HAL
open science

Yet another distributed election and (minimum-weight)spanning tree algorithm

Ivan Lavallee, C. Lavault

To cite this version:

Ivan Lavallee, C. Lavault. Yet another distributed election and (minimum-weight)spanning tree algorithm. RR-1024, INRIA. 1989. inria-00075534

HAL Id: inria-00075534

https://inria.hal.science/inria-00075534

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNITE DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél (1) 39 63 55 11

Rapports de Recherche

N° 1024

Programme 2 - 3

YET ANOTHER DISTRIBUTED ELECTION AND (minimum-weight) SPANNING TREE ALGORITHM

Ivan LAVALLEE
Christian LAVAULT

Avril 1989

RR_1024

YET ANOTHER DISTRIBUTED ELECTION AND (minimum-weight) SPANNING TREE ALGORITHM

UN NOUVEL ALGORITHME DISTRIBUÉ D'ÉLECTION, ET DE CONSTRUCTION D'UN ARBRE COUVRANT

(éventuellement minimal)

Ivan LAVALLÉE [1] and *Christian LAVAULT* [2]

Mots clés: Algorithme, système distribué, asynchrone, message, complexité, arbre couvrant, élection.

Key words: Algorithm, distributed system, asynchronous, message, spanning tree, leader election.

---

[1] INRIA, Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex. France.
email : lavallee@seti.inria.fr
Département d'Informatique, Université Paris VIII. France.

[2] INRIA, Domaine de Voluceau, Rocquencourt, B.P.105, 78153 Le Chesnay Cedex. France.
email : lavault@seti.inria.fr
Département d'Informatique, Université Paris XII. France.

Abstract.

A strongly improved version of Lavallée & Roucairol's distributed asynchronous algorithm for the construction of a minimum-weight spanning-tree in arbitrary networks [LAV,ROU-86] is given. The present algorithm is shown to have $\frac{1}{2}n\lg n + O(n)$ message complexity in the worst case and $O(n)$ average message complexity in complete meshed networks, and an upper bound of $2e + 7/2\ n\lg n + O(n)$ message complexity in arbitrary networks, with size of messages $O(\lg i)$.

Résumé

L'algorithme que nous proposons est une amélioration trés substantielle de l'algorithme distibué asynchrone de Lavallée et Roucairol [LAV,ROU-86] pour la construction d'une arborescence couvrante de poids minimum dans les réseaux d'interconnexion quelconques. Cet algorithme posséde, dans les réseaux à maillage complet, une complexité en messages de $\frac{1}{2}n\lg n + O(n)$ dans le pire des cas et une complexité en messages de $O(n)$ en moyenne. Dans les réseaux d'interconnexion quelconque, un bon majorant de sa complexité en messages est $2e + 7/2\ n\lg n + O(n)$. La taille maximale des messages y est $O(\lg i)$.

1. Introduction

1.1 Framework and Problem

A distributed system may be considered as a collection of n processes connected by e direct communication links. Each process has a local non-shared memory and clock, and can communicate by sending messages to and receiving messages from its neighbours. With respect to the communication behaviour, distributed systems can be classified as *asynchronous* or *synchronous* ones. In an asynchronous system, transmission and queuing delays experienced by a message along a link are finite but unpredictable. In a synchronous network, all clocks are assumed to tick simultaneously and any message sent at time t is received and processed at time $t+1$ (the communication subsystem can be further refined and distinguish also the cases of *partial synchrony* and *partial asynchrony* ; see [DWO,LYN,STO-88]). We also assume throughout the processes and the communication subsystem to be error-free, and that the links operate in a FIFO-manner.

Two closely related basic computations in a distributed environment are the **election**, and the **spanning tree construction** processes (ST).

The distributed election procedure consists in changing from an initial system configuration where every process is in the same state (say, *candidate*), to a final configuration where exactly one process is in a predefined state (say, *elected*) and all other processes are in another predetermined state (say, *defeated*). Note that, in such a *symmetry breaking* procedure, there is no a priori restriction on which process should become elected.

The distributed spanning tree construction roughly operates as the construction of a certain routing tree within a given interconnection network by choosing links which can be easily determined during the election process. The elected node-process is thus the root of the constructed spanning tree for the underlying graph of the network. These two distributed processes are at the basis of most control and coordination mechanisms employed in distributed systems (e.g. mutual exclusion,, synchronization, reset of a system after a possible failure, etc.) ; they are also closely related to other basic distributed computations (e.g. minimum finding, traversal, etc.). The election problem, and hence the ST problem, can be solved deterministically only if the each node-process has associated a *distinct* identification value, its *identity*, from a (possibly infinite) index universe I. With unique node-processes' identities, the election problem and the ST problem can be solved.

Both problems have been extensively studied in the recent literature. However, an exact average-case analysis of distributed election and/or ST algorithms in arbitrary networks remains a challenging mathematical problem

The distributed algorithm for minimum-weight spanning trees (MST) of Gallager et al. [GAL,HUM,SPI-85] is worst-case optimal (in $\Theta(n\lg n) + \Theta(e)$) with at most $5n\lg n + 2e + O(n)$ messages, and runs in at most $O(n\lg n)$ time. Lavallée & Roucairol's algorithm [LAV,ROU-86], which worst-case message complexity is $3n\lg n + 2e + O(n)$, represents only a slight improvement (with respect to the constant factor of $n\lg n$) over the latter result, whereas its $O(\lg n)$ time complexity really improves on Gallager's.

Distributed election and ST algorithms in arbitrary networks have been proposed, for example in [LAV-86], [HEL,MAD,RAY-87] and [LAV-87], which also run in at most $O(\lg n)$ time and use at most $O(n\lg n) + O(e)$ messages or $O(n^2)$ respectively. An attempt to compute the average-case message complexity in [LAV-87], involving the notion of phase of an algorithm (see subsection 3.2.1) and the diameter and the maximum degree of the underlying graph of the network, achieves an $O(n\lg n)$ average message complexity.

