Computer Communication Technologies for the 90's
J. Raviv (Editor), Elsevier Science Publishers B.V. (North-Holland)


# STEPWISE CONSTRUCTION OF AN EFFICIENT DISTRIBUTED TRAVERSING ALGORITHM FOR GENERAL STRONGLY CONNECTED DIRECTED NETWORKS

or: Traversing One Way Streets with no Map

(Extended Abstract)

**Shay Kutten**
*IBM T. J. Watson Research Center*
*Yorktown Heights, NY 10598.*

## ABSTRACT

We study the problem of distributively traversing directed networks, and constructing directed spanning trees. New lower bound on the message complexity is given. Then we present an algorithm whose message complexity approaches that lower bound in some important cases. In the worst case this message complexity is not worse than that of the known algorithm. Such an algorithm is important for solving many problems in directed networks. Among others are leader election (i.e. mutual exclusion), and performing a broadcast, either with or without acknowledgements. Such an algorithm is also needed in radio networks in order to coordinate transmissions.

Our algorithm also constructs two spanning trees. The first consists of routes from the traversal initiating node to all other nodes. The other spanning tree consists of routes from all nodes to the traversal initiating node. Such trees are needed for efficient routing in directed networks, and thus are a basic communication tool in communication networks. Except for their usage for routing users' mail, they can be used by distributed algorithms and network's managing programs. For example- some papers which deal with the detection of distributed termination, assume (but do not construct) the existence of some mechanism to collect information in directed networks. Using our trees, such a mechanism becomes straightforward.

The traversal algorithm is constructed in a systematic way, step by step, starting from a very simple algorithm to traverse directed rings. The algorithm is developed by going on changing the assumptions on the graph, and changing the algorithm as a result.

## 1. INTRODUCTION

Consider the graph theoretic problem of traversing with no map a city of one way streets. In terms of distributed algorithms consider a network with processors connected only by unidirectional links. A token is generated by one node, and it is required to route that token on a path which includes all the network's edges. Subsection 1.1 gives the formal model and problem definition. Subsection 1.2 gives motivations, describes the results of this paper and previous results, and outlines the rest of the paper.

### 1.1 Formal Model and Problem Definition

The network is viewed as a directed graph $G(V,E)$ where $|V| = n$ and $|E| = m$. The graph's nodes represent the network's processors and a directed edge represents an unidirectional link. Neither $n$ nor $m$ are known to any processor. Each node has a unique identity (denoted *id*) known only to itself. Each processor knows the lines connected to itself and their directions, (but not the identities of its neighbors).

Every processor executes a copy of the algorithm presented in this paper. The algorithm executed by a node, $v$, includes operations of (1) sending a message over an edge directed out from $v$; (2) receiving a message over an edge directed to $v$; and (3) processing information in $v$'s (local) memory. We assume that the messages on each line arrive in a finite time, with no error, and are kept until processed. Each message contains $O$ (log maximum id). The complexity measure is the number of messages required.

A distributed algorithm $A$ is *covering* on a class $\Gamma$ of graphs if for every graph $G = (V,E)$ in $\Gamma$, and for every execution of $A$ on $G$, a message is sent over every edge in $E$ 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 denied from the node when it transmits a message.

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

Note that a distributed traversal algorithm is, in some sense, equivalent to a non-distributed search algorithm. our aim in this paper is to construct an efficient traversal algorithm for directed networks.

### 1.2 Motivations, Results and Outline of the paper

This problem is a generalization of the traversing problem solved by the very famous serial (non-distributed) Depth First Search (see e.g. [E79]). The latter cannot be used here, since it retreats against the direction of edges. (We call our algorithm the Non-Retreating Traversal.) This problem is also another form of the graph Eulerization problem, i.e. duplicating edges s.t. the graph is transformed into an Euler Graph [IIPR81].

Such a traversal algorithm is useful for several reasons. First, it can be installed in the modular technique for constructing leader election algorithms, [KKM85] and thus yield a leader election algorithm. In [KKM85], the more efficient is the traversal, the more efficient is the leader election. The algorithms to elect a leader in the large family of strongly connected directed graphs have a high message complexity [S83, GA84]. This motivates the design of efficient traversal algorithms for this family.

