A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms

E. KORACH, S. KUTTEN, and S. MORAN
Technion, Israel Institute of Technology

A general, modular technique for designing efficient leader finding algorithms in distributed, asynchronous networks is developed. This technique reduces the problem of efficient leader finding to a simpler problem of efficient serial traversing of the corresponding network. The message complexity of the resulting leader finding algorithms is bounded by $[f(n) + n)(\log_2 k + 1)$ (or $(f(m) + n)(\log_2 k + 1)$], where n is the number of nodes in the network [m is the number of edges in the network], k is the number of nodes that start the algorithm, and $f(n)$ [$f(m)$] is the message complexity of traversing the nodes [edges] of the network. The time complexity of these algorithms may be as large as their message complexity. This technique does not require that the FIFO discipline is obeyed by the links. The local memory needed for each node, besides the memory needed for the traversal algorithm, is logarithmic in the maximal identity of a node in the network. This result achieves in a unified way the best known upper bounds on the message complexity of leader finding algorithms for circular, complete, and general networks. It is also shown to be applicable to other classes of networks, and in some cases the message complexity of the resulting algorithms is better by a constant factor than that of previously known algorithms.

Categories and Subject Descriptors: C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design; F.2.2 [**Analysis of Algorithms and Program Complexity**]: Nonnumerical Algorithms and Problems

General Terms: Algorithms, Design

Additional Key Words and Phrases: Asynchronous leader election, modularity, optimal message complexity, traversal algorithms

1. INTRODUCTION

The problem of efficiently electing a leader in distributed networks has been studied in many papers [1, 2, 4, 6, 7–10, 12–14, 16–18, 19, 21, 23, 27]. Some of the more efficient algorithms in this list are quite sophisticated and specially designed for specific classes of networks. In this paper a general, modular technique for designing efficient leader finding algorithms in such networks is

A preliminary version of this paper was presented at the *4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Ont., Canada, Aug. 1985).
Authors' address: Department of Computer Science, Technion, Israel Institute of Technology, Haifa, Israel 32 000.

developed. This technique greatly simplifies the design of distributed leader finding algorithms, without increasing the order of their message complexity. Moreover, the message complexity of several algorithms obtained by this technique is shown to be better than that of specially designed algorithms. Unfortunately, the time complexity of the resulting algorithms may be as large as their message complexity. It remains an interesting open problem whether a similar general technique that achieves the same message complexity with a better time complexity can be found.

The model under investigation is a network of n processors with distinct totally ordered identities *identity*(1), *identity*(2), ..., *identity*(n), and m communication links connecting pairs of the processors. No processor knows any other processor's identity. Each of the communication links can be either bidirectional or unidirectional, and each processor knows the links connected to itself and their directions, but not the identities of its neighbors. The processors communicate by sending messages along the permitted directions of the communication links. The processors are identical in the sense that all the processors that are working on some common task execute the same algorithm. Such an algorithm may include operations of (1) sending a message to a neighbor, (2) receiving a message from a neighbor, and (3) processing information in the processor's local memory.

We assume that the messages on each link arrive in a finite time, with no error, and are kept until processed. Unlike some algorithms for finding a leader (e.g., [4, 7, 9]), our algorithm does not assume that the FIFO discipline is obeyed by the links. Note that existing communication protocols do not necessarily guarantee the FIFO discipline (see [25, ch. 4]). For networks that obey the FIFO discipline, the algorithms can be simplified, and the length of the message can be slightly reduced. We also assume that all the processors are initially asleep and that any nonempty set of processors may be awakened spontaneously and start the algorithm; a processor that is not a starter remains asleep until a message reaches it.

A communication network can be viewed as a mixed graph $G = (V, E)$ (i.e., E might contain both directed and undirected edges) with $|V| = n$ and $|E| = m$. We refer to algorithms for a given network as algorithms acting on the underlying graph, and we use the terms "graph" and "network" to denote the same entity.

When no processor knows the value of n, a spanning tree (and hence a leader) is found in [11, 12] in $O(n \log n + m)$ messages for a general undirected graph (the algorithm in [12] actually finds a minimum weight spanning tree). When n is known to every processor, a leader in such a network is found in [10], in an expected number of messages which is $O(n \log n)$ (independent of m), and the worst case is $O(nm)$. In [21] a spanning tree construction algorithm is presented for the case where each node knows its neighbors, and messages are permitted to be very long. It is claimed there that the message complexity of that algorithm is $3n \log n + O(n)$ (independent of m) for the average case. The worst case message complexity of that algorithm is said to be $O(n^2)$.

$\Omega(n \log n)$ lower bounds and $O(n \log n)$ upper bounds on the number of messages required for distributively finding a leader in a circular network of processors (directed and undirected) and in a complete undirected network are known; see [2, 4, 6, 13, 17, 22, 23, 27] for the circular networks and [1, 8, 14, 16] for the

complete undirected network. An algorithm that can find a leader in general strongly connected unidirectional networks in $O(nm)$ messages is given in [24]. Another such algorithm (using fewer bits per message, to the total of $O(nm)$ bits) is developed in [7]. Most of these algorithms also construct spanning trees on which messages can be routed from the leader to all nodes, and from all nodes to the leader.

In this paper we present a general, modular technique for constructing leader finding algorithms for any class of graphs. This technique yields algorithms which are competitive, or even better (in the message complexity) than known algorithms, designed for special classes of networks. This technique solves the problem in two stages:

Stage 1. Construction of a traversal algorithm (to be defined).

Stage 2. Construction of a leader finding algorithm, which uses the algorithm of stage 1 as a distributed subroutine. (The exact relation between the leader finding algorithm and its distributed subroutine are explained in the sequel.)

In fact, we present a single, general algorithm for the second stage, which is independent of the specific features of the algorithm of the first stage. Thus, the problem of leader finding algorithms in any class of graphs is reduced to the problem of traversal algorithms for the same class, which in general is much simpler. The origins of the ideas used here can be found in [11], in which a traversal algorithm and the idea of tracing appear as a part of a leader finding algorithm in general undirected networks. The technique used there appears to be applicable to undirected graphs only. Following our paper, the idea of modular construction of distributed algorithms was recently used also in [3].