For MST and election algorithms, Santoro [SAN-84], Korach et al. [KOR,MOR,ZAK-84], and Pachl et al. [PAC,KOR,ROT-84] (respectively) established $\Omega(n\lg n)$ lower bounds on messages in various networks, and in [GAF,LOU,TIW,WES,ZAK-86] a worst-case $\Omega(e)$ lower bound is proved for "common knowledge" distributed algorithms.

The most recent distributed synchronous election algorithm in complete networks [CHA,CHI-88] elects a leader using $2n\lg n$ messages in the worst case, $O(n)$ messages on the average, and $2\lg n + O(1)$ time. Up until now this is the best result achieved in a particular topology for comparison election algorithms. Note that throughout, lg denotes the logarithm to the base 2.

1.2 Results

In this paper, a distributed asynchronous algorithm is presented and analysed for building a (possibly minimum-weight) ST in an asynchronous distributed system. At the same time the algorithm is electing the root of the constructed spanning tree as the leader node-process in the network. Thus, one and the same algorithm solves both the election problem and the ST construction problem. There is no central controller in the distributed system and every process only has *local* information about the network topology ; namely it a priori *only* knows its direct neighbours in the network and the size n of the network is unknown to the processes.

The present algorithm is based upon a basic specific technique introduced in a previous algorithm proposed in [LAV,ROU-86]. Yet, it represents a strong improvement over this latter algorithm since the length of messages is here $O(\lg i)$, where **i** is the identity of the root of the ST (the elected node), whereas the preceding algorithm used messages of length at most $O(n)$, while the message complexity is improved (in the worst case and on the average) and the time complexity remains equivalent *in arbitrary networks* : viz. at most $7/2\ n\lg n + 2e + O(n)$ and $O(\lg n)$, respectively. The previous algorithm presented in [LAV,ROU-86] could actually entail the expense of an *unbounded number of messages* (for a given n), within a specific worst-case configuration of the network.

Moreover, although it is designed for asynchronous arbitrary networks, this algorithm actually achieves the worst-case, average message complexity and time complexity of [CHA,CHI-88] *in complete meshed networks* : viz. $\frac{1}{2}n\lg n + O(n)$, $O(n)$, and $O(\lg n)$, respectively. The algorithm also performs quite well *on a ring-based topology*, since it uses at most $3n\lg n + O(n)$ messages and $\frac{1}{2}n\lg n + O(n)$ messages on the average, and runs in $O(\lg n)$ virtual time on rings.

2. The Algorithm

We adopt throughout the standard model of an asynchronous distributed system as described in subsection 1.1. Node-process may start executing the ST algorithm either voluntarily at any arbitrary moment or upon receiving a message which triggers their execution of the algorithm. The same algorithm is assumed to reside at all nodes.

2.1 Description of the Algorithm

The ST algorithm uses Sollin's local property [SOL-63]. It performs a progressive enlargement of the subtrees in the network by successively combining larger and larger *fragments* (that is directed subtrees), with respect to the strict total order relation existing in the universe I of the n nodes' identities. Using the strict total order binary relation between nodes which is induced by the son-to-father relation in the network, fragments may be considered as directed subtrees. Within these fragments, the node without father is called the (unique) *root* of the corresponding fragment. Thus, before the algorithm starts, each single node in itself constitutes an entire fragment of which it is of course the root.

2.1.1 The Combination of Fragments

In contrast with all other ST or MST algorithms we use a specific technique for the combination of the fragments : (*i*) a given fragment F is a *candidate for a merging process into* some other fragment G and the root of F makes a *connection request* to G (the combination process does

not work in the reverse direction), and (*ii*) such a request to combining F into G is only accepted (or rejected) by *the first process of the fragment G which has received the request to merging* F into G initiated by the root of fragment F.

More precisely, at any stage of the algorithm, any fragment selects at random an outgoing edge and sends a request-message of merging along the selected link to the node which is at the extremity of this latter edge. This very node accepts or rejects the combination, according to the strict order relation existing between the identities (all the fragments are uniquely identified by the identy of their root). Thus, the algorithm can perform the combination of rooted fragments.

2.1.2 The Steps of the Combination Process

Let the fragments F and G be connected by an edge (*a*,*b*). Merging F into G,which depends of the order relation F < G, produces a new fragment in which the node *a* is a son of the node *b*. This process reverses the order relation all along the path from the node *a* to the root of F (say, *i*).

The combination process is thus completed in three steps :

1. The root *i* of F sends a combination request-message **conn** to node *b* in another fragment G via some path in F ending with edge (*a*,*b*) (*a* being a node in fragment F).

2. Node *b* sends back to *i* a combination accept-message **ok** in the reverse direction within fragment F.

3. The root *i* of fragment F sends a message **merge** to the root *j* of fragment G.

As a consequence, any node in a fragment should be able to identify : *i*) the fragments having a larger identity, i.e. the fragments to which combination request-messages may possibly be sent, and *ii*) the fragments having a smaller identity, i.e. the fragments which may possibly send a combination request-message.

Note that whenever there eventually exist no more such fragments (as *i*) and *ii*)), the current fragment actually is a spanning tree for the whole network and the algorithm is obviously terminated.

Note also that in the case when both identities of the sending and the receiving node are equal (without father-to-son relation), then the receiving node sends back the message **cousin**.

Whenever a root has sent a connection request-message it cannot send another connection request-message before receiving an answer **nok** from the previous one.

2.1.3 Messages and Variables

a)The Pattern of the Messages

The pattern of the messages sent from a node *i* is <*a,b,c,d*> with the following meaning of the four records : *a* is the identity of a root, *b* is a keyword, *c* is the boolean value of (*i* is *free*), and *d* is the boolean value of (*i* is *open*). In the following, MAJ denotes the procedure in which booleans arrays and variables are handled at any node upon receipt of a keyword record. The keyword is an element of the set {**conn, ok, nok, merge, opening, newroot, cousin, end**}

c) The Variables *free* and *open*, the Arrays ACTIVE and CANDIDATE

Definitions and Notations

The binary predicates "is *active for* " and "is *candidate for*" concerning two neighbour-nodes, and the general unary predicates "is *free*" and "is *open*" are defined as follows.

If a node *x* has at least one out-neighbour (i.e. a neighbour which does not belong to the fragment of *x*) or one free son, *x free*. In the reverse case, it is *complete*.

