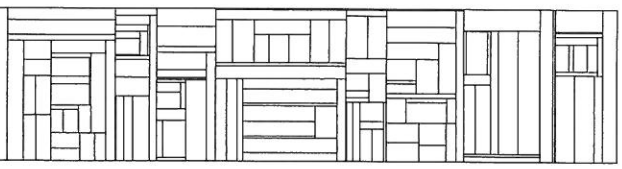
ISSN 0105-8517

A Distributed Election and Spanning Tree Algorithm Based on Depth First Search Traversals

Sven Skyum

DAIMI PB – 232
August 1987

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 6 12 83 55 *Telex: 64767 aausci dk*



A Distributed Election and Spanning Tree Algorithm Based on Depth First Search Traversals

Abstract

The existence of an effective distributed traversal algorithm for a class of graphs has proven useful in connection with election problems for those classes. In this paper we show how a general traversal algorithm, such as depth first search, can be turned into an effective election algorithm using modular techniques. The presented method also constructs a spanning tree for the graph.

1. Introduction

The problems of election and constructing spanning trees distributively have been dealt with in several papers. In general the problems require $\Theta(E+N\log N)$ messages to be sent ([2, 3, 5]). The best known upper bound is $2E+3N\log N$ ([4]) for general networks. Better bounds are known for e.g. rings, meshes and complete networks ([5, 6, 7, 8]). Spanning tree algorithms have mostly been of the "Gallager" type. That is, fragments (or kingdoms) examined and conquered by different kings are combined into larger fragments ending up with only one fragment, which has a built-in structure of a spanning tree. Korach, Kutten and Moran ([5]) introduced an elegant and simple approach to the election problem, namely constructing election algorithms from effective traversal algorithms using a modular technique. Given a class of graphs such that each graph can be traversed using O(f(N)) messages (f a convex function), an election algorithm using O(f(N)logN) messages was constructed. Attiya ([1]) refined the technique and obtained the bound $O(\Sigma_k(f(N/k))$. Both constructions yield non optimal algorithms for graphs, where only O(E) or $O(N^2)$ traversal algorithms are known. The purpose of this paper is to present a general algorithm using modular technique. Depth first search is used as the underlying O(E) traversal algorithm. By extending the technique, introducing *pruning*, we obtain an election and spanning tree algorithm using only $2E+2N\log N+O(N)$ messages of small size (O(log(maxid), where maxid is the maximal identity of a node).

The algorithm is based on the commonly used model. We consider an undirected connected graph G = (V(G),E(G)) without selfloops. N is the number of nodes and E is the number of edges. Each node corresponds to a processor with a unique identity, and each edge corresponds to a two way channel. Each channel has (input) buffers at either endpoints organized as queues. Processors can send and receive

messages via channels. The computation as well as communication is asyncronous. Messages may be arbitrarily but finitely delayed. Initially all processors are in a sleeping state. They wake up spontaneously after a finite time and start execution of the algorithm.

The algorithm is presented in Section 2. Section 3 contains the proof of correctness while Section 4 contains the analysis of the algorithm.

2. Description of the algorithm

Outline of the algorithm

When a processor v wakes up, it will typically initiate its computation by creating a token <*,0,v> at level 0 and let this token start a depth first search traversal. The first coordinate in the token indicated with an * contains information about whether the token is moving forward, backward or chasing another token (see below). Each time a token makes a backward step along an edge in the depth first search traversal (backtracking) it will either close the edge (a back edge) or declare it a tree edge, and the edge should not be traversed any more - the graph is **pruned**. If a token <*,L,u> enters a processor, which has already been traversed by a token at a higher level, the token <*,L,u> disappears. If it enters a processor traversed by a token <*,L,w> at the same level then if u < w the token <*,L,u> stops in v and waits for another token to catch up with it. This is done by changing the status of processor v to be a candidate at level L. If on the other hand u > w, <*,L,u> will stop its traversal and chase <*,L,w> as long as it cannot be detected that this token is already being chased by a third token at level L in which case <*,L,u> will stop again and change the status of the processor where it stops to a candidate at level L. If two tokens at the same level L meet, that is if a token enters a processor being a candidate at the same level, a new token at level L+1 is created and will start a depth first traversal in the pruned graph, while the two tokens at level L will disappear. If a token successfully finishes its traversal, it can declare the processor as leader where it stops and the tree edges will then form a spanning tree for G.

