ALGORITHM FOR DISTRIBUTED SPANNING TREE CONSTRUCTION IN DYNAMIC NETWORKS

by

E. Korach and M. Markowitz

Technical Report #401

February 1986
Algorithms for Distributed Spanning Tree Construction in Dynamic Networks

(Extended Abstract)

by

Ephraim Korach and Moshe Markowitz

Computer Science Department
Technion - Israel Institute of Technology
Haifa, Israel 32000

ABSTRACT

The main result of this paper is an algorithm that elect a leader and construct a spanning tree in a distributed network with $n$ processors and $m$ links, where links may fail during the execution of the algorithm. This algorithm is obtained by a natural two step extension of the Korach-Moran-Zaks algorithm for leader election in a complete network.

At the first extension step, an algorithm that choses a leader (and construct a spanning tree) in a general reliable network. The complexity of this algorithm is $2m + 4n \log k$, where $k$ is the number of processor that were spontaneously awakened. (It is a slight improvement of the known $2m + 5n \log n$ spanning tree construction of Gallager-Humblet-Spira algorithms, although their algorithm constructs, in fact, a minimum weight spanning tree).

In the second extension step is an algorithm that elect a leader and construct a spanning tree in a network where edges may fail during the execution of the algorithm. The message complexity of this case is $O(tm + n \log k)$, where $t$ is the number of links that failed during the execution. This algorithm could be further extended to deal with links that may also rise up. This result is better then the $O(t(m + n \log k))$ message complexity of a naive approach to this problem.

Some other useful algorithms for different models of unreliable networks are also presented.
1. INTRODUCTION

The goal of this paper is to present an algorithm for leader election and spanning tree construction in a dynamic asynchronous network. We start with a basic model of a reliable complete network, for which an algorithm exists, and gradually extended the model side by side with its algorithm, up to a dynamic network.

Our basic model is a complete network of processors which are distinguished only by their identities. A processor has communication lines which connects it with all the other processors. Initially, the processors do not know any other processor's identity and are all in capable to perform the same algorithm. In the leader election problem an arbitrary subset of processors, awaken spontaneously at arbitrary times and start the execution of the algorithm. The algorithm may include sending and receiving messages through the communication lines. A processor that was not a starter remains asleep until it receives a message. A message reaches its destination within a finite time and with no error. After the message exchange terminates, exactly one leader is elected. The algorithm we use for this model is from [KMZ], which require at most $5n \log k + O(n)$ messages.

Now we extend our complete network model to a general connected network For this purpose we give an extension to [KMZ] algorithm which finds a leader in at most $2m + 4n \log k$ messages ($m = |E|$), where $k$ is the number of nodes that are spontaneously awaken ($1 \leq k \leq n$). The known algorithm for a spanning tree construction, given in [GHS], requires at most $2m + 5n \log^2 n$ messages. The algorithm that is presented here is an improvement by a constant factor and by replacing the log argument $n$ by $k$ (Note that if only $o(1)$ nodes starts the algorithm then, the message complexity becomes $O(m)$.) However our algorithm constructs only a spanning tree - not a minimum weighted one.

On the next step we assume that links may fail during the leader election process. A failure when happens, is observed by the processors at both ends of the link. The algorithm for a general reliable graph is extended to deal with this kind of failures. If the number of failures during the process is guaranteed to be finite, then the spanning
tree construction and leader election will terminate successfully.

In order to have the full model of dynamic network, we assume that links may also be added to network during the process. The extension need to be made on the latter version of the algorithm, although not given here, is very easy to figure out.

The technique used to extend the basic algorithm to a general graph could be generalized also for some other special classes of graph where the message complexity remains $O(n \log_2 k)$.

In particular, this technique is shown to be useful for devising an efficient adaptive algorithm. Consider a network class that originally has some kind of complete property (e.g. complete graph, complete k-partite graph), for which an $O(n \log n)$ algorithm does exist. These complete classes has a threshold condition where up to the threshold point, the network may loose links and still, leader election could be done in $O(n \log n)$. It might be that due to links failure (occured before the algorithm have started), the complete model or even the threshold condition do not hold hence, we must assume that the network is a general graph. Applying the algorithm for a general graph from the beginning, solves the problem, but will cost $O(n^2)$, even if the network eventually happens to be complete. Checking of the threshold condition and then applying the right algorithm, also requires a great communication effort and will be very inefficient (in fact, it is as hard as leader election). What we suggest, is an adaptive algorithm that its total behavior on a specific case, will be as if the special purpose algorithm for this case would have been applied from the beginning.

