Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

312

J. van Leeuwen (Ed.)

Distributed Algorithms

2nd International Workshop
Amsterdam, The Netherlands, July 1987
Proceedings

Springer-Verlag

Lecture Notes in
Computer Science

Edited by G. Goos and J. Hartmanis

312

J. van Leeuwen (Ed.)

Distributed Algorithms

2nd International Workshop
Amsterdam, The Netherlands, July 8–10, 1987
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo

Editor
J. van Leeuwen
Department of Computer Science, University of Utrecht
P.O. Box 80.012, NL-3508 TA Utrec

CR Subject Classification

ISBN 3-540-19366-9 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-19366-9 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

PREFACE

The 2nd International Workshop on Distributed Algorithms was held at the Centre for Mathematics and Computer Science (CWI) in Amsterdam, July 8-10, 1987. The workshop was intended to provide a forum for researchers and other interested parties to discuss the recent results and trends in the design and analysis of distributed algorithms on communication networks and graphs.

Papers were solicited describing original results in all areas of distributed algorithms and their applications including, e.g distributed combinatorial algorithms, distributed optimization algorithms, distributed algorithms on graphs, distributed algorithms for control and communication, routing algorithms, design of network protocols, distributed database techniques, algorithms for transaction management, fail-safe and fault-tolerant distributed algorithms and other related fields. The organizational committee for the Workshop consisted of

E. Gafni	(UCLA, Los Angeles),
M. Raynal	(IRISA, University of Rennes),
N. Santoro	(Carleton University, Ottawa),
J. van Leeuwen	(University of Utrecht),
S. Zaks	(the Technion, Haifa).

Out of the submissions the organizational committee selected twenty-nine papers for presentation in the Workshop. The selection reflects several current directions of research that are representative for the area of distributed algorithms, although certainly not all aspects could be covered in the three-day Workshop. The participants received draft versions of all papers as part of their Workshop materials, by way of working proceedings for review.

The present volume contains the revised version of all papers presented in the Workshop, together with one additional paper (by Y.Afek et al.) that was presented as a general colloquium (by B. Awerbuch) at the Centre for Mathematics and Computer Science (CWI) outside of the Workshop program. The revised versions are based on the comments and suggestions received by the authors during and after the Workshop. Several papers are in the form of preliminary reports on continuing research, and it is expected that more elaborate versions will eventually appear in standard scientific journals. While the papers presented in this volume were primarily selected to serve the purposes of the Workshop, we hope that the papers give a good impression of the current work in distributed algorithms and stimulate further research.

The organizational committee is grateful to the Centre for Mathematics and Computer Science in Amsterdam for allowing the use of its excellent facilities for the Workshop, to the Departments of Computer Science at Carleton University, the University of Utrecht, the University of Rennes, the University of California at Los Angeles, and the Technion in Haifa for assisting in and supporting the organization of the Workshop, and to Geraldine Leebeek (Dept. of Computer Science, University of Utrecht) for her invaluable assistance in all matters related to the Workshop.

J. van Leeuwen (Organizational Chairman)

Utrecht, March 1988.

CONTENTS

A Distributed Spanning Tree Algorithm

Karl Erik Johansen, Ulla Lundin Jørgensen, Svend Hauge Nielsen,
Søren Erik Nielsen and Sven Skyum
Computer Science Department, Aarhus University,
DK-8000 Aarhus C, Denmark

Abstract

We present a distributed algorithm for constructing a spanning tree for connected undirected graphs. Nodes correspond to processors and edges correspond to two way channels. Each processor has initially a distinct identity and all processors perform the same algorithm. Computation as well as communication is asyncronous. The total number of messages sent during a construction of a spanning tree is at most 2E+3NlogN. The maximal message size is loglogN+log(maxid)+3, where maxid is the maximal processor identity.

