 1, the Arvy protocol reduces to the Arrow protocol if the new parent returned in Line 7 is always u and to Ivy if the new parent returned is always v. It is not hard to see that Arrow's competitive ratio is $\Omega(n)$ on rings. It is known that any spanning tree on a ring has a pair of points with *stretch* $\Omega(n)$ [12], where stretch is the ratio of the distance between the points on the spanning tree and the shortest distance between the points on the ring. A request sequence that alternates across such a pair of points results in a competitive ratio of $\Omega(n)$, since the spanning tree remains fixed during the operation of Arrow protocol.

We can also show an $\Omega(n)$ lower bound for Ivy on rings. Consider that the ring contains unit weight edges and n nodes $v_1, v_2, \ldots, v_n$. The initial spanning tree consists of the directed edges $v_i \rightarrow v_{i+1}$ for $1 \leq i \leq n-1$ and a self-loop at v_n, which is the root and holds the token. Consider the sequence σ of requests $v_1, v_2, v_3, \ldots, v_n$. A request v_i moves the token to v_i so every request in the sequence takes unit cost optimally. Thus, the optimal cost $OPT(\sigma) = n$. For ivy, the tree is a star rooted at v_1 after the request v_1. Every request after v_2 kills a spoke of the star and the tree returns to the initial tree at the end of the sequence. The first request by v_1 traverses round the ring from the node v_n and incurs a cost of n. Afterwards, every edge (v_1, v_i) for $2 \leq i < n$ is traversed twice and the edge (v_1, v_n) is traversed once. Thus, Ivy's cost

$$\begin{aligned} IVY(\sigma) &= n + 2 \cdot \Sigma_{i=2}^{n-1} d(v_1 v_i) - 1 \\ &= \Theta(n^2). \end{aligned}$$

As $OPT(\sigma) = n$, the competitive ratio of Ivy on rings is $\Omega(n)$ and we can state the following result.

Lemma 8. *The competitive ratio of the Arrow or Ivy protocol is $\Omega(n)$ on rings of size n.*

7 DISCUSSION

In this paper, we propose the Arvy distributed directory protocol, which is a generalization of both classic directory protocols: Arrow and Ivy. We show that Arvy is correct, i.e., each request finds the shared token in an asynchronous and reliable network where the requests to the token can be issued concurrently. We remark that in the original Arrow or Ivy protocols, the parent pointers except the self-loops, must coincide with an edge of the original network. The Arvy generalization gets rid of this assumption, which implies, for instance, that Ivy would also work correctly on networks that are not complete.

We show that Arvy has a constant competitive ratio on a ring of n nodes, where Arrow or Ivy have $\Omega(n)$ competitive ratio. Thus, the protocol not only preserves the simplicity of Arrow or Ivy protocols but is also more efficient. Arvy also has low space overhead per node, an uncommon feature of state of the art protocols. It would be interesting to analyze the competitive ratio of the protocol for other networks. In general, we think that the protocol is not only practical but also opens up very interesting theoretical questions for future research.

ACKNOWLEDGMENTS

We would like to thank Sebastian Brandt, Klaus-Tycho Foerster, Darya Melnyk, Thatchaphol Saranurak and Yuyi Wang for the discussions about this problem.

REFERENCES

[1] Baruch Awerbuch and David Peleg. 1990. Sparse Partitions. In *31st Annual Symposium on Foundations of Computer Science (FOCS), St. Louis, MO, USA* .
[2] Baruch Awerbuch and David Peleg. 1995. Online tracking of mobile users. *Journal of the ACM (JACM)* (1995).
[3] A. Bouabdallah and M. Trehel. 1988. A Characterization of the Dynamical Ascending Routing. In *Workshop on the Future Trends of Distributed Computing Systems in the 1990s, Hong Kong.*
[4] Costas Busch, Ryan LaFortune, and Srikanta Tirthapura. 2007. Improved Sparse Covers for Graphs Excluding a Fixed Minor. In *26th Annual ACM Symposium on Principles of Distributed Computing (PODC), Portland, Oregon, USA.*
[5] Michael J. Demmer and Maurice Herlihy. 1998. The Arrow Distributed Directory Protocol. In *12th International Symposium on Distributed Computing (DISC), Andros, Greece.*
[6] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar. 2003. A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics. In *35th Annual ACM Symposium on Theory of Computing (STOC), San Diego, CA, USA.*
[7] Abdolhamid Ghodselahi and Fabian Kuhn. 2017. Dynamic Analysis of the Arrow Distributed Directory Protocol in General Networks. In *31st International Symposium on Distributed Computing (DISC), Vienna, Austria.*
[8] David Ginat, Daniel D. Sleator, and Robert E. Tarjan. 1989. A Tight Amortized Bound for Path Reversal. *Information Processing Letters* (1989).
[9] Maurice Herlihy and Ye Sun. 2005. Distributed Transactional Memory for Metric-Space Networks. In *19th International Conference on Distributed Computing (DISC), Cracow, Poland.*
[10] Fabian Kuhn and Roger Wattenhofer. 2004. Dynamic Analysis of the Arrow Distributed Protocol. In *16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Barcelona, Spain.*
[11] Kai Li and Paul Hudak. 1986. Memory Coherence in Shared Virtual Memory Systems. In *5th Annual ACM Symposium on Principles of Distributed Computing (PODC), Calgary, Alberta, Canada.*
[12] Y. Rabinovich and R. Raz. 1998. Lower Bounds on the Distortion of Embedding Finite Metric Spaces in Graphs. *Discrete and Computational Geometry* (1998).
[13] Kerry Raymond. 1989. A Tree-Based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems (TOCS)* (1989).
[14] Gokarna Sharma, Costas Busch, and Srivathsan Srinivasagopalan. 2012. Distributed Transactional Memory for General Networks. In *26th International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China.*