DISTRIBUTED ALGORITHMS FOR ELECTION IN UNIDIRECTIONAL AND COMPLETE NETWORKS

Yehuda Afek

November 1985
CSD-850036

UNIVERSITY OF CALIFORNIA

Los Angeles

Distributed Algorithms for Election in Unidirectional and Complete Networks

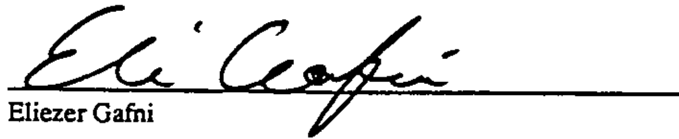
A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in Computer Science

by

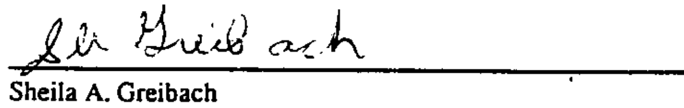
Yehuda Afek

1985

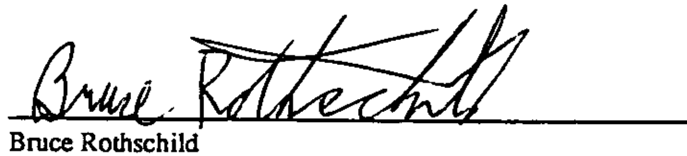
The dissertation of Yehuda Afek is approved.

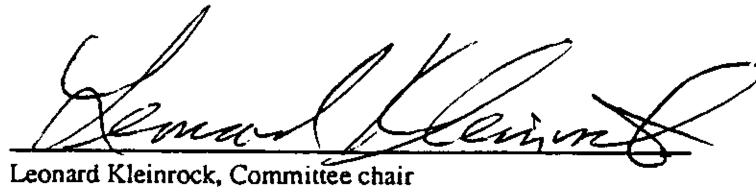

Eliezer Gafni

Mario Gerla


Sheila A. Greibach

James MacQueen


Bruce Rothschild


Leonard Kleinrock, Committee chair

University of California, Los Angeles

1985

Table of Contents

page

1. INTRODUCTION 1
- 1.1. Data Communication Networks 2
- 1.2. Distributed Algorithms 3
- 1.3. Election and Traversal 5
  - 1.3.1. The Election Problem 5
  - 1.3.2. The Traversal Problem 5
- 1.4. Models 6
- 1.5. Performance Measures 10
- 1.6. Previous Work 11
  - 1.6.1. Election in Ring Networks 12
  - 1.6.2. Election in General Networks 14
  - 1.6.3. Election in Complete Networks 17
  - 1.6.4. Election in Unidirectional Networks 17
- 1.7. Dissertation Overview 18

2. ALGORITHMS FOR ELECTION IN COMPLETE NETWORKS 22
- 2.1. Introduction 23
- 2.2. The Synchronous Algorithm 23
  - 2.2.1. Description of the Algorithm 24
  - 2.2.2. Time and Message Complexities 25
- 2.3. Asynchronizing the synchronous algorithm 27
  - 2.3.1. Description of the algorithm 28
  - 2.3.2. Time and Message Complexities 30
- 2.4. Algorithms for Election in Asynchronous Complete Networks 36
  - 2.4.1. Algorithm A 37
  - 2.4.2. Algorithm B 41
  - 2.4.3. Algorithm C 44
- 2.5. Conclusions 48

3. LOWER BOUNDS FOR ELECTION IN COMPLETE NETWORKS 50
- 3.1. Introduction 50
- 3.2. Lower Bounds 51
  - 3.2.1. Definitions and Assumptions 51
  - 3.2.2. A Lower Bound on Message Complexity 53
  - 3.2.3. A Lower Bound on Time Complexity 55
- 3.3. Conclusions 57

4. TRAVERSAL OF UNIDIRECTIONAL NETWORKS 59
- 4.1. Introduction 59
- 4.2. Traversal-1: a simple traversal algorithm 62
- 4.3. Traversal-2: Simulating Directed Depth First Traversal 65
  - 4.3.1. Bidirectional directed depth first traversal 66
  - 4.3.2. Unidirectional depth first traversal, using a spanning in-directed tree 71
  - 4.3.3. On the fly in-tree construction 73
- 4.4. Traversal-3: an algorithm for a network of finite automata 78
  - 4.4.1. Reducing the communication complexity of Traversal-2 to

$O(n \cdot |E| + n^2 \cdot \log n)$ **bits** 79
4.4.2. A finite automata implementation of Traversal-2 82
4.5. Lower Bounds 83
4.6. Applications 84
4.6.1. Producing a spanning out-tree 85
4.6.2. Applications of the traversal algorithm 86
4.6.2.1. Broadcast with Echo 87
4.6.2.2. Messages sending 88
4.6.2.3. Emulating bi-directional distributed algorithms 89

5. ELECTION IN UNIDIRECTIONAL NETWORKS 90
5.1. Introduction 90
5.2. A Unidirectional Election Algorithm 93
5.2.1. Definitions and Outline 93
5.2.2. Selection of a Cluster-Outgoing Link 96
5.2.2.1. Distributed Depth First Traversal of Unidirectional Networks 96
5.2.2.2. Selecting a cluster outgoing link 98
5.2.3. Cycle Detection 99
5.2.4. Cycle Contraction and Cluster Synchronization 100
5.2.4.1. Merging the in-trees 101
5.2.4.2. Acknowledging the broadcast 101
5.2.5. Termination 102
5.3. Complexity of the Election Algorithm 102
5.3.1. Cluster Synchronization Cost 103
5.3.2. Cycle Detection Cost 103
5.3.3. The Cost of Cluster Outgoing Link Selection 104
5.3.3.1. Cluster-Head Election 105
5.4. The Traversal Algorithm as a Special Case of the Election Algorithm 107
5.4.1. Deriving Traversal-2 from the election algorithm 107
5.4.2. Deriving Traversal-3 from the election algorithm 110
5.5. Concluding Remarks 111

References 113

List of Figures

page

Figure 2.1: The Synchronous Algorithm 26

Figure 2.2: The $O(n)$ time scenario 32

Figure 2.3: The waiting chain of lemma 2.1 35

Figure 2.4: Algorithm A 39

Figure 2.5: Algorithm B 43

Figure 4.1: An example for the exponential complexity of Traversal-1 64

Figure 4.2: Traversal-1 65

Figure 4.3: The bidirectional directed depth first traversal algorithm 67

Figure 4.4: The active path 69

Figure 4.5 71

Figure 4.6: Traversal-2, The unidirectional depth first traversal algorithm 76

Figure 4.7 78

Figure 4.8: Backtracking in Traversal-3 80

Figure 4.9: A network for the $\Omega(n \cdot |E|)$ lower bound 84

Figure 4.10: An example of an out-tree 85

Figure 5.1: Clusters in the Election Algorithm 97

Figure 5.2: Rerooting an in-tree 102

Figure 5.3: Clusters in the Traversal Algorithm 109

Acknowledgements

I wish to express my gratitude to Eli Gafni for his collaboration in this work and for sharing with me the excitement of creative thinking. The many hours spent arguing and talking in his office and outside will be valued forever. I am indebted to Leonard Kleinrock, the Chairman of my Dissertation Committee, for providing me with continuous warm encouragement, inspiration, motivation and advice. His guidance for the past three years was essential. I also wish to thank the other members of the committee, Mario Gerla, Sheila Greibach, James MacQueen and Bruce Rothschild for their comments on an early version of the dissertation. In particular, I thank Mario Gerla, in whose class in 1982 I first learned about distributed algorithms, and Sheila Greibach, who offered great constructive criticism on early variations of the work during her seminars. During my studies, my "comrades" and officemates, Joe Green and Hanoch Levy, shared in many discussions, much advice and lots of levity. Special thanks to Lillian Larijani for taking such good care of me at UCLA. If it were not for her help, given through numerous readings of earlier versions and countless administrative operations, this dissertation would not have been possible for a long time. The comments of John Marberg, who read through the material, helped improve the presentation enormously. The unique friendship of Uri Zernik and our interesting and enlightening discussions provided me with a great amount of insight and perspective. The support of the Defense Advanced Research Projects Agency of the Department of Defense under Contract MDA 903-82-C-0064 and the IBM Graduate Fellowship is greatly appreciated. For my wife Liesbeth, who provided me with essential encouragement, security, and love as well as the pleasant distractions of Ofra and Noa, I express my sincere gratitude. Finally, I would like to dedicate this work to my parents, Miriam and Menachem Pinkhof.

VITA

September 30, 1952	Born, Haifa, Israel
1978	B.S., Technion - Israel Institute of Technology
1978-1980	Computer Engineer, Israeli Air Force
1982	M.S, University of California, Los Angeles
1982-1983	Teaching Assistant, University of California, Los Angeles
1983-1985	IBM Graduate Fellowship
1980-1985	Research Assistant, USC/Information Sciences Institute, Los Angeles

PUBLICATIONS

Eli Gafni and Yehuda Afek "Election and Traversal in Unidirectional Networks," *Proceedings, ACM Symposium on Principles of Distributed Computing,* Vancouver, Canada, August 1984.

Yehuda Afek and Eli Gafni "Time and Message Bounds for Election in Synchronous and Asynchronous Complete Networks" *Proceedings, ACM Symposium on Principles of Distributed Computing,* Minacki, Ontario, August 1985

Yehuda Afek and Eli Gafni "Simple and Efficient Distributed Algorithms for Election in Complete Networks," *Proceedings, 22nd Annual Allerton Conference,* 1984.

Eli Gafni, Leonard Kleinrock and Yehuda Afek "Fast and Message Optimal Synchronous Election Algorithm for Complete Networks," CSD-840041, UCLA, Los Angeles, CA 90024 (October 1984).

Mario Gerla, Leonard Kleinrock and Yehuda Afek "A Distributed Routing Algorithm for Unidirectional Networks," *Proceedings, GLOBECOM '83, San Diego,* November 1983.

Yehuda Afek "COCO: Control Units Silicon Compiler," Master's Thesis, University of California, Los Angeles, Computer Science Department, 1982.

ABSTRACT OF THE DISSERTATION

Distributed Algorithms for Election in
Unidirectional and Complete Networks

by

Yehuda Afek

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 1985

Professor Leonard Kleinrock, Chair

Consider a data communication network of n nodes, each of which has a unique identifier (*id*); otherwise the nodes are identical. The nodes are asleep and have no global information about network topology, number and ids of other nodes, etc. A distributed election algorithm is a means by which the nodes of the network distinguish one among them as the *leader*.

The problem of distributively electing a leader in a network is viewed as a problem of synchronization among potential candidates for leadership. Each candidate tries to capture all the nodes. To guarantee that only one succeeds, all but one candidate are killed. Following this view election algorithms in a general, two component framework are designed. Component one is a capturing and termination detection mechanism, assuming only one candidate. Component two is a synchronization mechanism, to eliminate all but one candidate.

In arbitrary networks the synchronization is complicated by the uncertainties of nodes about the network topology and the relative location of candidates. Two network models are considered: first, a complete network in which a bidirectional communication link connects every node with every other, thus eliminating topological uncertainties; and second, the opposite extreme in which topological uncertainties are at maximum -- a strongly connected unidirectional network with some or all links transmitting messages in one direction only.

The study produces an $O(n \cdot \log n)$ messages $O(\log n)$ time *synchronous* and $O(n \cdot \log n)$ messages $O(n)$ time *asynchronous* election algorithm in complete networks. For unidirectional networks we derive a distributed election algorithm whose communication complexity is $O(n \cdot |E| + n^2 \log n)$ bits, where $|E|$ is the total number of links.

We also establish that $\Omega(n \cdot \log n)$ is a lower bound on the total number of messages transmitted for achieving election in synchronous complete networks. Moreover, it is shown that the time complexity of message-optimal synchronous algorithms is $\Omega(\log n)$, hence the optimality of our synchronous complete network algorithm. It remains open whether a sublinear time, message-optimal, asynchronous complete network election algorithm exists.

CHAPTER 1.

INTRODUCTION

The rapid growth of computer networks and their applications in resource sharing, data distribution and exploiting parallelism in complex calculations have increased the demand for distributed network control algorithms. To enable reliable and efficient use of networks, their computers have to be coordinated to cooperatively achieve common global objectives. In an effort to accomplish this, many distributed algorithms have been developed over the last decade.

The solution to the election problem, i.e., the problem of distributively distinguishing one computer from all the others is a basic building block in many distributed algorithms and systems. For example, it is used to replace a faulty coordinating center in distributed algorithms, such as a routing center in routing algorithms, a lock-coordinator in a distributed data-base, or a primary site in a replicated distributed file system.

In this thesis we study distributed algorithms for election in two models of data communication networks. First, we study a complete network in which every node is connected to every other node. This network is, topologically, the most simple one, thus revealing basic principles of distributed election algorithms when the topological uncertainties are removed. Second, we study arbitrary-topology, strongly-connected unidirectional networks in which some or all the links can transmit messages only in one direction. Unlike the complete network, the unidirectional network topology is the most general one, since every other network topology

can be modeled as a unidirectional network. The study of these two models provides insight into the basic elements of the design of distributed algorithms in general, and election algorithms in particular.

1.1. Data Communication Networks

A data communication network (network, in short) consists of a set of autonomous processors (nodes) connected by communication lines (links). Each autonomous computer has its own memory and is capable of carrying out its own local computations regardless of the status of any other computer in the network. Each communication line connects a distinct pair of nodes, thus enabling these two nodes to exchange messages. Message exchange is the only form of communication between the nodes of the network.

An example of a data communication network is the ARPANET whose nodes are dispersed throughout North America and Europe. Some of today's supercomputers, are examples of a network of micro-computers, all situated in one room (e.g. the Cosmic Cube [Sei85]).

Computer networks can be used to facilitate: (1) resource sharing; (2) data distribution and; (3) the exploitation of parallelism in complex calculations. Examples of such tasks are: controlling a telephone system, connecting branches of a bank, distributed data base systems and distributed simulation. To perform these tasks, the nodes of the network are coordinated to achieve cooperation in solving a common problem. The objective of most distributed algorithms is to control and coordinate the nodes of the network. These algorithms are then used as building blocks in the implementation of these distributed systems. Distributed algorithms efficiently solve problems such as: finding all shortest paths in the network, distin-

guishing a unique node from all the others and constructing minimum weight spanning trees. Coordinating the nodes of a network is the major task of distributed algorithms.

1.2. Distributed Algorithms

A *distributed algorithm* is a means by which the nodes of a network cooperatively achieve a common objective. The algorithm itself is a collection of identical programs, one copy in each node of the network. To perform the algorithm, the programs communicate with each other via message exchange. Unlike centralized algorithms, the execution of a distributed algorithm can be started by any subset of the nodes at any time. Although started by a few nodes, the algorithm has a unique objective, and all the nodes participating in the algorithm are coordinated to efficiently achieve that objective. Thus, in a distributed algorithm, each node of the network performs part of the total computation required to achieve the algorithm's objective.

Before a distributed algorithm starts executing, the nodes of the network are assumed to have only local information about the network. Since networks are very large and frequently change, no global knowledge is assumed at any one node. *Initially, no node knows the total number of nodes in the network or the network topology.*

Nodes start their participation in a distributed algorithm in two ways, either by being spontaneously awakened at an arbitrary time, in which case it is called an *initiator*, or by receiving a message of the algorithm. The spontaneously awakened nodes are awakened by an attached host, or a user operator, or some other event which is external to the network.

Unlike centralized algorithms, distributed algorithms exhibit two forms of non-determinism. First, an execution of the algorithm may be started by any subset of the processors at different times. Second, once started, the distributed algorithm progresses in a non-deterministic fashion. At any given time, neither the location nor the time of arrival of the next message is known.

There are two simple, straightforward approaches in the design of distributed algorithms: broadcasting and semi-centralized algorithms. With broadcasting, all information required to solve the problem is broadcast throughout the network. Each node then employs a centralized algorithm to solve the problem. In semi-centralized algorithms, a particular node is selected ahead of time to synchronize and coordinate the processors of the network. This central node collects all the required information about the network, such as the topology, and uses centralized algorithms to solve the problem locally and distribute the results to all the other nodes.

Considering that a new broadcast must be initiated for every node or link failure in order to update the other nodes of the change, the broadcasting solution is impractical. The result of this will be a huge flow of messages, which will degrade the performance of any network, in particular of large networks where a high frequency of failures is expected.

