Theory Comput. Systems **36**, 479–495 (2003)
DOI: 10.1007/s00224-003-1084-8

Theory of Computing Systems

Deterministic Resource Discovery in Distributed Networks*

Shay Kutten,[1] David Peleg,[2] and Uzi Vishkin[3]

[1]Faculty of Industrial Engineering and Management, Technion,
Haifa 32000, Israel
kutten@ie.technion.ac.il

[2]Department of Computer Science and Applied Mathematics, The Weizmann Institute,
Rehovot 76100, Israel
peleg@wisdom.weizmann.ac.il

[3]Electrical and Computer Engineering Department,
University of Maryland Institute for Advanced Computer Studies,
College Park, MD 20742, USA
vishkin@umiacs.umd.edu
and
Computer Science Department, The Technion,
Haifa 32000, Israel

Abstract. The *resource discovery* problem was introduced by Harchol-Balter, Leighton, and Lewin. They developed a number of algorithms for the problem in the weakly connected directed graph model. This model is a directed logical graph that represents the vertices' knowledge about the topology of the underlying communication network.

The current paper proposes a deterministic algorithm for the problem in the *same* model, with improved time, message, and communication complexities. Each previous algorithm had a complexity that was higher at least in one of the measures. Specifically, previous deterministic solutions required either time linear in the diameter of the initial network, or communication complexity $O(n^3)$ (with message complexity $O(n^2)$), or message complexity $O(|E_0| \log n)$ (where E_0 is the arc set of the initial graph G_0). Compared with the main randomized algorithm of Harchol-Balter, Leighton, and Lewin, the time complexity is reduced from $O(\log^2 n)$ to

* A preliminary version of this paper appeared in *Proc. SPAA* '01, July 4–6, 2001, Hersonissos, Crete. Shay Kutten was supported in part by a grant from the Israel Ministry of Science and Art and by the Technion Fund for the Promotion of Research. David Peleg was supported in part by grants from the Israel Science Foundation and the Israel Ministry of Science and Art. Uzi Vishkin was supported by NSF Grant 9820955.

$O(\log n)$, the message complexity from $O(n \log^2 n)$ to $O(n \log n)$, and the communication complexity from $O(n^2 \log^3 n)$ to $O(|E_0| \log^2 n)$.

Our work significantly extends the connectivity algorithm of Shiloach and Vishkin which was originally given for a parallel model of computation. Our result also confirms a conjecture of Harchol-Balter, Leighton, and Lewin, and addresses an open question due to Lipton.

1. Introduction

The *resource discovery problem* was introduced by Harchol-Balter, Leighton, and Lewin in [1] as a part of their work on web caching. They developed a randomized algorithm for the problem in the weakly connected directed graph model, that was implemented within the Laboratory of Computer Science at MIT, as part of a project to build a large-scale distributed cache, and then licensed to Akamai Technologies. The motivation is to build an Internet-wide content-distribution system that speeds up the access of users to web pages of major content supplier web sites. In order for the machines of that system to cooperate they must first locate each other [1].

Recently, many other distributed *Peer to Peer* (P2P) applications were introduced that require the same task. For example, in Gnutella [2], [3] users on the Internet wish to share files with each other, without using a central server. Each user may tell its local Gnutella process (which is both a client and a server) the Internet addresses of some other client/server processes. Those may know other such processes, and so forth. There is no way for any process to know all the other processes in advance, since users appear and disappear without consulting any central server. Thus, to be able to share and swap files, the client–server processes need to cooperate to locate each other. A very partial list of related P2P applications includes [4]–[8]. A system of intelligent agents (in the context of Artificial Intelligence), by the name Retsina, was developed at CMU, and its description can be found in [9] and [10]. In that system, the agents need to locate each other. A general platform for P2P applications (together with distributed location protocols) is being developed in a project led by SUN Microsystems [11], [12]. Since the actual set of P2P-related projects grows very fast, it is hard to list them here. A more up to date partial listing may be found in [13].

1.1. *The Model*

Following [1], the system consists of a set V of n machines (vertices). Each machine $v \in V$ has a distinct identity label $ID(v)$, which is a number of length logarithmic in n, representing its "address." At any time t, the machines are logically connected via a directed graph $G_t(V, E_t)$. In particular, the input to the task performed here is the directed graph at time 0, G_0, that we term the *initial graph*. When no confusion arises, we omit t and use the notation $G(V, E)$. We use some auxiliary graphs, but, to avoid confusion, we refer to them explicitly by their names.

A vertex u can send a message to another vertex v if and only if there exists at that time a directed arc $\langle u, v \rangle$ in E pointing from u to v. Such an arc exists if and only if u knows the *ID* of v, that is, v's *ID* is included in a list of *ID*s u has. (To see the

intuition behind the definition above, consider, e.g., the case that an *ID* corresponds to an IP address; then, if u knows v's IP address it can send v messages in the Internet.)

Initially, each vertex v knows only its own *ID* and the *ID*'s of its outgoing neighbors in some *initial* directed graph $G_0(V, E_0)$ (i.e., the values $ID(w)$ for every vertex w such that $\langle v, w\rangle \in E_0$).

Note that the directed graph G can "grow" dynamically by the following operation that adds an arc to G. We say that a vertex v *learns* the address $ID(u)$ of another vertex u by receiving a message, in either of the following two cases: (1) if the message received by v was sent by u itself, or (2) if that message was sent by some other vertex w who knew the *ID* of u and included this *ID* in the message it sent to v. This changes the directed graph G, since a directed arc $\langle v, u\rangle$ pointing from v to u is added to E. (Neither this paper nor [1] allow the deletion of logical arcs.)