Out-neighbours and free sons of a node *i* are *active for i*.

If an out-neighbour of a node *i* belongs to a fragment which may absorbe the fragment of *i*, it is *candidate for i*.

If a node *i* possesses a *candidate for i* out-neighbour or if at least one son of *i* is *open*, then the node *i* is *open*. In the reverse case, *i* is *closed*.

Any open son of a node *i* is *candidate for i*.

Note that a connection request-message can only be sent from a node *a* to a node *b* iff *b* is *open for a*.

For each node, handling the predicates *free* and *open* requires two boolean arrays (indexed on its neighbours) and two boolean variables, viz. ACTIVE[ ], CANDIDATE[ ], *free*, and *open*.

When the algorithm starts, the logical arrays CANDIDATE and ACTIVE at each node are the characteristic vectors of its neighbours. In the initialization phase, (y is a neighbour of x) $\Leftrightarrow$ (CANDIDATE(y) = true and ACTIVE(y) = true)

Also define the variables *open* and *free* as follows.:

$$open = \begin{cases} \text{true, if there is at least one element of the array CANDIDATE} \\ \text{which is set to true.} \\ \\ \text{false, if there is no element of the array CANDIDATE which is set to} \\ \text{true.} \end{cases}$$

$$free = \begin{cases} \text{true, if there is at least one element of the array ACTIVE which is set} \\ \text{to true.} \\ \\ \text{false, if there is no element of the array ACTIVE with is set to true.} \end{cases}$$

The value of an element of the logical array CANDIDATE is thus modified in the following cases :

i) If node x sends back a message **nok** to node y, x sets CANDIDATE(y) to true (if necessary). If the variable *open* has to change value (from false to true), the corresponding message **opening** must be sent to the father of x.

ii) Whenever node x receives a message **opening** from a son (say, z), x updates the value CANDIDATE(z) to true.

iii) Whenever node x receives a message **nok** from node y, x setsCANDIDATE(y) to false and recomputes the value of the variable *open* := $\vee$ CANDIDATE(). If the value of *open* happens to be false, it only means that no message **conn** can be sent to any neighbour. Hence, x sends back a message **nok** to its father.

The following first property.directly follows from the above rules.

Property 1. *Let (a,b) be an outgoing edge and* $\text{CANDIDATE}_a[b]$ *and* $\text{CANDIDATE}_b[a]$ *be the values of the items corresponding to the arrays* CANDIDATE *for both nodes a and b indexed by b and a, respectively. We then have :*

$$(\text{CANDIDATE}_a[b] = 0) \Rightarrow (\text{CANDIDATE}_b[a] = 1).$$

The value of an element of the logical array ACTIVE is modified upon receipt of the messages

- **cousin.** The current node both learns that no other message **conn** will ever be received from port y, and that it is useless to send any message **conn** throught this port (the node connected to y being a cousin). Hence, the current node sets ACTIVE(y) to false.

- **ok.** Whenever node x receives a message **ok** from node y, the identity of its new father is y, and x sets ACTIVE(y) to false .

- **merge & nok.** Whenever node x receives a message **merge (nok)** from node (a son) y (respectively), x updates ACTIVE(y).

Clearly, if all items of an array ACTIVE at some node x are set to false, then every node in the fragment of root x is connected to another fragment. Thus, we may claim the two following properties.

Property 2. *For any node, the formula* (CANDIDATE[i] = 1) $\Rightarrow$ (ACTIVE[i] = 1) *holds.*

Proof. Immediate, since for any node x, CANDIDATE[i] = 1 means that the node at the other side of the link connected with the port i belongs to another fragment than x. ❑

Property 3. *Let r be the root of a fragment F. If* ($\forall y$) ACTIVE(y) *= false at the root r , then the computation is finished and r is the elected leader.*

Proof. Direct consequence of how the array ACTIVE is handled. If ACTIVE(y) = false at the root r, for all y, then there exists no outgoing edge connected to F. Hence, no modification of the array ACTIVE is still possible ; such a root is called complete.❑

If, for a node x in a fragment F, CANDIDATE[i] = 0 and ACTIVE[i] = 1,

ACTIVE[i] = 1 means that there exists at least one path of origin x which extremity is a node in a fragment G≠F containing the node i, and such that this path contains the node-process connected with x throught the port i (here and in the sequel, we consider the name of the port as identical to the identity of the node to which it is connected). In other words, the path contains an outgoing edge. In the case when the path contains one only edge, this must be (x,i) which is then the outgoing edge.

CANDIDATE[i] = 0 means that, throught each path of origin x containing an outgoing edge in its extremity which contains i, x has received a message **nok**.

Property 4. *Let (a,b) be an outgoing edge, and* ACTIVE$_a$[b]*and* ACTIVE$_b$[a] *be the values of the items corresponding to the arrays* ACTIVE *for both nodes a and b indexed by b and a, respectively We then have :* ACTIVE$_a$[b] = ACTIVE$_b$[a].

Proof. Immediate from the definitions of $ACTIVE_a[b]$ and $ACTIVE_b[a]$: $ACTIVE_a[b] = 1 \Leftrightarrow ACTIVE_b[a] = 1$. The reverse is also obvious, and $ACTIVE_a[b] = 0 \Leftrightarrow ACTIVE_b[a] = 0$.

❑

2.1.4 The Behaviour of Roots and non-Roots Nodes

a) Roots

As previously noted a complete root is the root of a completed ST. Hence, a complete root broadcasts to all its sons the message **end** and stops. Conversely, an open root *r* selects one *candidate for r* neighbour and sends it a connection request-message <*i*, **conn**, ø, ø>. It can send no other connection request-message but upon receipt of the answer **nok** to its previous message.

Three answer messages to a connection request-message sent from *x* to *i* are possible,

<ø, **cousin**, false, false> : *i* and *x* are cousins, and *i* executes the procedure MAJ ;

<ø, **nok**, *c*, *d*> : the connection request is rejected according to the strict total order relation in ***I*** ;

<*j* , **ok**, *c*, *d*> : the connection request is accepted ; *x* becomes the father of *i*, *i* executesthe procedure MAJ, it broadcasts to all its sons the message <*j*, **newroot**, ø, ø>, and it sends its father the message <ø,**merge**,*free*, *open*>. From now on, *i* is a non-root node and will never be root any more.