A formal description of the algorithm

Messages sent will be **tokens** of the form **<M,L,v>**, where M is the mode of the token. M can be either **F** meaning that the token is sent forward in the traversal, **B** meaning that the token is sent back along a tree edge, **Cl** meaning that the token is sent back along a back edge, or **Ch** meaning that the token is chasing another token.

L is the level of the token. L will be an integer in [0,logN]. v is the identity of the node which initiated the depth first search traversal of the token <*,L,v>, by which we mean all tokens of the form <M,L,v>, where M belongs to {F, B, Cl, CH}. Tokens <*,L,v> are ordered lexicographically by (L,v) only. The two different meanings of *token* should hopefully not create any confusions.

During computation a node v classifies locally its adjacent edges as **open**, **closed** or **tree** edges (O(v), C(v), and T(v) resp.).

The **status** of a node v will keep a small amount of the history of v. Only O(log(maxid)) bits are required for the status information in each node, where maxid is the maximal identification of a node.
The status is a five-tuple **(state,level,nodeid,inedge,outedge)**. state can be one of **IDLE, INIT, TRAV, CH, CAND** or **TERM**, level is an integer in [-1,logN], nodeid is a node identification, while inedge and outedge are edges adjacent to v. The various histories bear the following meanings:

(IDLE,-1,v,e,e): v has not started its computation yet. e is not used for any purposes. Initially all nodes have this status.
(INIT,L,v,e,e): The last action of v was to initiate a depth first search traversal at level L by sending a token <F,L,v> along an adjacent edge e.
(TRAV,L,u,e_1,e_2): The last action of v was to pass on token <F,L,u> along e_2. <F,L,u> was received on e_1.
(CH,L,u,e_1,e_2): The last action of v was to send a token <Ch,L,w> along e_2 chasing the token <*,L,u>. v will have had the status (INIT,L,u,e_1,e_2) where u = v and $e_1 = e_2$ or (TRAV,L,u,e_1,e_2) preceding the present status.
(CAND,L,u,e_1,e_2): v has with the status (IT,L,u,e_1,e_2) (IT being either INIT or TRAV) received a token <F,L,w> with w < u, or with status (CH,L,u,e_1,e_2) received a token <F,L,w>. In both cases token <*,L,w> was annihilated by v.
(TERM,L,u,e_1,e_2): v has no more open adjacent edges. e_2 is a tree-edge along which the last token was sent. If u = v, node v is the leader of the network and the set of tree edges forms a spanning tree.

The initial status for a node v is (IDLE,-1,v,e,e), O(v) equals the set of adjacent edges and
T(v) = C(v) = Ø. After a finite time node v initiates its computation by reading an input (if it exists) from an input buffer, or by sending <F,0,v> along one of its edges e in O(v) and enters state (INIT,0,v,e,e).

Thereafter nodes repeatedly check their input buffers for input and act upon them according to the following rules.

If the status of v is status(v) and v receives a token <M,L,u> along e (<M,L,u> is read from input buffer e by v) then one of the following 13 statements is executed as an atomic step by v:

if status(v) = (*,L_1,*,*,*) & L<L_1 then

1: noop;

if status(v) = (*,L_1,*,*,*) & L>L_1 & e in C(v) $\cup$ T(v) then

2: if e in C(v) then send <Cl,L,u> along e else send <B,L,u> along e;

if status(v) = (*,L_1,*,*,*) & L>L_1 & e not in C(v) $\cup$ T(v) & e_1 in O(v)-{e} then

3: send <M,L,u> along e_1; status(v):=(TRAV,L,u,e,e_1);

if status(v) = (*,L_1,*,*,*) & L>L_1 & O(v)={e} then

4: send <B,L,u> along e; O(v):=O(v)-{e}; T(v):=T(v)+{e};
status(v):=(TERM,L,u,e,e);

if status(v) = (IT,L,w,e_1,e_2) & IT in {INIT,TRAV} & u<w then

5: status(v):=(CAND,L,u,e_1,e_2)

if status(v) = (IT,L,w,e_1,e_2) & IT in {INIT,TRAV} & u>w then

6: send <Ch,L,u> along e_2; status(v):=(CH,L,w,e_1,e_2);

if status(v) = (IT,L,u,e_1,e_2) & IT in {INIT,TRAV} & e$\neq e_2$ then

7: send <Cl,L,u> along e; O(v):=O(v)-{e}; C(v):=C(v)+{e};