The traversal token can carry a broadcast message s.t. the broadcasting node knows when its message has already been received by all nodes (i.e. when the token returns on the termination of the traversal). Also, any traversal algorithm can be used to construct two spanning trees rooted at the traversal initiator. One is a tree of paths from the root to all nodes, the other is a (different) tree of paths from all nodes to the root ffIKM85". These trees enables an efficient communication, e.g., a broadcast can be performed on them using $O(n)$ messages instead of $O(m)$. If acknowledgements are desired from all nodes, then $O(n)$ messages are used instead of $O(n\,m)$. Such trees are also required for efficient routing (e.g of users' mail) [H83].

A path via all edges, or all nodes, (such as the path constructed by a traversal algorithm) is used in several papers (but not constructed) to detect termination [CM85, M83, K85]. Another use is to coordinate nodes transmissions in radio networks [GF82, GF86]. Directed edges there arise from differences in nodes' transmission radiuses [CK85]. Also, solutions of problems in undirected graphs, can easily be adapted to directed graphs, once this method to acknowledge is established (e.g. the algorithm of [FS86]).

The message complexity of the traversal algorithms suggested so far is $O(n\,m)$ [GA84, K84a]. This upper bound is not tight. For example: there is no strongly connected directed graph in which $n = m$ which cannot be traversed in less then $O(n\,m)$ messages. In fact, every such graph is a simple cycle, and its traversal requires only $n$ messages.

In this paper we define a new graph parameter, the *deficit*, denoted by $x$ , and show that the lower bound on the message complexity of traversing is $O(m + n\,x)$. For some graphs $x = O(m)$, but in large families of graphs $x$ is very small. For example in directed Euler graphs (including cycles) $x = 0$.

Next we introduce the Non-Retreating Traversal algorithm (NRT), which traverses any strongly connected directed graph in $O(n + n\,m)$ messages. However, in many cases, including the important case of Euler graphs, it uses only $O(n + n\,x)$, thus reaching the lower bound. The algorithm is developed step by step, starting from a very simple algorithm to traverse directed rings. The changes in every step are made in such a way that the algorithm remains correct, even though some assumption on the graph are changed, relaxed or implemented in a different way. We omit the formal description from this extended abstract.

In the second section we present the lower bound. The third section is dedicated to the development of the algorithm. The proof of the upper bound is given in the forth section.

## 2. LOWER BOUND

**Definition:** Let, $x_v$, the *deficit* of node $v$ , be the absolute value of the difference between the number of edges entering $v$ and the number of edges leaving $v$ , i.e. $x_v = |d_{in}(v) - d_{out}(v)|$. The *Deficit* of a graph $G(V,E)$, denoted by $x$, is: $x = \sum_{(v} \in V\} x_v$

**Theorem 2.1:** For every possible combination of $n,m,x$ there exists a graph which cannot be traversed using less then $m + n\,x$ messages.

Note that $x \leq 2m$ .

**proof:** Ommtted.

## 3. A STEP BY STEP DEVELOPMENT OF THE NRT ALGORITHM

We now develop the Non-Retreating Traversal algorithm. Instead of describing what does a node do when it receives a message with a token in it, we describe the algorithms by means of instruction to an imaginary traveling agent (the token). It is not hard to construct the translation between the two description methods. More examples of this description method can be found in [GA84, K84b, KKM85, BKWZ86].

We develop the algorithm step by step, starting from the very simple algorithm to traverse the simplest family of strongly connected directed graphs: directed simple cycles (i.e. rings). We argue that it is correct (but without a completely formal proof). Next we start to revise the algorithm to adapt it to more general types of graphs. We motivate each change, then we make the change taking care s.t. the arguments for the correctness of the previous step need only slight changes.

### 3.1 Traversing Directed Rings and 8-like Graphs

Ommitted.

### 3.2 Traversing Directed Euler Graphs

An Euler graph consists of one or more edges disjoint cycles. When there are more than one cycle, each cycle is connected to some other cycles via nodes they share. We have seen that when the algorithm to traverse a cycle ends, it is known whether there are edges not on this cycle. We change the algorithm such that, when such an edge is encountered during retraversal of the first cycle $C_1$ , the retraversal is postponed, and the algorithm is recursively applied to the unused edge. Call the cycle which is now discovered $C_2$.