Independently, a root may receive the following other messages : <ø, **opening**, ø, true> and <ø, **merge**, *c*, *d*>, then *i* executes the procedure MAJ, or <*k*, **conn**, ø, ø>, and then the behaviour of root *i* is identical to a non-root node's.

b) Non-Roots Nodes

A non-root node *i* can only act upon receipt of a message. There are eight possible messages a node *i* can receive from a node *x*.

1. <*k*, **conn**, ø, ø>. This message leads to three possible different situations :

- *i* and *x* are cousins. *i* sends the message <ø, **cousin**, false, false> to *x*, and sets the values of ACTIVE[*x*] and CANDIDATE[*x*] to false.

- *x* is the father of *i*. If *i* is *open*, it simply passes the message at random to an open neighbour, else *i* sends back the message <ø, **nok**, ø, false> to its father.

- x and i are out-neighbours. According to their fragment identity, either i accepts the connection request-message, x becomes the son of i, i sends the message <*root_of_i*, **ok**, false, false> to its father and assigns the value false to CANDIDATE[x] (this implies that the induced merge message should pass through i), or i rejects the connection request-message, i sends back the message <ø,**nok**,ø,ø> to x, and assigns the value true to CANDIDAT[x] ; if the variable *open* changes its value, i sends the message <ø, **opening**, ø, true> to its father.

2. < , **cousin**, false, false> or < , **nok**, ø, false>. i executes the procedure MAJ. This message is an answer to a connection request-message sent from i to x : if i is *open*, it passes the connection message to an open neighbour, else i sends the message <ø, **nok**, ø, false> to its father.

3. < , **ok**, false,false>. i executes the procedure MAJ and passes the message to its father. x becomes the father of i, so i broadcasts to all its sons the message <k, **newroot**, ø, ø>.

4. < , **merge**, g, h>. i executes MAJ and passes the message to its father.

5. < , **opening**, ø, true>. i executes the MAJ, and if (but only if) the variable *open* changes its value, i passes the message to its father.

6. <k, **newroot**, ø, ø>. i broadcasts the message to all its sons and changes its own root 's identity, if it is was different from k.

7. < , **end**, ø, ø>. i broadcasts the message to all its sons and stops.

Notice that the arrays ACTIVE[] and CANDIDATE[] make the **conn** messages move in the reverse direction (backtracking) throught a fragment.

Note about the Termination.

There is another way to ensure a good termination of the algorithm. Assume each node-process knows the size n of the distributed system. In that case, every root r knows at any time the size of its own fragment F, and whenever F merges into another fragment G, r sends its size to the root of G. Thus, whenever a fragment eventually achieves the size n, its root knows that it is elected as the root of the constructed ST, and the algorithm is terminated. A local variable *counter of nodes* replaces then the variable *free* and the array ACTIVE[] ; it is initialized to the value one and updated for each combination between two fragments.

2.2 Specification of the Algorithm

The specification of the algorithm uses a C.S.P.-like syntax (see [LAV-86]). The differences between the standard C.S.P. syntax of [HOA-78] lies in the fact that we consider communication primitives as non-blocking primitives. Thus, the primitive P_i !! <*x*> means that the message *x* is sent to the process which identity is *i* (such a communication being non-blocking). In other words, a FIFO-queue is attached to P_i in which incoming messages are stored. We only assume here that the size of such a queue remains "reasonable". Note that this assumption has been fully verified in experimental tests of the algorithm on a network of transputers.

Similarly, P_j ?? <*y*> means that a current process reads in its queue the message *y* sent from process P_j.

The syntax of the algorithm is also simplified by the use of set notations as $\cup_{j \in SON} P_j$!! <*x*>, which means that the current process broadcasts message *x* to all elements of the set SON.

The value of the variable *root* is the identity of the root of the fragment which contains the current node-process.

The value of the variable *pred* is the identity of the father of the current node-process.

The value of the variable *req* is true if the current node-process has previously sent a connection message.

Algorithm

```
Procedure MAJ ::      CANDIDATE[y] := false;
                      open := ∨ CANDIDATE[y] :
                      ACTIVE[y] := false;
                      free := ∨ ACTIVE[k];

Proc i ::
root := i ; pred := nil ; req := false ; fils := ø ;
∪j ∈ NEIGHBOUR ACTIVE[j] := true ; free := true ;
∪j∈ NEIGHBOUR CANDIDATE[j] := true ; open := true ;
*[(root = i ∧ req = false ∧ open = true) → x := select() ; Px !! <i,conn,ø,ø> ; req := true
▮
(root = i ∧ req = false ∧ free = false) → ∪x ∈ SON Px !! <ø,end,ø,ø> ; STOP.
▮
```

```
P_y ?? <a,b,c,d> → [    b = end → ∪_{x ∈ SON} P_x !! <ø,end,ø,ø> ; STOP.
                        ▮
                        b = conn ∧ y = pred → [open = true → x := Select() ; P_x !! <a,b,c,d>>
                                               ▮
                                               open = false → P_y !! <ø,nok,free,open>
                                               ]
                        ▮
                        b = conn ∧ y ≠ pred → [a = root →P_y !! <ø,cousin,ø,ø> ; MAJ ;
                                               ▮
                                               a < root → P_y !! <root,ok,ø,ø>; SON := SON ∪ {y} ;
                                                          CANDIDATE[y] := false ;
                                               ▮
                                               a > root → P_y !! <ø,nok,ø,ø> ; CANDIDATE[y]:= true ;
                                                     [open = false ∧ i ≠root → P_pred !! <ø,opening,ø,ø>] ;
                                                     open := true
                                               ]
                        ▮
                        b = ok ∧ root =i → [y ∈ SON → SON := SON - {y}] ; root:= a; pred: y ; MAJ ;
                                  ∪_{x ∈ SON} P_x !! <a,newroot,ø,ø> ; P_y !! <ø,merge,free,open>
                        ▮
                        b = ok ∧ root ≠i → [y ∈ SON → SON := SON - {y}] ; P_pred !! <a, b,c,d> ; root:= a ;
                              ∪_{x ∈ SON} P_x !! <a,newroot,ø,ø> ; SON := SON ∪ {pred} ; pred := y ; MAJ ;
                        ▮
                        b = nok → CANDIDATE[y] := false ; open := ∨ CANDIDATE[k] ;
                                  [y ∈ SON ∧ c = false → ACTIVE[y] := false ; free := ∨ ACTIVE[k]  ] ;
                                  [root = i → req := false
                                  ▮
                                  root ≠i → [  open=true → x := Select() ; P_x !! <root,conn;ø,ø> ;
                                               ▮
                                               open=false → P_pred !! <ø,nok,free,ø>
                                            ]
                                  ]
                        ▮
                        b = cousin ∧ root ≠ i → MAJ ;  [open=true → x := Select() ;
                                                                P_x !! <root,conn,ø,ø> ;
                                                        ▮
                                                        open = false → P_pred !! <ø,nok,free,ø>
```