The message complexity of the resulting algorithm is at most $(f(n) + n)(\log_2 k + 1)$ [or $(f(m) + n)(\log_2 k + 1)$], where the convex function $f(n)$ [$f(m)$] is an upper bound on the complexity of certain, simple executions of the traversal algorithm, and k is the number of nodes that spontaneously start the algorithm. The messages of the resulting algorithms and the local memory used at each node are of a length which is logarithmic in the maximal identity of a processor (this does not include the memory and the length of the messages used by the assumed traversal algorithms, which are usually small). It is also shown that a leader in a network can use any traversal algorithm to construct a spanning tree of routes from all nodes to itself, and a spanning tree of routes from itself to all nodes (clearly, in an undirected network, any spanning tree can be used for both purposes).

The algorithms constructed by this technique are shown to unify and generalize the results on leader finding algorithms mentioned above in a nontrivial sense. For instance, they provide simple constructions of $O(n \log n)$ distributed algorithms for finding leaders in circular and complete networks, and in other classes of networks of more complex structure, for which no such algorithms were designed before. The message complexity achieved by this technique for complete graphs is better in a constant factor than the previous results. A simple generalization of this technique achieves a $2m + 3n \log k + O(n)$ leader finding algorithm for general undirected networks; the message complexity of that

algorithm is better than that of the algorithms in [12, 15]. Our results for general networks are applicable also for different models such as the one appearing in [21] (and in fact improve their result).

This technique enables the simple construction of an efficient leader finding algorithm in directed Euler networks. Another algorithm for this task was obtained independently in [9] (the communication complexity of the algorithm presented here is smaller by a constant factor than that of [9]).

The time complexity of the algorithms constructed by our technique is, in some cases, as large as their message complexity, which is inferior to that of existing algorithms for similar tasks. For example, there are linear time leader election algorithms for complete and circular networks [1, 23]. This disadvantage seems to be a price we must pay for assuming networks of arbitrary type (i.e., each link can be directed or undirected) and for using the carrier traversal algorithm as a "black box." An interesting question is whether there is a different general technique for constructing leader finding algorithms with better time complexity.

The rest of the paper is organized as follows: In Section 2 some basic definitions used in this paper are given. In Section 3 the modular technique for leader finding is presented and proved, and in Section 4 various applications of this technique are given.

2. PRELIMINARIES

In this section we give the definitions needed for our results and introduce some of the basic tools we use.

Let A be a distributed algorithm acting on a graph $G = (V, E)$. An execution of A consists of *events*, each being either sending a message, receiving a message, or doing some local computations. We assume message-driven algorithms, in which a node may send a message only in response to awakening or to the arrival of a message. A distributed algorithm A is *global* on a class Γ of graphs if for every graph $G = (V, E)$ in Γ, and for every execution of A on G, every node v in V either receives a message or sends a message during this execution.

A *rooted* execution of an algorithm A is an execution in which exactly one node was awakened spontaneously. An algorithm A is *serial* if in every rooted execution of A, at any given moment, at most one message is sent in the network, and the next message is always sent by the last node that received a message. An equivalent way to describe a serial algorithm is the following: A node gets a permission to transmit a single message in a rooted execution of such an algorithm either on its spontaneous awakening or by receiving a message from another node. This permission (viewed as a *token*) is taken away from the node when it transmits a message. Another use of a token to generate serial executions of distributed algorithms is given in [26].

The following type of distributed algorithm is the main tool for our results.

Definition. A *traversal algorithm* is a distributed algorithm which is both serial and global.

The *message complexity* $m_{A,G}$ of an algorithm A acting on a graph G is the maximal number of messages sent in any executions of A on G.

Let Γ be a family of graphs, A an algorithm acting on graphs in Γ, and $f(x, y)$ a function of two variables. Then A is said to be of a message complexity $f(n, m)$ if for each graph $G = (V, E)$ in Γ, $m_{A,G} \leq f(n, m)$ $(n = |V|, m = |E|)$.

Let B be a distributed algorithm acting on some graph G. The *carrier algorithm* induced by B, denoted B^c, is the algorithm obtained from B by having each message M sent by B be replaced by a message (M, w), where w is an arbitrary string received from an outside source (i.e., an upper layer algorithm located at the same node). w will be denoted as the *attachment* carried with M by B. The attachments have no effect on the execution of B.

Let A and B be two distributed algorithms acting on the same graph G. We say that A *uses* B *as a carrier* (and A is the *master* of B) if A at each node may invoke B^c (at the same node), and whenever B^c at a given node receives a message, it first transfers its attachment to A (at the same node), and then waits for instructions from A. More specifically, the master algorithm A at node i can perform the following operations on the carrier algorithm B^c at the same node:

Operation 1. Initiating B^c. The effect of such an initiation on B^c is the same as that of a spontaneous awakening.

Operation 2. Appending an attachment w to a message M to be sent by B^c.

Operation 3. Deleting an attachment w from a message M received by B^c.

Operation 4. Instructing B^c to continue its execution.

Operation 5. Destroying a message to be sent by B^c. (In the case that B is a traversal algorithm with a single initiator, this means an abortion of B^c.)

Operation 6. Repeating the last sending executed by B^c, after replacing the attachment of the corresponding message by a new one. In this case we say that the new attachment is *chasing* the previous one.

All carrier algorithms in this paper are induced by traversal algorithms, and we use "traversal algorithm" instead of "carrier algorithm induced by the traversal algorithm." Moreover, each execution of such an algorithm will be a rooted execution (which may be aborted, as described in operation 5 above). Note that this does not exclude the possibility that a carrier algorithm is invoked simultaneously in several nodes—each invocation will be considered a distinct rooted execution of the carrier algorithm. For this purpose all such invocations must be made distinguishable, for example, by including distinct names in their attachments. Readers who are familiar with operating systems may view the master–carriers mechanism described above as a generalization of some single-computer operating system mechanisms (e.g., in UNIX) in which a single father-process may create and monitor several son-processes, some of which may use the same code simultaneously. It also may be viewed as a generalization to the common practice in communication networks, where higher layer protocols rely on an environment consisting of lower layer protocols (see, e.g., [25]).