if status(v) = (IT,L,u,e_1,e) & IT in {INIT,TRAV} & e_2 in O(v)-{e_1,e} then

8: send <F,L,u> along e_2; O(v):=O(v)-{e}; if M = Cl then C(v):=C(v)+{e} else
T(v):=T(v)+{e}; status(v):=(TRAV,L,u,e_1,e_2);

if status(v) = (INIT,L,u,e,e) & O(v)={e} then

9: O(v):=O(v)-{e}; T(v):=T(v)+{e};
{v is the leader and the algorithm has terminated}

if status(v) = (TRAV,L,u,e_1,e) & O(v)={e_1,e} then

10: send <B,L,u> along e_1; O(v):=O(v)-{e,e_1}; T(v):=T(v)+{e,e_1};
status(v):=(TERM,L,u,e,e_1);

if status(v) = (CH,L,w,e_1,e_2) then
11: status(v):=(CAND,L,u,e_1,e_2);

if status(v) = (CAND,L,w,e_1,e_2) & e_3 in O(v) then
12: send <F,L+1,v> along e_3; status(v):=(INIT,L+1,v,e_3,e_3);

if status(v) = (TERM,L_1,w,e_1,e) then
13: send <B,L,u> along e;

3. Correctness of the algorithm

An edge e = {v,w} is called **nonclosed** if it belongs to (O(v) $\cup$ T(v)) $\cap$ (O(w) $\cup$ T(w)) and e is called **open** if it belongs to O(v) $\cap$ O(w). Finally e is called a **tree-edge** if it belongs to T(v) $\cap$ T(w). A path is called a **nonclosed path** if it consists of nonclosed edges. A path is called an **open path** if it consists of open edges.

Without loss of generality we may assume the time to be discrete 0,1,... .

Lemma 3.1
If a node v receives a token <B,*,*> from w along {w,v} then all nonclosed paths from w to nodes with (locally) open adjacent edges contain {w,v} as the first edge.

Proof
Follows using structural induction.

Lemma 3.2
No cycles of tree-edges are formed during a computation.

Proof
Follows by Lemma 3.1 and the properties of depth first search.

Lemma 3.3
If a node v on reception of the token <F,L,u> from a node w along e = {w,v} executes statement 7 above, then there is an open path from w to v not containing e and passing nodes with status of the form (*,L_1,x,*,*), where either (L_1,x) equals (L,u) or $L_1 > L$.

Proof
Since e is an open edge when v receives <F,L,u>, it follows by Lemma 3.1 that no edges on the path from v to w traversed by <F,L,u> has been made tree-edges. That closure of edges along that path does not affect the validity of the Lemma follows by induction primarily on increasing time and secondarily on decreasing token size:
The first time statement 7 is executed is the first time any edge is closed. For the case in which a token of maximal size is received, the Lemma is obviously valid. Assume now that node v receives token <F,L,u> at time t and that the Lemma holds true for time instances less than t and for nodes y executing statement 7 at time t on reception of tokens <F,L',z>, where (L',z) > (L,u). If any nodes on the path from w to v traversed by <F,L,u> have been visited by other tokens changing the status of these nodes then the level has increased or (L,u) is still part of the status (at time t). In the case where an edge {a,b} on the path has been closed, it is necessarily closed by a token <M,L',x>, where L' > L. The situation is shown on Figure 1. Since by assumption the Lemma is true for node a in that case, there is an open path from a to b via nodes with status of the form (*,L_1,*,*,*), where $L_1 \geq L' > L$, and the Lemma follows.