A different type of extension applied on [KMZ] algorithm yield an algorithm for leader electing in a complete network were up to $d$, ($d < n/2$) processors may fail before the algorithm starts. A processor that fails, stops sending massages without the awareness of its neighbors. Our algorithm uses no more then $5n \log_2 k + dk$ messages which is an improvement of [KW] result for this problem ($12 \log_2 k + 2dk$).
2. LEADER ELECTION IN A GENERAL NETWORK

2.1. The Basic Protocol Mechanism

We now present the basic mechanism used by the algorithm for electing a leader, which is essentially the [KMZ] algorithm for a complete network.

Each node in the network has a state, that is either KING or CITIZEN. Initially every node is a king (i.e. \(\text{state}(i) = \text{KING}\)), and - except for one - everyone will eventually become a citizen. A node \(i\) with \(\text{state}(i) = \text{KING}\) is called king \(i\). The algorithm starts by a \text{WAKE} message, received by any non-empty subset of nodes.

During the algorithm, each king is the root of a directed tree which is its kingdom. All the other nodes of this tree are citizens of this kingdom, and each node knows its father and sons. Each node also stores the identity \(\text{king}(i)\) and the phase \(p(i)\) of its king, which are updated during the execution of the algorithm. \(\text{status}(i) = (p(i), k(i))\) is called the status of node \(i\). We say that \(\text{status}(i) < \text{status}(j)\) if either (a) \(p(i) < p(j)\) or (b) \(p(i) = p(j)\) and \(k(i) < k(j)\). Before the algorithm starts \(k(i) = \text{identity}(i)\) and \(p(i) = -1\) for each node \(i\). The following variables are also used: \(\text{unused}(i)\) denotes the set of all unused edges of node \(i\), \(\text{father}(i)\) denotes the edge connecting \(i\) to its father, and \(\text{sons}(i)\) denotes the set of edges connecting \(i\) to its sons.

A king is trying to increase its kingdom by sending messages towards other kings (possibly through their citizens), asking them to join, together with their kingdoms, its kingdom. A citizen, upon receiving a message originated by a king, delays it, ignores it, or transfers it to (or from) its king.

When king \(i\) receives a message asking it to join the kingdom of king \(j\), it does so if \(\text{status}(i) < \text{status}(j)\). Otherwise, the message is simply ignored, and the sending king will eventually become a citizen. The process of joining \(j\)'s kingdom is combined of two stages: first king \(i\) sends a message to king \(j\) along the same path which transferred \(j\)'s message to \(i\), telling it that it is willing to join its kingdom; during this stage the directions of the edges in this path are reversed. In the second stage, if \(p(i) < p(j)\) then king \(j\) announces to its new citizens that it is their new king, or if \(p(i) = p(j)\) then it first increases its phase by 1 and then sends an appropriate updating message.
towards all its citizens (new and old).

A node $i$ in the CITIZEN state has a unique father edge leaving it (which may be changed during the algorithm) and zero or more son edges. It may be in one out of two substates: CITIZEN(regular) and CITIZEN(waiting). It enters the waiting substate upon receiving an $ASK$ message with status higher than its own status, and it returns to the regular substate upon getting a status which is higher or equal to the status of that $ASK$ message. While in the waiting substate, $i$ remembers the status $(p(j), k(j))$ of the last $ASK$ message it received, and also the edge $e'$ along which it received this message. Also, in this substate it receives messages only from its father, and delay the reception of other messages (e.g., by moving them to the end of a queue), until it enter the regular substate again (if at all). This basic protocol ends when $king_i$ knows that all its neighbors are in its kingdom. At this stage in a complete network, $king_i$ would announce itself as leader, however for general graph some additional mechanism is required.