3. THE GENERAL ALGORITHM FOR FINDING A LEADER

In this section we present a general technique for a modular construction of efficient leader finding algorithms. The section is divided into four subsections.

In the first we present a leader finding theorem; in the second we present the technique to construct the algorithms stated in that theorem; in the third we prove the correctness and the complexity of these algorithms; and in the last we show how traversal algorithms can be used to efficiently construct spanning trees of certain types.

3.1 Leader Finding Theorem

First let us define the *traversability* property and the *edge traversability* property for any class of graphs. Then we present a theorem which connects these properties with the complexity of leader finding algorithms for this class.

Definition. Let Γ be a class of graphs and $f(x)$ a real-valued function. Then Γ is *f traversable* (*f edge traversable*) if there exists a traversal algorithm B such that in any rooted execution of B on any graph $G \in \Gamma$, and for any positive integer x, after sending $\lfloor f(x) \rfloor$ messages, B must have visited at least $\min\{x + 1, n\}$ distinct nodes [$\min\{x + 1, m\}$ distinct edges]. That is: at least $\min\{x + 1, n\}$ distinct nodes [$\min\{x + 1, m\}$ distinct edges] were involved in the sending/receiving of these $\lfloor f(x) \rfloor$ messages.

It follows from [7, 20] that every class of graphs is $O(x^3)$ traversable [$O(x^2)$ edge traversable]. Moreover, it can be shown that for every $1 \leq \alpha \leq 3$ [$1 \leq \alpha \leq 2$] there is a class of graphs which is $O(x^\alpha)$ traversable [edge traversable] but not $o(x^\alpha)$ traversable [edge traversable].

LEADER FINDING THEOREM. *Let a class of graphs Γ be f traversable [f edge traversable], where f is a convex function (i.e., for all x and y it holds that $f(x) + f(y) \leq f(x + y)$). Then there exists a distributed leader finding algorithm whose message complexity on any graph $G = (V, E) \in \Gamma$ is at most $(n + f(n))(\log_2 n + 1)$ if G is f traversable, and $(n + f(m))(\log_2 n + 1)$ if G is f edge traversable.*

Note that, trivially, the converse of the above theorem does not hold (e.g., for classes of graphs for which the message complexity of leader finding algorithm is less than $O(n \log n)$, such as stars).

The proof of the leader finding theorem will be given in the sequel by presenting a leader finding algorithm that uses a carrier B^c based on a given traversal algorithm B and proving the properties of the combined algorithm. We prove the theorem only for the traversability property. The proof for the edge traversability property is similar.

3.2 Presentation of the Algorithm

We outline here the general leader finding algorithm, which uses a given traversal algorithm B^c as a carrier. This algorithm is designed for networks in which messages sent along a link do not necessarily obey the FIFO discipline; it will be noted later that if the FIFO discipline is obeyed, then the algorithm can be simplified, and the length of the messages sent by it can be reduced.

Initially all nodes are asleep and are at *phase* $= -1$. Assume that one node a is awakened and starts the algorithm. Node a raises its phase to 0 and initiates a rooted execution of B^c, with an attachment w that contains a's phase, a's identity,

and a *hop-counter* h which is initially zero (i.e., $w = (p, a, h)$, and initially $p = 0$ and $h = 0$). The pair $(0, a)$ of the phase and the identity is viewed as a *token* in *annexing* mode, and the hop-counter counts the number of times this token is sent. (The hop-counter, and other related variables which are defined later, are needed only when the FIFO discipline is not assumed. They are used to detect when a chasing message bypasses the message it chases.) This token traverses the graph, *annexing* each node of lower phase it passes to its *domain*. We say that a node a *belongs* to (the domain of) token (b, p) if (b, p) is the last token that annexed a. The annexing is done by having the annexed node store the phase and identity of the token. The value of the hop-counter is increased by one each time the token is sent (using operations 3 and 4), and each node annexed by this token records the maximal value of the hop-counter it had seen, in a local variable *MaxHop*. If a is the only node that was awakened spontaneously, then eventually the token $(0, a)$ will complete the traversal of the graph at some node d, having annexed all the nodes it passed. At this moment d declares itself as a leader and initiates another execution of B^c to announce its leadership.

Assume now that exactly two nodes, a and b, are spontaneously awakened. Then each node initiates a rooted execution of B^c, as before. In the case that token $(0, a)$ reaches a node c that was already annexed by token $(0, b)$, it stops the traversing and acts as follows:

(1) If $(0, b) > (0, a)$ lexicographically (i.e., $b > a$), then token $(0, a)$ stays at c and becomes a *candidate*.
(2) Otherwise, token $(0, a)$ becomes a *chasing* token, and it *chases* token $(0, b)$ (using operation 6). The attachment of a chasing token contains the identity of the chased token, $(0, b)$. In addition, it contains a variable *LastMaxHop*, which is the value of the variable *MaxHop* of the last node it passed. This chasing token marks every node it visits during this chase by "*chased*(0)", meaning that a chasing token has already passed this node at phase 0.

Since node a was annexed by token $(0, a)$ and node b by $(0, b)$, no token will annex all the nodes of the graph. Without loss of generality, assume that $b > a$. Then token $(0, b)$ eventually reaches a node which either was annexed by token $(0, a)$, or where $(0, a)$ stays as a candidate. In the former case, $(0, b)$ will start chasing $(0, a)$. Token $(0, a)$, on the other hand, will not chase token $(0, b)$, since $b > a$. Instead, if token $(0, a)$ reaches a node c that is either marked "*chased*(0)" or annexed by token $(0, b)$, token $(0, a)$ will wait as a candidate at c until it is reached by the other token, which may be either in the annexing mode or in the chasing mode.