1. Introduction

Construction of spanning trees for communication graphs has proven useful and has been considered in a number of papers ([2, 3, 6, 7, 8, 9]). Other problems such as termination, extrema finding, and election of a leader all reduce to spanning trees. Most papers on spanning trees deals with construction of minimum spanning trees. Santoro has shown that O(E+NlogN) is a lowerbound on the message-complexity for the problem ([9]). The best known upper bound is 2E+5NlogN ([2]). Recently Lavellee and Roucairol have presented an algorithm that constructs a spanning tree in a general network with message complexity N-1+3NlogN provided the algorithm behaves in a balanced manner ([8]). Their worst-case complexity is again 2E+5NlogN and their message-size is very large. Finally Korach, Moran and Zaks have shown that finding a spanning tree in a complete graph might be easier than finding a minimal spanning tree ([7]). In this paper we present an algorithm, having worst-case complexity 2E+3NlogN, that constructs a spanning tree in an arbitrary network of processors. The construction is very similar to a construction due to the first four authors of this paper ([4])[1], which in turn was inspired by a spanning tree algorithm for complete graphs presented in ([6]) by Korach, Moran and Zaks.

The algorithm is based on the commonly used model. We consider an undirected connected graph without selfloops. Each node corresponds to a processor with unique identity, and each edge corresponds to a two way channel. Each channel has (input) buffers at either

[1]In ([5]) Korach and Markowitz have independently presented a 2E+4NlogN algorithm for the same problem.

endpoint organized as queues. Processors can send and receive messages via channels. The communication is asyncronous. Messages on a channel are received at an input buffer for a processor in the order they are sent from the neighbour, they 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 based on a finite state machine where states have a little memory. In the papers referenced above processors can either wake up spontaneously or be waked up by receiving messages. This difference in presentation makes no "visible" difference in the behavior, since messages can be delayed arbitrarily.

Section 2 contains various concepts and notions needed to explain the algorithm. In Section 3 an overall description of the algorithm is given. The algorithm is given in details in Section 4. Section 5 contains the proof of correctness while Section 6 contains the analysis of the algorithm.

2. Definitions and notations

Let $G=(V(G),E(G))$ be an undirected connected graph without selfloops. We will use $\{.,.\}$ to denote undirected edges and $(.,.)$ to denote directed edges. Each node v in G corresponds to a processor with the unique identity $id_v=v$. $V(G)\subset\{1,2, \ldots\}$. Let $N=|V(G)|$ and $E=|E(G)|$.

A **fragment** is a connected subgraph, $F=(V(F),E(F))$ of G, with no cycles (an undirected tree). A **spanning tree** for G is a fragment F, where $V(F)=V(G)$. A **spanning forrest** is a set $(F_1,F_2,\ldots,F_k)$ of fragments, such that $V(F_i) \cap V(F_j)=\emptyset$ for $i\neq j$ and $V(G)=V(F_1) \cup V(F_2) \cup \ldots \cup V(F_k)$. Given a fragment $F=(V(F),E(F))$ and a node v in V(F) let $F_v=(V(F),D(F_v))$ denote the rooted tree where v is the root, and edges in $D(F_v)$ are directed towards the root v. Each fragment F is equipped with two not necessarily different orientations by naming two roots. One is called the **centre** of F and denoted **c(F)**. If (v,w) is an edge in $D(F_{c(F)})$ then {v,w} in E(F) will be called an **in-edge for** v and an **out-edge for** w. The other is called the **king** of F and denoted **k(F)**. If (v,w) is an edge in $D(F_{k(F)})$ then we call {v,w} an **up-edge for** v and a **down-edge for** w. Each fragment F has a unique identification **<level,k(F)>** called the **colour** of F. level will be an integer in [0,logN].