2.2. The Additional Protocol Mechanism

The idea is to move the king in its kingdom in a DFS search. The DFS move consist of two basic steps: (1) The move forward when the king moves itself toward one of its unsearched sons. This is done after the king has communicated with all its neighbors (i.e. each of those either become its son or acknowledged of being its citizen). (2) The move backward is done when all the sons have been searched.

While moving, the king reverses the direction of the tree edge he moved on and makes its new place to be the root of the spanning tree. The equality $identity(i) = k(i)$ holds for the king only when he is in its original place. The node that the king has just left keeps in $father(i)$, the direction to the current king, and in $king\_origin(i)$, the direction of the original father edge. (Note that there are always two directed spanning trees, with the same edges, to a kingdom. The original one with the king's origin node as a root and on which the search is being done. And the second one which is directed to the current place of the king.)
The king in its new node, continues to try to increase its kingdom, by using the same basic protocol as before.

The search is completed when the king is back in its origin node and all its sons have been searched. Now the king announces himself as leader by propagating a LEADER message through the spanning tree.

In searched(i) the processor keeps the edges that king went in and has return back. This variable do not change when another king takes over the king of the same node. So once the king has return from an edge he wont go through it again. From this we can see that the elected king, in its DFS does not necessarily visit all the nodes in the network.

2.3. The Messages used by the Algorithm

(1) \textit{ASK}(\textit{status}(i)) : this message is sent by king \(i\) through an unused edge in an attempt to increase its kingdom, and might be transferred onwards by citizens. Each \textit{ASK} message has a \textit{status}, which is the status \(p(i)\cdot k(i)\) of the king that originated it (at the time it was originated).

(2) \textit{ACCEPT}(p(i)) : this message is sent by king \(i\) in return to an \textit{ASK} message from another king, telling it that it is willing to join its kingdom. (this message also might be transferred onwards by citizens.)

(3) \textit{UPDATE}(\textit{status}(i)) : this message is sent by king \(i\) (after receiving an \textit{ACCEPT} message from another king) updating its new (and in some cases also its old) citizens of its identity and phase.

(4) \textit{YOUR\_CITIZEN} : this message is returned by a citizen as an answer to \textit{ASK} message originated by its own king.

(5) \textit{MOVE} : this message is sent by a king to one of its sons in order to make a forward step of the DFS. The sender becomes a citizen and the receiver becomes a king.

(6) \textit{RETURN} : this message is sent by a king to its original father in order to make backward step of the DFS.

(7) \textit{LEADER} : this message is sent by the leader to all other nodes, announcing its leadership and terminating the algorithm.
2.4. The Algorithm for a King

Before the algorithm starts every node $i$ has the following initial values. Local variable $z$ at node $i$ is defined as $z(i)$.

- $state(i) = \text{KING}$
- $status(i) = (p(i), k(i)) = (-1, \text{id}(i))$
- $unused(i)$ is the set of all its adjacent edges,
- $sons(i) = \emptyset$
- $\text{king\_origin}(i) = \text{null}$
- $searched(i) = \emptyset$

We now give the formal description of the algorithms to be performed by node $i$ as long as it is a king.

(K0) Upon node $i$ becomes a king it performs procedure check.

Node $i$ reacts to a message $m$ it receives along an edge $e$ according to $m$'s type as follows:

(K1) $m = \text{WAKE}$: Node $i$ increases its phase to zero and performs the procedure check, given at the end of this subsection.

(K2) $m = \text{ASK}(\text{status}(j))$: If $\text{status}(j) < \text{status}(i)$ then node $i$ ignores this message. Otherwise, it sends an $\text{ACCEPT}(p(i))$ message along $e$, and changes its state to $\text{CITIZEN}(\text{waiting})$, with $\text{father}(i) = e$.

(K3) $m = \text{YOUR\_CITIZEN}$: Node $i$ performs the procedure check.

(K4) $m = \text{ACCEPT}(p(j))$: Node $i$ does the following:

1. Adds $e$ to $sons(i)$.
2. If $p(j) < p(i)$ it sends an $\text{UPDATE}(\text{status}(i))$ message along $e$. Otherwise (i.e., $p(j) = p(i)$) it increases its phase by one and sends an $\text{UPDATE}(\text{status}(i))$ message along all the edges in $sons(i)$.
3. It performs the procedure check.