If the FIFO discipline is not obeyed, then the chasing token may bypass the chased token $(0, a)$ on some edge. Thus the chasing token arrives at node d at the other end of this edge before $(0, a)$ does. In this case one of the following must hold:

(1) Either d does not belong to the domain of $(0, a)$ and it does not contain a candidate, or
(2) d belongs to the domain of $(0, a)$ and the value of *MaxHop* in node d is smaller than the value of *LastMaxHop* of the chasing token.

Upon recognizing that one of these cases has happened, the chasing token stays at d as a candidate, and it is guaranteed that the bypassed token $(0, a)$ will eventually arrive at d.

In any case, one token will become a candidate at some node, and another token will meet it at that node. At this moment both tokens are destroyed, and a new token, at a phase higher by one, is created. The identity of this new token is the identity of the node in which it was created, d. Now we are left with only one token, which, as in the first case, is going to start traversing the graph. This token ignores lower phase tokens and will not reach any node visited by another token at a phase greater than or equal to its own. Thus this token will complete a full traversal of the graph at some node v. At this time v is elected as the leader.

In the general case, each token (p, a) is in one of three modes:

(i) *Annexing* mode. A token in this mode is trying to annex all the nodes in the network to its domain. For this, the token is using B^c to traverse the network, and it annexes the nodes it passes during the traversal.

(ii) *Chasing* mode. A token in this mode is chasing some token (p, b) $(b \neq a)$ in the annexing mode, attempting to reach it and then to create a token in a higher phase.

(iii) *Candidate* mode. A token in this mode has a phase p and is waiting to be met by a chasing or annexing token in the same phase.

Whenever an annexing or chasing token reaches (or is created at) a node c, the following rules are applied, according to the mode of the token.

Annexing Mode. Whenever a token (p, a) in the annexing mode reaches (or is created at) a node c, which belongs to a token (q, b), the following rules are applied.

(a1) The annexing is continued if: (1) the corresponding execution of B^c is not terminated, and (2) one of the following holds: $q < p$ or else $(p, a) = (q, b)$ and node c is not marked "*chased*(p)". If this condition is satisfied, node c performs the following: (i) It joins the domain of (p, a) (if it is not yet there), (ii) it increases the value of the hop-counter of the token by one, and sends it forward (using operations 2–4), and (iii) it records the value of the updated hop-counter in the local variable *MaxHop*.

(a2) If the traversal is completed successfully (by annexing all nodes) then c is the elected leader. In all other cases token (p, a) is destroyed (using operation 5), and one of the following applies:

(a3) If $p < q$, then token (p, a) is destroyed, and no more steps are taken.

(a4) If c contains a candidate token in phase p, then both (p, a) and the candidate token are destroyed, and a new token $(p + 1, c)$ in the annexing mode is created. This new token starts the annexing process, beginning from the node c where it was created. Its hop-counter is initialized to zero.

(a5) If $p = q$, and either node c is marked "*chased*(p)" or $b > a$, then token (p, a) enters the candidate mode, and waits at c.

(a6) Otherwise (i.e., $p = q$, $b < a$, and node c is not marked "*chased*(p)"), token (p, a) is destroyed, and a token in the chasing mode, which is chasing (p, b), is created.

Chasing Mode. Whenever a token in this mode, which is chasing a token (p, a), reaches (or is created at) a node c which belongs to a token (q, b), the following rules are applied.

(c1) The chasing is continued if (1): $(p, a) = (q, b)$, (2) the value of the variable *LastMaxHop* carried with the chasing token is smaller than the value of the variable *MaxHop* stored at the node c, and (3) node c is neither marked "*chased*(p)", nor does it contain a token at phase p in the candidate mode. If these conditions are satisfied, node c performs the following: (i) it marks itself by "*chased*(p)", (ii) it sets the value of *LastMaxHop* of the chasing token to the value of *MaxHop* of itself, and (iii) it continues the chasing (using operation 6 on the message containing the chased token). In all other cases, the chasing is stopped, and one of the following applies.

(c2) If $p < q$, then the chasing token is destroyed, and no other steps are taken.

(c3) If c contains a candidate token at phase p, then both the candidate and the chasing tokens are destroyed, and a new token $(p + 1, c)$ is created, as in (a4).

(c4) Otherwise (i.e., either (1) $(p, a) = (q, b)$, and either node c is marked "*chased*(p)" or the value of *MaxHop* at node c is not larger than the value *LastMaxHop* at the chasing token, or (2) $(p, a) \neq (q, b)$, and $p \geq q$): the chasing token enters the candidate mode, and waits at c.

After a leader is elected, one more execution of B^c to announce its leadership to all other nodes might be needed.

3.3 Complexity and Correctness Proofs

LEMMA 1. *The number of distinct tokens at phase p created in an execution of the algorithm is at most* $k \cdot 2^{-p}$, *where k is the number of nodes that start the algorithm spontaneously.*

PROOF. Lemma 1 is proved by the facts that any annexing token at phase $p > 0$ is created by destroying two tokens at phase $p - 1$ (rules (a4) and (c3)), and that a chasing token at phase p is created by destroying an annexing token of the same phase. □

LEMMA 2. *In every execution of the algorithm, at most one node is declared as a leader.*

PROOF. Let (p, a) be the first token that declares some node c as a leader. Then (p, a) is the first token that has completed a traversal, which implies, by (a1), that (p, a) has not encountered any node that belonged to another token at phase $\geq p$. Moreover, no other token at phase $\geq p$ will ever be created. Also by (a1), no token (q, b) with $q < p$ could complete the traversal. □

LEMMA 3. *At any phase, the number of messages sent by the algorithm with a token in this phase is at most* $f(n) + n$.