**Definition:** If the traversing of cycle $C_2$ started during the retraversal of cycle $C_1$ then call cycle $C_2$ a *descendant* of cycle $C_1$, and $C_1$ an *ancestor* of $C_2$. The descendants of cycle $C_2$ are also descendants of cycle $C_1$, and cycle $C_1$ also their ancestor. The relations between the cycles thus form a tree, the root of which is the first cycle. The leaves of this tree are the cycles in the retraversal of which no new cycle was discovered. (They have no descendants.) See Fig. 2.

Consider a cycle, $C_4$ , which is a a leaf in the cycles tree. Assume that the traversal of cycle $C_4$ had started in node $u_4$, during the retraversal of some cycle, $C_3$. See Fig 2. Clearly the retraversal of cycle $C_4$ is never postponed and ends in node $u_4$. Thus the retraversal of cycle $C_3$, which had been postponed in node $u_4$ , can be resumed.

Remove from the graph any leaf cycle and the same argument can be applied to cycles which has no descendants in the new graph. Thus, eventually $C_2$ is (recursively) traversed and retraversed. At that time the token is again at node $u_2$. Thus the postponed retraversal of $C_1$ can be resumed. When the retraversal of the first cycle is completed, the retraversal of all the graph is completed. To summarize:

**Algorithm 1 (for the token):**
*Traverse a cycle starting from some edge.*
*Retraverse the last cycle*
**Whenever** during retraversal you encounter
an unused edge **do**
postpone the retraversal.
Activate the algorithm recursively
for the unused edge.
When the recursive activation terminates,
resume the postponed retraversal.

The following arguments are both the properties of the algorithm, and the sketch for the proof of correctness.
**Argument 1:** The traversal of a cycle always ends in the node from which it has started. Thus when the traversal of a cycle ends, the retraversal of the cycle can start.
**Argument 2:** An unused edge discovered during the retraversal of a cycle causes the traversal of a descendant cycle.
**Argument 3:** Eventually a cycle is retraversed, from which no new unused edges will be discovered. Thus its retraversal terminates.
**Argument 4:** When the retraversal of a cycle ends, the token is again in the node from which the traversal of that cycle has started.
**Argument 5:** When a retraversal of a cycle ends the postponed retraversal of its immediate ancestor can be resumed (from the same node).
**Argument 6:** A cycle, all of which descendant are retraversed, its retraversal can also be completed.
**Argument 7:** The retraversal of descendant cycles will always be completed before the retraversal of their ancestors will be completed.
**Argument 8:** When the retraversal of a cycle ends, all its descendants are retraversed.
**Argument 9:** Every cycle which is traversed will be retraversed.
**Argument 10:** No unused edge emanates from a node on a fully retraversed cycle.
**Argument 11:** Eventually the first cycle will be fully retraversed. at that time every edge in the graph will already be retraversed.

### 3.3 Using Paths Instead of Cycles

The first step is to replace the cycles by general simple paths (simple in edges). Clearly arguments 1, 4, and 5 do not hold any more. This can be corrected by some subroutine which will bring the token from the end of a path *backward* to the node from which the path starts. This change has the effect of creating virtual circuits, each consists of a traversed path, and of a second path which closes the cycle. Passing an edge to close a cycle will not be considered a part of the traversal. For example, the edge will not change its mark (used, unused, traversed). It can be said that a duplicate of the edge is created and used (for the closing path), and this duplicate does not belong to the graph to be traversed. Once we have introduced the method to find the closing paths, the correctness of the algorithm will follow from the correctness of the algorithm for the Eulerian case. (Substitute the word "cycle" in the arguments, by the word "path".) The formal algorithm is ommitted.

### 3.4 Partitioning a Path to several Paths

Coming to implement algorithm 2, we note the following. When a traversal of a path ends, the edges which belong to the closing path may still be unused. As we

have no global knowledge, it is not reasonable to expect that a closing path via yet unused edges, will be known. Let us thus look for a route (*backward paths*) which will lead not necessarily to the starting node of the path, but, rather, to some other node on the path, say node $v$. Thus retraversal can be accomplished for the suffix of the path starting from node $v$ and on.