]

▮

b = **cousin** $\wedge$ *root* = i $\rightarrow$ MAJ ; *req* := false

▮

b = **opening** $\wedge$ *root* $\neq i$ $\rightarrow$ CANDIDATE[y] := true ;
[*open*=false $\rightarrow$ P_{pred} !!<$\emptyset$,**opening**,$\emptyset$,$\emptyset$>] ; *open* := true

▮

b = **opening** $\wedge$ *root* = i $\rightarrow$ CANDIDATE[y] := true ; *open* := true

▮

b = **newroot** $\rightarrow$ *root* := a ; $\cup_{x \in SON} P_x$!! <a,**newroot**,$\emptyset$,$\emptyset$>

▮

b = **merge** $\rightarrow$ CANDIDATE[y] := h ; *open* := $\vee_k$ CANDIDATE[k] ; ACTIVE[y] := g ;
free := $\vee_k$ ACTIVE[k] ; [*root* $\neq i$ $\rightarrow$ P_{pred} !! <a,b,*free*,*open*>]

]

]

3. ANALYSIS

3.1 Correctness of the Algorithm

The correctness proof of the algorithm is completed in proving that the constructed pattern is a forest, and then by proving the convergence of the algorithm.

3.1.1 Subtrees as Invariant.

To prove the correctness of this algorithm we must show that no cycle is built during the computation, and that the result is indeed a spanning tree.

Lemma 1. *When the algorithm starts, each single node in itself constitutes an entire fragment of which it is the root .*

Proof. Immediate. As consequence of definitions. ❑

Lemma 2. *At any stage of the algorithm, no cycle can be built.*

Proof. The strict total order relation existing in I makes the set of combining fragments a subset of a superhalf lattice ordered by the accepting relation (see [BIR.]). The representative graph of such a subset is a forest. Which implies that there is no cycle in the resulting graph. ❑

Theorem 1. At any stage of the algorithm the pattern built in the network is a forest.

Proof. Immediate. Theorem 1 is a straightforward consequence of lemma 2. ❑

3.1.2 Convergence of the Algorithm

Lemma 3. *For any node in the network, if* ACTIVE[i] = *1 then in a finite (but impredictible) amount of time,* ACTIVE[i] = *0 shall eventually hold.*

Proof. First assume that ACTIVE[i] = 1 affects a port i of a node x and that the edge (x,i) is an outgoing edge. It follows from lemma 2 that no message can circulate forever in a loop. Moreover, in any fragment there is a message **conn** alive. This implies that a message **conn** is sent out of a fragment throught a port such that CANDIDATE[i] = 1, whence, by Property 2, ACTIVE[i] = 1. Two possibilities might then arise.

Either the answer to message **conn** is **ok** and lemma 3 holds,

or the answer to message **conn** is **nok**, and the value of CANDIDATE[i] becomes 0 while the value of ACTIVE[i] remains unchanged. In such a case, from Property 1, CANDIDATE[x] = 1 at node i, and thus, a message **conn** is still alive in the fragment containing i. Hence, this message **conn** will eventually pass throught port x at node i. Thus, the two previous possibilities may still happen. Yet, the number of nodes in the network is finite, and whence the number of fragments is finite too. As a consequence, the fist case shall eventually take place and the first possibility is inevitable to occur.

Next, assume that (x,i) is not an outgoing edge and also assume that CANDIDATE[i] = 1 and ACTIVE[i] = 1 at node x, and CANDIDATE[x] = 1 and ACTIVE[x] = 1 at node i. Then, a message **conn** shall eventually pass throught the link (x,i). The answer-message to **conn** is **cousin**, whence CANDIDATE[i] = 0 and ACTIVE[i] = 0 at node x, while CANDIDATE[x] = 0 and ACTIVE[x] = 0 at node i. ❑

Note that we can assume that there remains only two fragments connected by only one edge (x,y) before the last stage of the algorithm. Since CANDIDATE[y] and CANDIDATE[x] cannot simultaneously be set to false, the algorithm can perform the last stage and stop.

Property 5. *The algorithm is starvation free.*

Proof. Direct consequence of Property 1 and Theorem 1. ❑

From lemma 3, we may now claim the following

Theorem 2. *In a connected network, the algorithm builds a spanning tree and elects a leader within a finite time. The elected node is the root of the constructed spanning tree.*

Proof. Consequence of lemma 3. Since the number of nodes is finite (and from lemma 3), the number of fragments is strictly decreasing and their respective size is strictly increasing during any execution of the algorithm. Hence, a unique fragment eventually remains within the last stage of the algorithm, which contains the whole set of nodes of the network ; this last fragment is the (finite time) constructed ST, and its root is the (unique) elected leader. □

3.1.3 The minimum-weight Spanning Tree.

The previous algorithm is easily transformed into a MST algorithm. Assume that when the algorithm starts, each node-process in the network knows the weight of its outgoing edges. The algorithm must simply choose the minimum-weight outgoing edge (or possibly one of the minimum-weight outgoing edges) for each node. Hence, every node has to handle a local variable containing the minimum value of the outgoing edge of the fragment of which it is the extremum, this according to the preorder father-to-son relation. The modification of the algorithm only entails larger messages to contain one edge weight as a new record. The extra number of bits is thus bounded from above by lgw, where w is the maximal value of an edge weight

3.2 Complexity of the Algorithm

The complexity of a computation in a distributed system is evaluated with respect to two basic parameters : *communication* and *time*. The *communication complexity* of an algorithm is measured as the number of communication activities (i.e. *message* transmission) performed in the system during the computation process of the algorithm, viz. the *message complexity*, or alternatively the *bit complexity*, of the algorithm.