Communication and computation are synchronous, and proceed in rounds. In each round a vertex can perform computations that depend on its current state (including dependence on the messages it received in the previous round), and send an arbitrary size message to each of its outgoing neighbors in G. Each message reaches its destination with no corruption, and before the next round. We count a round as one time unit. We do not deal with faults.

1.2. *The Resource Discovery Problem*

As defined in [1], the task is to compute the connected components in the underlying graph of G_0 (where the underlying graph is the undirected graph obtained from G_0 by removing the direction from all arcs). More formally, the input to the problem is a directed graph $G_0(V, E_0)$. Each vertex (network node) knows (has a list of) all its outgoing arcs (but not its incoming arcs).

A distributed algorithm is said to solve the *Resource Discovery Problem* if the following applies to every weakly connected component C in the directed graph G when the algorithm terminates:

(a) there exists a vertex (termed *root*) v in C such that for every other vertex u in C, G contains a directed arc (v, u) (or, in other words, v knows all the *ID*s in C);
(b) every vertex u in C "designates" vertex v as the unique *root* of the component (in the implementation a variable called $PTR(u)$ is set to the *ID* of v).[1]

1.3. *Motivation for Model Assumptions*

The problem, and the model, as presented in [1], are a specific instance of a problem that is inherent in distributed networks. The topology may change from time to time. Thus, in order to solve topology-related problems, an algorithm is required to learn the new topology. See, e.g., [14] and [15]. The separation between the logical graph (here

[1] It is rather straightforward to convert the algorithm into one that outputs a different representation of the connected component, e.g., one where every vertex in C knows, upon termination, the *ID*s of all the vertices in C. See the discussion of different output representations, at the end of Section 4.

the directed graph G) and the underlying communication graph (advocated, in [16]) is becoming common for today's fast networks. The logical graph (here the directed graph G) represents the vertices' knowledge about the topology of the underlying communication network. In [1] the notion of topology knowledge is simplified, by modeling it as a knowledge of the *ID*s of other vertices. In general, such knowledge may include a whole route, as well as any additional information needed in order to establish connection (e.g., a password, a cryptographic public key, etc.) An algorithm can increase the connectivity of the logical graph by learning more about the topology. On the other hand, the environment typically decreases the logical connectivity by introducing topological changes such as the addition and deletion of links and vertices, state-corrupting faults, cooperation (or a break in cooperation) between different domains (such as security domains, ownership domains, etc.), different autonomous systems, and different networks, etc.

Recall (Section 1.2) that the problem, as defined (here, as well as in [1]) is to find all the members of a maximal component that is at least weakly connected (in other words, a connected component of the underlying graph of G). Note that weak connectivity is a necessary condition for the solvability of the problem. In the specific application of [1] the weak connectivity is motivated by the scenario where every newly added machine is given a pointer to at least one previously added machine. In more general cases this can result from more general topology changes. For example, topology knowledge can be lost as a result of a loss of a connection to a name server, or because of changes (introduced by the environment) that make the knowledge of some vertex u irrelevant, so it no longer knows how to reach some other vertex v. On the other hand, if, e.g., the name server used by v is still intact, then vertex v may still know how to reach vertex u. Obtaining knowledge is also not uniform (especially in its timing), since it is done by distributed algorithms. Thus, vertex u may have already "learned" about the existence of v (and thus have a logical arc to v) while v is yet to learn about the existence of u.

Dealing efficiently with a weakly connected graph was in fact the main contribution in [1]. The alternative of transforming the initial directed graph G_0 into an undirected one, and then solving the problem on the resulting undirected graph, may lead to efficient solutions if $E_0 = O(n)$, since efficient solutions for undirected graphs are possible. For example, if the directed graph G_0 is transformed into an undirected one, it is possible to use [17] to solve for an undirected graph with message complexity of $O(|E_0| \log n)$. However, E_0 could in practice be rather large, since many practical distributed systems attempt to deal with the case that the network may be partitioned by maintaining a large E_0, to enable the disconnected components to regain connectivity.

Note that some of the assumptions made by [1] (and repeated here in Section 1.1) imply that the underlying communication network is, actually, modeled as a complete undirected graph over the set V of machines. (We shall not use this graph in this paper.) Indeed, the lowest layer of the Internet is not a complete graph. However, at the application layer (where our algorithm is intended to run) every vertex is reachable from any other by establishing a TCP connection. Moreover, the cost of establishing a connection often dominates the difference in costs between a short route connection and a long one [16]. Thus, it makes sense to ignore the length of the route and model the communication graph at the application layer as a complete graph.

1.4. *Complexity Measures*

In [1] the communication cost is expressed in terms of two new complexity measures, called "connection" complexity and "pointer" complexity. These are specialized measures defined for the resource discovery problem. However, it turns out that optimizing according to these measures is equivalent to optimizing according to the usual message and communication (bit) complexity measures, respectively. Specifically, the connection complexity of [1] is the same as the message complexity, and the pointer complexity of an algorithm equals its communication complexity divided by a factor of $\log n$ (bits per pointer).

The main goal is to develop an algorithm efficient simultaneously in all of the measures used by [1], or, equivalently, in each of the following three standard distributed complexity measures: time, messages (counting the overall number of messages sent throughout the execution), and communication (counting the overall number of bits sent).