This motivates the following change in the algorithm: When the traversal of a path ends, having traversed some edge $e_{last}$, a "go-back" subroutine is used to find a closing path- not necessarily to the starting node of the path. Instead it will lead to the tail of some edge, $e$, on the path. The walk that the token performed from $e$ to $e_{last}$ (including) will be called a *subpath*. The token will first retraverse this subpath. Then it will return to the starting node of the subpath and use the "go-back" subroutine in order to find a closing-path to a still earlier edge. This enables the token to retraverse a second subpath of this path. Etc. Eventually all the path will be retraversed.

It is not hard to generalize the arguments in order to prove the correctness of this algorithm. The algorithm and the example figure are ommitted from this abstract.

Note that we do not require that the token knows when it ends the traversal of one subpath, and starts the traversal of the next subpath of the same path. The path will be partitioned into subpaths only later, when the closing paths will be found.

### 3.5 How to Collect Information in order to Construct Closing Paths

The edges are numbered: (no. of path, no. of edge in the path). Each subpath is passed a third time, in order to find the lowest edge to which it is known how to arrive from this subpath. This information, needed to construct the closing paths, will be explained in the following section. Due to space limitation we omit here (and henceforth) the new quasi formal version of the algorithm.

### 3.6 Closing the Last Subpath of Each Path

Let node $v$ be the node in which the traversal of path $P_1$ ends. As no part of $P_1$ has already been retraversed, (i.e. no closing path information has been accumulated), we define the first closing path of $P_1$ to be empty. That is, if none of the edges emanating from node $v$ belongs to $P_1$, then $S_{last}(P_1)$, the last subpath of $P_1$, is empty. Otherwise the lowest edge which emanates from node $v$ and belongs to $P_1$, is taken to be the first edge of $S_{last}(P_1)$. In both cases, the token is in the first (and last) node on $S_{last}(P_1)$, and no closing path is needed.

In Fig. 6, node $w$ represents the first case, and node $v$ and edge $(P_1, 3)$ the second case. The last subpath of path $P_2$ is empty. The last subpath of $P_1$ contains the edges $(P_1, 3)$, $(P_1, 4)$, $(P_1, 5)$, $(P_1, 6)$, $(P_1, 7)$, and $(P_1, 8)$.

### 3.7 Collecting Information in order to Construct the Next Closing Path

When the retraversal of $S_{last}(P_1)$ is finished the token is again in the first node on the subpath. The re-retraversal of the subpath can thus start. As in the Eulerian case we deal first with the case of a leaf subpath (i.e. no new subpath has been discovered during the retraversal of subpath $S_{last}(P_1)$). By the strong connectivity of the graph there is a route from node $v$ to node $u$ from which the traversal of the graph has started (if $v = u$ then the graph's traversal has terminated.)

If node $u$ is on $S_{last}(P_1)$ then this route consists of edges in increasing numbering, ending in some edge $e_{elevator}$ which enters node $u$. In this case let us call the lowest edge leaving node $u$ the *low point* of node $v$ at that time. (In this case it must be edge $(P_1, 1)$.) Also let us call $e_{elevator}$ the *elevator entree* of node $v$ at that time. In Fig. 7 the low point is edge $(P_1, 1)$, and the elevator entree is edge $(P_1, 7)$.

Otherwise the route from node $v$ to node $u$ may start with edges on $S_{last}(P_1)$ but it must proceed with an edge not on $S_{last}(P_1)$. By argument 10 this edge is already used. By the assumption that no new paths were discovered from $S_{last}(P_1)$, this edge must belong to a former subpath, and thus must be lower than any edge on subpath $S_{last}(P_1)$. See Fig. 8.

Let us generalize the definition of the *low point* of node $v$ at that time, to be the lowest edge emanating from a node on $S_{last}(P_1)$. The *elevator entree* is defined as the highest edge entering the node from which the low point emanates. The sole roll of the re-retraversal is to enable the token to find (and bring to node $v$) the values of the low point and the elevator entree.

In Fig. 8 the low point of $S_{last}(P_2)$ is edge $(P_1, 4)$, and the elevator entree is edge $(P_2, 5)$.

Now let us explain how to close the subpath which ends in node $v$, (in which $S_{last}(P_1)$ starts). First the token is routed to the elevator entree. This can be done by repeating the original traversal. However a more economic way is to avoid the loops in this route in the following method. In every node, $s$, consider the edges which are lower than the elevator entree. let $e_{last}$ be the last (so far) among them to be used by the token to leave node $s$. The token is sent over $e_{last}$. Clearly this method will bring the token to the elevator entree using no more than $n$ messages (one per a node on the route). In Fig. 8 the token is not routed over edge $(P_2, 7)$, since it is higher than the elevator entree. It is also not routed from node $w$ over $(P_2, 3)$, since edge $(P_2, 5)$, which also emanates from $w$, is higher than $(P_2, 3)$, and still not higher than the elevator entree..