Procedure check:

- If $\text{unused}(i) \neq \emptyset$ then

  - delete an edge $e'$ from $\text{unused}(i)$ and send an $\text{ASK}(p(i), k(i))$ message along $e'$.

- Else

  - If $sons(i) - searched(i) \neq \{\text{king\_origin}(i)\}$ then

    1. Choose $e \in sons(i) - searched(i) - \{\text{king\_origin}(i)\}$.
    2. Send $\text{MOVE}^\text{a}$ along edge $e$.
    3. $\text{sons}(i) = sons(i) - \{e\}$.
    4. $\text{father}(i) = e$.
    5. $\text{state}(i) = \text{CITIZEN}(\text{regular})$.

  - Else

    1. If $k(i) = \text{id}(i)$ then


send a LEADER message to all your sons, and terminate.

else
(1) Send RETURN along edge king origen(\(i\)).
(2) \(\text{father}(i) = \text{king origen}(i)\).
(3) \(\text{sons}(i) = \text{sons}(i) - \{\text{king origen}(i)\}\).
(4) \(\text{state}(i) = \text{CITIZEN}(\text{regular})\).

2.5. The Algorithm for a Citizen

Node \(i\) reacts to a message \(m\) it receives along an edge \(e\) according to \(m\)'s type and its substate, as follows:

**regular substate:**

(CR1) \(m = \text{ASK}(\text{status}(j))\) and \(e \neq \text{father}(i)\): \(i\) does the following:
1) If \(k(i) = k(j)\) and \(e\) is not a son edge, it returns \(\text{YOUR CITIZEN}\) along \(e\).
2) If \(\text{status}(j) > \text{status}(i)\) and \(k(i) \neq k(j)\), it sends \(m\) to its father and enters the waiting substate.

(CR2) \(m = \text{UPDATE}(\text{status}(j))\) and \(e = \text{father}(i)\): It updates its status to \(\text{status}(j)\); and forwards \(m\) to all its sons.

(CR3) \(m = \text{ACCEPT}(\text{p}(j))\): It makes \(e\) its son edge; and forwards an \(\text{UPDATE}(\text{status}(i))\) message along \(e\).

(CR4) \(m = \text{MOVE}\): Make \(e\) your son, set \(e\) as \(\text{king origen}(i)\) and change state to \(\text{KING}\).

(CR5) \(m = \text{RETURN}\): Make \(e\) your son, put \(e\) into \(\text{searched}(i)\) and change state to \(\text{KING}\).

(CR6) \(m = \text{LEADER}\): Send \(m\) to all your sons and terminate the algorithm.

**waiting substate:**

(Recall that: (1) \(\{\text{p}(j), k(j)\}\) is the status of the last \(\text{ASK}\) message received by \(i\), (2) \(e'\) is the edge along which this \(\text{ASK}\) message was received, and (3) in this substate \(i\) receives messages only from its father):

(CW1) \(m = \text{UPDATE}(\text{p}(k), k(k))\): \(i\) does the following:
1) Updates its status to \(\{\text{p}(k), k(k)\}\) and forwards \(m\) to all its sons.
2) If \(k(k) = k(j)\) and \(e'\) is not a tree edge, it sends a \(\text{YOUR CITIZEN}\) message along \(e'\) and returns to the regular substate.
3) If $(p(k),k(k)) \geq (p(j),k(j))$ it returns to the regular substate.

(CW2) \( m = ACCEPT(p(k)) \): Make \( e \) (which is your father) your son, make \( e' \) your father and forward \( m \) along \( e' \).

(CR3) \( m = MOVE \): Make \( e \) your son, set \( e \) as \( king_{origin}(t) \), send \( ASK(p(j),k(j)) \) to your self and change state to \( KING \).

(CR4) \( m = RETURN \): Make \( e \) your son, put \( e \) into \( searched(i) \), send \( ASK(p(j),k(j)) \) to your self and change state to \( KING \).

(CR5) \( m = LEADER \): Send \( m \) to all your sons and terminate the algorithm.

2.2 Message Complexity

In this section we will give a complexity analysis of the algorithm.