The *time complexity* of the algorithm is measured as the total delay from the time the first process starts the computation to the time the last process terminates the computation.

Note that since the maximum number of bits contained in any message is $O(\lg i)$, where $i \in I$, denotes the identity of the root of the constructed spanning tree, we will only consider the message complexity of the algorithm (indeed, the *bit complexity* of the algorithm can be obviously derived from the latter).

Consider an *asynchronous* distributed system. Each process is distinguished by a unique identity, taken from some index universe I. We assume throughout that the processes work fully asynchronously and cannot use global clocks nor time-outs. Hence, we can assume in the following analysis that the algorithm is message-driven : except for the first message upon initialization, any process can only perform actions as a result of the receipt of a message.

As to the notion of *time*, we assume in the analysis of the algorithm that all message delay times are bounded and equal, i.e. each message takes unit time along any link of the distributed system.

Further assume that whenever a process receives several messages (from several neighbours) at the same moment, it handles the messages with respect to the strict total order relation existing in I between the identities of these message-sending neighbours. This breaks the implicit non-determinism associated with asynchrony, and thus, as far as complexity measures are concerned, we may consider the algorithm as an asynchronous, message-driven algorithm, running in synchronous distributed systems. Hence, the distributed asynchronous algorithm is more easily analysed using the virtual notion of *phases* of the algorithm. A phase may actually be considered as the equivalent of a (global) clock pulse, or simultaneous clocks' ticks (as defined in the Introduction) in a synchronous distributed system. During each current phase of the algorithm, nodes (posssibly) receive messages, perform local computation, and send messages destined to be received at the beginning of the next phase.

3.2.1 Number of Phases and Time Complexity of the Algorithm

As in [LAV-86], we will assume that the behaviour of the algorithm implies a well-balanced growth of the successively combined fragments. So that, in any phase of the algorithm, the number of answer to the connection request-messages is roughly equally distributed : one half being composed of messages **ok**, and the the other half of messages **nok**. Now the number of phases of the algorithm is easily evaluated.

General Lemma *The maximum number of phases of the algorithm is at most* $\lceil \lg n \rceil + 1$.

Proof. At phase zero, each single node constitutes an entire fragment : each fragment is of size one. At phase one, a set of fragments with one edge each is constructed, and there are $n/2$ fragments. At the second phase, there are two edges in each fragment and $n/4$ fragments. At the current phase φ, there are 2^{φ} edges in $n/2^{\varphi}$ fragments. Obviously, the algorithm eventually ends when $\varphi = \lceil \lg n \rceil$, and the algorithm builds the ST and elects a leader within at most $\lceil \lg n \rceil + 1$ phases. ❑

Theorem 3. *Whatever assumptions on the network, the time complexity of the ST-Election (and of the MST) algorithms is measured with at most the maximum number of phases of the algorithm, viz.* $O(\lg n)$.

Proof. Immediate from the above lemma. Yet, note that up to a constant factor, the time complexity measure ranges between at least $2\lg n + O(1)$ for the virtually synchronized algorithm

running in complete meshed networks (the time complexity of [CHAN,CHIN-88]), and at most $\lambda \lg n$ (λ real) for arbitrary networks and general assumtions. □

3.2.2 Upper Bound on the Message Complexity of the Algorithm in an Arbitrary Network

We determine here an upper bound on the number of messages exchanges during any execution of the algorithm. Note again that the most complex message contains $O(\lg i)$ bits.

Assume the size n of the network to be unknown to all the node-processes.

Theorem 4. *In the case when all the node-processes have no knowledge ofn, the number of messages required by the ST-Election algorithm is at most* $2e + 7/2\ n\lg n + O(n)$.

Proof. Since an edge can be rejected only once, and each rejection requires one message , there are at most e reject messages in any execution of the algorithm.

Next, at the first phase, each node sends a connection request-message **conn**, and thus n messages are sent. Assuming the growth of the fragments to be well-balanced, there are $n/2$ ok and $n/2$ **nok** answer messages. The $n/2$ **ok** messages yield $n/2$ **merge** messages and $n/2$ **newroot** messages as well. Taking this upper bound for each phase, and adding the n - 1 **end** messages broadcasted at the end of the general computation yields a number of at most $3n\lg n + n - 1$ messages (which is the upper bound obtained in [LAV,ROU-86]). Taking now also into account the messages **cousin** and **opening** adds $\frac{1}{2} n\lg n$ messages to our grand total. The number of messages used by the algorithm is whence $e + 7/2\ n\lg n + O(n)$. □

Assume next the size n of the network to be known by *at least one* node-process. Note that such a global information at only one node-process is enough to imply the global knowledge of n in the whole network.

Theorem 5. *In the case when one node-process knows n, the number of messages required by the ST-Election algorithm is at most* $7/2\ n\lg n + O(n)$.

Proof. In the case, the knowledge of n makes it unnecessary for the messages to traverse all the edges of the network ; the amount of extra messages is here at most $O(n)$. Hence, adding the results of the previous computations of the algorithm without any knowledge of n leads to an upper bound on the number of messages exchanged during any execution of the ST-Election algorithm : viz. $7/2\ n\lg n + O(n)$. □

3.2.3 Message Complexity and Time Complexity in Complete Meshed Networks

In the pseudo-synchronized variant of the algorithm and with regard to the election process, each candidate (root of a fragment) is either captured (by some other candidate) or (still being candidate) increases the extend of its own fragment within each phase.
The current phase is denoted by φ $(0 \leq \varphi \leq \lceil \lg n \rceil)$ and the algorithm runs in at most $O(\lg n)$ phases.

a) Worst-case message complexity.