Now the token is in the node from which the low point emanates. If the low point belongs to the same path, then we define it to be the first edge of the next subpath to be traversed. The closing path in this case was the route which ended with the elevator entree.

**Example:** In Fig. 9 $S_{last}(P_1)$ is edges $(P_1, 4)$, $(P_1, 5)$, $(P_1, 6)$, and $P_1$, 7). The low point is $(P_1, 2)$ and the elevator entree is $(P_1, 5)$. The next subpath is edges $(P_1, 2)$ and ( $P_1$, 3). (For the sake of simplicity we did not bother to make the graph of Fig. 9 strongly connected. If it were strongly connected, then it would have been possible to close, later, also the subpath which contains edge $(P_1, 1)$.) This concludes the example.

In the complementary case the low point is lower than the first edge of the current path. See Fig 8. In that case the token is routed next to that first edge. The routing is done by the same method as before, i.e. the token is sent over the last edge used by the token to leave each node, among the edges which are still lower than the first edge of the current path. (By "current path" we mean the path that the token is currently traversing, retraversing, re-retraversing, or walking one of its closing paths.)

**Example:** In Fig. 8 the token is routed over $(P_1, 4)$, and then over $(P_1, 1)$, even though $(P^1, 3)$ is higher than $(P_1, 1)$ and still lower than $(P_2, 1)$. This is because $(P_1, 1)$ was the last (so far) to be passed from node $u$- this happened when $(P_1, 1)$ was retraversed. $(P_1, 3)$ has not yet been retraversed. This concludes the example.

### 3.8 Closing Other Subpaths

The mechanism to close the next subpaths, is a generalization of the method used above. As we have seen, from the first node on a leaf subpath there is a route of the leaf subpath's edges, leading to a lower edge. When the retraversal of the leaf subpath, $S$, ends, its low point and elevator entree numbering (i.e. the specification of the back subpath) are stored in the first node on $S$. This node belongs to the immediate ancestor subpath of $S$. When the retraversal of this ancestor ends, the re-retraversal of all its descendant has already ended (argument 7). Thus by re-retraversal this ancestor subpath, the token can determine the lowest low point of the descendant subpaths. This low point is compared to edges emanating from the current subpath. The lowest value is taken to be the low point of the current subpath. The elevator entree associated with the chosen low point is the elevator entree of the current subpath.

**Example:** In Fig. 10, $S_{last}(P_3)$ contains edges $(P_3, 3)$, $(P_3, 4)$, $(P_3, 5)$, and $(P_3, 6)$. Its retraversal is postponed in node $z$, in order to discover and traverse path $P_4$. Next the token returns to node z, using the closing path (P sub 3, 2), (P sub 3, 3). Next $P_4$ is retraversed and re-retraversed. Then the names (numbering) of the low point, i.e. $(P_3, 2)$, and the elevator entree, $(P_3, 3)$, are stored in node $z$. Likewise the values $(P_2, 6)$ and $(P_3, 4)$ are stored in node $y$ after the re-retraversal of path $P_5$, and the values $(P_2, 2)$ and $(P_3, 5)$ are stored in $x$.

When the token now completes the retraversal of $S_{last}(P_3)$ in node $v$, it re-retraverses $S_{last}(P_3)$ . It finds a name of a potential low point $(P_3, 2)$ in $z$. The token carries this value until it encounters the name of the better candidate for a low point $(P_2, 6)$, stored in $y$. This is the name the token now carries (together with the name of the corresponding elevator entree $(P_3, 4)$. In node $x$ this values are replaced in the token by $(P_2, 2)$ and its corresponding elevator entree $(P_3, 5)$. When the token finishes the re-retraversing again in node $v$, it decides that this last value is the best. It thus goes to $x$, then to $w$, then to $t$, and thus closes path $P_3$.

Now it can retraverse the next subpath of $P_3$, which contains $(P_3, 1)$ and $(P_3, 2)$. This concludes the example.