The semi-centralized approach has three drawbacks. First, the central processor becomes a critical element of the network. The correct and reliable operation of the entire network then depends on the reliability, availability and correct functioning of one node. Second, the central processor serializes the operation of the distributed algorithm in an environment specifically intended to support parallelism. Third, the central processor and its neighborhood will be swamped with such messages as topology updates and service requests. The second and third problems

exacerbate each other since, consequently, the whole network operates at the rate of the central processor. Henceforth, we will not consider broadcasting or semi-centralized solutions in this work. Rather, we will consider algorithms in which every node performs only part of the total computation. Before these algorithms start no node is distinguished to play any special role in the algorithm. The amount of communication in the algorithms that we consider is considerably less than that in the broadcasting solution.

1.3. Election and Traversal

This dissertation addresses two problems in computer networking: (1) the problem of distributively electing a leader, and (2) the traversal problem. We present distributed algorithms for solving these problems in complete networks (in which every node is connected to every other node) and unidirectional networks (in which some or all the links can transmit messages only in one direction).

1.3.1. The Election Problem

In the *election* problem, a single node, called the leader, is to be selected from a set of nodes which differ only by their identifiers (*ids*). Initially, no node is aware of all the other ids. In the distributed election algorithm an arbitrary subset of nodes wakes up spontaneously at arbitrary times and starts the algorithm by sending messages over the network. When the message exchange terminates, a leader is distinguished from all other nodes.

1.3.2. The Traversal Problem

In the *traversal* problem, one node, called the *root*, initiates a single process (which can be viewed as a token) which must visit all the nodes in the network, one

at a time. If necessary, the process may traverse any link or visit any node several times.

Since every node is assumed to have knowledge only of its own incident links, a traversal algorithm has (1) to detect the time when it has seen all the nodes, and (2) to efficiently reach the nodes not yet visited. In order to do this, the traversal process marks nodes visited and carries along some information.

Distributed algorithms for traversal and election are strongly related to each other. On the one hand, four of the six election algorithms presented in this dissertation use some sort of traversal algorithm as a building block. On the other hand, if initiated only by one node, these election algorithms are turned into a traversal algorithm. The other two algorithms, when initiated only by one node, are turned into a traversal in which a few nodes are visited simultaneously by different tokens.

A modular technique to design efficient election algorithms on a network for which a traversal algorithm is given was presented by Korach et al. [Kor85]. Applying their algorithm, which was developed independently of this study, to a complete network yields algorithm B of Chapter 2. However, their algorithm lucks the improvements which we have introduced in algorithm C of Chapter 2.

1.4. Models

Distributed algorithms for three different models of data communication networks are presented in this dissertation. The three models are: the *synchronous complete network*, the *asynchronous complete network*, and the *unidirectional strongly connected network*. The three models are based on the traditional message-passing model of data communication networks [Bur80, Lyn81, Seg83]. In this Section we first present the traditional model and then discuss the variations used in the

dissertation.

A computer communication network is a set of n processors (nodes) connected by a set, E, of bidirectional communication lines (links). Each link connects a distinct pair of processors. Each processor has a set of input ports and output ports. Each communication line is modeled by connecting an <output ; input> pair of ports of one processor to an <input ; output> pair of ports of another processor. The following assumptions are made:

1. Associated with each node is a unique identifier number (*id*). We assume that every *id* can be written in, at most, $O(\log n)$[1,2] bits.

2. Within each processor, each input and each output port has a unique port id which is known to the processor. Thus each port can be uniquely identified by its processor.

3. Initially (before the algorithm starts), aside from its port ids each processor knows nothing about the network. In particular, the ids of processors connected on the other side of each link are not known to the processor. Moreover, processors initially have no global knowledge such as the network topology or the total number of nodes.

4. The communication lines are reliable. Messages transmitted over the communication lines incur an unpredictable, but finite, delay and arrive at the input port in the order sent. Queueing delays are included in the messages

---

1– A function of n, $T(n)$, is $O(F(n))$ ("is oh $F(n)$") if there are positive constants c and n_0 such that for $n \geq n_0$, we have $T(n) \leq c \cdot F(n)$.

2– Unless otherwise specified all logarithms are to the base 2. Note that $O(\log n)$ does not depend on the base of the logarithm since $\log_a n = c \cdot \log_b n$, where $c = \log_a b$.

delay.

5. All messages received at a node are stamped with the identification of the port (link) through which they arrived. Messages from all input ports are transferred into a central queue. The processor receiving the messages processes them one at a time in the order that they arrive at the central queue.

6. The processing time of a message is negligible compared to its communication delay.

Several variations on the above model are possible. In particular, we consider the following variations:

1. Communication lines can be either unidirectional or bidirectional.

2. The underlying topology, which is known to the algorithm designer, can be either arbitrary strongly connected unidirectional network topology, or bidirectional complete network topology.

3. The communication mode can be either synchronous, or asynchronous.

Unlike a bidirectional communication link, a *unidirectional* communication link from node v to node u, can carry messages only from v to u. A *unidirectional network* is a network in which some or all the links are unidirectional. In such networks a communication line is modeled by a connection of an <output> port of one processor to an <input> port of another processor. A unidirectional network is called *strongly connected* if there is a directed path from every node in the network to every other node. A unidirectional link from v to u is called an *outgoing* link of v and *incoming* link of u.

In practice, networks in which communication is unidirectional appear in a few forms. For example, due to antenna power differences in packet radio networks, the hearing matrix is not symmetric, i.e., some links are unidirectional [Kah78]. Examples of point to point unidirectional networks are found in fiber optic networks and microwave communication networks.

Two topologies are considered in this work, the complete network, and the strongly connected unidirectional network. In a *complete network* of n nodes, every node is connected by $n-1$ bidirectional communication links to all other nodes. *All the links incident to a given node on which no message was sent or received are indistinguishable to this node.*

Considered here are two modes of communication, *synchronous* and *asynchronous*. In the *synchronous* mode, a global clock is connected to all the nodes in the network. The time interval between two consecutive pulses of the clock is a *round*. At the beginning of each round, each node decides, according to its state, what messages to send and on which links to send them. Each node then receives any messages sent to it in this round and uses the received messages and its state to decide on its next state. Spontaneously awakened nodes start a distributed algorithm by entering an initial state and then waiting for the beginning of the next round. Further variations on the assumptions in the synchronous mode are possible and are discussed in more detail in Chapter 3. In the *asynchronous* mode there is no global clock, and messages incur arbitrary but finite delay.

In Table 1.1 the different combinations of parameters used in this work are summarized. In particular, we consider

(1) synchronous complete bidirectional networks,

(2) asynchronous complete bidirectional networks and

(3) arbitrary-topology strongly-connected unidirectional networks.

Dissertation Summary					
	Model-Parameters				
Ch.	Synchronous/ Asynchronous	Uni-/Bi- Directional	Topology	Problem	Storage bits/node
§2.2	Synch	Bi	Complete	Election	$O(\log n)$
§2.3, 2.4	Asynch	Bi	Complete	Election	$O(\log n)$
§3	Synch	Bi	Complete	Election Lower bound	$O(\log n)$
§4.2, 4.3	Asynch	Uni	Arbitrary	Traversal	$O(\log n)$
§4.4	Asynch	Uni	Arbitrary	Traversal	$O(1)$
§5	Asynch	Uni	Arbitrary	Election	$O(\log n)$

§Chapter.Section

Table 1.1: Models Summary

1.5. Performance Measures

The interesting performance measures of distributed algorithms are: the amount of communication and the amount of time which are required in the execution of the algorithm. Hence, two measures of performance are used to analyze distributed algorithms -- communication complexity, and time complexity.

Communication complexity is the total number of messages sent, in the worst case, by the algorithm. Each message is assumed to contain no more than $O(\log n)$ bits. Thus preventing an algorithm from sending fewer but large messages. $O(\log n)$ is the number of bits which we assume required to represent one node id.

Alternatively we also use *bit complexity* as a measure of communication complexity. The bit complexity of an algorithm is the total number of bits of all messages transmitted by the algorithm, in the worst case.

Time complexity is the worst case length of the time interval from the first to the last message transmission due to the algorithm. As was stated before, processing time is assumed to be zero and therefore we do not consider processing time complexity in this study.

The time complexity of a synchronous algorithm is well defined by the above definition; however the time complexity of an asynchronous algorithm is unbounded, in the worst case, since messages can incur an arbitrary, but finite, delay. To analyze the time complexity of asynchronous algorithms, our assumptions must be modified. Thus, in the asynchronous mode of communication, and only for the purpose of time complexity analysis, we assume that the transmission of a message over any link incurs at most one time unit delay. In arguing the time complexity, we shall allow a message to traverse a link in any fraction of the time unit. This enables us to construct scenarios in which some messages are delivered as fast as we want relative to other messages.

1.6. Previous Work

Distributed algorithms as a solution to the problem of distributively electing a leader first appeared in 1977 in two different topologies. One group of researchers [Dal77, Spi77, Gal77] tackled the problem for arbitrary-topology bidirectional networks, while another group [LeL77, Cha79] treated the problem for both bidirec-

tional and unidirectional rings [1].

1.6.1. Election in Ring Networks

In [LeL77], Le Lann faced the problem of electing a leader in a unidirectional ring while designing a scheme for mutual exclusion in a distributed environment. Le Lann studied a system of n controllers taking turns allocating resources to users. The controllers, each of which has a unique id, are connected in a virtual unidirectional ring. Mutual exclusion among the controllers is achieved by circulating a single token around the ring. The problem is, then, to design an algorithm which will elect one (unique) controller to initiate a new token in case the previous token is lost. Le Lann proposed an $O(n^2)$ message algorithm for the problem.

Following Le Lann's work, Chang and Roberts [Cha79] proposed an algorithm with an average message complexity of $O(n \cdot \log n)$; however, the worst case complexity is $O(n^2)$ messages. Subsequently, Hirschberg and Sinclair [Hir80] gave an $O(n \cdot \log n)$ message, in the worst case, election algorithm for bidirectional rings. Burns [Bur80] proved a lower bound of $\Omega(n \cdot \log n)$[2] messages for election in bidirectional rings. In [Hir80], Hirschberg and Sinclair conjectured that, for unidirectional rings, $\Omega(n^2)$ messages is the lower bound. However, Dolev et al. [Dol82] and Peterson [Pet82] both disproved the conjecture by presenting a sequence of unidirectional algorithms, each improving on the other. The last improvement, given in [Dol82], obtained a $1.356 n \cdot \log n$ message algorithm. A lower bound of

---

1– A *ring* topology is a circular arrangement of processors in which every processor is connected by a link to each of its two neighbors.

2– A function of n, $T(n)$, is $\Omega(F(n))$ ("is omega $F(n)$") if there exists a positive constant c such that $T(n) \geq c \cdot F(n)$ infinitely often (for infinite number of values of n). This definition, taken from [Aho83] , is not symmetric to the big-oh notation. Because an algorithm can be efficient on many but not all values of n. However in this work the symmetric definition would be sufficient, i.e., there exist positive constants c and n_0 such that $T(n) \leq c \cdot F(n)$ for all $n \geq n_0$.

$0.693n \cdot \log n$ messages for unidirectional rings was obtained by Pachl et al [Pac82].

Recently, Frederickson and Lynch [Fre84] addressed the problem of electing a leader in *synchronous* rings. They showed that, for synchronous networks, one should distinguish between two types of algorithms: *general*, in which nodes may perform any computation on the values of their ids; and *comparison*, in which the values of ids can be used only for comparison with each other. On the one hand, they gave an $\Omega(n \cdot \log n)$ message lower bound for comparison algorithms. On the other hand, they presented an $O(n)$ general algorithm (which was also independently discovered by P. Vitanyi [Vit84]), thus showing that general algorithms are strictly more powerful than comparison algorithms in synchronous rings. However, the time complexity of the general algorithm is exponential in the value of the smallest id around the ring. Noticing the discrepancy between the time complexity of the linear message complexity general algorithm and the $O(n \cdot \log n)$ message comparison algorithm, they proved the following relation between three parameters of general election algorithms in a synchronous ring which are: (1) the upper bound on the time complexity, (2) the lower bound on the message complexity, and (3) the size of T, the set of ids from which ids for nodes around the ring are selected. Specifically they proved, that if the time complexity of a general algorithm is upper bounded by d, then there exists T, such that the message complexity of the algorithm is lower bounded by $\Omega(n \cdot \log n)$. The relation they proved is such that the size of T, $|T|$, grows very fast with both d and n. More recently, Gafni [Gaf85] presented an $O(n \log^* n)$ message, $O(G^{-1}(G(n))|T|)$ time general algorithm for election in synchronous rings ($G(n)=\log^* n$)[1], thus improving on the time complexity of Frederickson and Lynch's algorithm.

1– $\log^* n$ is the minimum number of times we have to take log from n to get a number smaller than 1. Alternatively, it is the inverse of the function $F(n)$ which is defined as follows: $F(0) = 1, F(i) = 2^{F(i-1)}$ (i.e., $G(n) = log^* n = F(n)^{-1}$).

1.6.2. Election in General Networks

Fault tolerance and broadcast protocols motivated the development of distributed election algorithms in arbitrary-topology bidirectional networks. In fault tolerance applications, a central controller of a distributed system sometimes need to be chosen to replace a faulty one. For example, in the TYMNET public network [Tym71, Rin77], at any given time there is one node, called SAM (Supervisor in Active Mode), which allocates virtual routes for new sessions between users distributed in the network. Periodically, all the nodes in the network update SAM with the delays on their incident links. SAM uses this information to issue any newly requested virtual routes while keeping the overall delay of the network down. When SAM goes down, a new node is elected to make the routing decisions. Other examples of applications of the election algorithm are situations where a faulty primary site in a replicated distributed file system [Als76] or a faulty lock coordinator in a distributed data base system [Men78] need to be replaced.

Distributed algorithms for spanning tree construction, rather than distributed election, were motivated by the design of efficient broadcast protocols. However, any algorithm to construct a spanning tree can be transformed into an election algorithm by sending $O(n)$ more messages as described below. Hence, any efficient algorithm for spanning tree construction is also an efficient election algorithm.

In a distributed spanning tree algorithm, every node marks some of its incident links; the collection of marked links constitutes a spanning tree. Once a spanning tree is constructed, an election phase is implemented by associating directions with each link in the tree. Every node, for which all incident links but one have already been directed, directs that one link outward. This process starts at the leaves and terminates when two nodes simultaneously try to direct the link connecting them

in opposite directions. The highest id node of these two is then elected as the leader.

In his Ph.D. dissertation, Dalal [Dal77] addressed the problem of distributively constructing a spanning tree in the design of efficient network broadcast protocols. One simple way to broadcast a message in a network is to send a copy of the message over each link; however, in large networks this mechanism could be too costly, in particular when considering frequent broadcastings by different nodes. To reduce the message complexity of broadcasting, Dalal suggested first defining a minimum weight spanning tree (MST) on the network (the link weights being the cost of transmitting one message over each link) and then broadcasting by sending one copy of the message over each link of the MST. Thus, for the price of one MST construction, Dalal reduced the message complexity of broadcasting from $O(|E|)$ to $O(n)$. In his dissertation, Dalal gives a distributed algorithm for MST construction. The message complexity of the algorithm was not analyzed but is believed to be worse than that of more recent algorithms.

Spira [Spi77] followed up on the algorithm of Dalal and obtained a distributed MST algorithm with average message complexity of $O(|E|+n \cdot \log n)$. In [Gal83], Gallager, Humblet and Spira have further improved on the algorithms of [Dal77, Spi77] to obtain an $O(|E|+n \cdot \log n)$ worst case message complexity algorithm.

In [Gal83], every awakened node starts to construct a subtree (fragment) of the MST by iteratively selecting the minimum weight edge adjacent to its already constructed fragment. A variable, called *level*, is associated with each fragment. Whenever two growing fragments meet, the level variables are used to economically combine them into one fragment. The algorithm terminates when the whole network is spanned by one fragment.

The time complexity of the algorithm in [Gal83] is $O(n \cdot \log n)$. Recently, Gafni [Gaf85] further improved the algorithm of Gallager et al. to reduce its time complexity to $O(n \cdot \log^* n)$ by modifying the definition of the level variables and the mechanism by which fragments merge. The algorithm of Gafni has thus established the best known time upper bound for message optimal election algorithms in asynchronous general networks.

While the main goal in [Dal77, Spi77, Gal83] was to construct an MST, an upper bound of $O(|E|+n \cdot \log n)$ messages on the election problem was already established by Gallager in 1977 [Gal77], when he presented an election algorithm with this complexity. In [Gal77], each spontaneously awakened node starts a depth first search process which tries to traverse all the links of the network. When two traversing processes meet, the one which has already visited more nodes kills the other and continues. The depth first search process which survives all the others elects its initiating node as the leader of the network.