At phase φ, the number of fragments is at most $n/2^{\varphi}$, and each fragment's node disjointly owns 2^{φ} within its fragment (this for $0 \leq \varphi \leq \lceil \lg n \rceil$)

Lemma 4. *The worst -case message complexity of the algorithm is* $\frac{1}{2} n \lg n + O(n)$.

Proof. Each node will then send out at most $n/2^{\varphi}$ messages **conn** in an attempt to merge its fragment into some other fragment. Thus, in a similar process to section 3.2.1 : at phase φ, the number of fragments is $n/2^{\varphi}$ which size is at most 2^{φ}. There are $n/2^{\varphi}$ messages **conn** sent, $n/2^{\varphi+1}$ **ok** and $n/2^{\varphi+1}$ **nok** answer messages. The $n/2^{\varphi+1}$ messages **ok** yield $n/2^{\varphi+1}$ messages **merge** and $(n/2^{\varphi+1})\mathbf{x}2^{\varphi}$ messages **newroot** (i.e. the current number of messages **ok** times the maximum size of a current fragment). There are also $n/2^{\varphi+1}$ messages **opening** and α messages **cousin** and β messages **nok**, with $\alpha + \beta \leq n/2^{\varphi}$. Adding n - 1 messages **end** broadcasted at the last phase of the computation to our grand total yields a number of messages N such that

$$N \leq \sum_{\varphi=0}^{\lg n} \{ n/2^{\varphi} + \alpha + \beta + 2n/2^{\varphi+1} + 2n/2^{\varphi+1} + (n/2^{\varphi+1})\mathbf{x}2^{\varphi} \} + n - 1$$

$$\leq \sum_{\varphi=0}^{\lg n} \{ 2n/2^{\varphi} + 4n/2^{\varphi+1} + \tfrac{1}{2}n \} + n - 1 \leq \tfrac{1}{2} n \lg n + n/2 + \sum_{\varphi=0}^{\lg n} 4n/2^{\varphi+1}$$

Thus, $$N \leq \tfrac{1}{2} n \lg n + \tfrac{1}{2} n + n + 4n + O(1) = \tfrac{1}{2} n \lg n + 11/2\, n + O(1),$$

and the worst-case message complexity is $\frac{1}{2} n \lg n + O(n)$. □

b) Average-case message complexity.

The crux of the proof lies in determining an upper bound on the expected number of roots to survive each phase, from phase 0 to phase $\lceil \lg n \rceil$.

Lemma 5. *Let p be the number of nodes disjointly belonging to each of the (at most) $n/2^{\varphi}$ candidate fragments at the start of phase φ, and q the number of messages sent by each of these candidate fragments at the start of phase φ. Then, the expected number of fragments to start phase $\varphi + 1$ is at most $1 + (n - p - q)/p(q - 1)$.*

Proof. Let $r_1, r_2, \ldots, r_m$ be the candidates to start phase φ arranged in decreasing identity order ($mp \leq n$) and p_i the probability that r_i survives phase φ. If r_i is to survive phase φ, it must send messages to q nodes *other than those already belonging to fragments* $F_1, F_2, \ldots, F_{i-1}$.

Hence, $p_i \leq \dfrac{\binom{n-ip}{q}}{\binom{n-p}{q}}$, and the expected number of candidates to start phase $\varphi + 1$ is :

$$\sum_{i=1}^{m} p_i = 1 + \sum_{i=2}^{m} p_i \leq 1 + \frac{\sum_{i=2}^{m} \binom{n-ip}{q}}{\binom{n-p}{q}}$$

The expected number of candidates is whence at most

$$1 + \frac{\sum_{j=2p}^{mp} \binom{n-j}{q}}{\binom{n-p}{q}} \quad \text{(with } mp \leq q\text{)}.$$

And

$$1 + \frac{\sum_{j=2p}^{mp} \binom{n-j}{q}}{\binom{n-p}{q}} \leq \frac{\sum_{j=p+1}^{n} \binom{n-j}{q}}{\binom{n-p}{q}} .$$

Now, $$\sum_{j=p+1}^{n} \binom{n-j}{q} = \sum_{j=0}^{n-p-1} \binom{j}{q} = \binom{n-p}{q+1}.$$

Hence, the expected number of candidates is

$$\leq 1 + \frac{1}{p} \frac{\binom{n-p}{q+1}}{\binom{n-p}{q}} = \frac{n-p-q}{p(q-1)} . \quad \square$$

Theorem 6. *The algorithm constructs a ST and elects a leader in a complete meshed network using* $\frac{1}{2}n\lg n + O(n)$ *messages in the worst case,* $O(n)$ *messages on the average, and* $O(\lg n)$ *time.*

Proof. The time complexity and the worst-case message complexity of the algorithm in complete meshed networks are already determined in the above Theorem 3 and lemma 3

An upper bound on the number of roots (candidates) is known from lemma 5. Now, since the roots send more messages than the other nodes, which number of propagated messages is certainly at most $O(n)$, an upper bound on the number of messages sent by the roots is given from lemma 5 with $p = 2^{\varphi}$ and $q = n/2^{\varphi}$,

$$(n + 4\,n/2) + \sum_{\varphi=2}^{\lg n} \left\{ 1 + \frac{n - 2^{\varphi} - n/2^{\varphi}}{2^{\varphi} \times n/2^{\varphi}} \right\} (1 + 2^{\varphi}) \leq 3n + \sum_{\varphi=2}^{\lg n} 2(1 + 2^{\varphi})$$

$$\leq 3n + 2\lg n + n + O(1).$$

Hence, the average complexity of the algorithm is bounded from above by $O(n)$. ❑

3.2.4 Message Complexity in a Ring-Based Topology

Theorem 7. *On a bidirectional ring without the global sense of direction and without knowledge of n, the present algorithm uses at most* $3n\lg n + O(n)$ *messages, and* $\frac{1}{2}n\lg n + O(n)$ *messages on the average.*

Proof. The only available messages on a bidirectional ring are the messages **conn**, **ok**, **nok**, **merge**, **newroot** and **end**, since the messages **cousin** and **opening** cannot be used on a circular configuration of processes.