PROOF. Assume that d tokens were created in phase p, and that the domain of the ith token, $1 \leq i \leq d$, contains n_i nodes. Since no two domains in the same

phase overlap, we have that

$$\sum_{i=1}^{d} n_i \leq n.$$

By the traversability property, the annexing token (p, i) occurred in at most $f(n_i)$ messages. Thus, we obtain by the convexity of f that the number M of messages with annexing tokens in a given phase satisfies

$$M \leq \sum_{i=1}^{d} f(n_i) \leq f\left(\sum_{i=1}^{d} n_i\right) \leq f(n).$$

Since every node that sends a chasing token at a given phase p is marked "*chased*(p)", and a node that is marked "*chased*(p)" does not send any chasing token at phase p (rule (c1)), the number of messages with chasing tokens at a given phase is bounded by n. The lemma follows. □

Note that the term n in the bound of the above lemma comes from the bound of at most n messages with chasing tokens at any given phase. It will be shown that in certain cases this bound can be reduced by simplifying the analysis of the chasing.

LEMMA 4. *The number of messages sent by the algorithm is at most* $(n + f(n)) \cdot (\log_2 k + 1)$, *where* k *is the number of nodes that start the algorithm spontaneously.*

PROOF. By Lemma 1, the number of phases is bounded by $\log_2 k + 1$. By Lemma 3, at most $n + f(n)$ messages are sent by the algorithm with tokens in any given phase. The lemma follows. □

LEMMA 5. *If there is more than one token at a certain phase* p, *then a token at phase* $p + 1$ *is eventually created.*

PROOF. Assume the contrary. Then there is a phase p such that there are at least two annexing tokens at phase p, and no token at phase $p + 1$ is ever created. We show that this is impossible.

Let (p, i) be a token in phase p with the maximum possible i. When (p, i) is created, it invokes an execution of B^c to traverse the graph. Since there are other annexing tokens at phase p, this execution cannot complete the annexing of all the nodes in the network and hence must be aborted upon reaching some node c (see rule (a1)). By the maximality of i and of p, one of the following must have happened at the time (p, i) reached c:

(1) c belonged to a token (p, j) with $j < i$, and hence by rule (a6), a chasing token at phase p, which chased (p, j), was created, or
(2) c was already marked "*chased*(p)".

In both cases, a chasing token at phase p must have been created. Out of all annexing tokens in phase p which are chased, let (p, t) be the one with the minimum possible t.

Out of all messages containing a chasing token which is chasing (p, t), consider the one with the maximal value of *LastMaxHop*. Denote this message by C and the value of its *LastMaxHop* by $l(C)$. Note that C is unique, as by (c1) no node sends more than one token that chases (p, t); therefore, no two messages with the same value of *LastMaxHop* chase a message with the token (p, t).

Let A be the message chased by C, that is, the unique message with an annexing token (p, t) whose hop-counter $h(A)$ equals $l(C)$. Let e be the edge that carried A (and hence also carried C), and let v be the node that received A (and hence also C).

By the maximality of $l(C)$, the chasing token carried by C was not forwarded by v. In other words, node v, upon receiving message C, destroyed the chasing token carried by it, by executing one of the operations (c2)–(c4). If it executed (c2) or (c3), then a node of higher phase was created—a contradiction. We conclude that v destroyed C by executing operation (c4), by which a candidate token of phase p was created at v. By the maximality of p, this candidate node was never destroyed (a candidate token at phase p which stays at node v is destroyed only when a token of higher phase is created at v or annexes v). We consider two subcases:

(1) Message C bypassed A on the edge e. Then A will eventually reach v, will find a candidate node of phase p in it, and by executing operation (a4) will create an annexing token of phase $p + 1$. A contradiction.

(2) Not (1), that is, A arrived at v before C. Then, eventually, C will also arrive at v. By the maximality of $l(C)$, C will not continue chasing the annexing token (p, t), carried by A. This implies that v did not forward this annexing token upon the receipt of A, which means that when v received A, the annexing token (p, t) was destroyed. By the maximality of p and the minimality of t, this could happen only if upon receiving A, v executed operation (a5), and thus created a candidate token at phase p which stayed at v. By the same argument as in (1) above, when C arrives at v, it will find this candidate token there, and a token of phase $p + 1$ will be created—a contradiction. □

PROOF OF THE LEADER FINDING THEOREM. Clearly, at the beginning of the algorithm there is at least one token. By Lemmas 4 and 5, the algorithm will eventually get to a situation where there is a unique token at some phase p, and no other token in this phase will ever be created. This token, which is an annexing token, will complete the traversing of the graph (completing B^c), and then will create a *leader declaring* token, which by Lemma 2 is unique. The complexity of the algorithm follows by Lemma 4. □

Note that most of the complication in the proof of Lemma 5 is due to the fact that we do not assume the FIFO discipline. Indeed, in the case that this discipline is guaranteed, the mechanism for counting and recording the number hops taken by annexing tokens (using the hop-counter and the variables *MaxHop* and *LastMaxHop*) is redundant. Consequently, the algorithm can be considerably simplified.

3.4 Construction of Spanning Trees

In this section it is shown how the traversal algorithm B^c can be used after a leader is elected to construct directed spanning trees, containing paths to (from) the leader from (to) all other nodes. For this construction alone (and not for the leader election which may have preceded it), we assume that a rooted execution of B^c always terminates at the node that initiated it. (This can always be achieved by at most doubling the message complexity of B^c, when it is used not for leader election, but for spanning tree construction.)

Let node c be the elected leader. Upon its election, c initiates a "leader declaring" execution of B^c. Associate with each node, excluding c, the edge on which the node had sent the last message in this execution. Clearly, this set of edges, directed in the directions of these last messages, constitutes a spanning tree in which there is a directed path from every node to the leader.

Note that in the description above, a node might not know when its participation in the leader declaration execution of B^c is completed, neither what edge leaving it belongs to the tree rooted at the leader. This problem is solved in the following way:

(1) In the leader declaration execution of B^c, attach to the token a hop-counter. Also, each node a records the value *MaxHop* of the hop-counter the token had the last time a sent it (as in the annexing mode of the leader finding algorithm).