$\Omega(|E|)$ is clearly a lower bound for election in asynchronous general networks (also, in rings) since no algorithm may terminate before sending at least one message over each link; otherwise, an untraversed link could be the only link connecting two parts of the network, each holding a separate election. Following [Bur80], $\Omega(n \cdot \log n)$ is also a lower bound. Thus, $\Theta(|E|+n \cdot \log n)$ [1] is both the upper and lower bound on the message complexity of the election in the asynchronous general networks.

1– A function of n, $T(n)$, is $\Theta(F(n))$ ("is theta $F(n)$") if it is both $O(F(n))$ and $\Omega(F(n))$, i.e., there exist positive constants c_1, c_2 and n_0 such that $c_1 \cdot F(n) \le T(n) \le c_2 \cdot F(n)$ for all $n \ge n_0$.

1.6.3. Election in Complete Networks

It was shown by Korach, Moran and Zaks [Kor84] that in complete networks, unlike in rings and general networks, the $\Omega(|E|)$ lower bound does not hold. The reason being that, in a complete network, an election algorithm can be terminated once some node has communicated with all its neighbors. Subsequently, Korach, Moran and Zaks presented a $5 \cdot n \cdot \log n + O(n)$ message $O(n \cdot \log n)$ time algorithm and an $\Omega(n \cdot \log n)$ message lower bound for election in asynchronous complete networks. Their algorithm is essentially the same as the MST algorithm of Gallager et al. [Gal83] with the observation of the simplified termination detection of complete networks.

1.6.4. Election in Unidirectional Networks

Prior to the algorithm given in this dissertation, no algorithm specifically designed for election in strongly connected unidirectional networks has been observed. However, two bidirectional distributed algorithms [Seg83, Gal76] can easily be turned into unidirectional election algorithms. In [Seg83], Segall presents a connectivity checking algorithm upon whose termination every node knows the ids of all the other nodes connected to it. The shortest path algorithm in [Gal76] exhibits the same property when it terminates. The communication complexity of the two algorithms is $O(n \cdot |E| \cdot \log n)$ bits, and each node is assumed to have $O(n \log n)$ bits of memory.

The unidirectional variation of the two algorithms proceeds in two phases: In the first phase, every node acquires the ids of its incoming neighbors; in the second, it acquires the ids of all the other nodes in the network. The details of this algorithm are postponed to the introduction of Chapter 5. The communication complexity of

the algorithm is $O(|E|^2 \cdot \log n)$ bits; however, assuming that messages sent over one link are received in the order transmitted, the communication complexity can be reduced to $O(n \cdot |E| \cdot \log n)$ bits.

In looking for a lower bound on the problem of electing a leader in unidirectional networks Gafni and Korfhage [Gaf84] designed an election algorithm for unidirectional Eulerian networks. The message complexity of their algorithm is $O(|E| \cdot \log n)$.

1.7. Dissertation Overview

In this dissertation we will present distributed algorithms for three different models: complete synchronous networks, complete asynchronous networks and asynchronous strongly-connected unidirectional networks (refer to Table 1.1).

Five algorithms for election in complete networks are presented in **Chapter 2** (see Table 2.1). In Section 2.3, we present a $3 \cdot n \cdot \log n$ message, $O(\log n)$ time synchronous algorithm.

In trying to apply the synchronous algorithm on an asynchronous network, the time complexity degrades to $O(n)$ and its message complexity to $5n \cdot \log n$ (Section 2.4). The asynchronous algorithm is an improvement over the considerably more complicated algorithm in [Kor84], whose time complexity is $O(n \cdot \log n)$ and message complexity is $5 \cdot n \cdot \log n + O(n)$.

In an effort to reduce the message complexity of the asynchronous algorithm to $2n \cdot \log n$ while maintaining its linear time complexity, we present a sequence of three more asynchronous algorithms (A, B and C, Section 2.5). The first two algorithms present tradeoffs between time and message complexities. Algorithm A

(which was also derived independently in [Hum84]) has $O(n)$ time complexity and $2.773 \cdot n \cdot \log n$ message complexity. Algorithm B has $O(n \cdot \log n)$ time complexity but $2 \cdot n \cdot \log n$ message complexity. Analyzing the communication and time complexities of the two algorithms, we derive a third algorithm, algorithm C, whose time complexity is $O(n)$ and communication complexity is $2n \cdot \log n$, an improvement on the $O(n \cdot \log n)$ time and $2n \cdot \log n$ message algorithm of [Pet84]. It remains an open question whether a sublinear-time, message-optimal ($O(n \cdot \log n)$ messages) asynchronous algorithm exists. We conjecture that such an algorithm does not exist, i.e., that the time complexity of any asynchronous message-optimal election algorithm is $\Omega(n)$.

In **Chapter 3** we prove two lower bounds on the problem of electing a leader in synchronous complete networks. First, we prove a lower bound of $\Omega(n \cdot \log n)$ on the message complexity. Second, we prove that any message-optimal synchronous algorithm requires $\Omega(\log n)$ time. In proving these bounds, we do not restrict the type of operations performed by nodes. The bounds thus apply to general algorithms and not just to comparison-based algorithms. This proves that the synchronous algorithm of Chapter 2 is optimal.

We prove that the message complexity of any election algorithm, comparison or general, in a complete synchronous or asynchronous network is $\Theta(n \cdot \log n)$. This proves that, for the problem of election in complete networks (unlike rings), general algorithms are not more powerful than comparison algorithms. The difference between synchronous rings and synchronous complete networks stems from the fact that in a ring all nodes can be distributively awakened with n messages, whereas in the complete network the awakening problem is as hard as the election problem, requiring $\Omega(n \cdot \log n)$ messages. If all the nodes of a complete network could be

awakened with n messages, then a general algorithm could take advantage of the synchronous mode of communication to elect a leader in a linear number of messages by using the principles suggested in [Gaf85].

We also prove an $\Omega(\log n)$ lower bound on the time complexity of any message-optimal election algorithm in synchronous complete networks. Specifically, we show that, if the time complexity of an election algorithm (whether comparison or general) is upper bounded by $\frac{1}{2}\cdot\log_c n$ rounds, then its message complexity is lower bounded by $\Omega(\frac{c-1}{2\cdot\log c} n\cdot\log n)$.

In **Chapter 4** three algorithms for traversal of unidirectional networks, Traversal-1, -2 and -3, are presented. Traversal-1 is simple but inefficient. In many networks, the process of Traversal-1 hops over an exponential number of links before terminating. Traversal-2, which is based on the centralized depth first search algorithm, makes at most $O(n\cdot|E|)$ hops on any network. Furthermore, we show that, in general, $\Omega(n\cdot|E|)$ is a lower bound on the number of hops.

In both Traversal-1 and -2 $O(\log n)$ bits of memory are required at each node and that same amount is carried along with the traversing process (i.e., message size is $O(\log n)$ bits). In some applications, such as VLSI, memory size and message length are restricted, and a question then arises whether a unidirectional traversal could be implemented using only a constant number of bits in every node, and on the traversing process (i.e., in a unidirectional network of finite automata). In Chapter 4 we answer the question in the affirmative by presenting Traversal-3, a traversal algorithm for unidirectional networks of finite automata. Traversal-3 makes at most $O(n\cdot|E|+n^2\cdot\log n)$ hops, which is optimal in the worst case (dense networks, in which $|E|=\Omega(n\log n)$).

Both Traversal-2 and -3 yield two spanning trees, both rooted at the root of the traversal, one an incoming tree and the other an outgoing tree. The structure defined by the union of these two trees is shown to be useful in various applications such as broadcasting, routing and termination detection.

In **Chapter 5** we present a distributed algorithm for election in strongly-connected unidirectional networks. The algorithm distinguishes a single processor from all other processors in the network. The algorithm requires $O(\log n)$ bits of memory in each processor, and its communication complexity is $O(n \cdot |E| + n^2 \log n)$ bits.

As with Traversal-2 and -3, the election algorithm yields two directed spanning trees, both rooted at the elected leader; one an incoming tree and the other an outgoing tree. The algorithm is an improvement on the connectivity checking algorithm of Segall [Seg83] and the shortest path algorithm of Gallager [Gal76], both of which can easily be modified to work on a unidirectional network (see Section 1.6.4). The communication complexity of the two algorithms is $O(n \cdot |E| \cdot \log n)$ bits, and each node is assumed to have $O(n \log n)$ bits of memory. Furthermore, unlike our algorithm, neither Segall's nor Gallager's algorithm provides the spanning trees.

CHAPTER 2.

ALGORITHMS FOR ELECTION IN COMPLETE NETWORKS

In this and the next chapter we address the problem of electing a leader in complete networks. In this chapter five election algorithms for synchronous and asynchronous complete networks are presented (see Table 2.1), while tight lower bounds on the message and time complexity for the synchronous case are given in Chapter 3.

The message complexity of the five algorithms presented in this chapter, is $O(n \cdot \log n)$, where n is the total number of nodes in the network. However, the time complexity of the synchronous algorithm, $O(\log n)$, is considerably better than $O(n)$, the time complexity of the asynchronous algorithms, thus suggesting that the synchronous mode of communication is more powerful than the asynchronous mode.

§	Communication Mode	Messages Complexity	Time Complexity
2	Synchronous	$3 \cdot n \cdot \log n$	$O(\log n)$
3	Asynchronous	$6 \cdot n \cdot \log n$	$O(n)$
4.1	Asynchronous	$2.77 \cdot n \cdot \log n$	$O(n)$
4.2	Asynchronous	$2 \cdot n \cdot \log n$	$O(n \cdot \log n)$
4.3	Asynchronous	$2 \cdot n \cdot \log n$	$O(n)$

Table 2.1

2.1. Introduction

In a complete network every node is connected to all the other nodes. Before the algorithm starts, no node has any information on any of the other nodes. Thus, the incident links of a node, on which no message was sent or received, are indistinguishable.

Consider the following straightforward election algorithm in complete networks. Every initiator starts the algorithm by sending messages, containing its id, to all its neighbors. All the initiators then elect the highest id initiator as the leader. The time complexity of this algorithm is two time units, and its worst case message complexity is $O(n^2)$ (the message complexity is $k \cdot n$ where k is the number of initiators). In Section 2.4 the message complexity of this simple algorithm is reduced to $O(n \cdot \log n)$ by slowing-down the rate at which initiators send messages to their neighbors to one message at a time. However, the reduced rate increases the time complexity of the algorithm to $O(n)$. In Section 2.2 we use the synchronous model of communication to design an $O(\log n)$ time, message-optimal algorithm. This is done by carefully selecting a dynamic rate at which initiators send messages to their neighbors.

2.2. The Synchronous Algorithm

In this section we present a $2 \cdot \log n$ rounds, $3n \cdot \log n$ messages synchronous algorithm. In the next chapter we will show that this algorithm is message optimal and is as fast as a message-optimal algorithm can be.

2.2.1. Description of the Algorithm

The algorithm is initiated by any subset of nodes, each of which is a *candidate* for leadership. Each candidate tries to capture all other nodes by sending messages on all the links incident to it. The candidate that has succeeded in capturing all its neighbors elects itself as the leader. To guarantee that only one node succeeds, all candidates but one are *killed*.

To simplify the algorithm every initiator node spawns two processes, the *candidate* process and the *ordinary* process. The two processes are connected to each other by a bidirectional logical link which behaves like a physical link. A node awakened by receiving a message of the algorithm spawns only an ordinary process. Candidate processes communicate only with ordinary processes and vice versa. Thus, the communication topology is a complete bipartite graph, on one side the candidate processes and on the other side n ordinary processes. Henceforth, the term *candidate* will be applied interchangeably to both the process and its initiating node. All messages received by a node are tagged according to the type of their sending process. Messages received from candidate processes are forwarded to the ordinary process. Messages received from ordinary processes are forwarded to the candidate process.

At every candidate the algorithm proceeds in levels. Every live candidate at level i, $i \geq 0$, tries to capture 2^i new ordinary processes by sending them messages containing its level and id. If in the second round of level i the candidate receives acknowledgments from all the ordinary processes it tries to capture, it proceeds as a candidate to the next level. On the other hand, if not all the acknowledgments are received, the process (and hence the node owning it) is eliminated form candidacy.

Every candidate has a variable called *level* which is incremented by one every two rounds. Every ordinary process has an *owner–level* and an *owner–id* variable which contain the level and id of the highest-level candidate the process has received a message from (level ties are resolved by selecting the highest id). In every round, every ordinary process first increases its owner-level by one, to reflect the owner's actual level, and then inspects the newly received messages to update its owner-level and owner-id if necessary. If an update occurred, the ordinary process acknowledges its new owner.

A formal description of the algorithm is given in Figure 2.1. E is the set of edges incident to a candidate process. Every candidate maintains a list of edges, called *untraversed*, which it has not yet traversed in any direction.

2.2.2. Time and Message Complexities

Let p be the largest id of a candidate from the set of oldest candidates (i.e., whose level is the largest). We observe the following three facts:

> *Fact 1: The owner-level* of every node strictly increases from one round to the next.
>
> *Fact 2 : At* most $\frac{n}{2^{i-1}}$ candidates reach level i, $1 \le i \le \log n$.
>
> *Fact 3 :* $2\log n$ *rounds* after it has started the algorithm, candidate p has captured all the nodes and is elected as the network leader.

Fact 1 follows immediately from the algorithm for ordinary node processes. Fact 3 holds because all the messages of p get acknowledged, and once a node has acknowledged p, it will not acknowledge any other message. Fact 2 follows from

```
Candidate program:
  untraversed ← E
  level ← -1 ;
  Each round do:
    level ← level + 1 ;
    If level is even
    Then
      If untraversed is empty
      Then
        ELECTED, STOP
      Else
        K ← Minimum ( 2^(level/2), | untraversed | ) ;
        Send (level, id) over K links from untraversed, and
        remove these links from untraversed ;
    Else  /* level is odd */
      Receive all acknowledgment type messages
      If received less than K acknowledgments
      Then
        Stop          /* Not a candidate any more */
  End each round.


Ordinary program:
  L* ← nil ;
  owner-level ← -1 ;
  owner-id ← id ;
  Each round do:
    Send an acknowledgment over L* ;
    owner-level ← owner-level + 1 ;
    Receive all candidate messages {(level,id) over link L};
    Let (level*, id*) be the lexicographically largest
    ( level, id) candidate message, and
    L* the link over which it arrived ;
    If (level*, id*)>(owner-level, owner-id)
    Then
      (owner-level, owner-id ) ← (level*, id*) ;
    Else
      L* ← nil ;
  End each round.
```

Figure 2.1: The Synchronous Algorithm

fact 1 and the observation that every ordinary node acknowledges at most one message in which the level is i, $0 \le i \le \log n$, i.e., the sets of 2^{i-1} nodes that are captured by each candidate that has reached level i are disjoint.

Following fact 3, the time complexity of the algorithm is $2\log n$. Since every node sends at most one acknowledgment to a candidate in level i, the total number of acknowledgments is $n \cdot \log n$, each of length $O(1)$ bits. Due to fact 2, the total number of candidate messages is $\sum_{i=1}^{\log n} \frac{n}{2^{i-1}} 2^i = 2n \cdot \log n$, each message containing $\log n + \log\log n$ bits. The total communication complexity is thus $3 \cdot n \cdot \log n$ messages.

A continuum of algorithms can be devised to close the gap between the trivial $O(1)$ time, $O(n^2)$ messages algorithm and the $O(\log n)$ time, $3n \cdot \log n$ messages algorithm. Each algorithm in the continuum is the same as the above, except that a candidate in level i is trying to capture c^i neighbors, $2 \le c \le n$. The time complexity of the algorithm is $2\log_c n$, and its message complexity is $2c \cdot n \cdot \log_c n$, thus proving that the lower bounds that will be presented in Chapter 3 (Theorem 3.2) are tight.

2.3. Asynchronizing the synchronous algorithm

In this section we apply the synchronous algorithm to an asynchronous complete network. To maintain the $O(n \cdot \log n)$ message complexity in the asynchronous communication mode, we are forced to increase the time complexity to $O(n)$. The increase in the time complexity seems unavoidable and it remains open whether a sublinear time, message-optimal asynchronous algorithm exists.

There are two basic differences between the asynchronous and synchronous modes of communication. First, in the asynchronous mode there is no global clock, and second, messages incur an arbitrary but finite delay. The arbitrary delay of messages (and not the absence of the clock) is the source of the increase in the time complexity of the algorithm. Essentially the synchronous algorithm could work without

a global clock, if all messages incur exactly the same delay (in which also the queueing and processing time are included). In such a model we assume that, if two nodes, P and Q, send messages at the same time to the same two other nodes, u and v, and the message of P arrives at u before the message of Q then, the message of P will arrive before also at v.