3. Description of the algorithm

Initially each node v in the network is in a **sleeping** state. It wakes up spontaneously and enters after initialization an **idle** state. Then it constitutes a fragment of size 1 at level 0. The colour is <0,v> and v is both the centre and the king. During execution of the algorithm each centre in the idle state attempts to send a **request**(colour)-message along one of its unprocessed edges in order to combine fragments into larger fragments . If a centre has no adjacent unprocessed edge, the centre is moved around in a fragment F in a depth-first fashion in $F_{k(F)}$ by sending **movecentre**-messages until an unprocessed adjacent edge is found or the algorithm terminates. Assume that we have two fragments F_1 and F_2 with colours $<L_1,k(F_1)>$ and $<L_2,k(F_2)>$. Colours of nodes in F_i are then at most, but not necessarily equal to $<L_i,k(F_i)>$. During computation they will all receive the colour $<L_i,k(F_i)>$ sooner or later. Assume furthermore that the centre c_1 in F_1 sends a request($<L_1,k(F_1)>$) to a node in F_2 along edge e (see Figure 1). After sending the request c_1 enters state **waiting_for_accept**. The request is routed in the direction of the centre c_2 in F_2 as long as $<L_1,k(F_1)>$ is greater than the colour of the nodes it passes. Nodes which pass

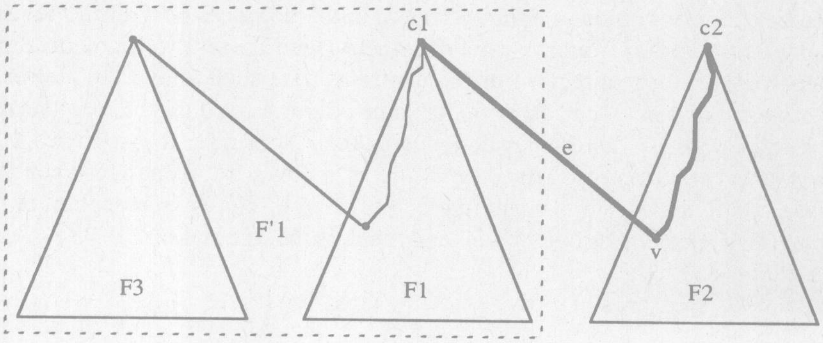

Figure 1. Connecting fragments.

on request-messages also enter the waiting_for_accept state and then only listen to the in-edge for an **accept**- or **newcolour**-message (see later). If a request arrives at a node with greater colour, the node does not react on the request. In that case nodes which passed on that request-message will receive a new colour and reenter the idle state. If the request reaches the centre c_2, call the route (in both directions) along which the request came for the **request-route**. If $<L_2,k(F_2)>$ is less than $<L_1,k(F_1)>$ then F_2 and what has become of F_1 (see later) are to be combined into a larger fragment F. To perform a combination c_2 sends an accept(L_2)-message back through F_2 along the request-route. Nodes on the route, including c_2 itself, enter state **waiting_for_new_colour**. When c_1 receives the accept-message it might have become centre in a larger fragment, than it was when it sent the request to F_2. It might also be in state waiting_for_new_colour in which case F_1 is in the process of being combined with yet another fragment F_3 and will receive a new colour in connection with that combination. In both cases c_1 has sent an accept-message after sending the request-message to F_2. This possibility is necessary to prevent dead-locks. If c_1 is waiting for a new colour, it waits until it has received a new colour before it goes on, otherwise it immediately initiates the final part of the combination of F'_1 and F_2, where F'_1 is the present fragment containing c_1 as its centre. The combination of F'_1 and F_2 will be a fragment F consisting of F'_1, F_2 and the edge e along which c_1 sent the request to F_2. The cen-

tre for F will be c_2, the king will be $k(F'_1)$ and the new level L will be L'_1, if L'_1 is greater than L_2, and L'_1+1 otherwise. (L'_1 is the level of F'_1). c_1 stops being a centre and starts colouring F_2 by sending a newcolour($<L,k(F'_1)>$)-message along e. The newcolour-message is broadcast to all nodes in F_2 via tree-edges. During colouring of F_2 the orientation with respect to up and down in $F_{k(F)}$ is updated. When c_2 receives its new colour, it reenters the idle state and we say that F has been formed. If $<L,k(F'_1)>$ differs from c_1's colour, F'_1 is also coloured by sending newcolour-messages. The reason for choosing $k(F'_1)$ as king for F is that nodes in F'_1 do not necessarily receive a new colour in connection with the combination.
If a centre c in a fragment F sends a request along an unprocessed edge e to a node v in the same fragment, then v's colour might be less than the colour sent with the request and v cannot know that it comes from the same fragment (see Figure 2). The request might therefore be forwarded in the direction of the centre as described above. Sooner or later the request-message will meet a node u with the same colour as c (it might be c itself) and will not be sent any further. When v later on receives the colour of c, it recognizes that the request it forwarded was received from the centre of its own fragment and it sends a close-message back along the edge e to c and marks the edge closed. Upon reception of the close-message c marks the edge e closed as well and reenters the idle state.