Theorem: If \( k \) nodes start the algorithm then the number of messages used by the algorithm is bounded by \( 2|E| + 4n \log_2 k + O(n) \).

Lemma: If \( k \) nodes start the algorithm and node \( i \) is announced as leader then when the algorithm terminates \( p(i) \leq \log_2 k \) holds.

Lemma: Exactly one \( RETURN \) message is sent on any edge of the final spanning tree.

Proof of the lemma is omitted.

Proof of the theorem: An upper bound for number of messages of each kind:

(1) \( LEADER \): exactly \( n - 1 \) messages.

(2) \( ASK \): Consider first ask messages send by a king. Since such \( ASK \) won't be sent twice on the same edge, except of maximum \( n - 1 \) that could be sent one against the other, we have at most \( m + n \) messages. Since a citizen transfers at most one \( ASK \) message per phase, the total number of these is bounded by \( n \log_2 k \).

(3) \( ACCEPT \): the total number of such a messages sent by a king is \( k - 1 \). At a given phase a citizen send at most one \( ACCEPT \), the total number of messages hence is bound by \( n \log_2 k \).
(4) **YOUR_CITIZEN**: since each edge of the graph can be rejected as a reply to an *ASK* message sent by a king, the total number of such message is bounded by $m$.

(5) **UPDATE**: Each citizen receives at most one such message per phase, hence the total number is bounded by $n \log_2 k$.

(6) **MOVE**: Each citizen receives at most one such message per phase, hence the total number is bounded by $n \log_2 k$.

(7) **RETURN**: By the lemma exactly $n - 1$ such message are being sent during the execution of the algorithm.

Total count of messages sent by the algorithm is no more then $2m + 4n \log_2 k + O(n)$.

3. **GENERALIZATION**

In the previous section we showed how a king, by searching its kingdom spanning tree, can cover all the graph. There are however classes of graphs where it is sufficient for the king to search only over a subset of nodes, in order to guarantee communication with all the rest.

In particularly in the cases where the size of subset needed to be searched, is a constant, we can easily devise a leader election algorithms that requires $O(n \log_2 k)$ messages. The modifications needed to be made in the original algorithm for general network are basically in the condition for a forward move (search condition) and in set of edges from which the chosen is done. In the algorithm for a general graph it was:

```
if there are unsearched sons then choose one of those.
```

In (3.1)-(3.4) we will see few examples that were discussed before in [KMZ] and in [KKM]:

3.1. **Complete Network**: This is a degenerated case were the size of the king's search is one. The search condition here is *false*. And since no MOVE or RETURN messages are being sent we have exactly the algorithm from [KMZ].
3.2. Complete K-Partite Graphs: In a $k$-partite graph the nodes are partitioned into $k$ disjoint subsets. Each node in any subset $U$ has links to all the other nodes but to those in $U$. In order for a king to dominate the graph it needs to visit in two connected nodes. The searched condition here is: 

$$\text{if } k(i) = \text{identity}(i) \text{ and } |\text{searched}| = 0 \text{ then choose one your sons.}$$

3.3. Graphs were any node with any $K$ neighbors sees the others: The search here is of size $k + 1$. The search condition is:

$$\text{if } k(i) \neq \text{identity}(i) \text{ and } |\text{searched}| < k \text{ then choose from unused adjacent edges.}$$

Here we see that the search does not necessarily follow the original edges of the spanning tree. In king’s forward and backward moving, to a node which is not its son, the used link is added to the spanning tree and the previous father of this son is disconnected, and the directed tree shape is preserved.

3.4. Graphs of Radius 1: Where at least one node is connected to all others. The king gets his neighbors degrees, (could be transferred by `YOUR_CITIZEN` and `ACCEPT` messages, as an additional parameter) and to move to the one with the maximum degree, unless he has the maximum degree.