To see that a straightforward application of the synchronous algorithm in the asynchronous model will not work consider the following situation: There are two competing candidates, C_1 and C_2, each had already successfully captured one node, and both proceed to capture the nodes v and u at the same time. A message of C_1 was the first to arrive at v which is then captured by C_1, while a message of C_2 was first to arrive at u. Following the rules of the synchronous algorithm, v positively acknowledges only C_1 and u positively acknowledges only C_2. Thus, both candidates are killed since none had all of its messages positively acknowledged. In the following section we present an algorithm which overcomes this and similar problems in the asynchronous case.

2.3.1. Description of the algorithm

As in the synchronous case, the asynchronous algorithm is started at arbitrary times by an arbitrary set of nodes, each of which is a candidate for leadership. Each candidate tries to capture the network by sending messages on all its incident links. To guarantee that only one candidate is elected, all candidates but one are killed. The candidate that has succeeded in capturing all its neighbors is elected as the leader of the network.

The *level* variable of a candidate is a function of the number of nodes that the candidate has already captured. A candidate at level l has already successfully cap-

tured 2^l-1 nodes. As in the synchronous algorithm, every captured node has *owner−level* and *owner−id* variables, which respectively are the highest level among the candidates from which it has received a message, and the id of one of these candidates (which is assumed to own it). The *potential−id* of a captured node is the id of a candidate which tries to capture it. All variables are initially set to nil.

Every candidate at level l, sends $2^{l+1}-1$ messages containing its level and id, $0 \le l < \log n$, to the 2^l-1 nodes it had already captured and on 2^l unused incident links. Unlike the synchronous case, a candidate in this algorithm waits to receive either positive or negative acknowledgment to each of these messages. If $2^{l+1}-1$ positive acknowledgments are received the candidate proceeds to level $l+1$. On the other hand, if any of the messages was acknowledged negatively, the candidate is killed. It then sends *relinquish* messages, containing its id, to all the nodes that it had ever tried to capture, informing them of its elimination from candidacy.

When a message $<level_c, id_c>$ from candidate C arrives at node v whose variables are $owner-level_v$, $owner-id_v$, and $potential-id_v$, we distinguish between three cases:

1. Either $<owner-level_v, owner-id_v>$ or $<owner-level_v, potential-id_v>$ are lexicographically greater than $<level_c, id_c>$, in which case the message is acknowledged negatively.

2. $owner-level_v$ is smaller than $level_c$, in which case candidate C is acknowledged positively, and the $<level_c, id_c>$ pair replaces the $<owner-level_v, owner-id_v>$ pair.

3. $owner-level_v$ equals $level_c$ and id_c is greater than both $owner-id_v$ and the $potential-id_v$. In this case id_c replaces $potential-id_v$ and node v waits for

one of the following two events to occur before acknowledging candidate C:

a. A relinquish message with the same id as $owner\text{-}id_v$ is received, in which case C is acknowledged positively, and the $<level_c, id_c>$ pair replaces the $<owner\text{-}level_v, owner\text{-}id_v>$ pair, or

b. Another message whose $<level, id>$ is lexicographically greater than $<level_c, id_c>$ arrives, in which case C is acknowledged negatively.

Note that in none of the above cases does a node put on hold more than one message. If a message arrives at a node which already holds one message then the lexicographically smaller one is acknowledged negatively (case 3 b).

2.3.2. Time and Message Complexities

First we shall prove that the algorithm is deadlock-free. Candidate C_1 can cause another candidate, C_2, to wait for it only if (1) both try to capture the same node, v, at the same level, and (2) C_1 captures v while C_2 becomes v's potential owner (because $id_{C_1} < id_{C_2}$). Then, C_2 is waiting to get either a positive or a negative acknowledgment from v. Node v will send a positive acknowledgement to C_2 if it receives a relinquish message from C_1. On the other hand, if v receives a higher level message, it will send a negative acknowledgement to C_2. Hence, the ids along any chain of waiting candidates must be increasing and the algorithm is deadlock-free.

To analyze the message complexity of the algorithm we note the following fact:

Fact: The sets of nodes captured at level $l-1$ by candidates which

reach level l, are disjoint.

This fact follows from the observation that at most one candidate from all the candidates which try to capture the same node at level l will proceed to level $l+1$ (this candidate gets a positive acknowledgment and does not send a relinquish at that level). Thus, the maximum number of candidates at level l is $\frac{n}{2^l-1}$. The total number of messages due to capturing attempts is then the number of candidates at level l times $2^{l+1}-1$ times 2, since each message is also acknowledged, i.e., $2 \cdot \sum_{l=1}^{\log n} \frac{n}{2^l-1} \cdot (2^{l+1}-1)$. Similarly, the maximum number of relinquish messages is $\sum_{l=1}^{\log n} \frac{n}{2^l-1} \cdot (2^{l+1}-1)$ which is bounded by $2 \cdot n \cdot \log n$. Hence, the message complexity of the algorithm is bounded by $6n \cdot \log n$.

The total delay of the algorithm is composed of two terms; The delay incurred by capturing messages, and the delay incurred while waiting for relinquish messages (which is the overhead introduced by the asynchronous model of communication). While the former contributes $O(\log n)$ delay, we will show that the latter takes $O(n)$.

To prove that the time complexity of the algorithm is $O(n)$ we first give a scenario which attains this complexity, and then prove that $O(n)$ is also the upper bound on the worst case time complexity.

In the following scenario $\frac{n}{3}$ candidates, $C_1, \ldots, C_{\frac{n}{3}}$, with ids $id_1, \ldots, id_{\frac{n}{3}}$ such that, $id_i < id_{i+1}$, $i=1, \ldots, \frac{n}{3}-1$, try to capture the same two nodes, v and u two time units after each other. The scenario starts with candidates, C_1 and C_2,

each of which has already captured one node and both try to capture nodes, v and u at the same time (see figure 2.2 and Table 2.2).

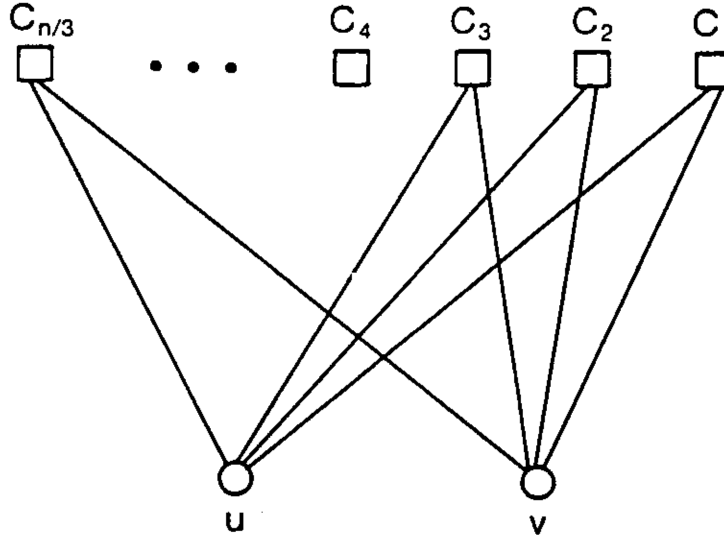


Figure 2.2: The $O(n)$ time scenario

The capturing message of C_1 arrives before the capturing message of C_2 at v, and after at u. Thus, C_2 captures u and becomes the potential owner of v, and C_1 captures v and becomes the potential owner of u. In the next two time units u sends a negative acknowledgment to C_1 which then sends a relinquish message to v. At the same time that C_1 sends the relinquish to v, candidate C_3 sends capturing messages to v and u. The messages are scheduled such that the message of C_3 arrives at v just before the relinquish of C_1. Thus, C_3 becomes the owner of v and the potential owner of u (which is now owned by C_2). In the next two time units v sends a negative acknowledgment to C_2 which then sends a relinquish message to u. But, at the same time that C_2 sends a relinquish message to u, candidate C_4 sends capturing messages to v and u. The scenario proceeds in this pattern for $2/3 \cdot n$ time units at which time all candidates except one are killed. In Table 2.2 the scheduling of the messages in the scenario is given.

Next we prove that $O(n)$ is also the upper bound on the worst case time complexity. To this end we claim that every two time units, either the highest level candidate increments its level by one, or one candidate is effectively eliminated. Since there are at most n candidates, and the highest level is $\log n$, the worst case time complexity is at most $O(n)$.

To prove the claim, consider the first time, T_l, and the time interval, Δ_l, in which l is the highest level in the network i.e., $\Delta_l = T_{l+1} - T_l$.

Time Units	Events
1	C_1 captures v C_2 captures u C_2 becomes v's potential-owner C_1 tries to capture u
2	u sends a negative acknowledgment to C_1
3	C_3 becomes v's potential-owner C_1 relinquishes v C_3 captures v C_3 becomes u's potential-owner
4	v sends a negative acknowledgment to C_2
5	C_4 becomes u's potential-owner C_2 relinquishes u C_4 captures u C_4 becomes v's potential-owner
6	u sends a negative acknowledgment to C_3

Table 2.2: The $O(n)$ time scenario

Clearly, $\sum_{i=0}^{\log n} \Delta_i$ is the time complexity of the algorithm. In the next lemma we argue that if l persists as the highest level in the network for Δ_l time, then a number of candidates linearly proportional to Δ_l have been effectively eliminated during this time. The term "effectively" is used since it might be that the nodes are notified of their elimination some time after T_{l+1}. The extra period of time is also linearly proportional to Δ_l and hence the total time complexity is $O(n)$.

Let N_T denote the number of live candidates in the network at time T. Formally, we claim:

Lemma 2.1: There exist positive constants k_1 and k_2 such that, $N_{T_l+k_1\cdot\Delta_l} \leq N_{T_l} - \left\lceil k_2\cdot\Delta_l - 1 \right\rceil + U_{\Delta_l}$, where U_Δ is the number of initiators (candidates) which start the algorithm in time interval Δ.

Proof: Basically we claim that for every two time units in Δ_l (where a time unit is the maximum delay of a message), at least one candidate at level l is either killed, or added to a chain of waiting candidates (a similar chain was used in the argument that the algorithm is deadlock-free). If a long chain is created, then within some time from T_{l+1} at least half the candidates in the chain are killed. The constant k_1 represents the time it takes the chain to unfold with at least half the candidates killed. The time it takes the chain to unfold is at most the time it takes a message to pass along the chain. If k_1=1 the time complexity is reduced by at most half, and thus we will henceforth assume for convenience that k_1=1. The constant k_2 represents two parameters: The rate at which the chain of either waiting, or dead candidates is created; and the fraction (at least half) of candidates in the above chain which get killed. By appropriately scaling the time units we may assume without loss of generality (w.l.o.g.) that k_2=1 as well.

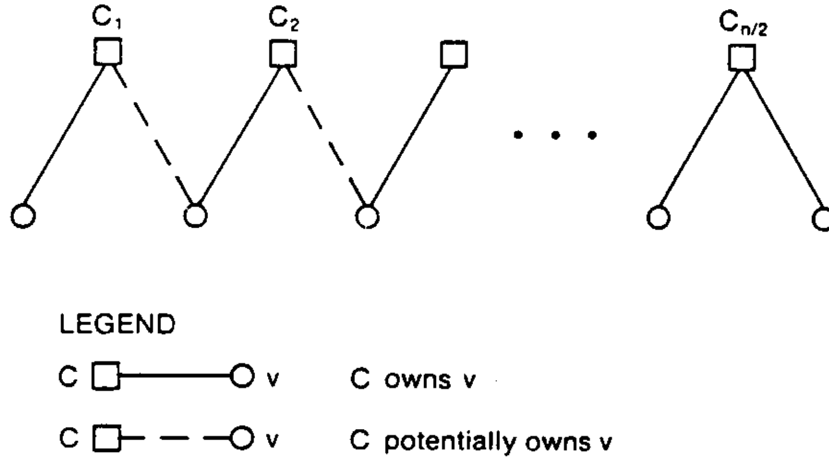


Figure 2.3: The waiting chain of lemma 2.1

Let us now prove that for every two time units in Δ_l at least one candidate is either killed, or added to a chain of waiting candidates. If Δ_l=2 we are done. Assume Δ_l>2, and let C_1 be the first candidate to reach level l at time T_l. Then, at time T_l+2 there must have been candidate C_2 such that C_2 has captured a node which C_1 tries to capture, and either C_2 already caused the death of C_1, or C_1 is waiting for C_2 to relinquish, or to advance to level l+1 (see figure 2.3). In the former case the id of C_1 is smaller than that of C_2 while in the latter it is larger. The proof thus continues inductively by adding for each two time units in Δ_l another candidate to a chain of either waiting or killed candidates.

When a chain of waiting candidates unfolds, at least half of the candidates along the chain are killed. Since, if candidate C_1 is waiting for C_2, either C_2 relinquishes, in which case C_1 advances to level l+1 (unless it is waiting for another candidate), or C_2 advances, in which case C_1 gets a negative acknowledgment from the node they both tried to capture, and C_1 relinquishes. ■

2.4. Algorithms for Election in Asynchronous Complete Networks

Our aim in this section is to derive a $2 \cdot n \cdot \log n + O(n)$ messages, linear-time asynchronous algorithm. To this end we present a sequence of three asynchronous algorithms (A, B, and C), each devised to circumvent the problems of the previous, so that algorithm C achieves the desired complexity.

The underlying mechanism for all three algorithms is similar. Each algorithm is initiated by any subset of nodes, each of which is a *candidate* for leadership. Each candidate spawns a process which tries to capture all the other nodes by successfully traversing in both directions all the links incident to its initiator. The term *candidate* will be applied interchangeably to both the process and its initiating node. The candidate which has succeeded in capturing all its neighbors becomes the leader. To guarantee that only one node is elected, all candidates but one are *killed*.

All candidates use a variable called *level* to estimate the number of nodes they have already captured. The level variable is used by candidates to contest each other. Captured nodes also have a level variable, which tracks the highest level candidate they have observed. All level variables are initialized to 0.

A candidate that arrives at a node with a larger level than its own is eliminated from candidacy. However, if the candidate's level is larger or equal, the node's level is replaced by the candidate's level. The candidate may then claim the node and try to eliminate the previous owner of the node. Upon being killed, the initiating node of a candidate functions like a regular captured node.

The three algorithms differ mainly in two parts: (1) The way that candidates determine their level, and (2) The rule candidates use to eliminate each other. In algorithm A, the level of a candidate is the number of nodes it has already captured

(following [Gal77].) In algorithm B, the level is the number of candidates it has killed. Algorithm A achieves a better time complexity while algorithm B achieves a better message complexity. In algorithm C, candidates use a combination of the above two level functions to attain the time complexity of A and the message complexity of B.

2.4.1. Algorithm A

Level: In this algorithm, the level of a candidate is the *number of nodes it has already captured.*

Capturing and Elimination Rule: To capture node v; (1) the (level, id) of a candidate must be lexicographically larger than the (level, id) of the previous owner of v, and (2) the previous owner must be killed.

When candidate P arrives at node v which is currently owned by candidate Q, the following rule is used:

If $(Level(P), id(P)) < (Level(v), id(Q))$, P is killed.

If $(Level(P), id(P)) > (Level(v), id(Q))$, (1) v gets P's level, and (2) P is sent to Q.

When P arrives at Q:

If $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed.

If Q has been killed already then P captures v.

If $(Level(P), id(P)) > (Level(Q), id(Q))$, then (1) Q is killed, and (2) P captures v.

Upon returning to its initiating node from a successful capturing, P increases its level by one.

Details

To keep track of its owning candidate, every captured node has two link pointers, *father* and *potential−father*. The father pointer points to the link through which the node was most recently captured, and the potential-father pointer points to the link through which a candidate which tries to claim the node from its father, has arrived.

A candidate, C, that arrives at an already captured node v whose level is smaller than its own, replaces v's level with its own and becomes v's *potential−father*. C is then sent to the father candidate of v. If C survives at v's father, and meanwhile no other candidate replaced C as the potential-father of v, then C becomes v's father. If v has not yet been captured, the potential-father automatically becomes the father of v.

A formal description of algorithm A is given in Figure 2.4. As in the synchronous algorithm every initiator spawns two independent processes, *candidate* and *ordinary*. The two processes are connected by a bidirectional logical link which behaves like a physical link.

Analysis

The algorithm is deadlock-free since candidates never wait for each other, and the (*level*, *id*) pair is lexicographically increasing along any chain of candidates which kill each other.

The time complexity of the algorithm is $O(n)$ since candidates never wait for each other and a candidate which has done more work is never killed by a candidate which has done less work. Thus, each killed candidate spent, in the worst case, less

```
Initially
level ← owner-id ← 0 ;
untraversed ← E ; father ←nil ;

Candidate ( id ) :
 while ( untraversed ≠ ∅ ) do;
      l ← any( untraversed ) ;
      send(id,level) on l ;
 R:   receive(id',level') over l' ;
      if (id' = id) then          /* successful capturing.*/
           level ← level + 1 ;
           untraversed ← untraversed - l ;
      else      /*another candidate tries to eliminate candidate id */
          if (level',id' < level,id)      /* lexicographically*/
          then  Discard the message, goto R ;
          else  /* Candidate id is eliminated */
                (1) send(id',level') over l';
                (2) discard all future messages;
end while;
announce(ELECTED,terminate the algorithm) , STOP ;

Ordinary:
for_ever do;
   receive(id',level') over l' ;
   case level', id' of :
   (1) level',id' < level,owner-id:
          Discard message ;
   (2) level',id' > level,owner-id :
          potential-father ← l' ;
          level ← level' ;
          owner-id ← id' ;
          if father = nil then father ←potential-father ;
          send(id',level') over the father link ;
   (3) level', id' = level,owner-id:
          father ←potential-father ;
          send(id', level') over the father link ;
   end case ;
end for_ever ;
```

Figure 2.4: Algorithm A

time than the one killing it.

To prove that the communication complexity of the algorithm is $O(n \cdot \log n)$ we use a Lemma which was introduced in [Gal77].

Lemma 2.2: For any given k, the number of candidates that own $\frac{n}{k}$ or more nodes

is at most k.

Proof: Let C_1 and C_2 be any two candidates which owned $\frac{n}{k}$ nodes at some point of time. We shall show that each of C_1 and C_2 must have owned at least $\frac{n}{k}$ nodes disjointly. If they never tried to claim a node from each other, we are done. The first time that C_1 (w.l.o.g.) tries to claim a node, say v, from C_2, either it causes the death of one of them, or C_2 has been already killed. If C_1, w.l.o.g., caused the death of C_2 then clearly it must have owned at least $\frac{n}{k}$ nodes disjoint from C_2, at the time of killing. If C_2 is already dead, C_1 must still own at least $\frac{n}{k}$ nodes in order to claim v to itself. ■

Corollary 2.1: The largest candidate to be killed by another candidate owns at most $\frac{n}{2}$ nodes, the next largest owns at most $\frac{n}{3}$ nodes, etc.

Lemma 2.3: The message complexity of algorithm A is $4 \cdot n \cdot \ln n$ ($= 2.773 \cdot n \cdot \log_2 n$) messages.

Proof: Since in capturing one node a candidate makes at most 4 hops, a candidate which owned k nodes incurs at most $4 \cdot k$ messages. By Corollary 2.1, the total cost is then bounded by $4 \cdot n \cdot \sum_{i=1}^{n} \frac{1}{i}$ messages. Note that each message of the algorithm contains at most $2 \cdot \log n$ bits. ■

The number of candidates at a particular level was constrained by the disjointness property. Hence, a candidate which captures many nodes from another candidate, tries to eliminate that other candidate as many times as the number of nodes it captures from it. This gives rise to the factor 4 in the message complexity.

In the next algorithm we remove the disjointness requirement and change the level function to reduce the message complexity to $2 \cdot n \cdot \log n$ messages.

2.4.2. Algorithm B

Level: In this algorithm, the level of a candidate is the *total number of other candidates that it has killed.*

Capturing and Elimination Rule: To capture node v the level of a candidate must be strictly larger than that of v, in which case the candidate captures v without killing the previous owner of v.

When candidate P arrives at node v which is currently owned by candidate Q, the following rule is used:

If $Level(P) < Level(v)$, P is killed.

If $Level(P) > Level(v)$, v is captured by P, and v gets P's level.

If $Level(P) = Level(v)$, P is sent to Q.

Upon arriving to Q:

If $(Level(P), id(P)) < (Level(Q), id(Q))$, P is killed.

If Q has already been killed, P is killed too.

If $(Level(P), id(P)) > (Level(Q), id(Q))$, then (1) Q is killed, (2) P increases its level by one, and (3) P captures v.

Details

A formal description of algorithm B is given in Figure 2.5. When a candidate arrives at node v whose level is the same as its own, and the id of v's father, Q, is smaller, it becomes v's potential-father. The potential-father is then sent to Q in an attempt to kill it. If another candidate at the same level with even higher id

arrives at v before the potential-father returns from Q, then this other candidate is killed. If the potential-father survives at Q it first increments its level by one, then returns to v and captures it, and only then, returns to its initiating node. However, if the potential-father finds that Q is already killed, it eliminates itself as well (since if Q was killed, there exists a higher level candidate in the network).

Analysis

Since at most half of the candidates at level k go up to level $k+1$, the maximum level achievable during the algorithm is $\log n$. Clearly, every time a node is recaptured its level is increased by at least one. Hence, the total number of capture messages possible is at most $n \cdot \log n$. Each capture uses 2 messages, which sums up to a total of $2 \cdot n \cdot \log n$ messages. The extra messages spent by candidates which go over father links to other candidates is at most $2 \cdot n$, since each such traversal results in the elimination of one live candidate. Thus, the message complexity of the algorithm is $2 \cdot n \cdot \log n + 2 \cdot n$ messages, each of length $\log n + \log \log n$ bits.

The time complexity of the algorithm is $O(n \cdot \log n)$ by the following scenario, in which $\frac{n}{2}$ of the nodes are captured serially $\log(\frac{n}{2})$ times. The algorithm is started by node v_0 which captures $\frac{n}{2}$ nodes in level 0. Then, a new node, v_1, spontaneously starts the algorithm, kills v_0, increases its level to 1 and recaptures the same $\frac{n}{2}$ nodes. After v_1 has captured the $\frac{n}{2}$ nodes, two new nodes spontaneously start the algorithm, try to kill each other, and the one which survives, v_2, reaches level 1. Node v_2 then kills v_1 and recaptures the $\frac{n}{2}$ nodes at level 2. The scenario continues until the entire network has been captured by $v_{\log \frac{n}{2}}$ which is

```
/* The variable size is for algorithm C only */

Initially:
      level ← size ← 0 ; owner-id ← potential-id ← 0 ;
      untraversed ← E ; father ← potential-father ← nil ;

Candidate ( id ) :
 while ( untraversed ≠ ∅ ) do;
      e ← any( untraversed ) ;
      send(level,id) on e ;
 R:   receive(level',id') over e' ;
      if (id' = id) then          /*successful capturing.*/
           level ← level'
           untraversed ← untraversed - e ;
      else-if (level',id' < level,id) /* lexicographically*/
      then      Discard the message, goto R ;
      else      (1) send(level',id') over e';
        (2) Discard all future messages;
 end while;
announce(ELECTED, terminate the algorithm ), STOP ;


Ordinary:
 level ← -1 ;
 while (not terminated) do;
     receive(level',id') over e' ;
     case level' of :
       (1) level' < level : Discard message ;
       (2) level' > level :    /* Replace the father */
             father ← e' ; level ← level' ; owner-id ← id' ;
             potential-id ← 0; potential-father ← nil ;
             send(level',id') over the father link ;
       (3) level' = level :
             if     (id' < owner-id) then Discard message;
             else-if ( id' = potential-id ) then
                  father ← potential-father ;
                  level' ← level' +1 ;
                  owner-id ← id' ;
                  potential-id ← 0 ;
                  potential-father ← nil ;
                  send(level',id') over the father link ;
             else-if there is already a potential-father
                   then Discard message ;
             else  /* there is no potential-father */
                  potential-id ← id' ;
                  potential-father ← e' ;
                  send(level',id') over the father link ;
      end case ;
end while ;
```

Figure 2.5: Algorithm B

elected as a leader.

The increase in time complexity of the algorithm is because unlike algorithm A, the level of a candidate here is not a function of the number of nodes it has already captured. A candidate which spent a lot of work (and time) accumulating nodes might be killed by a candidate which did not spend nearly as much. Although algorithm A does not suffer from this problem, it has the problem that candidates could be "killed" many times. In the next algorithm we eliminate both problems by employing both techniques simultaneously in one algorithm.

2.4.3. Algorithm C

Here we make two modifications to algorithm B in order to achieve a linear-time complexity with no increase in the communication cost. First, we incorporate an estimate of the amount of work spent by each candidate into the level function of algorithm B. Second, we enable candidates with a high level ($>\log n$) to capture many nodes in parallel (in one time unit). We start describing the algorithm with the first modification. The second modification will be introduced during the performance analysis.

Level: In this algorithm the level of a candidate is increased according to two rules. First, the same rule as in algorithm B is used, and second after each capturing the candidate increases its level to be at least log(*total number of nodes captured*), i.e., after returning from a successful capture the level is set to *MAX* (log *(# nodes captured), present level)*.

Capturing and Elimination Rule: Same as in algorithm B.

Details

The formal description of the algorithm is similar to that of algorithm B. A variable, called *size*, is used to count the number of nodes captured by a candidate. The only change is to replace the first "then" clause, within the "while" loop of a candidate program to:

then

  $size \leftarrow size + 1$;

  $level \leftarrow \max(level', \log size)$;

  untraversed $\leftarrow$ untraversed - e ;

Analysis

To analyze its performances we will first show that:

Lemma 2.4: The maximum *level* reachable during any execution of algorithm C is $\log n + \log\log n + 1$.

Proof: Let N_i be the total number of candidates that reach level i during the execution of the algorithm. Consider the maximum number of candidates which could possibly pass from level $i-1$ to level i. There are two ways in which a candidate can go from level $i-1$ to level i. First, by capturing 2^{i-1} nodes at level $i-1$ for $i \leq \log n$, and second, by killing another candidate which is at level $i-1$. We note that N_i is maximized if as many candidates as possible pass from level $i-1$ to level i by capturing other nodes (i.e., $\frac{n}{2^{i-1}}$ candidates) and the rest of the candidates (i.e., $N_{i-1} - \frac{n}{2^{i-1}}$) kill each other in pairs. Hence,

$$N_i \le \frac{(N_{i-1} - \frac{n}{2^{i-1}})}{2} + \frac{n}{2^{i-1}} \tag{1}$$

Solving (1) for N_i we get:

$$N_i \le \frac{n}{2^i} \cdot (i+1) \tag{2}$$

Substituting $N_i = 1$ in (2) and solving for i gives us the maximum level, which is $\log n + \log \log n + 1$. ■

Using the same argument as in algorithm B we find that the message complexity of algorithm C is $2 \cdot n \cdot (\log n + \log \log n + 2)$ messages, each of length $\log n + \log(\log n + \log \log n)$ bits.

With the above modification it can be shown that the time complexity of algorithm B is reduced to $O(n \cdot \log \log n)$. In order to further reduce the time complexity to $O(n)$, processes at levels higher than $\log n$ will try to capture $n / \log n$ nodes in parallel. Thus a candidate which has reached level $\log n$ will send messages over $n / \log n$ untraversed links incident to it. Each of these messages carries the (level, id) of the candidate. When a message arrives at an adjacent node the node compares its level to that of the message. If the message level is higher, the node replaces its (level, id) with the message, thus making the candidate the new father of the node. The node then sends the candidate an acknowledgment of successful capture. If the message level is smaller, it returns no message. Finally, if the message level is the same as that of the node but the message id is higher, a notification to that effect is sent back to the candidate.

The candidate waits for all the $n/\log n$ acknowledgments. If all the acknowledgments indicate a successful capture, the candidate proceeds to the next $n/\log n$ untraversed incident links. If, on the other hand, some of the acknowledgments indicate that they have encountered the same level, one of the links is arbitrarily chosen and a process that behaves as in algorithm B is sent along that link. If the process returns, the candidate increases its level and proceeds to the next $n/\log n$ untraversed links (links on which no successful capture was reported are not considered traversed).

To analyze the algorithm with this modification we make two observations: First, the maximum attainable level in the algorithm is still bounded by $\log n + \log\log n + 1$. Second, by substituting $i = \log n$ in equation (2), we find that the maximum number of candidates which reach level $\log n$ is $\log n$.

The last modification has increased the communication complexity of the algorithm by at most $O(n)$ messages. Each node is still captured at most $\log n + \log\log n + 1$ times, however the death of a candidate at level greater than $\log n$ might be associated with at most $2 \cdot n/\log n$ messages. Since there are at most $\log n$ such candidates the increase due to killings is bounded by $O(n)$.

To show that the time complexity of the algorithm is $O(n)$ we arrange the candidates in a rooted tree. Each level of the tree corresponds to the candidates that have reached that level in the algorithm, i.e., the nodes at level i in the tree correspond to the candidates that have reached level i in the algorithm. The parent of a candidate at level i in the tree is either the candidate that caused the death of the given candidate or, the same candidate at the next level. The time delay of the algorithm is the sum of the delays incurred by candidates along the path from the first initiator (candidate) to wake up, at level 0, to the root.

To evaluate this time delay we note that no candidate that either survives or is killed at level i spends more than 2^{i-1} time units in level i, $i \le \log n$. In level i, $\log n < i \le \log n + \log\log n$ no candidate spends more than $\log n$ time units since it captures nodes at a rate of $n/\log n$ per time unit. Hence, the total time delay of the algorithm is bounded by $\sum_{i=1}^{\log n} 2^i + \log n \cdot \log\log n = n + \log n \cdot \log\log n$. Note that we scale a time unit to be the maximum delay it takes to capture one node, which is a constant.

In the above calculation we did not include the actual time it takes candidates to kill each other. Since there are at most n candidates and no candidate tries to kill a dead one (unlike algorithm A), this delay is also bounded by $O(n)$.

2.5. Conclusions

An $O(n \cdot \log n)$ messages $O(\log n)$ time synchronous and $O(n \cdot \log n)$ messages $O(n)$ time asynchronous election algorithms were presented. It remains open whether a $O(n \cdot \log n)$ messages sublinear time asynchronous election algorithm exists. We conjecture that such an algorithm does not exist and hence that the synchronous mode of communication is more powerful than the asynchronous mode.

Three asynchronous election algorithms (A, B, and C) were presented. The simplicity of the complete network topology, and, hence, of termination detection, enabled us to concentrate on the synchronization among contending candidates. With each of the three algorithms we can associate an analogous algorithm for arbitrary topology networks, which uses the corresponding method to synchronize different initiations of the algorithm but a different method to traverse the network (i.e., to detect termination). The analogy to algorithm A is given in [Gal77]. In [Gal83] the same level function as in algorithm B was used, however, there candidates merge

their "territories" (rather than kill each other) when they meet. In [Gaf85] the time complexity of [Gal83] is improved by replacing its level function with that of algorithm C. Each of the methods can be applied to other classes of networks. For example, applications of method B in different classes of topologies are presented in [Kor85]. As in algorithm B, the time complexity of the algorithms in [Kor85] is $O(n \cdot \log n)$, whereas similar applications, but of methods A or C, improve the time complexity of [Kor85] while maintaining the message optimality.

CHAPTER 3.

LOWER BOUNDS FOR ELECTION IN COMPLETE NETWORKS

3.1. Introduction

Two algorithms for election in synchronous complete networks were discussed in the previous chapter. The first is a $O(n^2)$ messages $O(1)$ time algorithm and the second is a $O(n \cdot \log n)$ messages $O(\log n)$ time algorithm. The two algorithms raise two questions: (1) Is $\Omega(n \cdot \log n)$ also the lower bound on the message complexity of election in synchronous complete networks, and (2) If $\Omega(n \cdot \log n)$ is the message complexity lower bound, then how fast can a message-optimal algorithm be, i.e., is there a $O(n \cdot \log n)$ messages $O(1)$ time algorithm, or is $\Omega(\log n)$ is the lower bound on the time complexity of any message-optimal synchronous algorithm.

In this chapter we answer these question by proving first, that $\Omega(n \cdot \log n)$ is a lower bound on the worst case message complexity of a synchronous algorithm, and second, by proving that $\Omega(\log n)$ is a lower bound on the time complexity of any message-optimal synchronous algorithm. In proving these bounds we do not restrict the type of operations performed by the nodes. The bounds thus apply to general algorithms and not just to comparison based algorithms. This proves that the synchronous algorithm of Chapter 2 is optimal.

Furthermore, in Chapter 2 we have presented a continuum of, $\frac{2c}{\log c} \cdot n \cdot \log n$ messages $2 \cdot \log_c n$ time, synchronous algorithms where, $c = 2, \ldots, n$. In Section

3.2.2 each algorithm in the continuum is shown to be optimal by proving that if an algorithm (whether comparison or general) elects a leader in at most $\frac{1}{2}\cdot\log_c n$ rounds, then its message complexity is at least $\frac{c-1}{2\cdot\log c}\cdot n\cdot\log n$.

3.2. Lower Bounds

To show the lower bounds, a scenario in which any synchronous (and hence also asynchronous) algorithm transmits at least $\frac{n}{2}\cdot\log n$ messages, is constructed using an adversary argument. A similar argument is then used to show that the delay of any message-optimal algorithm is at least $O(\log n)$ rounds.

3.2.1. Definitions and Assumptions

Consider an arbitrary election algorithm on the synchronous model defined above. An *event* is the sending of a message over a previously unused link (Two messages sent in the same round in opposite directions over a previously unused link are considered two separate events). With each event we associate a pair (s,d), where s is the source node and d is the destination node of the corresponding message. With each round i of the algorithm we associate a set of events, R_i. A sequence $E=(R_0,R_1,\cdots)$ is called an *execution*. An *execution−prefix* E_j is a prefix, $(R_0,R_1,\ldots,R_j)$, of an execution E. With each run of the algorithm we associate an execution, called a *legal−execution*, that includes all events which occurred in the run, arranged in order of the corresponding rounds. Henceforth, any mention of a message refers to an event.

A *cluster* in an execution-prefix E_j is a maximal subset of nodes spanned by a connected subnetwork of links which were used by events which occurred in E_j.

The *degree* of a node v in an execution-prefix E_j is the number of links incident to v which were used by events in E_j. The *potential–degree* of node v in an execution-prefix E_j is the degree of v in E_j plus the number of times that v is a source node of an event in R_{j+1}. The *potential–degree* of a set of nodes is the maximum potential-degree among its nodes.

For the purpose of proving the lower bounds we introduce a slightly different model, called the *stopping–model*. The stopping model allows us to withhold the clock pulse, at the beginning of round j from cluster, C, in E_{j-1}, given that no node in C is expected to receive a message in round j from a node not in C. The nodes in C are then said to be *frozen* in round j. Therefore, a frozen node in a round neither sends nor receives any message in that round; nor does it change its state. The stopping-model will be used to prevent large differences in the clusters' growth rates.

A *stopping–execution* is an execution which corresponds to a run of the algorithm in the stopping-model. A stopping-execution is called a k *stopping–execution* if the cumulative number of pulses withheld over all clusters throughout the run is k. Obviously, a 0-stopping-execution is a legal-execution.

Lemma 3.1: For any k stopping-execution E, there exists a $k-1$ stopping-execution E' which contains exactly the same events as E does.

Proof: Let l be the minimum index of a round in which any cluster is frozen, and let C be a cluster which is frozen in l. An execution-prefix E' which satisfies the lemma can be obtained from E by shifting all events which occurred before round l and involve nodes in C, one round forward. This affects neither any event in later rounds nor any event which involves nodes not in C. Because, neither in E nor in

E' there is an event connecting a node in C with a node not in C in any round R_j, $j \le l$, and because R_{l+1} in E is identical to R_{l+1} in E'. Notice that in E' the nodes in C are awakened one round later than in E. ■

Corollary 3.1: For any stopping-execution there exists a legal-execution which contains the same events.

In the next two sections we will prove the lower bounds on the stopping model. Using Corollary 3.1, these bounds apply also to the non stopping model. In our proofs we do not restrict the type of operations performed by the nodes, hence proving the bounds for general algorithms.

3.2.2. A Lower Bound on Message Complexity

At the end of any election algorithm all nodes know who the leader is, hence any such algorithm has to send messages along the links of a spanning subnetwork. In other words, by the end of the algorithm the whole network is contained in one cluster. Thus, no cluster in the algorithm can defer indefinitely the sending of messages to nodes not in the cluster, as the rest of the network might not wake up spontaneously.

In the following proof of the lower bound we will use an adversary argument to construct a stopping-execution which contains at least $\frac{1}{2} n \log n$ events. In the beginning of each round, the adversary first determines which clusters to freeze and then determines the destination of messages sent in this round over previously unused links. The first feature is used to delay the formation of larger clusters until later rounds in the run, thus avoiding large differences in the clusters' growth rates; the second feature is used to send as many messages as possible within one cluster.

The second feature is possible since links incident to a given node on which *no* message was sent or received are indistinguishable to this node.

Theorem 3.1: A stopping-execution of an election algorithm in a synchronous complete network of n nodes contains at least $\frac{n}{2} \cdot \log n$ events, in the worst case.

Corollary 3.2: The message complexity of any election algorithm in a synchronous complete network of n nodes is at least $\frac{n}{2} \cdot \log n$.

Proof of Theorem 3.1: Assume w.l.o.g. that $n = 2^q$. We define a sequence of partitions $(P_0, \ldots, P_q)$ of the nodes such that each subset in partition P_0 contains one node, and each subset in P_j contains two subsets from P_{j-1}, $1 \le j < q$. Hence, each subset in P_j contains 2^j nodes.

We construct, in q phases, a sequence of stopping-execution-prefixes $(E_{i_0}, \ldots, E_{i_q})$, $i_0=0$, each being a prefix of the next. E_{i_0} is an empty execution-prefix in which all nodes have been awakened and the potential-degree of each node is at least 1. This is done by withholding the clock pulse from any node whose potential-degree is at least 1 until there is no node with potential-degree 0. Inductively we assume that: (1) Any cluster in E_{i_j} is contained within one subset in P_j. and (2) The potential-degree, in E_{i_j} of every subset in P_j is at least 2^j. Obviously, E_{i_0} satisfies these assumptions.

Assuming that $E_{i_{j-1}}$ has been constructed, we describe how the adversary constructs E_{i_j}, $j=1, \ldots, q-1$. In each round of phase j, we freeze all the subsets in P_j whose potential-degree $\ge 2^j$. When all subsets are frozen, phase j is complete. The source and destination nodes of any message sent in this phase are both in the

same subset in P_j. This is always possible since every node that has a potential-degree $\geq 2^j$ is frozen. Clearly, E_{i_j} satisfies the inductive assumptions. In the $q-th$ phase no freezing takes place. After that phase, the network is contained in one cluster and the algorithm is assumed to produce no more events.

Clearly, there are at least $n/2^j$ nodes whose degree at the end of the algorithm is at least 2^j, for $j=0, \ldots, q-1$. Thus, the total number of events is at least $\frac{n}{2} \cdot \log n$. ■

Given that the message complexity of any election algorithm on a synchronous complete network is $\Omega(n \cdot \log n)$, the question arises how fast can a message-optimal algorithm be. In the next section we prove that the time complexity of any message-optimal algorithm is $\Omega(\log n)$.

3.2.3. A Lower Bound on Time Complexity

In this section we will extend the techniques of the previous section to prove that the shorter the length of the execution the larger the lower bound on the number of events it must contain.

Theorem 3.2: Any stopping-execution of an election algorithm in a synchronous complete network of n nodes which terminates in less than $\frac{1}{2} \cdot \log_c n$ rounds, contains at least $\frac{c-1}{2 \cdot \log c} \cdot n \cdot \log n$ events.

Corollary 3.3: The time complexity of any message-optimal election algorithm in a synchronous complete network of n nodes is $\Omega(\log n)$ rounds.

Proof of Theorem 3.2: Consider an election algorithm whose time complexity is at

most $\frac{1}{2}\cdot\log_c n$. Assume w.l.o.g. that $n=c^q$. A construction similar to the proof of Theorem 3.1 will be used here. We construct, in q phases, a sequence of stopping-execution-prefixes $(E_{i_0}, \ldots, E_{i_q})$, $i_0=0$, each being a prefix of the next, and a sequence of partitions $(P_0, \ldots, P_q)$, the subset of each partition containing c subsets of the previous. Each subset of P_0 contains one node, thus each subset of P_j contains c^j nodes. E_{i_0} is an empty execution-prefix in which all nodes are awakened spontaneously. Inductively we assume that: (1) Any cluster in E_{i_j} is contained within one subset of P_j, and (2) The potential-degree in E_{i_j} of every subset in P_j is at least c^j. Obviously, E_{i_0} and P_0 satisfy these assumptions.

Assuming that $E_{i_{j-1}}$ has been constructed, the adversary constructs E_{i_j} by first defining the subsets of P_j, and then constructing E_{i_j}. Let $(S_1, \ldots, S_k)$, $k=n/c^{j-1}$ be the subsets of P_{j-1} indexed in nondecreasing order of their potential-degrees in $E_{i_{j-1}}$. Then the $i-th$ subset of P_j is defined as the union of $S_{(i-1)c+1}, \ldots, S_{ic}$ $i=1,\ldots,n/c^j$. This implies that if subset S in P_j contains one subset from P_{j-1} whose potential-degree is at least c^j, then all subsets from P_{j-1} in S have potential-degree at least c^j, with the exception of at most one subset of P_j, called the *boundary* subset.

In each round of phase j, $j=1, \ldots, q-1$, we freeze all the subsets in P_j whose potential-degree $\geq c^j$. When all subsets are frozen phase j is complete. The destinations for messages to be sent by node v are selected from the subsets which included v in partitions $P_0, \ldots, P_j$, in that order of priority. This is always possible since every node that has a potential-degree $\geq c^j$ is frozen. Clearly, E_{i_j} and P_j satisfy the inductive assumptions. After the $q-th$ phase, the network is contained in one cluster and the algorithm is assumed to produce no more events.

We now show that every node is the destination of at least $\frac{1}{2}\cdot(c-1)\log_c n$ events in E_{i_q}. As the time complexity is at most $\frac{q}{2}$, every node in E_{i_q} must have been frozen in all the rounds of at least $\frac{q}{2}$ phases. Otherwise, the legal-execution corresponding to E_{i_q} would contain more than $\frac{q}{2}$ rounds, contradicting the assumption on the time complexity. If node v is frozen in all the rounds of phase j, it will later receive one message from every subset in P_{j-1} which does not contain v and is with v in a subset of P_j (unless v is in a boundary subset). Thus, for each phase that v is frozen in all its rounds, v is the destination of $c-1$ events. The total number of events in E_{i_q} is thus at least $\frac{c-1}{2\cdot\log c}\cdot n\cdot\log n - n\cdot c$. The term $n\cdot c$ is due to the nodes in the boundary subsets (since due to the boundary subset in phase j at most $c^{j-1}\cdot(c-1)$ events should be discounted). ■

3.3. Conclusions

The effect of synchronous and asynchronous communication on the problem of distributively electing a leader in a complete network was examined in the last two chapters. On the one hand, it was proved that the message complexity is not affected by the choice of the communication mode. In both modes of communication, the message complexity was shown to be $\Theta(n\cdot\log n)$. On the other hand, it remains open whether or not the choice of communication mode affects the time complexity of a message-optimal algorithm. With synchronous communication, the time complexity of message-optimal algorithms was proved to be $\Theta(\log n)$, whereas with asynchronous communication, only an $O(n)$ upper bound on the time complexity was obtained. The lower bound on time for asynchronous communication remains an open question and is the subject of the following conjecture:

Conjecture: The time complexity of any message-optimal asynchronous election algorithm on a complete network is $\Omega(n)$.

The implication of the conjecture is that synchronous communication is faster by a factor of $n/\log n$ than asynchronous communication. An analogous result was obtained in [Arj83], where a particular synchronous system of parallel processors was proved to be faster by a factor of $\log n$ than the corresponding asynchronous system.

CHAPTER 4.

TRAVERSAL OF UNIDIRECTIONAL NETWORKS

4.1. Introduction

In this chapter we address the problem of traversing a unidirectional network. In the traversal problem, one node, called the *root*, initiates a single process (token) which has to visit all the nodes in the network, one at a time. If necessary the process may go over any link more than one time.

A traversal algorithm is an identical program residing at each node in the network. When the process arrives to a node, the program decides which of its incident links will be the next in the traversal. The algorithm also detects when the process has visited all the nodes. To this end, the program in each node uses local variables to mark the node and its incident links. The marks are used by the program upon the next arrival of the process. In addition, the nodes of the network may use the process to carry messages between them.

We consider two types of unidirectional networks. The two networks differ in the amount of memory available for the algorithm at each node. In the first model $O(\log n + d_v)$ bits of memory are available for the algorithm in node v, where n is the total number of nodes in the network and d_v is the degree of node v. In the second model only $O(d_v)$ bits of memory are available.

Clearly, to be able to traverse a bidirectional network, the network has to be connected. Similarly, to traverse a unidirectional network, the network has to be strongly connected, i.e., there should be a directed path from every node to every other node.

$\Omega(|E|)$ is obviously a lower bound on the number of messages transmitted by any traversal algorithm. This is because every link in the network has to be traversed, or otherwise an untraversed subnetwork could reside on the other side of any untraversed link.

The problem of distributively traversing a bidirectional network is essentially the same as the centralized problem of searching a graph under the restriction that any two consecutively visited nodes are neighbors in the network. One search algorithm which satisfies this restriction is the Depth First Search (DFS) algorithm [Tar72, Hop73] and thus it can be the basis for a distributed bidirectional network traversal. In the resulting traversal the process makes $2 \cdot |E|$ hops, and the number of memory bits it uses at each node is linear in the degree of the node.

In the bidirectional DFS algorithm, the root spawns a process which visits all the nodes in the network. Upon arriving at node v for the first time, say through link l, the process marks v and sequentially traverses each of v's incident links, except l. If the process arrives at an already marked node, it backtracks to the node from which it came. After backtracking from all of v's incident links, except l, the process backtracks from v over link l. The traversal is completed when all the links incident to the root were backtracked.

In solving the unidirectional traversal problem we would like to adapt the bidirectional DFS traversal. However, it is not obvious how to backtrack in a uni-

directional network. In spite of this difficulty, the final unidirectional traversal algorithm presented in this chapter is based on the DFS algorithm. The difficulty of backtracking is surmounted by constructing, on the fly, a structure called *in-directed forest*. An in-directed-forest is a subnetwork in which there is a unique path from any fully-backtracked node to exactly one visited node which is not yet fully-backtracked. In constructing the forest we will use a technique similar to the one used in the strongly connected components algorithm of Hopcroft and Tarjan [Hop73].

In this chapter, three traversal algorithms are presented. Traversal-1 is simple but inefficient. In many networks the process of Traversal-1 hops over an exponential number of links before terminating. Traversal-2, which is based on the DFS algorithm, makes at most $O(n \cdot |E|)$ hops on any network. Furthermore, we show that $\Omega(n \cdot |E|)$ is a lower bound on the number of hops, in general.

In both Traversal-1 and Traversal-2 $O(\log n + d_v)$ bits of memory are required at each node v and that much is also carried along with the traversing process (i.e., messages size is $O(\log n + d_{\max})$ bits, where $d_{\max}$ is the maximum degree in the network).

In some applications, such as VLSI, memory size and message length are restricted and the question then arises, could a unidirectional traversal be implemented using only a constant number of bits in every node and with the traversing process (i.e., in a unidirectional network of finite automata). We answer the question positively by presenting Traversal-3, a traversal algorithm for unidirectional networks of finite automata. Traversal-3 makes at most $O(n \cdot |E| + n^2 \cdot \log n)$ hops which is optimal in the worst case (dense networks, in which $|E| = \Omega(n \log n)$).

Throughout the discussion we make a distinction between **unidirectional** and **directed** networks. A unidirectional network, as defined before is a network in which some or all the links are unidirectional links. A directed network is a bidirectional network in which a unique direction is associated with each link. The link directions are given as part of the problem definition.

4.2. Traversal-1: a simple traversal algorithm

The algorithm is composed from two mechanisms: a termination detection mechanism, and a routing mechanism. The termination detection mechanism enables the traversing process to detect that it has traversed all the links in the network. The routing mechanism is used at each node to select the next link on which to send the process such that in a finite number of hops the process will detect termination.

The termination detection mechanism is implemented by a counter, called the *debt−counter* which is carried by the process. The counter is incremented by one whenever the process arrives at a node for the first time. It is decremented by one just before leaving a node through its last untraversed outgoing link. After leaving a node at least once through each of its out-going links, the debt counter is never changed again at this node. To start the traversal the root initiates the debt counter to zero and sends the process to itself.

Lemma 4.1: The debt-counter returns to zero when and only when all links are traversed.

Proof: Clearly, for each newly visited node the process increments the counter by one. Similarly, for each visited node whose outgoing links have been all traversed, the process decrements the counter by one. Hence, when all links are traversed, the

counter has been incremented and decremented the same number of times and is therefore back to its initial value. This proves the sufficient condition ("when").

To prove the "only when" assume that the counter has returned to zero but not all the links in the network are traversed. Since the network is strongly connected there must be an untraversed link l whose tail node, v, was visited. Hence, the counter was incremented at least once more than it was decremented, which leads to contradiction. ■

Using Lemma 4.1, the process may traverse a unidirectional network and tell whether or not it has traversed all the links. However, to guarantee termination, the process needs to employ a routing rule. The routing rule will route the process to any remaining untraversed link. We present different routing rules, which lead to different traversal algorithms, presented in this and the next section.

The routing rule used in traversal-1 is the following: Every node orders its outgoing links cyclically, i.e., the first link in the order follows the last one. Each time that the process arrives at a node it is sent out on the next outgoing link according to the cyclical order.

Lemma 4.2: Using the above routing rule the process eventually traverses all the links.

Proof: Assume the contrary. Then, since the network is strongly connected, there must exist an untraversed link, l, leaving visited node, v, whose out degree is d. Since v was visited once, and since the traversal cannot terminate, v will be visited infinitely many times. Hence, following the above rule l must be selected in the $d-th$ visit to v. Contradiction. ■

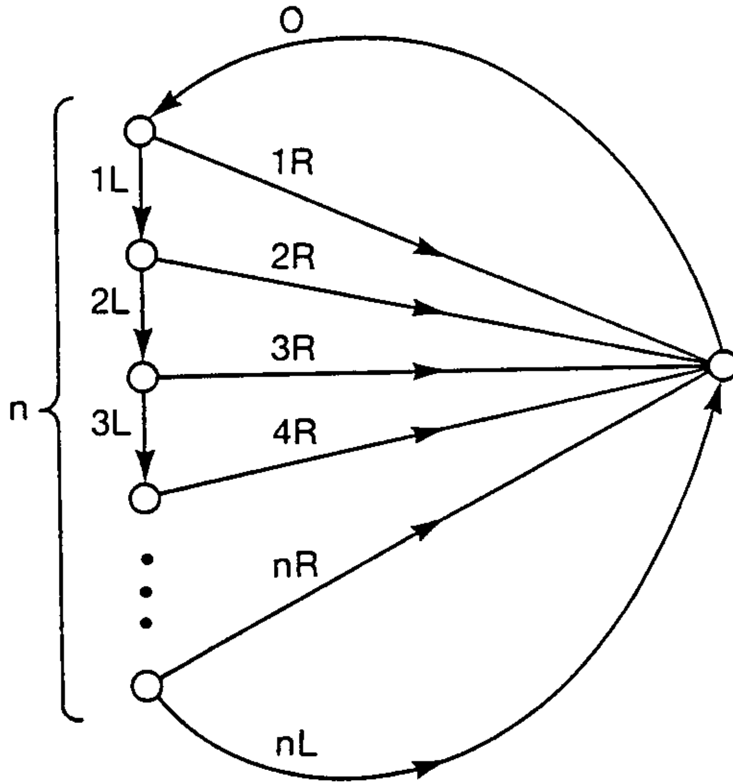


Figure 4.1: An example for the exponential complexity of Traversal-1

Lemmas 4.1 and 4.2 result in a correct traversal algorithm. Lemma 4.1 provides for termination detection while, Lemma 4.2 enables us to route the process in a way which guarantees termination. However, the communication complexity of the resulting algorithm is exponential, as will be argued next.

Figure 4.1 is an example of a network on which the algorithm requires 2^{n-1} link traversals. Let N_l be the number of traversals over link l in some execution of the algorithm. Clearly, $N_0=2 \cdot N_{1L}=2 \cdot N_{1R}$ and $N_{iL}=2 \cdot N_{(i+1)L}=2 \cdot N_{(i+1)R}$ for $i=1 \cdots n-1$. Since N_{nL} and N_{nR} are both equal to 1 by the end of the traversal, $N_0=2^{n-1}$.

Response to receiving the process at node v.

```
If v is unmarked
then begin
        mark v
        increment the Debt-Counter on the process by 1.
    end
else if Debt-Counter = 0 then stop

let l be the next link in the cyclic order of v

If l is unmarked
then begin
        mark l
        if l was the last unmarked link of v
        then decrement the Debt-Counter by 1
    end

Send the process over l
```

Figure 4.2: Traversal-1

The source of the traversal's inefficiency is the routing procedure. In the example of figure 4.1 the process hops over an exponential number of already traversed links before traversing the last untraversed link. A different routing rule is employed in the next section to derive a traversal which requires $O(n \cdot |E|)$ hops.

A semi-formal description of the algorithm is given in figure 4.2. Initially all nodes and links are assumed to be unmarked. To start the algorithm the root initiates a process with a debt counter set to zero and sends the process to itself (i.e., it places the process in its input queue).

4.3. Traversal-2: Simulating Directed Depth First Traversal

In Traversal-1, each time the process arrives at a node, we changed the link through which the process leaves the node. In the following algorithm we modify the routing strategy to use the same link to leave a node as long as the set of selected

links does not contain a cycle. Upon detecting a cycle, we select another untraversed outgoing link only at the node which was explored last.

We present the algorithm of Traversal-2 in three stages. First, a bidirectional depth first traversal algorithm is described. Second, a unidirectional implementation of the first algorithm is presented by assuming that a structure, called *spanning in-directed tree*, is predefined. Finally, a mechanism to build the in-directed tree on the fly is given, thus providing a unidirectional traversal algorithm.

4.3.1. Bidirectional directed depth first traversal

In a bidirectional directed network every link can be used to pass messages in both directions, however an arbitrary direction is associated with it. Here we assume that the directed graph resulting from the directions associated with the links is strongly connected (i.e., there is a directed path form every node to every other node in the network).

In the bidirectional directed depth first search algorithm [Hop73], the root spawns a process which visits all the nodes in the network. Upon arriving at node v for the first time, say through link l, the process marks v as *active*, l as the *father link* of v, and iteratively traverses each of v's incident out-going links. If the process arrives at an already marked node, it backtracks to the node from which it came. After backtracking on all of v's incident out-going links, the process marks v as *fully-backtracked* and backtracks from v on the incoming link, l. The traversal is completed when the root is marked fully-backtracked.

A formal description of the algorithm is given in figure 4.3. Note that the process passed between the nodes is merely a token. It does not carry any information except its actual location. Unlike this traversal, in the next sections we will use

the process to carry essential information between the nodes.

Initially all nodes and links are unmarked.

To start, the root s performs:

```
    mark s active ;
    select a link, l', outgoing from s ;
    mark l' active ;
    send the process over l' ;
```

Response to receiving the process at node v over incoming link l.

```
if v is marked
then
    send the process back over l ;
else
    mark v active ;
    mark l father ;
    select a link, l', outgoing from v ;
    mark l' active ;
    send the process over l' ;
end
```

Response to receiving the process at node v over outgoing link l.

```
mark l backtracked ;
If there is an unmarked outgoing link l'
then
    mark l' active ;
    send the process over l' ;
else
    mark v fully-backtracked ;
    if there is no father link
    then stop ;
    else send the process over the father incoming link ;
end
```

Figure 4.3: The bidirectional directed depth first traversal algorithm

The following two observations are used in the next section to implement the unidirectional algorithm. Let the *father node* of every node v, except the root, be the node from which the process arrived at v for the first time. At any given time, the link through which the traversing process left an active node for the last time is

called *active link*.

> *Observation 1: The* active nodes together with the active links form a simple directed path, called the *active path*. The first node on the active path is the root and the last link in the path either closes a cycle of active links (see figure 4.4), or leads to a fully-backtracked node. The most recently marked node among the active nodes is called the *focal point* of the traversal (e.g. see figure 4.4).
>
> *Observation 2: All* backtrackings are over the last link of the active path, i.e., either from an active node, or from a fully-backtracked node to the last active node on the active path.

Observation 1 follows inductively from the fact that every node has at most one active out-going link, and if it has one it must have an active incoming link (the father link). Observation 2 follows immediately from observation 1 and the algorithm.

For the sake of completeness a formal proof of the algorithm is included. The proof is a simple modification to a proof given in [Eve79] , the proof there is for the centralized undirected DFS algorithm. Let a *forward* traversal of a link be a traversal in the direction associated with the link, and likewise, a *backward* traversal of a link be a traversal in the opposite direction. First, we shall prove that no link is traversed more than once in each direction and then, that if the underlying graph is strongly connected then every link is traversed in every direction (similar to [Eve79]).

Lemma 4.3: The bidirectional directed depth first process traverses every link in the

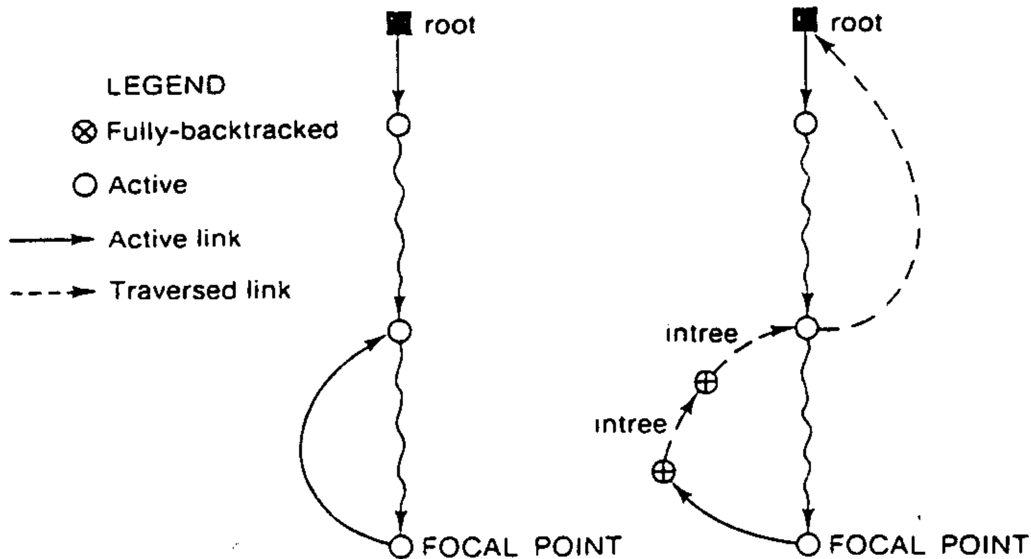


Figure 4.4: The active path

network at most once in each direction.

Proof: By the algorithm definition the process is never sent forward more than once on the same link. Similarly, a non father link is traversed backward once for each forward traversal. Hence, only the traversal of a father link in the backward direction still needs to be proved. Assume that link l, directed from v to u is the *first* father link to be traversed backward twice. Since, the father link is traversed backward only after the process has backtracked from another node to u and when all the out-going links of u are marked, the process must have backtracked to u twice on some link. However, it neither could backtrack twice on a non father link (as argued above) nor could it backtrack twice on any father link (by the assumption), contrad-

iction. Hence, every link is traversed at most once in every direction.■

Corollary 4.1: The depth first traversal must terminate.

Lemma 4.4: The bidirectional depth first process traverses every link in the network once in each direction.

Proof: Since the network is strongly connected it is enough to prove the following claim: For every node all its incident out-going links are traversed once in each direction.

The number of times node v is entered via its out-going and father links is equal to the number of times v is left via these links. This is because whenever the process arrives at v and v is marked, through an incoming link, it is sent back on this incoming link and otherwise, v never sends the process over an incoming link (except the father through which it was also entered). Since the algorithm terminates, all the out-going links incident to the root, s, are marked. Obviously no link can be traversed backward before it has been traversed forward, thus all the incident out-going links of s were traversed forward. As the number of times s was left through an out-going link equals to the number of times that s was entered on an out-going link, and no out-going link could be entered twice (by lemma 4.3), the claim holds for s.

Assume the claim does not hold for all the nodes. Let S be the set of nodes for which the claim holds (see figure 4.5). Since the network is strongly connected and $s \in S$, there must be a link l from v to u such that l is the father link of u and such that $v \in S$ and $u \in V-S$. Hence u was backtracked but not all of its incident out-going links were traversed once in each direction. Since u's father link was backtracked all u's out-going links must have been traversed forward. But, the

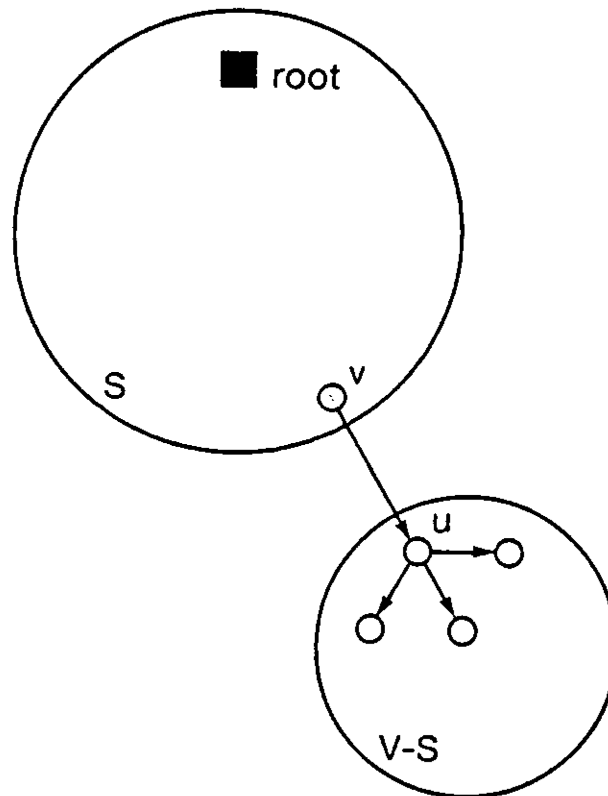


Figure 4.5

number of times the out-going links are traversed forward equals to the number of times they are traversed backwards, thus by lemma 4.3 $u \in S$, contradiction. ■

Corollary 4.2: The number of hops made by a traversing process in the bidirectional depth first search is exactly $2 \cdot |E|$.

4.3.2. Unidirectional depth first traversal, using a spanning in-directed tree

In this section we use the two observations of the previous section to implement the bidirectional depth first traversal on a unidirectional network in which an in-directed spanning tree is defined. The traversal requires, in the worst case, $n \cdot |E|$ hops (messages) of the traversing process.

An *in-directed tree* (or, in-tree) is a subnetwork in which every node, except one node, called the *root*, has exactly one out-going link and the underlying undirected graph is a tree. Since every node in the in-tree has exactly one outgoing link, there is a unique path from every node in the in-tree to the root. An *in-directed spanning tree* is an in-tree which spans the network.

The difficulty in implementing the depth first traversal on a unidirectional network is that the traversing process cannot backtrack on any link. To overcome this difficulty, we note that whenever the process wants to backtrack over link l, a directed cycle, called the *backtracking cycle* is defined by concatenating: l, the unique path in the in-tree from the head node of l to the root, and the active path. Thus, to backtrack over link l (from the head node of l to its tail node) the process goes along the backtracking cycle until it arrives at the tail node of l. To this end, the unique *ids* of each node are used by the process to identify the tail node of l. Note that shortcuts are possible if the unique path of the in-tree intersects the active path before reaching the root (i.e., whenever the cycle is not simple). In particular, if the head node of l is active the process needs to follow only the active path in order to backtrack to l's tail node.

A formal description of the traversal algorithm is given in figure 4.6, ignoring the lines marked with $*$. The lines marked "I" are the code that the initiator has to execute in order to start the traversal.

To implement the backtracking mechanism, whenever the traversing process traverses link l from node v to node u it carries the id of v. If u is unmarked (unvisited yet) then node u remembers that v is its father. If node u is already marked, the process follows the cycle until it arrives back to v. When node u becomes fully-backtracked the traversing process is sent along the backtracking

cycle to u's father, v.

Lemma 4.5: The number of hops made by a traversing process in the unidirectional depth first traversal is at most $n \cdot |E|$.

Proof: In Lemma 4.2 we saw that every link is backtracked exactly once. The lemma follows since in each such backtracking the process goes around a cycle of length at most n. ■

In Section 4.5 it will be shown that $\Omega(n \cdot |E|)$ is also the lower bound on the number of hops.

The communication cost of the traversal has two components, one is due to the hops that the process makes in the forward mode, and the other is due to the hops that the process makes in the backtrack mode. The process backtracks over $|E|$ links. Each backtracking requires, in the worst case, the traversal of an $O(n)$ long cycle. In each hop that the process makes it carries $O(\log n)$ bits which are used to identify the node which it wants to reach. The forward hops incur $O(|E| \cdot \log n)$ bits, while the backtracking hops add $O(n \cdot |E| \cdot \log n)$ bits to the communication complexity of the traversal (the number of bits transmitted by the algorithm).

4.3.3. On the fly in-tree construction

In this section the assumption of the previous subsection, that an in-tree is predefined on the network, is relaxed. The assumption is relaxed by constructing the in-tree on the fly, while the process is traversing the network.

The essential use of the in-tree in the previous section was to backtrack from a fully-backtracked node. Backtrackings from active nodes could be accomplished by using only the active path. Thus, every node decides on its unique out-going link

in the in-tree, called *intree*, while it is active. While active a node might change the intree mark a few times. The intree link of a fully-backtracked node does not change.