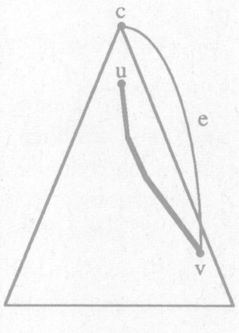

Figure 2. The route followed by a request from a centre to a node in its own fragment

The preceding description gives the overall picture of the algorithm but due to parallelism, messages might cross each other, which adds tedious details to the algorithm.

4. The algorithm

During computation, nodes (or processors) mark their adjacent edges (or channels) with attributes. More attributes can be attached to each edge and the attributes at the two endpoints might differ. The possible attributes are:

open. All edges adjacent to a node are marked open by that node when it wakes up. Messages will only be sent along open edges. (Edges which are only marked open have been and will be referred to as unprocessed).
closed. No messages will be sent along closed edges.
branch. A node marking an edge, with the attribute branch, knows that the edge is part of the fragment and will be part of the final spanning tree. (Edges marked branch at both ends have been and will be referred to as tree-edges).
in, out, up, and **down**. The edge is an in-edge (out-, up-, or down-edge resp. See Section 2). Only branch-edges will be attached those attributes.

Nodes can send a number of different messages along open edges. There are five different message types that can be send, namely:

request, accept, close, newcolour, and **movecentre**. Request and newcolour carry a colour as parameter, accept has a level as parameter, while close and movecentre are without parameters.

Nodes can be in a number of different states <st_1,st_2>. Apart from the various attributes attached to edges mentioned above, only information about at most one edge and one colour has to be stored in a node. We have chosen to store that information within the states. st_1 is either **centre** or **ordinary** while st_2 is one of the following:

sleeping. Initially all nodes are sleeping.
waiting_for_accept(colour,e). The node is a centre and has sent a request(colour) along e or it is not a centre and has forwarded a request(colour) which was received along e.
waiting_for_new_colour(e). The node has sent (or forwarded) an accept-message along e and waits for a new colour.
terminated. The node knows all its adjacent tree-edges and has finished its participation in the distributed computation.
idle. The node takes part in the computation, but is in none of the former four states.

At termination of the distributed algorithm (all nodes are in state terminated) all edges will be marked closed at both ends. Edges will be either tree-edges or not being marked branch in either endpoint in which case they are called cross-edges.

Each node or processor v in state <st_1,st_2> executes the following algorithm. The fragment including v is referred to by F and has colour=<L,id>. A(v) denotes the adjacent edges. The edges in A(v) are ordered such that, if we choose an edge with a specific property, it is assumed that we always choose the first edge in the ordering with that property. This is important when requests are sent from idle centres.

```
repeat
case <$st_1$,$st_2$> of
<*,sleeping> :
  <$st_1$,$st_2$>:= <centre,idle>; colour:=<0,v>;   for all e in A(v) do mark(e):={open} od;

<centre,idle> , <centre,waiting_for_accept(colour,e)> :
  if inputbuffer for an edge $e_1$ in A(v) is nonempty then m:=read buffer $e_1$;
    case m of
    close: {if $st_2$ = waiting_for_accept then $e_1$=e}
      mark($e_1$):=mark($e_1$)-{open}+{closed};   $st_2$:=idle;
    request($colour_1$):
      if colour < $colour_1$ then send accept(L) along $e_1$;
        $st_2$:=waiting_for_new_colour($e_1$) fi;
    accept($L_1$): {if $st_2$ = waiting_for_accept then $e_1$=e}
      if $L_1$ = L then L:=L+1; colour:=<L,id>;
        for all $e_2$ in A(v) where out in mark($e_2$) do
          send newcolour(colour) along $e_2$ od fi;
      send newcolour(<L,id>) along e; mark(e):={open,down,in,branch};
      <$st_1$,$st_2$>:=<ordinary,idle>;
    endcase
  else if $st_2$=idle then
    if there is an e in A(v) where mark(e)={open} then
      send request(colour) along e;   $st_2$:=waiting_for_accept(colour,e)
    else if there is an e in A(v) where {open,down} < mark(e) then
        send movecentre along e;   mark(e):=mark(e)-{out}+{in};   $st_1$:=ordinary
      else if there is an e in A(v) where {open,up} < mark(e) then
          send movecentre along e;   mark(e):=mark(e)-{open,out}+{closed,in};
          <$st_1$,$st_2$>:=<ordinary,terminated>
        else $st_2$:=terminated  fi fi fi fi;

<*,waiting_for_new_colour(e)>:
  if inputbuffer for e is nonempty then m:=read buffer e;
    case m of
    newcolour($colour_1$):
      colour:=$colour_1$; mark(e):={open,up,out,branch};
      for all $e_1$≠e in A(v) where branch in mark $e_1$ do
        send newcolour(colour) along $e_1$;
        if up in mark($e_1$) then mark($e_1$):=mark($e_1$)-{up}+{down} fi od;   $st_2$:=idle;
    request($colour_1$): {skip};
    endcase fi;
```

```
<ordinary,idle> :
  if an inputbuffer for an edge e in A(v) is nonempty then m:=read buffer e;
    case m of
    request(colour$_1$) :
      if colour < colour$_1$ then
        for the in-edge e$_1$ in A(v) do
          if e$\neq$e$_1$ then send request(colour$_1$) along e$_1$;
            st$_2$:=waiting_for_accept(colour$_1$,e) fi od
      else if (colour = colour$_1$) and mark(e) = {open} then
          send close along e;   mark(e):= {closed} fi fi;
    newcolour(<L$_1$,id$_1$>) :
      colour:=<L$_1$,id$_1$>;
      if (down in mark(e)) and (id$_1$$\neq$id) then mark(e):=mark(e)-{down}+{up} fi;
      for all e$_1$$\neq$e in A(v) where branch in mark(e) do
        send newcolour(colour) along e$_1$; if (up in mark(e$_1$)) and (id$_1$$\neq$id) then
          mark(e$_1$):=mark(e$_1$)-{up}+{down} fi od;
    movecentre :
      if down in mark(e) then mark(e):=mark(e)-{open}+{closed} fi;
      mark(e):=mark(e)-{in}+{out};   st$_1$:=centre
    endcase fi;

<ordinary,waiting_for_accept(colour$_1$,e)> :
  if the inputbuffer for the in-edge e$_1$ in A(v) is nonempty then m:=read buffer e$_1$;
    case m of
    request(colour$_2$): {This might occur if v has just been a centre - skip}
    accept(L) :
      send accept(L) along e;   st$_2$:=waiting_for_new_colour(e)
    newcolour(colour$_2$) :
      if colour$_1$ <= colour$_2$ then st$_2$:=idle;
        if (colour$_1$= colour$_2$ ) and mark(e)={open} then
        send close along e;   mark(e):={closed} fi fi;
      for all out-edges e$_2$ in A(v) do send newcolour(colour$_2$) along e$_2$;
        if (up in mark(e$_2$)) and (id$_2$$\neq$id) then
          mark(e$_2$):=mark(e$_2$)-{up}+{down} fi od;
      colour:=colour$_2$;
    movecentre:
      if down in mark(e$_1$) then mark(e):=mark(e)-{open}+{closed} fi;
      mark(e):=mark(e)-{in}+{out};   send accept(L) along e;
      <st$_1$,st$_2$>:=<centre,waiting_for_new_colour(e)>;
    endcase fi;

endcase until st$_2$=terminated;
```

5. Correctness of the algorithm

For each node at most one message is read during an execution of a cycle of the algorithm.We may therefore w.l.o.g. assume that time is discrete (-N,-N+1,...,0,1,2,....) and exactly one node executes one cycle of the algorithm for each time instance t. We may furthermore assume that all nodes are awake and that no messages have been read at time 0.