(2) After the leader declaration execution of B^c is terminated at node c, it initiates another execution of B^c which is identical to the previous one. When the hop-counter of the token in this execution upon leaving a node a equals the value of *MaxHop* from the previous execution at a, it terminates its part in the algorithm and takes the edge that carried the last message to be the one directed to the leader.

B^c can also be used to construct a spanning tree of directed paths from the leader to all other nodes: In the undirected case the direction of the edges in the tree described above are reversed. In directed or mixed graphs this can be done in the following way:

(1) Execute B^c from the leader, with an edge counter, that counts the distinct edges used by it. This assigns a unique name (number) to each edge used by B^c. This name is known to both ends of the edge.

(2) Each node records the name of the edge on which it received the first message. Clearly, the set of these edges constitutes a tree having the desired properties.

(3) Another two executions of B^c (using messages of length $O(n \log n)$) are used to accumulate the edges of the tree at the leader, and then to spread them to all nodes.

The message complexity of both algorithms is $O(f(n))$ $[O(f(m))]$. However, in the latter algorithm attachments of $O(n \log n)$ bits might be needed. Alternatively, it is possible to modify this latter algorithm to work with $O(nm + f(n))$ $[O(nm + f(m))]$ messages of $O(\log n)$ bits.

4. RESULTS OBTAINED BY USING THE GENERAL ALGORITHM FOR FINDING A LEADER

In order to demonstrate the power of the leader finding theorem we now present traversing and edge-traversing algorithms for various classes of networks, and use them to obtain leader finding algorithms for these classes of networks.

4.1 Finding a Leader Using $O(x)$ Traversals

Five examples of $O(x)$ traversable classes of networks are presented below. Applications of the leader finding theorem on these classes yield algorithms for finding a leader in $O(n \log n)$ messages. For two of these classes, circular and complete networks, there are known leader finding algorithms which use only $O(n \log n)$ messages. For the other three cases no such algorithms were published before. In the known two cases quite sophisticated special algorithms were designed [1, 2, 4, 8, 13, 14, 16, 17, 23], whereas it is a trivial task to design suitable traversing algorithms for these classes of networks.

(1) *Circuits (Unidirectional and Bidirectional).* Circuits are trivially x traversable (the initiator sends a message to one of its neighbors, and the message is forwarded around the circle until stopped by its initiator). Thus our leader finding algorithm will find a leader in any circle using no more than $2n \cdot (\log_2 n + 1)$ messages. Note that the message length required by the traversal algorithm itself is only $O(1)$ bits.

(2) *Complete Undirected Graphs.* The class of complete graphs is $2x$ traversable. The initiator sends a message to one of its neighbors and waits for an acknowledgment. This operation is repeated until all the initiator's neighbors are traversed. At first glance this traversing algorithm, together with the leader finding theorem yields an algorithm for finding a leader in complete undirected graphs in $(n + 2 \cdot n)(\log_2 n + 1)$ messages. However, a simple elaboration of the analysis yields a better bound. To see this recall the computation of the complexity of the leader finding algorithm: the term $(n + f(n))$ appears in $(n + f(n))(\log_2 n + 1)$ as in each phase $f(n)$ messages may be used for traversal of tokens in annexing mode, and n for chasing. In this special case of complete undirected graphs one can easily verify that at most three messages are used for the chasing of a token. The message complexity for finding a leader thus becomes $(2n \cdot \log_2 n) + O(n)$. (The $O(n)$ term is for the chasing: up to $3n \cdot 2^{-p}$ messages in phase p.) This is better in a constant factor than the complexity in [8, 14, 16].

Note. After the first version of this paper was completed, we have received a revised version of [8] in which two other algorithms were presented which also achieve the same message complexity $((2n \cdot \log n) + O(n))$.

(3) *Complete Undirected Bipartite Graphs.* A node initiating a traversing in a complete bipartite graph will send a message serially to all its neighbors, as in the case of a complete graph (i.e., at any given time there is at most one unacknowledged message). Next the initiator will ask its last visited neighbor, say b, to send serially messages to all b's other neighbors. Clearly, now, the class of complete bipartite graphs is $2x$ traversable, yielding a $3n \log n$ leader finding algorithm. However, the chasing of a token in this case is done by

at most 4 messages, and hence the complexity of leader finding is reduced to $(2 \cdot n \cdot \log n) + O(n)$ messages.

(4) *Undirected Graphs of Radius* 1. These are graphs that contain a node which is a neighbor of all other nodes. A node i, initiating a traversing in a graph of radius 1, will also send a message serially to all its neighbors, as in the case of complete graphs. In this case, however, each acknowledgment will include the number of neighbors of the acknowledging neighbor. When this process ends, any acknowledgment containing the largest number of neighbors must have come from some node c which is a center. Node i thus sends a message to c, asking it to send serially messages to all c's neighbors. Clearly, now, the class of graphs of radius 1 is $4x$ traversable, yielding an $O(5n \log n)$ leader finding algorithm. However, since chasing is done in this case with at most four messages, the complexity is reduced to $(4 \cdot n \cdot \log n) + O(n)$ messages.

(5) *Complete K-Partite Graphs.* These are graphs in which the nodes are partitioned into $K \geq 2$ sets, and each node in any set U is connected to all nodes except those in U. Clearly, this set is $4x$ traversable by a method similar to the method described in (3) above. Also, the chasing is done in four messages. Hence a leader is found in $4n \log n + O(n)$ messages. This algorithm can be generalized to the class of graphs in which each node, together with any r of its neighbors (for some fixed r) form a dominating set (note that for K-partite graphs, $r = 1$). The message complexity for general r is $2(r + 1)n \log n + O(n)$.

4.2 Finding a Leader Using $O(x)$ Edge Traversal

In this subsection we use the edge traversability property to derive an algorithm for finding a leader in directed Euler networks. The complexity of this algorithm is $O(m \log n)$ (and is smaller, by a constant factor, than that of an algorithm for the same task that was obtained independently in [9]). This is a generalization of the known results for directed circuits.