3.5. An Adaptive Algorithm for Unreliable Complete Network

Consider a complete network, where any number of links may become faulty, before the algorithm starts and with the awareness of its both ends. The threshold point for election could still be done in $O(n \log n)$ is that at least one node in the network remains connected to all the others. Since we can’t assume that one node is fully connected, the algorithm for graph of radius 1, is not to be applied here directly. However, since $n$ assumed to be known (a processor knows its degree before some links become not connected), instead of using directly the algorithm for a general graph, (that would require $O(n^2)$ messages for this case), we will devise the following king search:

$$\text{if } k(i) \neq \text{identity}(i) \text{ and } \text{degree}(i) < n-1 \text{ then}$$

$$\text{if you have a neighbor } j \text{ degree}(j) = n-1 \text{ then choose } j's \text{ edge to move for-}$$
nowledged back, causes to another `ASK` to be sent on new unused edge.

(2) The termination condition is if the number of neighbors that answered back is \( \lceil n/2 \rceil \).

4.3. Message Complexity

**Theorem:** If \( k \) nodes starts the algorithm, then the number of messages used by the algorithm is bounded by \( 5n \log k + kd + O(n) \).

**Proof:** An upper bound for number of messages of each kind:

(1) \textit{LEADER}: exactly \( n - 1 \) messages.

(2) \textit{ASK}: First let's consider `ASK` messages send by a king. At a phase \( p \) a king can send `ASK` messages that are finally answered, no more then the size of its kingdom. And since a node can't have the same phase again, we get no more then \( n \) replied `ASK` for each phase and \( n \log k \) totally. A king, during its life time, can have at most \( d+1 \) unreplied `ASK` messages. So we count at most \( k(d+1) \) such messages. A citizen transfers at most one `ASK` message per phase, hence a total number of these is bounded by \( n \log k \).

(3) \textit{ACCEPT}: the total number of such \( A \) messages sent by a king is \( k - 1 \). At a given phase a citizen send at most one `ACCEPT`, the total number of messages is hence is bound by \( n \log k \).

(4) \textit{YOUR\_CITIZEN}: No more then the replied `ASK` messages. Hence \( n \log k \).

(5) \textit{UPDATE}: A citizen receives at most one such message per phase, hence the total number is bounded by \( n \log k \).

To conclude, the total number of messages is no more then \( 5n \log k + kd + O(n) \).

5. LEADER ELECTING IN PRESENCE OF DYNAMIC LINKS FAILURE

The model discussed here is a general network were link failure may occur at any moment during a communication process. The processor notices the failure at the moment it happens and stop sending messages on this link. A trivial leader election algorithm for this model, could be using any leader election algorithm for a reliable
network and when a node notices a failure, it broadcasts this fact all over the kingdom and all nodes start the algorithm again. This solution requires $O(t[m+n \log k])$ messages, where $t$ is the number of link failures.

The algorithm we present here do not cancel the spanning tree that was built up to the failure moment and hence requires only $O(tm+n \log k)$ messages.

5.1. The Algorithm

The algorithm based on the algorithm for a general graph from (2), with the following additions:

1. We assume that a link transmits a FAIL message to its both ends just before it becomes faulty. This message, if coming from an edge not in the tree can be ignored.

2. A citizen $i$, upon receiving this message from its father edge, delete this edge from its edges list makes him self a king, increases its phase by one and then it sends to all its new kingdom an special update message $\text{RESTART}(k(t),p(t))$.

3. A citizen $i$, upon receiving FAIL message from its son, delete this edge from its edges list and send $\text{REMOTE_FAIL}$ toward its king.

4. The king, in response to $\text{REMOTE_FAIL}$ message, increase its phase by one and broadcasts $\text{RESTART}(K(t),p(t))$ on its tree.

5. Receiving $\text{RESTART}$ message causes the node to update its status and send this message to its sons.

6. Any node involves in the above described protocol, disregards all the information about edges not in tree.

5.2. Remarks

- One of the main achievements of this algorithm is that it is in capable to maintain a spanning tree, after its construction was completed, in a faulty edges environment where a tree edge may fail even after the leader was elected. A link failure causes a new king to be born. The two kings are now trying to connect the broken tree, using other edges.
• Note, that if $tm = o(n \log k)$ then message complexity do not increase.

• If the graph become disconnected during the execution of the algorithm and has two or more connected components, then in each of these a king is elected (if it has an awaken node).

• The algorithm could be easily extended to deal with networks where links may also be added to the network.

REFERENCES