1.5. *Previous Results and Our Results*

A number of algorithms are presented in [1] for the problem. The deterministic solutions for the weakly connected directed networks presented therein require either time linear in the diameter of the initial network G_0 or communication complexity $O(n^3)$ (with message complexity $O(n^2)$). On the other hand, the *randomized* solutions of [1] require either time complexity $O(\log^2 n)$, message complexity $O(n \log^2 n)$, and communication complexity $O(n^2 \log^3 n)$ (or, equivalently, $O(n^2 \log^2 n)$ pointer complexity), or communication complexity $O(n|E_0|)$ (again with message complexity $O(n^2)$). These complexity bounds hold with high probability. (A new randomized algorithm announced recently [18] uses a different model: it operates only for the *strongly connected* case; one variant of that algorithm has time complexity $O(\log n)$ and message complexity $O(n^2)$, and another variant has time complexity $O(\log^2 n)$ and message complexity $O(n)$.)

Other previous results that should be mentioned appear in [17] and [16]. In [17] a parallel algorithm is presented to compute the connected component of a graph represented by the processors and the pointers they keep to each other. Our algorithms borrows some ideas from [17]. In addition, a part of the contribution of this paper is the adaptation of ideas introduced for the parallel model (in particular, in [17]), into the distributed domain, and the suggestion that additional problems may benefit from similar adaptations. We note that one of the differences between the models is that the one in [17] is undirected, while here a lot of the difficulty arises from the directed nature of the model (and, even more, from assuming only weak connectivity). Other relevant differences between the models are discussed in Section 4.3.

In [16] a distributed model is presented, where a vertex may send a message directly (i.e., fast) to another vertex, if it knows the route to the other vertex. This model has similarities to the models used here. Moreover, the algorithm presented there for leader election can be adapted for use here with small changes, except that (like the algorithm of [17]) the algorithm of [16] relies on the virtual graph being undirected, and we do not see a way to apply it efficiently in a directed graph. For the *undirected case*, the message complexity of the algorithm of [16] is optimal $O(n)$, but the time complexity is very high—$O(n)$.

Neither this paper nor any of the previous papers mentioned above, allow the

deletion of logical arcs. (The case of arc removal by the environment is implicitly handled in [1] by rerunning their algorithm from time to time, after the environment had already removed some arcs from the directed graph; similarly, our algorithm can be rerun too, with the advantage that one can choose to rerun our algorithm only after it terminates.)

The current paper presents a deterministic distributed algorithm for the weakly connected case that is efficient in all three complexity measures, with time complexity $O(\log n)$, message complexity $O(n \log n)$, and communication complexity $O(|E_0| \log^2 n)$.

Our algorithm also enjoys some other advantages over that of [1]. Since our algorithm is deterministic, it does not rely on sources of random numbers, which are sometimes not easily available. Moreover, since our algorithm detects its termination, as opposed to the algorithm of [1], some decisions regarding the use of the algorithm output may be simplified. For example, assume that one would like to rerun the algorithm from time to time, after some faults forced the discarding of some logical arcs from G. It makes sense to wait until the algorithm terminates, and only then rerun it.

Structure of This Paper. Section 2 contains an overview of the algorithm. Section 3 contains the formal description. The analysis appears in Section 4. In Section 4.3 we discuss the techniques used, in order to shed some light on the comparison between the model of parallel computations, and the distributed computing model used here. Conclusions are given in Section 5.

2. Overview of the Algorithm

Similar to [17], the main data-structure is yet another graph, namely, a directed *pointer graph* $\mathcal{G}_t = (V, P_t)$, where $P_t \in E_t$, $P_t = \{PTR_v \mid v \in V\}$ at time t is a set of pointers, one per vertex. When t is clear from the context we omit it and refer to $\mathcal{G}$ and P. The value of PTR_v will always be the *ID* of some vertex u known to v. We then say that v *points* to u. Note that (by the definition of G, see Section 1.1), at any time t during the execution, the set of directed arcs in $\mathcal{G}$ is a subset of the directed arcs of the directed graph G at that time t. Initially every vertex v points to v itself. At any given time during the execution, the algorithm ensures that

(i) each vertex points to some vertex, and
(ii) $\mathcal{G}$ is the union of two sets of directed arcs: (1) a directed forest, and (2) a set of self-loops, one per root of a tree in the directed forest.

Ignoring the self-loops, $\mathcal{G}$ defines, at each time t, a forest of directed trees, each of which is a maximal weakly connected component of $\mathcal{G}$. (However, G includes additional arcs that do not belong to $\mathcal{G}$, and each may connect two maximal connected components of $\mathcal{G}$.) Let $Tree(v, t)$, the *tree of a vertex* v at time t, be the maximal weakly connected component of v in $\mathcal{G}_t$. (When the time t can be understood from the context we use, instead, the notation $Tree(v)$.) A tree in the pointer graph $\mathcal{G}$ is called a *star* if all its vertices point directly to its root. Initially, every vertex is a root (namely, points to itself). At the end of the algorithm, the pointer graph $\mathcal{G}$ consists of a single star.

The algorithm progresses by having vertices repeatedly (1) learn new *ID*s by receiving messages and (2) switch from pointing to one vertex to pointing to another. Pointers are switched for one of two purposes:

(i) *Merging trees*: A root u could choose to point to another vertex (and stop being a root) if it has an outgoing arc (in G) leading to a vertex in another tree. Following [17] we term this operation *hooking*.
(ii) *Path shortening*: A non-root vertex may advance its pointer (to an ancestor in its tree) in order to reduce its height, i.e., shorten its pointer path to the root of its tree. We term this operation *Shortcut*.

Initially all the vertices are ***active***. A root u (or, equivalently, its tree $Tree(u)$) becomes ***passive*** once its tree stops changing. That is:

(a) $Tree(u)$ is a star, so its height cannot be reduced any further (and u already learned all the outgoing arcs (in G) of all the vertices in $Tree(u)$);
(b) all the outgoing arcs of u in G lead to vertices which point to u; so no such arc can offer trees into which to merge;
(c) all offers to join received so far have been dealt with.

Once a root became passive, it stays passive until it is offered to join as a vertex in some other (active) tree. A non-root vertex in a passive tree (a tree rooted at a passive root) is considered passive too. Note that such a vertex must be both a leaf and a child of the root.

2.1. *Algorithm Structure*

We now proceed with a more detailed exposition of the structure of the algorithm. Each vertex v has a pointer variable $PTR(v)$, used for representing the forest structure discussed above. Throughout the execution of the algorithm, each tree root is in either an ***active*** or a ***passive*** state. If the root is ***active*** then the tree is said to be active, as well as every node in the tree (and similarly for ***passive*** roots).

Informally, Phase i consists of the following steps:

Step 1. Each ***active*** star-root r tries to form a larger tree by way of hooking (becoming a child of) onto another vertex, or by helping others to hook onto itself. The basis for the hooking operations is the set of vertices adjacent to the rooted star; but it is important to understand that r could end up using data from the set of vertices adjacent to *another* rooted star (such data would be obtained through **join** invitation messages received at r).

To save messages, star-root r only invites (by **join** messages) some a_i of its adjacent vertices in phase i, where the function a_i grows exponentially with i.

Star-root r seeks to hook itself onto the ***active*** vertex with the smallest *ID* which can be obtained with the help of the sets of the adjacent vertices mentioned above. (The smallest *ID* does not necessarily have to be smaller than the *ID* of r.) Star-root r hooks itself on another ***active*** vertex, *unless* one of the following two cases applies:

(a) (Preventing cycles of length 2 in the pointer graph $\mathcal{G}$)
Another ***active*** star-root w hooks itself on r, w has the smallest *ID* among the

active vertices which r found to be adjacent to it, but the *ID* of r is smaller than the *ID* of w. In this case, r is the one that remains (a root and) ***active***.

(b) (No other active vertex to hook to)
All the outgoing arcs of r (in G) lead to ***passive*** roots and all of them chose to hook on r.

(In Case (b), r *may* become passive (see Step 4(i)) if a_i already equals the total number of outgoing arcs of r in G.)

Case (b) requires further explanation. If an arc (of G) leads from an ***active*** star-root r to a vertex v of a ***passive*** star-root w, then w is notified of r; w then chooses to hook on the star-root with the smallest *ID* among r and other such ***active*** star-roots. Star-root r is then notified of w's choice. Had w chosen to hook on another ***active*** vertex, say s, over r, the *ID* of s must have been smaller than that of r. This would have implied that r itself should have chosen to hook on s, upon receiving w's message telling r about s. So, the fact that r is still a root implies that this did *not* happen. In other words, root r does not know of any lower *ID* active tree to hook to.

Step 2. Trees that are not stars reduce their height by having each vertex v copy to its variable $PTR(v)$ the value of the pointer variable of v's parent, i.e., $PTR(PTR(v))$. This operation is referred to as "pointer shortcut."

(In what follows we show that if the height h is even, then it is halved by this operation; the least reduction occurs for $h = 3$.)

Step 3. This step is used for each parent to learn about its new children and their state. In particular, the tree root learns the state of the tree after the hooking and shortcuts: (1) Is the tree now a star? (2) If so, does the star root know all the outgoing neighbors of vertices in the tree?

Step 4. A tree becomes passive if

(i) it has considered all its outgoing arcs (in G),
(ii) its height did not decrease geometrically, and
(iii) its size (i.e., its number of vertices) did not increase geometrically.

While we explained the geometric decrease in height, we defer to later (a) an explanation of what is meant by geometric increase in size (to be discussed in Lemma 4.4), and why a root is not allowed to consider all its outgoing arcs in a given phase (to be used in Lemma 4.8), and (b) a procedure for increasing the number of outgoing arcs (of G) that a root may consider in each phase (see Step 1 of the algorithm in Section 3).

Note that once no active roots remain, the algorithm terminates. We will show that the last active vertex is the root of a star that includes all the vertices.

To detect the termination of the execution, some additional information is needed. For example, if the number of vertices is known in advance, then the last active root can detect termination by counting the number of vertices in its star. Alternatively, suppose that an upper bound on the number of vertices is known in advance. Our analysis below

provides a deterministic upper bound on the number of phases as a function of the number of vertices in the input graph G_0. So, we can determine that the algorithm has terminated after an appropriate number of phases.

3. The Algorithm

We now present a precise description of the algorithm. Initially, $PTR(v) = v$. In each phase i (starting from $i = 0$):

Step 1.

Step 1.1: Each (***active***) star-root r sends out **join** messages on its first a_i outgoing arcs (of G), for $a_i = \min(m(r), (\frac{3}{2})^i)$, where $m(r)$ is the number of the outgoing arcs (of G) from r to a vertex outside its star at that line.

Step 1.2: A vertex z with $PTR(z) = v \neq z$ that receives such a **join** message, but not from z's child, forwards the **join** message to v.