Lemma 5.1

When a node v terminates then all nodes u, which can be reached from v following down-edges (coincides with out-edges), will be terminated as well (and have no open adjacent edges).

Proof

For t=0 no node is terminated, so the Lemma trivially holds true. If a node v terminates at time t, then v has at most one open adjacent edge (an up-edge). Since all down-edges {v,w} are closed v has received a movecentre-message along these edges indicating that "down-neighbours" w are terminated. Thus the Lemma follows by induction in t.

Corollary 5.2

If a node u has an open adjacent edge, a nonterminated centre c (possibly u itself) can be reached from u following in-edges marked open.

Lemma 5.3

If there is more than one centre at time t, then every pair of centres c_1 and c_2 are connected by an open path (a path where all edges are marked open at both endpoints).

Proof

The Lemma holds true for t=0 because the network is connected and all edges are open. The Lemma will remain true from time t to t+1 if the node v executing its cycle at time t does not mark any new edge closed. Therefore assume that the Lemma holds true at time t and that node v closes an edge at time t.

There are four possibilities: (1) v closes an up-edge and terminates, (2) v closes a down-edge {v,w} after receiving a movecentre-message along {v,w}, (3) v closes {v,w} if w is centre in the fragment containing v at time t and v has received a request from w with v's colour or v has received a newcolour-message with the same colour as the colour of an earlier request from w, or (4) v closes {v,w} after receiving a close-message along {v,w}.

Ad (1): By Lemma 5.1 no path of open edges goes through v so the connectivity of centres with respect to open edges is not affected by this operation.

Ad (2): Receiving a movecentre-message along a down-edge {v,w} indicates that w has terminated and again by Lemma 5.1 we get that {v,w} does not contribute to the connectivity of centres.

Ad (3): Before v closes {v,w} at time t, there exists a cycle of open edges containing v and

w. Breaking this cycle does not affect the connectivity either.
Ad (4): Similar to case 3.

Lemma 5.4
No cycle of tree-edges can be formed.

Proof
A cycle could only be formed if a centre c would accept a request that was initiated by c itself. That will never happen since the colour of a node is nondecreasing during computation.

Lemma 5.5
The number of tree-edges equals the difference between the total number of nodes and the number of centres.

Proof
The Lemma holds true initially and creation of a new tree-edge and deletion of a centre happen at the same time instance for a node in state <centre,waiting_for_accept(∗,∗)> after receiving an accept-message.

Theorem 5.6
The algorithm will terminate (all nodes are terminated). At termination all edges are closed and the tree-edges form a spanning tree.

Proof
The analysis of the number of messages sent (see Section 6) implies that the network will reach a stable situation, where no more messages will be sent and no more computing go on. Let the network be stable at time t . Assume that c is a nonterminated centre of maximal colour present in the network at time t, if such one exists. c cannot be idle because an idle centre can execute a cycle of computation in all circumstances. If c is waiting for an accept along {c,v}, then {c,v} would be open and an open path from v to a nonterminated centre c_1 (possibly c) along in-edges would exist (Corollary 5.2). The maximality of c's colour then implies that an accept- or close-message will be sent along {c,v} to c at a later time than t, which is a contradiction. If c is waiting for a new colour, it will eventually receive one, so this is impossible as well. Thus all present centres will be terminated and by Corollary 5.2 all edges will be closed. Lemma 5.4 then implies that exactly one centre exists. It finally follows from Lemmas 5.3 and 5.5 that the set of tree-edges form a spanning tree.

Remark. At the time of termination one node knows that the algorithm is terminated, namely the node terminating into state <centre,terminated>.