The routing of the token to the elevator entree is done as in a leaf subpath. Again, if the low point is lower than the first edge of the current path then the token is routed to the first edge of the current path. Recall that the closing paths are considered to consist of new edges (duplicates of graph edges) which do not have to be traversed. Thus when looking for the last edge used by the token to leave a certain node, we look only for the last use of an edge for traversing, retraversing or re-retraversing, and not for cycle closing.

When the token arrives at node $u$, the first node of some path $P_3$, having re-traversed the subpath which starts from node $t$, the retraversal of the whole path is finished. If path $P_3$ was not the first path then when path $P_3$ was discovered, the retraversal of a subpath of some lower path, $P_2$, was postponed. (In Fig. 10 it is the retraversal of $S_{last}(P_2)$ which was postponed in node $t$.) This retraversal is now resumed. (However, first the values of the low point and the elevator entree are stored in node $t$). If, however, $P_3$ was the first path then the algorithm is terminated.

## 4. COMPLEXITY ANALYSIS

The traversal, retraversal and re-retraversal are all done once per an edge. This yields $3m$ messages. The other messages are used for closing cycles.

**Lemma 4.1:** The union of the closing paths of one path, passes (leaves) each node at most three times.

**Proof:** Ommitted.

Since every path contains, at least, one edge, the complexity is $O(m\,n)$. To see on what kinds of graphs does the algorithm perform better, consider Euler graphs. In such graphs the deficit is zero. Thus, the lower bound is $O(m)$. Since no paths need be closed- this is also the complexity of NRT. Adding a few edges may increase the deficit. If the number of paths equals the deficit then, by the above lemma, the complexity will also match the lower bound. It may happen that a path is discovered from a node with no deficit. In this case the complexity may be higher. However, for many graphs this cannot happen, or can happen only a limited amount of times.

**Acknowledgements:**
It is a pleasure to thank Ephraim Korach, Shlomo Moran and Ronny Roth for useful discussions, and Ayla Matalon for proof reading.

## REFERENCES


[BKWZ86] Bar-Yehuda, R., Kutten, S., Wolfstahl, Y., and Zaks, S., "Making Distributed Algorithms Fault-Resilient", to appear in the *Proceedings of the 4th Symp. on Theoretical Aspects of Computer Science (STACS)*, Passau, FRG.

[CK85] Chlamtac, I., and Kutten, S., "On Broadcasting in Radio Networks- Problem Analysis and Protocol Design," *IEEE Trans. on Communications*, December 1985, pp. 1240-1246.

[CM85] Chandy, K.M., and Misra, J., "A Paradigm for Detecting Quiescent Properties in Distributed Computations," TR-85-02, CS. Dept., Univ. of Texas, January 1985.

[E79] Even, S., *Graph Algorithms*, Computer Science Press, 1979.

[FS86] Francez, N., and Shavit, N., "A New Approach to Detection of Global Quiescence," in the "*Proceedings of the 13th International Colloquium on Automata, Languages and Programming*, Rennes, France, July 1986, pp. 344-358.

[GA84] Gafni, E., and Afek, Y., "Election and Traversal in Unidirectional Networks," *3rd ACM symp. on Principles Of Distributed Computing (PODC)*, Vancouver, Canada, August 1984, pp. 190-198.

[GF82] Gold, Y., and Franta, W.R., "An Efficient Collision-Free Access-Protocol for Stationary Radio Networks with Less-Than-Full Connectivity," *3rd IEEE International Conference on Distributed Computing* Systems," Fort Lauderdale, USA, October 1982.

[GF86] Gold, Y., and Franta, W.R., "A Scheduling-Function-Based Distributed Access Protocol that Uses CDM to Relay Control Information with Hidden Nodes," to appear in IEEE Trans. on Computers.

[H83] Humblet, P.A., "A Distributed Algorithm for Minimum Weight Directed Spanning Trees," *IEEE Transactions on Communications, Vol. COM-31, No. 6, June 1983, pp. 756-762*

[ILPR81] Itai, A., Lipton, R.J., Papadimitriou, C.H., and Rodeh, M., "Covering a Graph by Simple Cycles," *SIAM J. on Comp.*, Vol. 10, 1981.