We use a simple algorithm to traverse the edges of an Euler graph in $2m$ messages: Start from the initiating node and proceed traversing unused edges as long as possible. When arriving at a node with no untraversed outgoing edges, the traversal of a (not necessarily simple) circuit has been completed. Retraverse this circuit in the same order, until a node v with an outgoing unused edge is met, or until the second traversing is completed. In the former case, repeat the procedure in a recursive manner, starting from v. Incorporating this traversal algorithm in the general algorithm yields an $(2m + n)\log n$ leader finding algorithm in Eulerian directed graphs. For comparison, the message complexity of the algorithms of [7, 24], when applied to Euler directed graphs, use up to $O(nm)$ messages. The message complexity of this algorithm is better, by a factor of 2, than that of an algorithm which was independently designed specially for the same task [9].

4.3 General Networks

In this section we use our leader finding theorem while generalizing the notion of traversal. This yields results which improve upon the known results for general undirected networks in the standard model, as well as in a certain nonstandard model.

4.3.1 *An Adaptive DFS Traversal.* The leader finding algorithm discussed in this paper repeatedly uses a given traversal algorithm on a given network. In this subsection we modify the traversal so as to use information it obtains during early executions in order to improve its performance in later executions.

If we use the depth first search algorithm (DFS) (see, e.g., [5, ch. 3]) as a traversal in general graphs in our leader finding algorithm, we obtain a leader finding algorithm with a message complexity of $O(m \log n)$, which is worse than the $O(m + n \log n)$ complexity of the algorithm in [12]. We show below how to improve it to yield a leader finding algorithm whose message complexity is smaller than that of the algorithm of [12] by a constant factor and is still better in the order of the message complexity when k, the number of initiators of the algorithm, is small. For this sake, we modify the DFS traversal to include operations of edge deletions from the graph, as follows (a similar idea appeared also in [11]):

Consider an annexing token (p, a), which traverses an edge e from node c to node d and finds that node d already belongs to its domain. In this case e is a DFS *back* edge, and the next move of the token, according to the DFS traversal, is moving back on e from d to c. We modify the traversal so that while traversing back, the token marks e *deleted* in d (upon departure) and in c (upon arrival).

In the leader finding algorithm, the only tokens which take deleted edges into consideration are annexing tokens. Suppose that such a token (p, a) arrives at a node d from a node c over an edge e and finds that some edge f is marked *deleted* in d. If (p, a) does not cease to be an annexing token upon arriving to d, then it continues the traversal according to the following rules:

(Case 1). $e \neq f$. The token ignores f.

(Case 2). $e = f$. (This means that e was deleted by another token that traversed it.) The token returns immediately to c (and ignores f).

Consider a given execution of the algorithm and call an edge *deleted* if it was marked *deleted* during this execution. Let $G_{\text{nondeleted}}$ be the subgraph of G that contains all the edges that are not deleted.

Claim. $G_{\text{nondeleted}}$ is a connected graph.

PROOF. Assume the contrary and let $E_{\text{connecting}}$ be the set of *deleted* edges such that each edge in $E_{\text{connecting}}$ connects two connected components of $G_{\text{nondeleted}}$. Among the edges in $E_{\text{connecting}}$, let e be one that was marked *deleted* by a token (p, a) with largest p, and let c and d be the ends of e. There is a path in G, between nodes c and d, which consists of the DFS *tree* edges of the traversal of token (p, a). Since c and d belong to different connected components of $G_{\text{nondeleted}}$, some edge f of this path was marked *deleted*, and there is no path in $G_{\text{nondeleted}}$ between the end points of f. Thus edge f belongs to $E_{\text{connecting}}$. However, f could be marked *deleted* only by a token in phase larger than p. A contradiction. □

Next we bound the message complexity of the algorithm. First note that the changes made in the traversal do not change the fact that a node never sends more than one token in the chasing mode at each phase, and hence the total number of messages with chasing tokens is bounded by $n \log n$.

It remains to bound the number of messages with annexing tokens. For this, we partition these messages to three kinds. Consider a message with an annexing token (p, a) sent along an edge e from c to d. Then this message is of one of the following types:

(1) Last message: If upon arrival at d, (p, a) ceases to be an annexing token.
(2) DFS tree message: If either (p, a) annexes node d (and hence add e to its DFS tree), or e was already in its DFS tree upon leaving c.
(3) Other message. The possibilities are
    (3.1) (p, a) finds that e is a back edge, or
    (3.2) (p, a) marked e deleted at c, following a message of type (3.1) from d to c, or
    (3.3) (p, a) finds that e was marked deleted at d, or
    (3.4) (p, a) is sent back on e following a message of type (3.3) from d to c.

Clearly, since there are at most $2n$ tokens (Lemma 1), there are at most $2n$ messages of type (1). To bound the messages of type (3), we have the following observation.

OBSERVATION. *There are at most* $2m$ *messages of type* (3).

PROOF. Consider an edge e that was marked deleted during the algorithm. Then the following must be true.

(1) Since in type (3) last messages are excluded, there is at most one message of type (3.2) or (3.4) on e, namely, the message with the token of the highest phase sent on e.
(2) Similarly, among all messages of types (3.1) or (3.3) sent on e, there is at most one with a token (p, a), which is not second to last message of this token in the annexing mode, namely, the message followed by the message in (1) above; all other messages of this type are followed by messages from d back to c (Case 2), which cease to be annexing token upon arrival to c, since they find that a token with a higher phase had already annexed c.

Consider all edges that were deleted during the algorithm. There are at most $m - n + 1$ such edges, and hence the number of messages of types (3.2) or (3.4) that were sent on them is at most $m - n + 1$. Also, there are at most m messages of types (3.1) and (3.3) with a maximal phase sent on them. All the remaining messages are second to last messages of the corresponding annexing tokens, and hence there are at most $2n$ such messages. Thus, the total number of messages of type (3) is at most $2m$. □

To bound the number of messages of type (2), we define the complexity of the DFS traversal to be the number of messages of this type it sends (i.e., we give the messages of type (2) weight 1, and the other messages weight 0). Under this weighted complexity, the DFS is a $2n$ node traversal. It is easy to see that the leader finding theorem remains valid for this notion of weighted complexity and hence it yields an upper bound of $2n \log n + O(n)$ on the number of messages of type (2). Thus, the total complexity of the algorithm is $2m + 3n \log k + O(n)$.