Step 1.3:

(i) If v is ***active***, then it responds (to r, the root who initiated the **join**) with the message **active**, to which it appends its *ID*.

(ii) If v is a ***passive*** root, it hooks itself on the ***active*** root w with the smallest *ID* which sent a **join** message to some vertex z in v's star.

Vertex v then responds to all the ***active*** vertices whose **join** messages it received with the message **passive**, to which it appends the *ID* of w.

Step 1.4: If an ***active*** star-root r found a non-empty set of other ***active*** vertices (by receiving *ID*s in either **active**, **join**, or **passive** messages), then r chooses, as a merging *candidate*, the active vertex v with the smallest *ID* in this set. Then r notifies v of this selection.

Step 1.5: If (1) r is not the *candidate* of v, or (2) r is the *candidate* of v but $ID(v) < ID(r)$, then r hooks into v, and notifies v about this hooking.

Step 2. Every vertex v performs a pointer shortcut operation, setting

$PTR(v) := PTR(PTR(v))$.

For that, every vertex v sends its pointer value to its new children (those who notified v in Step 3.3 of the previous phase or Step 1.5 of the current phase), and every non-root vertex assigns the pointer value received from the parent, to its own pointer variable.

Step 3.

Step 3.1: A vertex u that for the first time points to a root other than itself (i.e., before this step $PTR(u)$ was either u or a non-root vertex), sends the list of its outgoing neighbors to the root.

Step 3.2: Every child of a root, that now has no children, notifies the root.

Step 3.3: Every (passive or active) vertex that now points to a new vertex, sends its *ID* to the new parent.

Step 4. Every ***active*** root r declares itself ***passive***, *unless* either:

(i) $m(r)$ was larger than a_i in Step 1.1 of the current phase, or
(ii) an ***active*** vertex, which prior to Step 1 of the current phase did not point to r, points to r now (i.e., following Steps 1 or 2 of the current phase).

4. Analysis

4.1. *Correctness*

We prove that when the algorithm terminates, $\mathcal{G}$ consists of one star-root, who knows the *ID*s of all its children. Lemma 4.1 establishes (using induction) the fact that progress is always possible, until this state is reached. In other words, a root of a star tree knows the outgoing neighbors of the tree, and thus can initiate a "join" of the trees. Lemma 4.3 establishes the fact that this "join" is indeed carried out. Lemma 4.2 establishes the fact that all the operations still retain the structure of trees, assumed by the two other lemmas.

The algorithm immediately implies the following.

Lemma 4.1 (Well-Defined Actions). *Each root always knows*

(a) *its children*,
(b) *all the outgoing neighbors* (*in* E_0) *of all its children*, *and*
(c) *whether it is a star-root*.

Proof. Item (a) follows from Steps 3.3 and 1.5. Item (b) follows from Step 3.1 for vertices who point to a root for the first time. For a vertex u that previously, at some time t, pointed to another root v, it is easy to prove by induction on the steps taken, that: (1) from time t on, vertices u and v always belong to the same tree, and that (2) for every tree that contains u and v after t, the distance of v from the root is never higher than the distance of u from that root. For the induction basis the distance of v from the root of the common tree is zero, at time t, and the distance of u is 1. The induction step follows from a simple case analysis. Item (b) now follows from Step 3.3 of either v or some ancestor of v after time t. Item (c) follows from Step 3.2. □

Lemma 4.2 (Safety). *No cycles are created in the pointer graph* $\mathcal{G}$.

Proof. Recall that only a star-root may hook itself. Thus, if a star-root r hooks itself into a vertex v who is not a star-root, no cycle can be created, since v's tree does not hook itself further.

Now, note that in Step 1, each arc $\langle u, v\rangle$ (of G) considered (for hooking) at vertex u is also considered at vertex v if v is a star-root. Thus if v does not use arc $\langle u, v\rangle$ (of G) for hooking, it is either because v's tree does not hook in that phase or because v's tree hooks onto a vertex with an *ID* that is smaller than that of u. Given that, to see why hooking onto the smallest outgoing neighbor avoids cycles, note that for every vertex v, the linked list starting at it and jumping two pointers must be monotonically decreasing

in *ID* size, namely, the *ID* of $PTR(PTR(v))$ is smaller than that of v; this is since the vertex $PTR(v)$ chose vertex $PTR(PTR(v))$ over vertex v. □

Lemma 4.3 (Liveness). *As long as not all vertices belong to a single star, there is at least one active root.*

Proof. The case where there is a single tree in the pointer graph $\mathcal{G}$ and the tree is passive but is not a star is obviously not possible, because of Step 2. Assume now, in contradiction, that there are two ***passive*** trees in the pointer graph $\mathcal{G}$ and there is a directed arc $\langle u, v\rangle$ (in G) from a vertex u in one of them to a vertex v in the other (since we assume that the directed graph G is weakly connected). By Lemma 4.1 u is the root of one of these trees, and v is either the root of the other or a child of the root.

Consider the phase in which $Tree(u)$ became passive. By Steps 1.1 and 4(i), u must have sent a **join** message on $\langle u, v\rangle$. At the last phase in which this was done, v's tree did not hook itself onto u, and, moreover, no other vertex became a child of u in that phase. Nevertheless, u did not hook itself onto v; a contradiction to Steps 1.2 and 1.5. □

4.2. *Complexity*

For the analysis, we define the height of each tree (a maximal connected component of $\mathcal{G}$) to be the larger among 1 and the length (counting arcs in the tree) of the longest path over the (directed) tree from a leaf to the root. (Hence a singleton vertex and a star are both defined to have height 1.)

Lemma 4.4 (Progress Measure). *Following Phase i, the size of an **active** tree is at least $(\frac{3}{2})^i$.*

Proof. The proof is based on the following inductive claim.

*Following Phase i, the sum of the heights of all the **active** trees in the pointer graph $\mathcal{G}$ is at most $s/(\frac{2}{3})^i$, where s is the total number of vertices in **active** trees.*

The base case in the proof is trivial.

The induction step:

- For ***active*** trees of height greater than 1, the pointer shortcut operation which decreases the height of (non-star) ***active*** trees will provide the "progress." Consider a node with distance $2l$ on the tree from the tree root. It is straightforward to show by induction on l that its distance from the root, after the shortcut, is l. Hence, if the height of a tree is even, then it is reduced to a half. Otherwise, it is reduced to a half, rounded up. Thus, the least progress is obtained when a tree of height 3 is transformed to be a tree of height 2. Hence the term $(\frac{2}{3})^i$ in the formula for the height.

  Note that the join of ***passive*** trees does not increase the height of an active tree: a ***passive*** root always joins as a child of the root of an ***active*** tree (at Step 1.3(ii)). Since the ***passive*** tree is a star, this may temporarily increase the height of the active tree to 2 (if it was 1 before that). However, then Step 2 reduces the height again to 1.

The only operation that can *increase* the height of an active tree is the join of multiple other active trees in Step 1 (a chain of, say k, joining trees can form a tree of hight k). However, this does not increase the *sum* of their heights. Then, in Step 2, the progress stated in the induction claim is achieved.

- For ***active*** trees of height 1, we need the following claim:
  *If at Phase i, only **passive** vertices are hooked on an **active** root r, then either its size increases by $(\frac{3}{2})^i$ vertices, or r becomes **passive**.*
  The claim follows from Step 4(i) (since the claim assumes that the condition for Step 4(ii) did not hold) and from the definition of a_i in Step 1.1.

  To prove now the induction claim for active trees of height 1, note that if the ***active*** root remained ***active***, then its contribution to the parameter s grew larger by the same factor that appears in the denominator (of the term $s/(\frac{2}{3})^i$ of the induction claim), thus offsetting the effect of the increase in i in the induction. At the same time, the height of the tree did not increase.

  If, on the other hand, the tree (say $Tree(q)$ for some root q) does become ***passive***, then recall that $Tree(q)$ is no longer counted in the term $s/(\frac{2}{3})^i$ of the induction claim (since s is the size of the ***active*** trees). Thus, consider the execution of the algorithm on a different system of vertices, that does not include the vertices of $Tree(q)$ (but is identical to G otherwise). The execution up to Phase i (including) on both systems is identical. Thus, the induction hypothesis continues to hold for the set of ***active*** trees that does not include $Tree(q)$.

Since the height of a tree is at least 1, the claim proved by the induction above implies the desired lower bound on the size of each ***active*** tree. Indeed, the claim was proven for the sum of the sizes of the trees, but we can use it for each tree separately. To see this, given a tree, $Tree(q)$, consider the execution of the algorithm on a different system of vertices, that includes only the vertices of $Tree(q)$. Consider the execution up to Phase i (including) projected on the vertices of $Tree(q)$. These execution projections on both systems are identical. Thus, the induction claim continues to hold for the $Tree(q)$ by itself.

The lemma follows. □

The above lemma immediately implies the following:

Corollary 4.5 (Progress). *Following Phase i, there are at most $n/(\frac{3}{2})^i$ active trees.*

Corollary 4.5 immediately implies the following:

Corollary 4.6. *The time complexity of the algorithm is $O(\log n)$.*

Lemma 4.7. *In Step* 2 *every tree arc (in $\mathcal{G}$) carries at most one message.*

Proof. Immediate from Step 2. □

Lemma 4.8 (Message Chains). *In Phase i, each root initiates $O((\frac{3}{2})^i)$ "chains" of messages* (*i.e., messages followed by answer messages, and so on*), *except for messages sent in Step* 2; *the number of messages in each chain is $O(1)$.*

Proof. Follows immediately from the algorithm. See Steps 1.1–1.5. □

Corollary 4.9. *The algorithm sends $O(n)$ messages per phase, and its overall message complexity is $O(n \log n)$.*

Proof. Follows from Lemmas 4.7 and 4.8 and Corollary 4.6. □

Lemma 4.10. *Per phase, each arc in E_0 is being transmitted $O(1)$ times. There are also some lower-order terms.*

Proof. Follows immediately by a simple case study on the steps of the algorithm. □

Corollary 4.11. *The algorithm sends $O(|E_0| \log n)$ bits per phase, and its overall communication complexity is $O(|E_0| \log^2 n)$ bits.*

Proof. Follows from Lemma 4.10 and Corollary 4.6. □

The following theorem sums up the previous lemmas and corollaries, and its proof is immediate from them.

Theorem 4.12. *When executed on a weakly connected graph $G_0 = (V, E_0)$, the algorithm terminates in time $O(\log n)$ and uses $O(n \log n)$ messages totaling $O(|E_0| \log^2 n)$ bits. Upon termination, the pointer graph $\mathcal{G}$ is a star tree, where the star-root knows the IDs of all the vertices, every other vertex knows the ID of the star-root, and the pointer PTR of every other vertex points to the star-root.*

4.3. *Techniques Used for Improving the Complexity Results*

In this section we briefly revisit the main techniques that enabled extending the parallel algorithm of [17] from a purely parallel model to the model of this paper. A brief description of the algorithm of [17] is given in the Appendix.

- The idea of breaking cycles (in the pointer graphs $\mathcal{G}$) using a method different from the common one (used, e.g., in [17]), as well as from the method used in [19] (within an algorithm for computing a minimum spanning tree, which is in itself a variant of [17]).

  Compared with [17], we use "hook on *smallest*" as opposed to "hook on *smaller*." This idea simplified the algorithm by saving a second round of hookings in each iteration. Such a second round was needed in [17] in order to hook those roots which did not "make progress" (i.e., that nobody hooked onto them in the first round, and their tree did not shortcut) in the current iteration; without

such a second round, the parallel running time went up from $O(\log n)$ to linear in n. Unlike the parallel computation model in [17], the distributed model of the current paper readily allows us to provide the smallest value.

Compared with [19] we use "hook on smallest known" as opposed to "hook on absolute smallest" used in [19]. We cannot use the method of [19] since in the directed model, a vertex may not be aware of its minimum outgoing arc. (The vertex may learn of that arc by a message from the other endpoint of that arc, however, we have to limit such messages if we want to obtain the order of message complexity we do obtain, thus we cannot use this method.)

- The idea of limiting the set of arcs that can be considered at each phase instead of using *all* adjacent arcs. This, in turn, enabled geometrically increasing the number of arcs for the join messages and thereby the message complexity bounds.
- The idea of waking up ***passive*** vertices and the updated analysis to establish that the size of an ***active*** root is at least $(\frac{3}{2})^i$ following Phase i. This enabled the time complexity bounds.
- The idea of forwarding the entire list of the outgoing arcs *once*. This is done at the first time in which a vertex points to (another) root. It enabled the communication complexity bounds.

Different Output Representations. Our algorithm, as described above, outputs the *ID*s of all the vertices of the connected component only at the root (and makes the *ID* of the root known to every vertex). It is rather easy to convert the algorithm so that it outputs all the *ID*s at all the vertices, if necessary. The conversion does not increase the order of the time complexity, nor the order of the message complexity. However, it does change the communication complexity somewhat, from $O(|E_0| \log^2 n)$ to $O(|E_0| \log^2 n + n^2 \log n)$ (which is an increase if $|E_0| \log^2 n = o(n^2 \log n)$).

This change is necessary, since $\Omega(n)$ *ID*s of size $\Omega(\log n)$ have to be delivered to each one of $\Omega(n)$ vertices in the worst case. Recall that the communication complexity of the algorithm of [1] is $O(n^2 \log^3 n)$. The extra effort can be designed to satisfy the following. When counted over the whole run of the algorithm, every *ID* of a vertex can be sent to every vertex at most once (in addition to the number of times this *ID* is sent by the non-modified algorithm). The details of the implementation of this change are straightforward.

5. Conclusion

We have shown the following:

Complexity improvements to [1]. In fact, the results confirm a conjecture in [1] that the complexity results (we obtained) are possible.

Lipton's open question. The fact that the algorithm is even deterministic also addresses Lipton's open question (per [1]): *how will a vertex know when to stop*? If due to prior knowledge the overall number of machines can be bounded from above, the resource discovery done within a time which is log of the upper bound would be final.

Connection to a parallel algorithm. We use exactly the same model as in [1]. Since the connection was not pointed out in [1], we feel that perhaps the connection, as demonstrated in this work, is an interesting aspect of this work. (To avoid misunderstanding, it is important to note that this paper does not claim to have found a general connection.) The importance of finding ways to connect parallel algorithms to distributed models is due to the fact that the knowledge base of parallel algorithms is second in magnitude only to the knowledge base of serial algorithms. If more ways to connect parallel algorithms with the [1] model (used in the current paper) can be demonstrated, the algorithmic techniques that will become available for distributed networks algorithms would be greatly enriched. Distributed networks and their modeling keep evolving. The (stable) fundamental knowledge base of a well-studied computation model could enrich the field of distributed networks as it continues to evolve.

The short Section 3 highlights the simplicity of the new algorithm. Yet, the algorithm of [1] is somewhat simpler. While this is a distinct advantage in a distributed protocol, one should note that any real application that will make use of such an algorithm usually includes thousands of lines of code, besides the resource discovery algorithm itself. Thus, it is reasonable to expect that the burden of some additional lines of code will be outweighed by the advantages of the new algorithm.

This is a theory paper. Some additional effort is needed in order to use either the algorithm of Harchol-Balter et al. or our algorithm in a real application. For example, the model used in both papers assumes synchronous communication, which is costly. Another desired improvement is a change in the algorithm such that it handles not only a "one time" task, but rather a situation of ongoing network changes, such as the removal of vertices (e.g., because of faults) and loss of knowledge.

Acknowledgments

We are grateful to Mor Harchol-Balter, Renu Tewari, Srinivasa Rao, and Faith Fich for helpful discussions, Amit Fisher, Ateret Anavi, and Ayelet Yifrach for pointing out unclear points in a previous version of the paper, and the anonymous reviewers for their comments.

Appendix. The Parallel Algorithm of [17]

Recall that in the PRAM models a process can access any location in the shared memory. In particular, process i can access the data of any other process. Simultaneous reading from the same location is allowed as well as simultaneous writing. In the latter case one processor succeeds but we do not know which one.

