STEPWISE CONSTRUCTION OF AN EFFICIENT DISTRIBUTED TRAVERSING ALGORITHM FOR GENERAL STRONGLY CONNECTED DIRECTED GRAPHS (or: TRAVERSING ONE WAY STREETS WITH NO MAPS)

by

Shay Kutten

Technical Report #431
August 1986
STEPWISE CONSTRUCTION OF AN EFFICIENT DISTRIBUTED TRAVERSING ALGORITHM

FOR GENERAL STRONGLY-CONNECTED DIRECTED GRAPHS

or: Traversing One Way Streets with no Map

(Extended Abstract)

by

Shay Kutten

Technion - Israel Institute of Technology.

ABSTRACT

We study the problem of distributively traversing directed networks, and constructing directed spanning trees. New tight lower and upper bounds on the message complexity are given. Using the algorithm we present, better solutions for problems in directed graphs can be achieved. Among others, a leader election algorithm, which is better than existing ones for a very large family of graphs, is achieved. Also, the message complexity of performing a broadcast, either with or without acknowledgements, is reduced. Such an algorithm is also needed in radio networks in order to coordinate transmissions. The traversing algorithm is constructed 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 only unidirectional links. 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 id known only to itself. 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. The token is allowed to carry a number of bits which is $O(\log \text{maximum id})$. The complexity measure is the number of messages required.

This problem is a generalization of the traversing problem solved by the very famous Depth First Search (see e.g. [E79]). The latter cannot be used here, as it retreats against the direction of edges. 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 [ILPR81].

Such a traversing 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 traversing, 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, GAB4]. This motivates the design of efficient traversing algorithms for this family.

The traversing 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 traversing). Also, any traversing algorithm can be used to construct two spanning trees rooted at the traversing initiator. One is a tree of pathes from the root to all nodes, the other is a (different) tree of pathes from all nodes to the root [KM85]. 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 $O(nm)$.

A path via all edges, or all nodes, (such as the path constructed by a traversing algorithm) is used in several papers (but not constructed) to detect termination [CM85, M83,
Another use is to coordinate nodes transmissions in radio networks [GPB82]. Directed edges there arise from differences in nodes' transmission radiiuses [CKB85]. Also, solutions of problems in undirected graphs, can easily be adapted to directed graphs, once a method to acknowledge is established (e.g. the algorithm of [FSB85]).

The message complexity of the traversing algorithms suggested so far is $O(n \cdot m)$ [GA84, KB4]. 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 than $O(n \cdot m)$ messages. In fact, every such graph is a 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 $O(m + n \cdot x)$. For some graphs $x=O(m)$, but in large families of graphs $x$ is very small. For example in directed Euler graphs $x=0$. It is conjectured that the probability of $x$ to be close to $m$, is very small.

Next we show that the above lower bound is tight, by introducing an algorithm which traverses any strongly connected directed graph in $O(m + n \cdot x)$ messages. 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. Due to space limitations, we omit from this extended abstract the formal description.

In the second section we present the lower bound. The third section is dedicated to the development of the algorithm. The fourth section analyzes the algorithm's complexity. Finally, open problems are given in the fifth 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 1: For every possible combination of $n,m,x$ there exists a graph which cannot be traversed using less than $m + n \cdot x$ messages.
Note that $z \leq 2m$. The proofs is deferred to the full paper.

3. A STEP BY STEP DEVELOPMENT OF THE ALGORITHM

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 that the arguments for the correctness of the previous step need only slight changes.

3.1. Traversing Directed Rings

Initially all edges are marked unused. Starting from some node, $u$, the token traverses unused edges, making them used. This process continues until the token is in a node from which emanates no unused edge. Clearly at that time the token is in node $u$ and the traversing has been completed.

If, however, it is not known whether this is a ring or a more complex graph, this can be checked by retraversing the cycle. If during the retraversal no unused edge was encountered, then the traversal has really ended. The algorithm follows.

Algorithm 0 (for the token):

Traverse a cycle (i.e. traverse until there are no unused edges).
Retraverse the cycle.
Whenever during retraversal you encounter an unused edge mark yourself (the token) "uncomplete".

The correctness proof is trivial. However, in the full paper we give some arguments to show the correctness, in order to generalize them after the first change in the algorithm.

3.2. Traversing Directed Euler Graphs

An Euler graph consists of one or more edges disjoint cycle. When there are more than one cycles, 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.)

Consider a cycle, \(C_3\), which is a a leaf in the cycles tree. Assume that the traversal of cycle \(C_3\) had started in node \(u_3\), during the retraversal of some cycle, \(C_4\). Clearly the retraversal of cycle \(C_3\) is never postponed and ends in node \(u_3\). Thus the retraversal of cycle \(C_4\), which had been postponed in node \(u_3\), 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 the retraversal of the first cycle is completed, the retraversal of all the graph is completed. To summarise:

**Algorithm 1 (for the token):**

- Traverse a cycle starting from some edge.
- Retrace the last cycle
  - Whenever during retraversal you encounter an unused edge do
    - postpone the retraversing.
    - Activate the algorithm recursively for the unused edge.
    - When the recursive activation terminates, resume the postponed retraversing.

**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 and the postponed retraversal of its immediate ancestor can be resumed.

**Argument 4:** A cycle, all of which descendant are retraversed, its retraversal can also be completed.

**Argument 5:** When the retraversal of a cycle ends, the token is again in the node from which
the traversal of that cycle has started.

**Argument 6:** The retraversal of descendant cycles will always be completed before the retraversal of their ancestors will be completed.

**Argument 7:** When the retraversal of a cycle ends, all its descendants are retraversed.

**Argument 8:** Every cycle which is traversed will be retraversed.

**Argument 9:** No unused edge emanates from a node on a fully retraversed cycle.

### 3.3. Using Pathes Instead of Cycles

The first step is to replace the cycles by general simple pathes (simple in edges). Clearly arguments 1 and 5 stop to hold. 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 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 pathes, 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 algorithm itself has now the form:

**Algorithm 2:**

1. **Traverse a path starting from some edge.**
2. **Return to the last traversed path's starting node.**
3. **Retraverse the last path.**
   - **Whenever during retraversal you encounter an unused edge do**
     - **postpone the retraversing.**
     - **Activate the algorithm recursively for the unused edge.**
   - **When the recursive activation terminates resume the postponed retraversing.**
4. **Return to the last retraversed path's starting node.**

### 3.4. Partitioning a Path to several Pathes

Coming to implement algorithm 2, we note the following. When a traversing 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.

Of course we can regard the part of the path from node $v$ on, as a path by itself. This
motivates the following change in the algorithm. While traversing unused edges, the token will
be permitted to end a path even if the path can be continued. If at that time the token is in a
node from which emanates an unused edge, then the token can start traversing another path
although the first path has not been retraversed yet. The second path is also considered to be
a descendant of the first, although it was discovered during traversing of the first, not during
retraversing. Several such pathes may be traversed till the token arrives at some node $v$
from which emanates no unused edge. Let us term the concatenation of the above pathes a
multipath. At that time a closing path must be found to the starting of the last path, to enable
its retraversing. The rest of the algorithm remains unchanged in the sense that when the
retraversing of a descendant path ends, the token returns to the path's starting node, which
is also a node on the path's immediate ancestor. If the ancestor belongs to the same mul-
tipath, then a closing path to its starting node must be found, in order to enable its retravers-
sal. Otherwise the retraversal of the ancestor has been postponed in that node, and can be
now be resumed. Clearly no change is needed in the Arguments. The algorithm is now:

**Algorithm 3:**

*Traverse a multipath (i.e. several consecutive pathes) starting from some edge.*
*While not the whole multipath is retraversed*
  *Return to the starting node of the last traversed but not yet retraversed path.*
  *Reraverse that last path.*
  *Whenever during retraversal you encounter an unused edge do*
    *postpone the retraversing.*
    *Activate the algorithm recursively for the unused edge.*
    *When the recursive activation terminates resume the retraversing.*
  *Return to the last retraversed path's starting node.*

Note that we do not require that the token knows when it ends the traversing of one path,
and starts the traversing of the next path of the same multipath. The multipath will be parti-
tioned into pathes only later, when the closing pathes will be found. This change also does not
change the correctness of the algorithm, as the information about the multipath's partition-
ing was not used before the time of the retraversal.
3.5. How to Collect Information in order to Construct Closing Pathes

Another (minor) change is now introduced. Assume the token has just now finished retraversing some path, \( P_1 \). This retraversal may have been postponed to traverse new multipathes, some edges of which may participate in the next closing path. The information on "where backward does a multipath lead?" can be accumulated while dealing with this multipath, say multipath \( M_2 \). However, when resuming the retraversal of \( P_1 \), the token can leave this information in the node. This is because the token is now in the node on \( P_1 \) from which the traversal of \( M_2 \) has started (and ended, using a closing path). This is done to make room in the token for information regarding the next multipath. Thus we change the algorithm s.t. each retraversal is followed by a re-retraversal. This third pass on every path is used to accumulate the information left, in order to use it to construct the next closing path. Clearly this change does not affect the correctness of the algorithm. The algorithm now becomes:

**Algorithm 4:**

1. **Traverse a multipath (i.e. several consecutive pathes) starting from some edge.**
2. **While not the whole multipath is retraversed**
   1. **Return to the starting node of the last traversed but not yet retraversed path.**
   2. **Retraverse this path.**
   3. **Whenever during retraversal you encounter an unused edge do**
      1. **postpone the retraversing.**
      2. **Activate the algorithm recursively for the unused edge.**
      3. **When the recursive activation terminates resume the retraversing.**
      4. **Return to the last retraversed path’s starting node.**
   4. **Re-retraverse the last retraversed path.**
   5. **Return to the starting node of the last re-retraversed path.**

In order to be able to tell "where backward does a path lead," so that closing pathes can be constructed, we number the edges as follows. The token carries a **multipath count** which is actually the level in the recursion. When the algorithm starts, the multipath counter is set to one. Every edge which becomes used is marked by this number. When the procedure is be applied recursively multipath count is increased. The token also carries an **edge count** used to mark each edge by its sequential number within the multipath. All the used edges of the graph are thus totally ordered lexicographically according to (multipath no., edge no.). Hence forward whenever we talk about an edge being low, lower, high, higher, etc. we refer to this order. Also induced by the edges order are orders between any two multipathes, or pathes and the order of the nodes within a path or, a multipath. Due to space limitation we
omit here (and hence forward) the new quasi formal version of the algorithm.

3.6. Closing the Last Path of Each Multipath

Let node \( v \) be the node in which the traversing of multipath \( M_1 \) ends. As no part of \( M_1 \) has already been retraversed, (i.e., no closing path information has been accumulated), we define the first closing path of \( M_1 \) to be empty. That is, if none of the edges emanating from node \( v \) belongs to \( M_1 \), then \( P_{\text{last}}(M_1) \), the last path of \( M_1 \), is empty. Otherwise the lowest edge which emanates from node \( v \) and belongs to \( M_1 \), is taken to be the first edge of \( P_{\text{last}}(M_1) \). In both cases, the token is in the first (and last) node on \( P_{\text{last}}(M_1) \), and no closing path is needed.

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

When the traversal of \( P_{\text{last}}(M_1) \) is finished the token is again in the first node on the path. The re-retraversing of the path can thus start. As in the Eulerian case we deal first with the case where no new path has been discovered during the retraversal of path \( P_{\text{last}}(M_1) \). 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 \( P_{\text{last}}(M_1) \) then this route consists of edges in increasing numbering, ending in some edge \( e_{\text{elevator}} \) 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. Also let us call \( e_{\text{elevator}} \) the elevator entree of node \( v \) at that time. Otherwise the route from node \( v \) to node \( u \) may start with edges on \( P_{\text{last}}(M_1) \) but it must proceed with an edge not on \( P_{\text{last}}(M_1) \). By argument 9 this edge is already used. By the assumption that no new multipathes were discovered from \( P_{\text{last}}(M_1) \), this edge must belong to a former path, and thus must be lower than any edge on path \( P_{\text{last}}(M_1) \). 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 \( P_{\text{last}}(M_1) \). The elevator entree is defined as the highest edge entering the node from which the low point emanates. The sole role 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 edge.

Now let us explain how to close the path which ends in node \( v \), (in which \( P_{\text{last}}(M_1) \) 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). Now the token is in the node from which the low point emanates. If the low point belongs to the same multipath, then we define it to be the first edge of the next path to be traversed. The closing path in this case was the route which ended with the elevator entree. Else the low point is lower than the first edge of the current multipath, and the token is routed 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 multipath.

3.8. Closing Other Paths

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

The routing of the token to the elevator entree is done as in a leaf path. Again, if the low point is lower then the first edge of the current multipath then the token is routed to the first edge of the current multipath. 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 multipath $M_1$, having retraversed the path which starts from node $u$, the retraversing of the whole multipath is
finished. If multipath \( M_1 \) was not the first multipath then when multipath \( M_1 \) was discovered, the retraversal of a path of some lower multipath \( M_2 \) was postponed. This retraversal is now resumed. (However, first the values of the low point and the elevator entree are stored in node \( u \).) If, however, \( M_1 \) was the first multipath then the algorithm is terminated.

4. COMPLEXITY ANALYSIS

The traversing, retraversing and re-retraversing are all done once per an edge. This yields \( 3m \) messages. The other messages are used for closing cycles.

Lemma: The union of the closing pathes of one multipath, passes each node at most twice.

The proof of the lemma is quite long and complicated, and thus it is deferred to the full paper. Let a multipath \( M \) end in edge \( e \) which enters node \( v \). Note that if closing pathes are needed for multipath \( M \); this is because \( e \) cannot be matched with an edge which emanates from node \( v \). The number of such edges is \( \frac{x}{2} \). Each cycle must be closed three times (for traversing, retraversing and re-retraversing). Thus the message complexity of the algorithm is bounded from above by \( 3(m + n x) \). (The constant factor can be improved by avoiding the re-retraversal phase.) Consider the family of strongly connected directed graphs, in which \( x = o\left(\frac{m}{\log n}\right) \). Using the results of [KKM85] a leader can be found in every such graph in \( o(m n) \), which is better than \( O(n m) \), the complexity of the known algorithms. It is conjectured that the probability of a graph to have a deficit, \( x \), which is not smaller than \( \frac{m}{\log n} \), is small.

5. OPEN PROBLEMS

It may be interesting to prove our conjecture from the previous section. Another question is whether or not can any directed tree be constructed using parallelism, with less messages than in our serial approach. (It is required that the node from which an edge emanates, will know whether or not this edge belongs to the tree). Yet another task is to perform (when spanning trees are not known) a broadcast s.t. the broadcasting node will know when all the other nodes have already received its message. We have matched the lower bound for doing it serially, but perhaps it can be performed more economically using parallelism.
REFERENCES