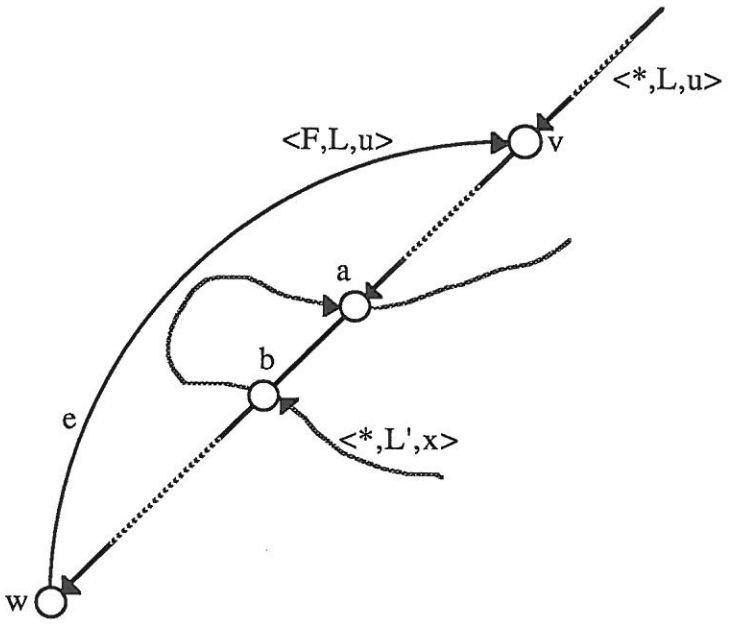


Figure 1 Crossing of traversals

Corollary 3.4
During a computation the graphs consisting of the nodes V(G) together with the nonclosed edges are all connected.

Corollary 3.5
During a computation the graphs consisting of the nodes with (locally) open adjacent edges together with the open edges (({v | O(v) ≠ Ø}, {{v,w} | {v,w} in O(v) ∩ O(w) })) are all connected.

Proof
Follows by Lemmas 3.1 and 3.3.

Lemma 3.6
If during a computation a token <Ch,L,*> is formed, then at least one token <*,L+1,*> is formed during the same computation.

Proof
For fixed L, let u be the minimal identity for which a token <Ch,L,u> is created during a computation. That token is chasing a token <*,L,v>, where v < u. Because of the minimality of u, the token <*,L,v> cannot start chasing a third token. Thus <Ch,L,u> will either catch up with <*,L,v> in which case a token <F,L+1,*> is formed or be annihilated by a node with a higher level in its status. In both cases a token <*,L+1,*> will be formed during the computation.

Lemma 3.7
During a computation a token <*,L,u> either finishes its traversal by returning to node u, which then executes statement 9 or a token at level L+1 is created during the same computation.

Proof
If a token <*,L,u> created during a computation does not finishes its traversal it stops traversing because it meets a node v, which has been visited by a token <*,L',w> at the same or higher level. If L' > L or the state of v is CAND, we are done. If L' = L and u > w then a token <Ch,L,u> is formed. If L' = L and u < w then the status of v becomes (CAND,L,u,e_1,e_2) for some e_1,e_2. In that case the token <*,L,w> cannot finishes it traversal because it before or later will return to v, in which case a token <*,L+1,v> is created. Now repeating the argument for the token <*,L,w> we will eventually end up with either a token at a higher level or a token of the the form <Ch,L,*>. Thus by Lemma 3.6 the Lemma follows.

Theorem 3.8
Exactly one token <*,L,v> will finishes its traversal. At that time no nodes will have open adjacent edges, the set of tree-edges form a spanning tree for the graph and v is the leader.

Proof
Each time a token <*,L,u> is formed for L > 0, at least two tokens of the form <*,L-1,*> have disappeared. Therefore by Lemma 3.7, at least one token <*,L,v> will finishes its traversal by returning to node v, which then executes statement 9. At that time no more edges are open (at either end) by Lemma 3.1, Corollary 3.4 and the fact $O(v) \neq \emptyset$ when v received <*,L,v>. When there are no more open edges, no more tokens can exists either, so statement 9 can only be executed once during a computation. Thus v can declare itself a leader. That the set of tree-edges forms a spanning tree finally follows by Corollaries 3.2 and 3.4.

4. Analysis of the algorithm

Lemma 4.1
The total number of different tokens <*,L,v> created during a computation is bounded by 2N-1, and L is bounded by logN.

Proof
Follows again from the fact that each time a token <*,L,v> is formed for L > 0, at least two tokens of the form <*,L-1,*> have disappeared.

Lemma 4.2
The total number of chasing tokens <Ch,*,*> sent during a computation is bounded by NlogN.

Proof
Each node can send at most one token <Ch,L,*> for each L.

Call a step of a token sent to a node v along e, where e is in $C(v) \cup T(v)$ on reception, and the step of immediately sending the token back along e from v (statements 2 and 13), for **bad** steps. Call the remaining steps for **good** steps.

Lemma 4.3
The total number of good steps, where tokens of the form <B,*,*> or <Cl,*,*> are being sent is bounded by E.

Proof
Clear since exactly one such token is sent along each edge.