The number of each of these messages is bounded from above by $n/2$ in each phase of the algorithm, and the number of messages exchanged is at most $(4 \times n/2)$ times the number of phases minus 1 plus $O(n)$.
In other words, an upper bound on the number of propagated messages is $3n\lg n + O(n)$ (i.e. $4.328\ldots\, n\ln n + O(n)$).

Next, the average number of messages **conn**, **ok**, **nok**, **merge**, **newroot** and **end** is respectively : (at phase φ, $0 \leq \varphi \leq \lceil \lg n \rceil - 1$) $n/2^{\varphi}$, $n/2^{\varphi+1}$, $n/2^{\varphi+1}$, $n/2^{\varphi+1}$, $[n/2^{\varphi+1}] \times 2^{\varphi}$, and $(n-1)$ in the last phase.

Hence, the expected number of messages is

$$\sum_{\varphi=0}^{\lceil \lg n \rceil -1} \{ n/2^{\varphi} + 3n/2^{\varphi+1} + (n/2^{\varphi+1}) \times 2^{\varphi} \} + n - 1 = \tfrac{1}{2} n \lg n + 3n \sum_{\varphi=0}^{\lceil \lg n \rceil -1} 2^{-\varphi} + O(n) ;$$

and since the sum is $O(n)$, the average number of messages exchanged in the algorithm is $\frac{1}{2} n \lg n + O(n)$ (i.e. $0.721\ldots\ n \ln n + O(n)$). □

4. Conclusions

The present algorithm is a strong improvement of [LAV,ROU-86], both within sparse and dense network topologies. In using the new variables *free* and *open* and the new messages **opening** and **cousin**, the algorithm is more efficient in arbitrary networks.

With regard to the upper bound on the number of messages, the algorithm strongly improves on [LAV,ROU-86] in arbitrary networks, since the latter algorithm could actually entail the expense of an *unbounded number of messages* (for a given n) for specific network topologies. In complete meshed networks, it also matches the algorithm of [CHI,CHA-88] in the worst case and on the average. Finally, with an average message complexity $\frac{1}{2} n \lg n + O(n)$ (i.e. $0.721\ldots\ n \ln n + O(n)$), the algorithm roughly matches the best distributed leader finding algorithms on bidirectional rings so far known : viz. the bidirectional variant of Chang-Roberts algorithm which uses $\frac{1}{2} \sqrt{2}\, n H_n$ (i.e. $0.707\ldots\ n \ln n + O(n)$) messages on the average. Moreover, the (virtual) time complexity of the algorithm $O(\lg n)$ is identical to the efficient ST-Election algorithms.

Aknowledgement. We express our thanks to Mr. Fanchon who programmed the algorithm in OCCAM, and runned it on a Transputer Network.

References

[BIR-67] **G. BIRKOFF,** *Lattice theory*, Am. Math. Society, Vol. XXIII, 1967.

[CHA,CHI-88] **M. Y. CHAN & F. Y. L. CHIN,** *Distributed election in complete networks*, Distributed Computing, Vol. 3, n°1, 1988, 19-22.

[CID,JAF,SID-87] **I. CIDON, J. JAFFE & M. SIDI,** *Local Distributed Deadlock Detection by Cycle detection and Clustering*, IEEE Trans. on Software Engineering, Vol. SE13, N° 1, Jan. 1987, 3-14.

[DWO,LYN,STO-88] **C. DWORK, N. A. LYNCH & L. STOCKMEYER,** *Consensus in the presence of partial synchrony*, J. ACM 35, 1988, 288-323.

[GAF,LOU,TIW,WES,ZAK-86] **E. GAFNI, M. C. LOUI, P. TIWARI, D. B. WEST & S. ZAKS,** *Lower Bounds on Common Knowledge in Distributed Algorithms*, Proc. 1st. Int. Workshop on Distributed Algorithms, Ottawa, Distributed Algorithms on Graphs (ed. Gafni & Santoro, Carleton Un. Press), 1986, 49-67.

[GAL,HUM,SPI-85] **R.G. GALLAGER, P. A. HUMBLET & P. M. SPIRA,** *A Distributed algorithm for minimum weight spanning trees*, ACM Trans. Prog. and Syst. 5, 1985, 66-77.

[HEL,MAD,RAY-87] **J-M. HELARY, A. MADDI & M. RAYNAL,** *Calcul distribué d'un extremum et du routage associé dans un réseau quelconque*, RAIRO Informatique théorique et Applications, Vol. 3, 1987.

[HOA-78] **C. A. R. HOARE,** *Communicating sequential Processes*, CACM, Vol. 21, n°8, Aug. 1978.

[KOR,MOR,ZAK-84] **E.KORACH, S. MORAN & S. ZAKS,** *Tight lower and upper bounds for some distributed algorithms for a complete network of processors*, Proc. 3rd ACM Symp. on Principles of Distributed Computing, ACM, New-York, 199-207.

[LAV-86.] **I. LAVALLÉE**, *Algorithmique parallèle et distribuée, application à l'optimisation combinatoire*, Thèse de Doctorat d'Etat , Université Paris 11, June 1986.

[LAV,ROU-86] **I. LAVALLÉE & G. ROUCAIROL**, *A Fully distributed (minimal) spanning tree algorithm*, I.P.L. 23, 1986, 55-62.

[LAV-87] **C. LAVAULT**, *Algorithmique et complexité distribuées*, Thèse de Doctorat d'Etat, Université Paris 11, Dec. 1987.

[MAT-87] **F. MATTERN**, *Algorithms for distributed termination detection*, Distributed Computing, Vol.2, n° 3, 1987, 161-175.

[PAC,KOR,ROT-84] **J. PACHL, E. KORACH & D. ROTEM**, *Lower bounds for distributed maximum finding algorithms*, J. ACM 31, 1984, 380-401.

[SAN-84] **N. SANTORO**, *On the message complexity of distributed problems*, Int. J. Comput. Inf. Science 13, 131-147.

[SOL-63] **M. SOLLIN**, Exposé du séminaire de C. Berge, I.H.P., 1961, *Méthodes et modèles de la Recherche Opérationnelle*, T. 2, ed. Kauffmann (Dunod-Paris), 1963, 33-45.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique