Deterministic Resource Discovery in Distributed Networks
[Extended Abstract]

Shay Kutten[1]
Technion

David Peleg[2]
Weizmann Institute

Uzi Vishkin[3]
Univ. of Maryland & Technion

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 nodes' "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 edge set of the initial graph). Compared to the main randomized algorithm of Harchol-Balter, Leighton, and Lewin, the time complexity is reduced from $O(\log^2 n)$ to $O(\log n)$, 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 [5] and addresses an open question due to R. Lipton.

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

[2]Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot, Israel, 76100. E-mail: peleg@wisdom.weizmann.ac.il. Supported in part by grants from the Israel Science Foundation and the Israel Ministry of Science and Art.

[3]Electrical and Computer Engineering Department and the University of Maryland Institute for Advnaced Computer Studies, College Park, MD 20742, and Computer Science Department, The Technion, Haifa, Israel, 32000. E-mail: vishkin@umiacs.umd.edu. Supported by NSF grant 9820955.

SPAA'01, July 4-6, 2001, Hersonissos, Crete, Greece.

1. INTRODUCTION

The *resource discovery problem* was introduced by Harchol-Balter, Leighton and Lewin in [5] 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 [5].

The system consists of a set V of n machines. Each machine $v \in V$ has a distinct identity label $ID(v)$, which is a number of length logarithmic in n, representing its "address." The machines are logically connected by a directed graph $G(V,E)$. A directed arc $\langle u,v\rangle$ pointing from a node u to another node v means that u "knows the address" of v, and can thus send it a message. Initially, each node 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 node w such that $\langle v,w\rangle \in E_0$). The logical graph G grows dynamically. When a node v learns the address $ID(u)$ of another node u, e.g., by receiving a message from it, a directed arc $\langle v,u\rangle$ pointing from v to u is added to E. (The case of edges removal by the environment is implicitly handled in [5] by rerunning the algorithm from time to time.)

The problem, and the model, in [5], are a current instance of a problem that is inherent in distributed networks: The topology may change. In order to solve topology related problems, an algorithm is required to learn about the new topology. See, e.g. [3, 7]. The separation between the logical graph and the underlying communication graph (e.g. [2]) is becoming common for current fast networks. This logical graph represents the nodes' "knowledge" about the topology of the underlying communication network. In [5] the notion of topology "knowledge" is simplified, by modeling it as a "knowledge" of an ID of another node. In the general case such a "knowledge" includes a whole route, as well as any additional information needed in order to establish a 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 edges and nodes, state corrupting faults, cooperation (or a break in cooperation) between different domains (security domains, ownership domains, etc.), different autonomous systems, and different networks, and other kinds of changes.

It is assumed (here, and in [5]) that the logical graph G is weakly connected. In the specific application of [5] this results from giving every newly added machine a pointer to at least one previously added machine [5]. 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 that make some "knowledge" irrelevant. The gain of "knowledge" is also not uniform (especially in its timing), since it is done by distributed algorithms. For example, the accessibility to a name server may be different at different nodes.

Note that weak connectivity is a necessary condition for the solvability of the problem. Dealing efficiently with a weakly connected graph was in fact the main contribution in [5]. The alternative of transforming the graph 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. However, E_0 could in practice be quite big, 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.

As in [5], the underlying communication network is modeled as a complete undirected graph over the set V of machines. Communication is synchronous, and proceeds in rounds. In each round, a node can send an arbitrary size message to each of the nodes whose ID it knows (namely, to its outgoing neighbors in G). Thus each message reaches its destination in one time unit. Indeed, the low layer of the Internet is not a complete graph. However, at the application layer (where our algorithm is intended to run) every node 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 [2]. Thus, it makes sense to ignore the length of the route and model the communication graph at the application layer as a complete graph.

The main goal is to develop an algorithm efficient simultaneously in all three standard distributed complexity measures, namely, time, messages (counting the overall number of messages sent throughout the execution) and communication (counting the overall number of bits sent). This translates directly to an efficient algorithm in the measures used in [5].

In particular, in [5], 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, they turn out to be closely linked to the usual message and communication (bit) complexity measures, respectively, and behave the same. (Specifically, connection complexity in the algorithm of [5] behaves exactly the same as message complexity, and pointer complexity translates into communication complexity, only multiplied by a factor of $\log n$ bits per pointer.)

A number of algorithms are presented in [5] 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 [5] 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 [6] 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 $O(n)$.)

The current paper presents a deterministic distributed algorithm 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 [5]. First, the protocol of [5], as described therein, relies on a network-wide synchronization both for starting the protocol and for every phase, and it is not obvious to us how this restriction can be lifted without a significant cost increase. Such global synchronization may be difficult and cumbersome to achieve on a network such as the Internet. This also means that the time of each round is large enough to allow this global synchronization. For the complexities of our protocols to be those calculated above, it really suffices that all the nodes start the algorithm around the same time. No phase synchronization is needed. (Furthermore, even if the start times differ for many nodes, the impact on the runtime of the algorithms will be limited to this difference only; moreover there will be no impact on the message and communication complexities.)

Secondly, since our algorithms are deterministic, they do not rely on sources of random numbers, which are sometimes not easily available. Moreover, since our algorithms detect their termination, as opposed to the algorithm of [5], some decisions regarding the use of the algorithm output may be simplified (for example, the decision to adapt to possible nodes failures by discarding some of the logical edges, and rerunning the algorithm).

An overview of the algorithm, given in Sect. 2, is followed by the algorithm itself in Sect. 3 and its analysis in Sect. 4. The somewhat lengthy overview should not hide the relative simplicity of the algorithm, as the brevity of Section 3 suggests.

2. OVERVIEW OF THE ALGORITHM

This section presents an overview of our algorithm. The input to the problem is a directed graph $G(V, E)$. Each vertex "knows" all its outgoing edges (but not its incoming edges).

The (weak) connectivity of the graph is computed. The output is of the following form. For each weakly connected component, there is one root vertex which is known to every other vertex in the component. The root knows the names of all the vertices in its component. (The identity of the root will be dictated by the computation.)

2.1 Preview

Similar to [8], the main data-structure is a *pointer graph*. At each point in the algorithm: (i) each vertex points to some vertex (possibly itself), and (ii) the algorithm ensures that the collection of pointers forms a directed forest plus self-loops at the roots; hence *the tree of a vertex* v is well-defined, and is denoted by $Tree(v)$. We refer to this forest as the *pointer graph* and denote it by $\mathcal{G}$. A tree in the pointer graph is called a *star* if all its vertices point directly to it. Initially, every vertex is a root (namely, points to itself). At the end of the algorithm, the pointer graph consists of a single star.

For the analysis, we define the height of each tree to be the larger among 1 and the length (counting edges) of the longest path from a leaf to the root. (Hence a singleton node and a star are both defined to have height 1.)

The algorithm progresses by having vertices 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:

(a) an outgoing edge of u (in G), or of another vertex in its tree, leads to a vertex in another tree; or

(b) an incoming edge (in G) from a vertex w, whose root is not u, leads into either u (in G), or into another vertex in $Tree(u)$, and an offer to join w's tree has been received by u.

(ii) *Path shortening*: A vertex may advance its pointer in order to reduce its height, i.e., shorten its pointer path to the root of its tree.

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) all the outgoing edges of u in G lead to vertices which point to u; so no such edge can offer trees into which to merge;

(b) all offers to join received so far have been dealt with;

(c) $Tree(u)$ is a star, so its height cannot be decreased any further.

Once a root became passive, it stays passive until it is offered to join as a vertex in some other (active) tree.

For upper bounding the communication complexity, it will be important to do the following. At the first opportunity in which a vertex u points to another vertex w which is a root, u will forward the entire list of its outgoing edges (including the list of outgoing edges previously forwarded to it, when it has been a root) to w.

The algorithm executes in phases. For upper bounding the message complexity, we limit the number of messages sent by each root to some f_i which increases geometrically with the phase number i.

The analysis of the time complexity, which is dominated by the number of phases, is similar to [8] for a parallel model of computation. Overall, a connection between an algorithm for a *parallel* model of computation and an algorithm for a *distributed* model of communication and computation is demonstrated. This connection is perhaps the most interesting methodical point in the current paper. Some of the technical details that were involved in making the connection are visited in Section 4.3.

Note that some of the actions implied by the above preview rely on data which is not readily available to a vertex in a distributed environment. Our later description tries to account for the extra effort needed to obtain these data.

2.2 Algorithm structure

Let us 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 *active* or *passive* state.

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 on another vertex, or by helping others to hook upon 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 a "join invitation").

Let $a_r = \min(m_r, (3/2)^i)$, where m_r is the number of the outgoing edges from r to a vertex outside its star which are available at r. The first a_r among the outgoing edges of r will be considered.

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

(a) (Preventing cycles in the pointer graph): Another *active* vertex 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 edges of r lead to *passive* roots and all of them chose to hook on r. Case (b) breaks into two subcases:

(b.1) $a_r = m_r$: then r becomes *passive* during the current phase.

(b.2) a_r is smaller than m_r: then r remains *active*.

Case (b) requires further explanation. If an edge 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 would be notified of w's choice. Had w been hooked on another *active* vertex, say s, over r, the ID of s must have been smaller; and this would have implied that r itself had to choose to hook on s. So the fact that r is still a root implies that this did *not* happen.

Step 2 Trees that are not stars reduce their height from h to $\lceil h/2 \rceil$, 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."

(Note that if h is even, then it is halved by this operation; the least reduction occurs for $h = 3$.)

Step 3 A tree becomes passive if

(i) its size (i.e., its number of vertices) did not increase geometrically, and

(ii) its height did not decrease geometrically, and

(iii) it has considered all its outgoing edges.

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 edges in a given phase (to be used in Lemma 4.7), and (b) a procedure for increasing the number of outgoing edges that a root may consider in each phase (see Step 1 of the algorithm in the 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 the number of phases as a function of the number of vertices in the input graph. So, we can determine that the algorithm has terminated after an appropriate number of phases.

3. THE ALGORITHM ITSELF

Let us now present a precise description of the algorithm. In each phase:

Step 1

Step 1.1: The vertex r sends out "join" messages on its first a_r outgoing edges. (Note that $a_0 = 1$.)

A vertex z receiving such a "join" message responds as follows. Suppose z currently points to vertex v.

(A) If z is *active* or points to a *passive* root, it notifies v about r.
If z is *active*, it also responds with the message "**active**," to which it appends the ID of v.

(B) If z is a *passive* root, it hooks itself on the *active* vertex w with the smallest ID which sent a "join" message to some vertex in z's star.
It then responds to all the *active* vertices that sent it a "join" message with the message "**passive**," to which it appends the ID of w.

Step 1.2: If an *active* star-root r found a non-empty set of other *active* vertices (either through the "join" messages it sent, or through ones it received from *active* vertices, or "**passive**" messages from passive vertices), then r chooses, as a candidate, the active vertex v with the smallest ID in this set.
If r is not the candidate of v, or r is the candidate of v but $ID(v) < ID(r)$, then r hooks into v.

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

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

For that, every vertex sends its pointer value to its children in the tree, 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 neighbors to the root.

Step 3.2: Every (passive or active) vertex that now points at a new root, sends its ID to the new root. Every child of a root, that now has no children, notifies the root.

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

(i) m_r was larger than a_r 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

The algorithm immediately implies the following.

LEMMA 4.1. [Well-Defined Actions]
Each root always knows

(a) its children,

(b) all the neighbors (in E_0) of all its children, and

(c) whether it is a star-root.

LEMMA 4.2. [Safety]
No cycles are created in the pointer graph.

PROOF SKETCH. First, note that in Step 1, each edge (u, v) considered at vertex u is also considered at vertex v (if v participates). To see why hooking on the smallest neighbor avoids cycles, note that for every vertex v, the linked list starting at it and jumping 2 pointers must be monotonically non-increasing in ID size, namely, the ID of $PTR(PTR(v))$ cannot be larger than that of v; this is since the vertex $PTR(v)$ chose vertex $PTR(PTR(v))$ over vertex v (or else they are the same). ∎

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 and the tree is passive but is not a star is obviously not possible. Assume now, in contradiction, that there are two *passive* trees in the pointer graph and there is a directed arc $\langle u, v\rangle$ from a vertex in one of them to a vertex in the other. By the phase in which $Tree(u)$ became passive, it must have sent a "join" message on $\langle u, v\rangle$. At the last phase in which this was done, u elected to become passive, rather than either (i) hook itself on another vertex, or (ii) being hooked upon from the other end; contradiction. ∎

4.2 Complexity

LEMMA 4.4. [Progress Measure]
Following Phase i, the size of an active tree is at least $(3/2)^i$.

PROOF. The proof is based on the following 2-item inductive claim:

Item 1. Following Phase i, the sum of the heights of all the active trees in the pointer graph is at most $t/(2/3)^i$, where t is the total number of vertices in active trees.