Lemma 4.4
The total number of good steps, where tokens of the form <F,*,*> sent, is bounded by
E + NlogN.

Proof
For fixed L, the number of good steps where tokens <F,L,*> are sent by a node v, is at most one more than the number of good steps after which v receives tokens <B,L,*> or <Cl,L,*>.

Lemma 4.5
The total number of bad steps taken during a computation is O(N).

Proof
Bad steps are taken in connection with statements 2 and 13. The two cases are more or less identical. Figure 2 shows a situation, where bad steps occur in connection with statement 13. Assume that at times t_i ($t_0 < t_1 < .... < t_k$) tokens $<F,L_i,v_i>$ are sent from w to v and that at times s_i ($s_0 < s_1 < .... < s_k$) tokens $<B,L_i,v_i>$ are received by w from v. Assume furthermore that $t_k < s_0$. For i = 0 the two steps taken by $<*,L_0,v_0>$ are good steps while the remaining 2k-2 steps are bad. By observing,

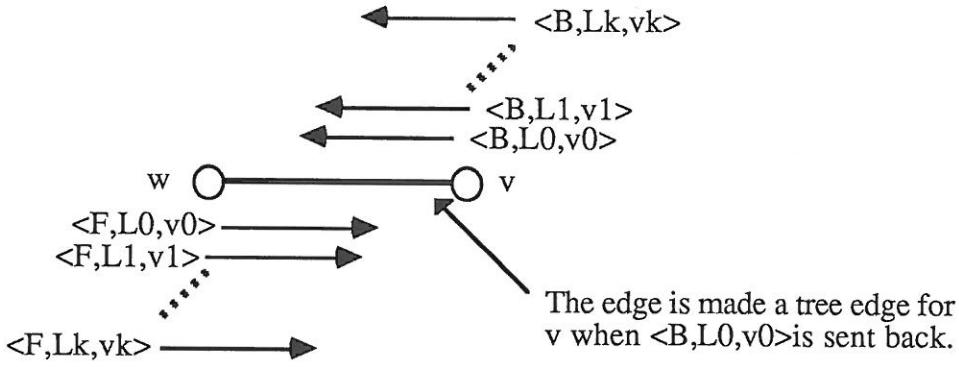


Figure 2. Example of bad steps

that for i < k the above mentioned steps are the two final steps of $<*,L_i,v_i>$'s traversal, the Lemma follows since the number of different tokens are bounded by 2N-1 (Lemma 4.1).

Theorem 4.6
The message complexity of the algorithm is 2E+2NlogN+O(N).

Proof
Lemmas 4.1 through 4.5.

Acknowledgement The author would like thank his colleaque Erik Meineche Schmidt for valuable comments and fruitful discussions.

References

[1] Attiya H.: Constructing Efficient Election Algorithms from Efficient Traversal Algorithms. Proc. of the 2nd International Workshop on Distributed Computing, Amsterdam 1987.

[2] Gallager R. G.: Finding a Leader in a Network with O(E+nlogn) Messages, internal memorandum, MIT.

[3] Gallager R. G., Humblet P. A., Spira P. M.: A Distributed Algorithm for Minimum-Weight Spanning Trees. ACM Trans. on Programming Languages and Systems. Vol. 5. 1983, pp. 66-77.

[4] Johansen K. E., Jørgensen U. L., Nielsen S. H., Nielsen S. E., Skyum S.: A Distributed Spanning Tree Algorithm. Proc. of the 2nd International Workshop on Distributed Computing, Amsterdam 1987.

[5] Korach E., Kutten S., Moran S.: A modular technique for the Design of Efficient Distributed Leader Finding Algorithms, Proc. of 4. Ann. ACM PODC 1985, pp 163-174.

[6] Korach E., Moran S., Zaks S.: Tight Lower and Upper Bounds for some Distributed Algorithms for a Complete Network of Processors, Proc. of 3. Ann ACM PODC 1984 pp. 199-207.

[7] Moran S., Shalom M., Zaks S.: A 1.44...NlogN Algorithm for Distributed Leader Finding in Bidirectional Rings of Processors. TECHNION Technical Report #389, November 1985.

[8] Peterson G. L.: Efficient Algorithms for Election in Meshes and Complete Networks, TR-140, University of Rochester, August 1984.

[9] Santoro N.: On the Message Complexity of Distributed Problems. Int. Journal of Comp. and Inf. Sci. 13. 1984, pp. 131-147.