This performance is better than the previous results [12] by a constant factor. This also implies a better order of worst-case performance when k (the number of initiators of the algorithm) is small. In contrast, the algorithm in [12] does not take advantage of small k (note, however, that in [12] a minimum weight spanning tree is constructed).

4.3.2 *Application in a Neighborhood-Knowledge Model.* In [21] a model in which each node knows its neighbors, and messages are permitted to be abitrarily long, is considered (we call this model "neighborhood knowledge"). The annexing token in this case can carry the whole description of its DFS tree, thus avoiding the traversing of back edges. The complexity of this traversal is $O(n)$, which yields, using our leader finding method, a $3n \log n + O(n)$ leader finding algorithm. The algorithm of [21] is shown there to have this message complexity only in the average case, under certain specific assumptions. The worst case message complexity in [21] is said to be $O(n^2)$.

ACKNOWLEDGMENTS

We would like to thank R. Gallager and two anonymous referees for their helpful comments.

REFERENCES

1. AFEK, Y., AND GAFNI, A. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of the 4th Annual ACM Symposium on Principles of Distributed Computing* (Minaki, Canada, Aug. 1985), pp. 186–195.
2. BURNS, J. E. A formal model for message passing systems. Tech. Rep. TR-91, Indiana University, Sept. 1980.
3. BOUGE, L., AND FRANCEZ, N. A compositional approach to superimposition. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Programming Languages* (San Diego, Calif., Jan. 1988), pp. 240–249.
4. DOLEV, D., KLAWE, M., AND RODEH, M. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *J. Algorithms 3* (1982), 245–260.
5. EVEN, S. *Graph Algorithms.* Computer Science Press, 1979.
6. FREDRICKSON, G., AND LYNCH, N. The impact of synchronous communication on the problem of electing a leader in a ring. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing* (Washington, D.C., 1984), pp. 493–503.
7. GAFNI, E., AND AFEK, Y. Election and traversal in unidirectional networks. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 1984), pp. 190–198.
8. GAFNI, E., AND AFEK, Y. Simple and efficient distributed algorithms for election in complete networks. In *Proceedings of the 22nd Annual Allerton Conference on Communication, Control, and Computing* (Monticello, Ill., Oct. 1984), pp. 689–698.
9. GAFNI, E., AND KORFHAGE, W. Distributed election in unidirectional Eulerian networks. In *Proceedings of the 22nd Annual Allerton Conference on Communication, Control, and Computing* (Monticello, Ill., Oct. 1984), pp. 699–700.
10. GALLAGER, R. G. Choosing a leader in a network. Unpublished memorandum, M.I.T., Cambridge, Mass., 1977.
11. GALLAGER, R. G. Finding a leader in networks with $O(E) + O(N \log N)$ messages. Internal Memo., M.I.T., Cambridge, Mass., 1978.
12. GALLAGER, R. G., HUMBLET, P. M., AND SPIRA, P. M. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 66–77.
13. HIRSHBERG, D. S., AND SINCLAIR, J. B. Decentralized extrema-finding in circular configurations of processors. *Commun. ACM 23*, 11 (Nov. 1980), 627–628.

14. HUMBLET, P. Selecting a leader in a clique in $O(n \log n)$ messages. Intern. Memo., Laboratory for Information and Decision Systems, M.I.T., Cambridge, Mass., 1984.
15. KORACH, E., AND MARKOVITZ, M. Algorithm for distributed spanning tree construction in dynamic networks. Tech. Rep. 401, Dept. Computer Science, Technion, Haifa, Israel, Feb. 1986.
16. KORACH, E., MORAN, S., AND ZAKS, S. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., Canada, Aug. 1984), pp. 199–207.
17. KORACH, E., ROTEM, D., AND SANTORO, N. A probabilistic algorithm for decentralized extrema-finding in a circular configuration of processors. Tech. Rep., University of Waterloo, Ont., Canada, 1981.
18. KORACH, E., ROTEM, D., AND SANTORO, N. Distributed algorithms for finding centers and medians in networks. *ACM Trans. Program. Lang. Syst. 6*, 3 (July 1984), 380–401.
19. KUTTEN, S. A unified approach to the efficient construction of distributed leader-finding-algorithms. Presented at the IEEE International Conference on Communication and Energy, Montreal, Canada, Oct. 1984.
20. KUTTEN, S. Traversing directed graphs—An upper and lower bound. Internal Memo., 1984.
21. LAVALLEE, I., AND ROUCAIROL, G. A fully distributed (minimal) spanning tree algorithm. *Inf. Processing Lett. 23* (Aug. 1986), 55–62.
22. PACHL, J., KORACH, E., AND ROTEM, D. Lower bounds for distributed maximum-finding algorithms. *J. ACM 31*, 4 (Oct. 1984), 905–918.
23. PETERSON, G. L. An $O(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Trans. Program. Lang. Syst. 4*, 4 (Oct. 1982), 758–762.
24. SEGALL, A. Distributed network protocols. *IEEE Trans. Inf. Theory IT-29*, 1 (Jan. 1983), 23–35.
25. TANENBAUM, A. S. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
26. TIWARI, P., AND LOUI, M. C. Simulation of chaotic algorithms by token algorithms. In *Distributed Algorithms on Graphs*, E. Gafni and N. Santoro, Eds. Carleton University Press, Northfield, Minn., 1986, pp. 145–152.
27. VITANYI, P. M. B. Distributed election in an Archimedean ring of processors. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing* (Washington, D.C., 1984), pp. 542–547.

Received April 1987; revised March 1988 and August 1989; accepted August 1989.