[KKM85] Korach, E., Kutten, S., and Moran, S., "A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms", *4th ACM symp. on Principles Of Distributed Computing (PODC)*, Minaki, Canada, August 1985, pp. 163-174.

[K84a] Kutten, S. "Traversing Directed Graphs- an Upper and Lower Bound", internal memo.

[K84b] Kutten, S., "A Unified Approach to the Construction of Efficient Distributed Leader Finding Algorithms", Proceedings of the 3rd International IEEE Conference on Communication and Energy, Montreal, Canada, October 1984. Also Technical Report #327, Computer Science Department, Technion, Haifa, Israel, July 1984.

[K85] Kumar, D., "A Class of Termination Detection Algorithms for Distributed Computations," *5th FST-TC*, New Delhi, India, December 1985 (*Lecture Notes on Computer Science*, Springer Verlag).

[L86] Lai, T.H., "A Termination Detector for Static and Dynamic Distributed Systems with Asynchronous Non-first-in-first-out Communication," in the *Proceedings of the 13th International Colloquium on Automata, Languages and Programming*, Rennes, France, July 1986, pp. 196-205.

[M83] Misra, J., Detecting Termination of Distributed Computations Using Markers," *2nd PODC*, Montreal, Canada, August 1983.

[S83] Segall, A., "Distributed Network Protocols", *IEEE Trans. on Information Theory*,file Vol. IT-29, No. 1, January 1983, pp. 23-35.


Shay Kutten has received his B.A. (cum laude), M.Sc and D.Sc. in Computer Science from the Technion, Israel, in 1981, 1984 and 1988, respectively. From 1983 he has been a teaching assistant, and later an acting lecturer in the department of Computer Science in the Technion. Now he is a Post-Doctorant in IBM T.J. Watson Research Center, NY, USA. His area of interest includes algorithms (especially distributed algorithms) and communication.

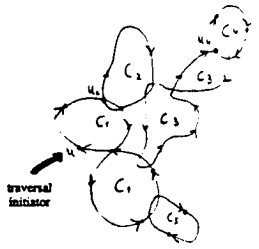


Fig. 2a
The original graph

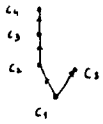

Fig. 2b
The cycles' tree

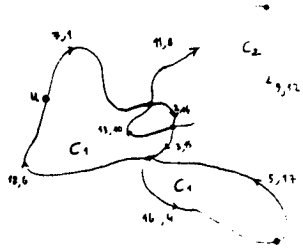

Fig. 3
The traversal starts in node *u*

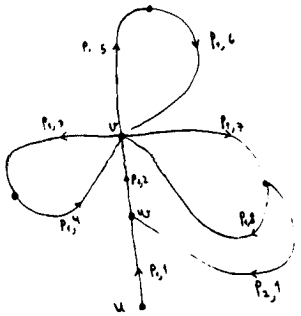

Fig. 6

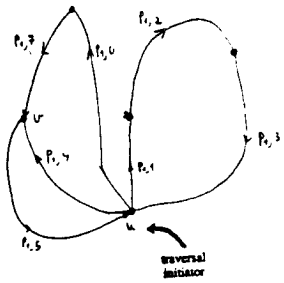


Fig. 7

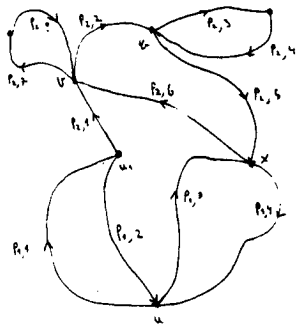

Fig. 8

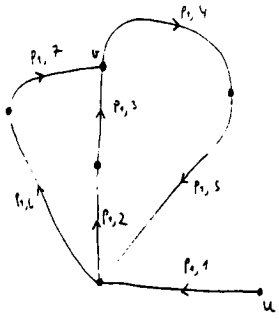

Fig. 9

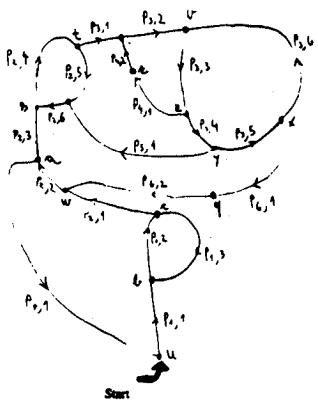


Fig. 10