In the algorithm of [17], a process $i \leq n$ plays the role of vertex i, while each process $i > n$ is assigned an arc $\langle i_1, i_2 \rangle$ of G. The current iteration (phase) number is kept in s. During Iteration s, Variable $Q(i)$ for vertex i equals s if after Step 2 (see below) of the iteration there exists at least one vertex j pointing to i that did not point to i before the iteration. $Q(i) < s$ otherwise. The directed tree constructed by the graph consists of one

pointer $D(i)$ at each node i. Initially $D_0(i) \leftarrow i$, $Q_0(i) \leftarrow 0$, where $D_t(i)$ (resp. $Q_t(i)$) is the value of $D(i)$ (resp. $Q(i)$) at time t (that is, just before Iteration t). In addition, $s \leftarrow 1$, $s' \leftarrow 1$.

While $s' = s$ do
  Step 1: if $i \leq n$
    then $D_s(i) \leftarrow D_{s-1}(D_{s-1}(i))$
      if $D_s(i) \neq D_{s-1}(i)$
      then $Q(D_s(i)) \leftarrow s$
  Step 2: if $i > n$ (*i plays arc $\langle i_1, i_2 \rangle$*)
    then if $D_s(i_1) = D_{s-1}(i_1)$
      then if $D_s(i_2) < D_s(i_1)$ (*i_2 points to smaller vertex*)
        then $D_s(D_s(i_1)) \leftarrow D_s(i_2)$ (*try to hook*)
          $Q(D_s(i_2)) \leftarrow s$. (*one of parallel attempts succeeds*)
  Step 3: if $i > n$
    then if $D_s(i_1) = D_s(D_s(i_1))$ and $Q(D_s(i_1)) \leq s$
      then if $D_s(i_1) \neq D_s(i_2)$
        then $D_s(D_s(i_1)) \leftarrow D_s(i_2)$
  Step 4: if $i \leq n$
    then $D_s(i) \leftarrow D_s(D_s(i))$.
  Step 5: if $i \leq$ and $Q(i) = s$
    then $s' \leftarrow s' + 1$ (*s' incremented only on progress*)
    $s \leftarrow s + 1$ (*if no progress $s > s'$ causes termination*)
end (*do loop*)

Synchronization in [17] was required before each line of the program; i.e., all the processors start the execution of each line together.

References

[1] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource Discovery in Distributed Networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pp. 229–237, May 1999.

[2] Gnutella home page. http://gnutella.wego.com.

[3] M. Ripeanu and I. Foster, Mapping the Gnutella Network: Macroscopic Properties of Large-Scale Peer-to-Peer Systems. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems*, Cambridge, MA, March 2002, pp. 85–93. LNCS 2429. Springer-Verlag, Berlin.

[4] Freenet home page. http://freenet.sourceforge.net.

[5] I. Clarke, T. W. Hong, S. G. Miller, O. Sandberg, and B. Wiley. Protecting Free Expression Online with Freenet, *IEEE Internet Computing*, Vol. 6, No. 1, pp. 40–49, 2002.

[6] PPWAN: Peer-to-Peer Without a Name. http://dividuum.de/p/ppwan/.

[7] ALPINE: Adaptive Large-scale Peer2peer Information Networking. http://cubicmetercrystal.com/alpine/.

[8] The GDNP: The G Diffuse Networking Protocol. http://web.g.diffuse-network.org/.

[9] Retsina: Discovery of Infrastructure in Multi-Agent Systems. http://www-2.cs.cmu.edu/softagents/retsina.html.

[10] K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. *Autonomous Agents and MAS*, Vol. 7, Nos. 1 and 2, pp. 29–48, July 2003.

[11] JXTA home page. http://www.jxta.org.

[12] L. Gong, JXTA: A Network Programming Environment. *IEEE Internet Computing*, Vol. 5, No. 3, pp. 88–95, May/June 2001.
[13] P2P seminar home page. http://iew3.technion.ac.il/kutten/p2p-seminar.html.
[14] I. Cidon, I. Gopal, M. Kaplan, and S. Kutten. Distributed Control for Fast Networks. *IEEE Transactions on Communications*, Vol. 43, No. 5, pp. 1950–1960, May 1995.
[15] J. McQuillan, I. Richer, and E. Rosen. The New Routing Algorithm for the Arpanet. *IEEE Transactions on Communications*, Vol. 28, No. 5, pp. 711–719, May 1980.
[16] I. Cidon, I. Gopal, and S. Kutten. New Models and Algorithms for Future Networks. *IEEE Transactions on Information Theory*, Vol. 41, No. 3, pp. 769–780, May 1995.
[17] Y. Shiloach and U. Vishkin. An $O(\log n)$ Parallel Connectivity Algorithm. *Journal of Algorithms*, Vol. 3, pp. 57–67, 1982.
[18] C. Law and K-Y. Siu. An $O(\log n)$ Randomized Resource Discovery Algorithm. Technical Report FIM/110.1/DLSIIS/2000, Technical University of Madrid, pp. 5–8, Oct. 2000.
[19] B. Awerbuch and Y. Shiloach. New Connectivity and MSF Algorithms for Ultracomputer and PRAM. *IEEE Transactions on Computers*, Vol. 36, pp. 1258–1263, 1987.

Received November 22, 2001, *and in revised form November* 30, 2002, *and in final form April* 5, 2003.
Online publication August 6, 2003.