The pointer shortcut operation which is applied to *active* trees will provide the "progress". Since the height of a tree is at least 1, Item 1 implies a lower bound on the size of each *active* tree; to see this, note that if we "isolate" the vertices and edges of one or more *active* trees, the same analysis would apply to the isolated part.

Item 2. If at PHASE i, only passive vertices are hooked on an active root r, then either its size increases by $(3/2)^i$ vertices, or r becomes passive.

The subtle point here is that the contribution of the tree to the parameter t grew larger, while the height of the tree did not increase.
The lemma follows. ∎

COROLLARY 4.5. [Progress]
Following phase i, there are at most $n/(3/2)^i$ active trees.

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

LEMMA 4.7. [Message Chains]
In phase i, each root initiates $O((3/2)^i)$ "chains" of messages (i.e., messages followed by answer messages, and so on); the number of messages in each chain is $O(1)$.

COROLLARY 4.8. *The algorithm sends $O(n)$ messages per phase, and its overall message complexity is $O(n \log n)$.*

LEMMA 4.9. *Per phase, each edge in E_0 is being broadcast $O(1)$ times. There are also some lower order terms.*

COROLLARY 4.10. *The algorithm sends $O(E_0 \log n)$ bits per phase, and its overall communication complexity is $O(E_0 \log^2 n)$ bits.*

4.3 Techniques used for improving complexity measures

In this section, we briefly revisit the main techniques that enabled extending the parallel algorithm of [8] from a purely parallel model to the model of this paper.

- The idea of breaking cycles (in the pointer graphs) using a method different from the common one (used, e.g., in [8]), as well as from the method used in [1] (within an algorithm for computing a minimum spanning tree, which is in itself a variant of [8]).

  Compared to [8], 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 [8] in order to hook those roots which did not "make progress" (i.e., that were not hooked upon 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 [8], the distributed model of the current paper readily allows to provide the smallest value.

  Compared to [1] we use "hook on smallest known," as opposed to "hook on absolute smallest" used in [1]. We cannot use the method of [1] since in the directed model, a vertex may not be aware of its minimum outgoing edge. (The vertex may learn of that edge by a message from the other endpoint of that edge, 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 edges that can be considered at each phase instead of using *all* adjacent edges. This, in turn, enabled geometrically increasing the number of edges 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 $(3/2)^i$ following phase i. This enabled the time complexity bounds.

- The idea of forwarding the entire list of the outgoing edges *once*. This is done at the first time in which a vertex points to (another) root. It enabled the communication complexity bounds.

5. CONCLUSION

We have shown the following.

Complexity improvements to [5] In fact, the results confirm a conjecture in [5] 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 [5]): *how will a vertex know when to stop?* If due to prior knowledge the overall number of machines can be upper bounded, the resource recovery done within a time which depends on that upper bound would be final.

Connection to a parallel algorithm We use exactly the same model as in [5]. Since the connection was not pointed out in [5], we feel that perhaps the connection, as demonstrated in this work, is a surprising aspect of this work. (To avoid misunderstanding, it is important to note that this submission 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 [5] 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 to 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 [5] 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, Leighton, and Lewin, or our algorithm, in a real application. For example, the model used in both papers assume synchronous communication, which is costly. In fact, under proper assumptions our algorithm can be shown to execute in an asynchronous system, with the same complexity as shown here.

Acknowledgements

We are grateful to Mor Harchol-Balter, Renu Tewari and Srinivasa Rao for helpful discussions.

6. REFERENCES

[1] B. Awerbuch, and Y. Shiloach. New connectivity and MSF algorithms for Ultracomputer and PRAM. *IEEE transactions on Computers, Vol. 36, pp. 1258-1263, 1987.*

[2] 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.*

[3] 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.

[4] R.G. Gallager, P.A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and Syst.*, 5(1):66–77, January 1983.

[5] M. Harchol-Balter, T. Leighton, and D. Lewin. Resource discovery in distributed networks. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, May 1999, pp. 229-237.

[6] C. Law and K-Y. Siu. An $O(\log n)$ randomized resource discovery algorithm. Unpublished manuscript, Oct. 2000.

[7] J. McQuillan, I. Richer, and E. Rosen. The new routing algorithm for the arpanet. *IEEE Transactions on Communications*, 28(5):711-719, May 1980.

[8] Y. Shiloach and U. Vishkin. An O(log n) Parallel connectivity algorithm. *Journal of Algorithms 3, pp. 57-67, 1982.*