The basic idea of the in-tree construction is as follows; Every node remembers whether or not its father incoming link has already participated in a backtracking cycle. When the father incoming link of node v participates in a backtracking cycle for the first time, all the active nodes from v to the end of the active path, select their present active link as their in-tree link. In the rest of this chapter a father link which has never participated in a backtracking cycle is called a *bridge*.

Aside form the in-tree construction, the traversal is the same as the depth first traversal of the previous subsection. It is assumed here, that whenever a shortcut in the backtracking cycle is possible it is done, i.e., the backtracking cycle is a simple directed cycle.

The mechanism to construct the in-tree can be viewed as an approximation of the mechanism to determine the low-points of vertices in a directed graph, which was introduced by Hopcroft and Tarjan [Hop73] in their algorithm for strongly connected components.

A formal description of the traversal algorithm, with the in-tree construction, is given in figure 4.6. The lines of the algorithm are marked $*$, + and I, to indicate the following; the $*$ lines implement the in-tree construction, i.e., by taking out the $*$ lines one obtains the unidirectional traversal algorithm given an in-tree. The "I" lines are the steps which the root executes in order to start the algorithm. The + marks will be explained in the next subsection. Basically they indicate the lines in which a variable of length $O(\log n)$ bits is used.

In this algorithm node v knows which of its incoming links is the father link by recording the id of the node on the other side of the father link, this is the father node of v. The first time that a father incoming link of node v participates in a backtracking cycle is easily detected, as it is exactly the second time that the process arrives at v through this link. To this end, a boolean variable, called BrgHd (Bridge Head), is used at every node to indicate whether or not its father incoming link is a bridge (i.e., if it has already participated in a backtracking cycle). Another boolean variable, called XBrdg (crossed bridge), is used on the traversing process to indicate whether or not a bridge is participating in the backtracking cycle. Whenever the process arrives at an active node v, in the backtracking mode, and the XBrdg indicator is on, v selects its active link to be its intree link. Aside from BrgHd, every node v has the following fields: *id* which is the id of v, *father* which is the id of the father node of v, *activelink* which points to the activelink of v, *intree* which points to the intree outgoing link of v. Aside form XBrdg, the traversing process, P, has the following fields: *mode* which indicates whether the process is now backtracking or not, *PreviousNode* which is the id of the node that P visited last, *FocalPoint* which is used in the backtracking mode and is the id of the backtracking destination node.

It remain to prove that the intree links selected by any fully-backtracked node span all these nodes and always lead to an active node.

Let us define an *in-directed forest* as a collection of disjoint in-trees.

Lemma 4.6: The intree marked links of the fully-backtracked nodes constitute an in-directed forest.

Before proving the lemma we note the following two implications of its proposition; First, if there are still active nodes, then the roots of in-trees in the forest

Response to receiving the traversal process, P, at node v:

```
If v is an unvisited node:
+       v.father ← P.PreviousNode ;
*       v.BrgHd ← true ; {BrgHd since v's father link have not yet been on a cycle}
I       v.activelink ← any unused out-going link ;
I       mark v active ;
I+      P.PreviousNode ← v.id ;
I       send P over v.activelink ;

If v is an Active node and P is in the Forward mode:
        P.mode ← backtrack ;
+       P.FocalPoint ← P.PreviousNode ;
+       P.PreviousNode ← v.id ;
        send P over v.activelink ;

If v is an Active node and P is in the backtrack mode:
*       if P.XBrdg then v.intree ← v.activelink ;
+       if v.id = P.FocalPoint
        then begin      { v is the destination of the backtracking }
*         P.XBrdg ← false ;
          if there are unused out-going links
          then begin
            v.activelink ← any unused out-going link ;
            P.mode ← Forward
          end
          else begin;   { No more unused out going links : }
            mark v fully-backtracked ;
            if v has no father then STOP ;  { v is the root }
+           P.FocalPoint ← v.father ;
        end end
*       else          {In the middle of backtracking, on the Active path }
*         if (v.BrgHd) and ( P.PreviousNode = v.father)
*         then begin {1-st time that v's father link is on a backtracking cycle}
*           P.XBrdg ← true ;
*           v.BrgHd ← false ;
*         end
+       P.PreviousNode ← v.id ;
        if v is still marked active
        then send P over v.activelink ;
        else send P over v.intree ;

If v is a Fully-Backtracked node:
        if P.mode = Forward
        then begin
          P.mode ← backtrack ;
+         P.FocalPoint ← P.PreviousNode ;
        end ;
        send P over the intree link ;
```

Figure 4.6: Traversal-2, The unidirectional depth first traversal algorithm

must be active nodes, which is exactly what we need for the backtracking process. This is because all the fully-backtracked nodes have an intree out-going link which obviously cannot lead to an unvisited node. Second, the proposition implies that when the algorithm terminates the intree links constitute an in-directed tree rooted at the traversal initiator, the root.

Proof of lemma 4.6: The claim will be proved by induction. Clearly, the lemma holds when the algorithm starts at a time when no node is fully-backtracked. Assume that the claim holds just before node v becomes fully-backtracked, and we will prove that it holds after v becomes fully-backtracked.

If v is the root then the claim certainly holds, since v selects no in-tree link. Henceforth v is not the root, and when v becomes fully-backtracked there is at least one active node in the network.

Assume to the contrary that after v becomes fully-backtracked the claim does not hold. Let T and S be the sets of nodes which were explored before and after v, respectively (see figure 4.7). Let L be the set of links which are directed from a node at S to a node at T. Clearly, $L \neq \emptyset$ since the network is strongly connected. By the definition of S and T there is no traversed link from T to S except the father link of v. Thus, all the in-trees in T are rooted at active nodes in T and the backtracking cycle, C_l, which is associated with each $l \in L$, passes from T to S on the father link of v. Clearly, at least one C_l, $l \in L$ passed on a bridge. Let the u to w link, l^*, be that link in L whose associated backtracking cycle, C_{l^*}, was the most recent to pass over a bridge among the backtracking cycles associated with the links in L. Then, all the links from v to w on C_{l^*} must have been marked as intree links. By the definition of l^* none of these marks could be removed in the future. Contradiction.

■

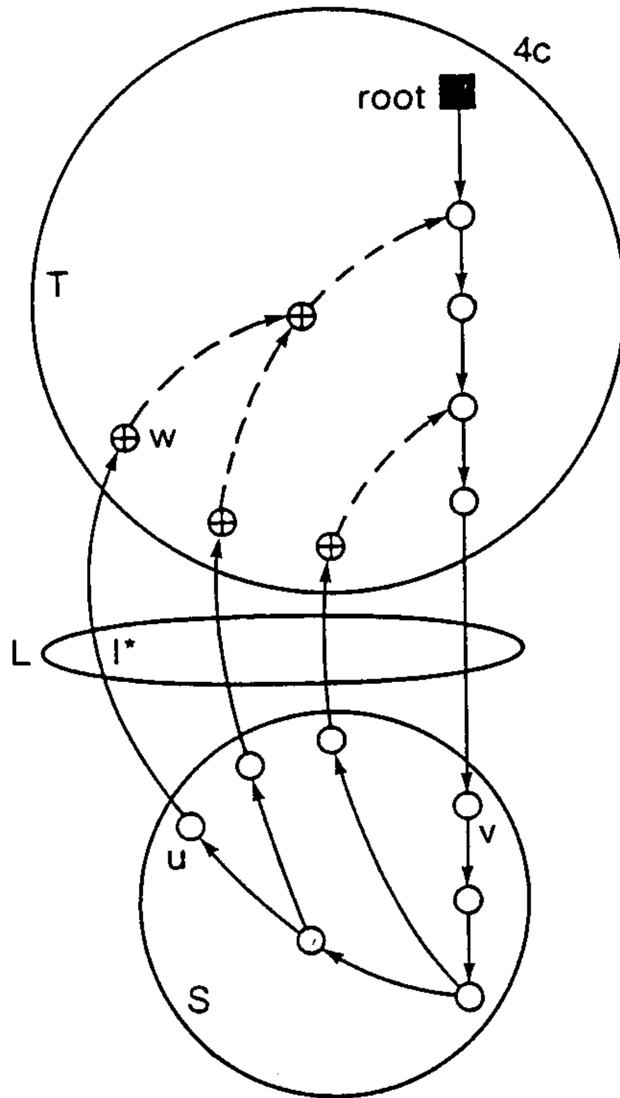


Figure 4.7

4.4. Traversal-3: an algorithm for a network of finite automata

In this section the assumption that every node has $O(\log n)$ bits of memory is relaxed. Instead, every node is assumed to be a finite automaton, i.e., to have constant size memory regardless of the network size. Since each node has a constant number of memory bits, the traversing process has to be of constant size too. We will show that with a constant size process the traversal requires at most $O(n \cdot |E| + n^2 \cdot \log n)$ hops which is also the bit complexity of the algorithm.

Traversal-3 is developed in two steps; First, the bit complexity of Traversal-2 is reduced to $O(n\cdot|E|+n^2\cdot\log n)$ (from $O(n\cdot|E|\cdot\log n)$) and second, an implementation with constant size memory is presented. In the first step it is shown that only in $O(n^2)$ out of a total of $O(n\cdot|E|)$ hops the traversing process has to carry $O(\log n)$ bits (in the rest it carries $O(1)$ bits). In the second step the $O(n^2)$ hops of size $O(\log n)$ are replaced by $O(n^2\cdot\log n)$ hops of a constant size process.

4.4.1. Reducing the communication complexity of Traversal-2 to $O(n\cdot|E|+n^2\cdot\log n)$ bits

In this section Traversal-2 is modified so in at most $2n-2$ of its backtrackings the process will carry $O(\log n)$ bits around the backtracking cycle. In the rest of the backtrackings the process need not carry more than a constant number of bits. Thus reducing the bit complexity from $O(n\cdot|E|\cdot\log n)$ to $O(n\cdot|E|+n^2\cdot\log n)$. A variation of the traversal presented here is also presented at the end of Chapter 5 in a much different setting.

The modification of the algorithm is as follows:

1. Every active node uses a boolean variable, called the *focal point*, to assert whether or not it is the focal point of the traversal. If the focal point variable of node v is false then v is not the focal point of the traversal. When v is visited for the first time it sets its focal point to true. When v becomes fully-backtracked it sets its focal point to false.

2. Whenever the process arrives in the forward mode at marked node, v, a two phase backtracking is started. In the first phase the process is sent around the backtracking cycle and back to v, counting whether there is one or more nodes on the backtracking cycle whose focal point is true. To this end only a

constant number of bits has to be carried around by the process.

3. If only one node on the cycle has its focal point "on" then, this node must be the node preceding v on the cycle (see figure 4.8 case 1), i.e., it is the backtracking destination. In this case, the process is sent around the backtracking cycle again with a constant number of bits, to the unique node which asserts to be the focal point. Hence, a complete backtracking is performed with the process carrying only a constant number of bits.

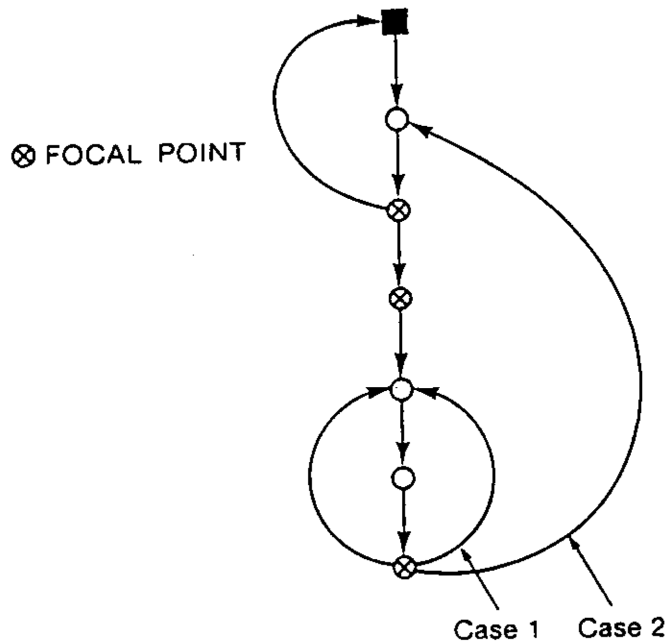


Figure 4.8: Backtracking in Traversal-3

4. If more than one node on the backtracking cycle has its focal point "on" then a bridge was included in the backtracking cycle (see case 2 figure 4.8), and a backtracking identical to the one used in Traversal-2 is initiated by v (with XBrdg set to true). In this phase of the backtracking all the nodes, aside from the last one on the active path, set their focal point to false. A bridge is included since the active link leaving any node whose focal point is true,

aside from the last one, must have led to a new node (i.e., it is the father link of the next node on the active path) and has never been on a backtracking cycle before (otherwise the focal point would not have been true).

5. The backtrackings from a fully-backtracked node remains the same as in Traversal-2 except that the focal point of the destination is set to true.

The main claim of this subsection is,

Lemma 4.7: By the above modification the process will have to carry $O(\log n)$ bits around a backtracking cycle only $2n-2$ times.

Proof: The process has to carry $O(\log n)$ bits around the backtracking cycle either when it backtracks from a fully-backtracked node to its father, or when the backtracking cycle goes over some bridge for the first time. Thus we can associate one such backtracking with each node that becomes fully-backtracked, and one with each bridge. Clearly there are $n-1$ nodes which become fully-backtracked (except the root from which the process never backtracks). Similarly, there are $n-1$ bridge links since each such link is the unique incoming father link of some node (except the root which has no bridge link entering it). ■

In the remaining $|E|-2n+2$ backtrackings the process carries only a constant number of bits. Since every backtracking requires at most n hops we get:

Corollary 4.3: The communication complexity of the modified Traversal-2 is $O(n \cdot |E|+n^2 \cdot \log n)$ bits.

4.4.2. A finite automata implementation of Traversal-2

In this section we show how each of the backtrackings in Lemma 4.7 can be implemented by a constant size process which will go around the backtracking cycle $O(\log n)$ times.

In each of these backtrackings a designated node on the backtracking cycle sends the process to the node preceding it on the cycle. The $O(\log n)$ bits were used to identify the preceding node.

To recognize the preceding node on the backtracking cycle using a constant size process, we use a solution to the following "last in the ring" puzzle: In a unidirectional ring of finite automata, design an algorithm by which a designated node, v, will distinguish the node preceding it, u, from all other nodes.

Lemma 4.8: The upper and lower bound on the bit complexity of the "last in the ring" puzzle is $\Omega(n \cdot \log n)$.

Proof: A solution to the puzzle works in phases. Initially, all nodes except v are candidates for the position. In each phase we eliminate half of the remaining candidates by sending a token around the ring, alternately marking the candidate nodes even and odd. When the token arrives at v, it remembers the parity of the candidate preceding v. In the next phase, the token eliminates all candidates whose parity differs from the parity of the desired node. The last phase is detected by the token when it sees that only one node has not been eliminated. Thus the token carries one more bit to indicate whether there are one or more uneliminated nodes on the cycle. Hence, $O(n \log n)$ is an upper bound on the bit complexity of the puzzle.

To prove that this is also the lower bound we note that if all n links have seen fewer than $\log n$ bits, then there are two distinct links, $u_0 \to u_1$, and $v_0 \to v_1$, such that both have seen the same sequence. Hence, u_1 and v_1 are in the same state and both have generated the same sequence on their out going links, which implies that their down neighbors u_2 and v_2 are also in the same state. Continuing this argument inductively we conclude that the node preceding the designated one has to be at an equal distance from both u_0 and v_0. This is a contradiction; hence, $\Omega(n \log n)$ is also the lower bound.■

Thus, whenever node v becomes fully-backtracked, or a backtracking cycle with more than one focal point is closed at v, v starts the algorithm described in the proof of Lemma 4.8 to send the process to the node preceding it. The total bit complexity of the traversal does not change with this modification; however, the number of hops made by the process is linear with the bit complexity of the traversal, i.e., $O(n \cdot |E| + n^2 \cdot \log n)$.

4.5. Lower Bounds

Arriving at an upper bound of $O(n \cdot |E| + n^2 \cdot \log n)$ bits on the communication complexity for traversal, we wonder what is the lower bound. Much research is still needed in establishing tight lower bounds on the unidirectional traversal problem in general. In this section we present one step in this direction. We show that $\Omega(n \cdot |E|)$ is a lower bound on the number of hops required by a single token traversal, i.e., when the traversal is restricted to send at most one message at a time.

Lemma 4.9: $\Omega(n \cdot |E|)$ bits is the lower bound to a single token traversal of a unidirectional graph of arbitrary topology.

Proof : (by example, figure 4.9). The result follows since each traversal of a link

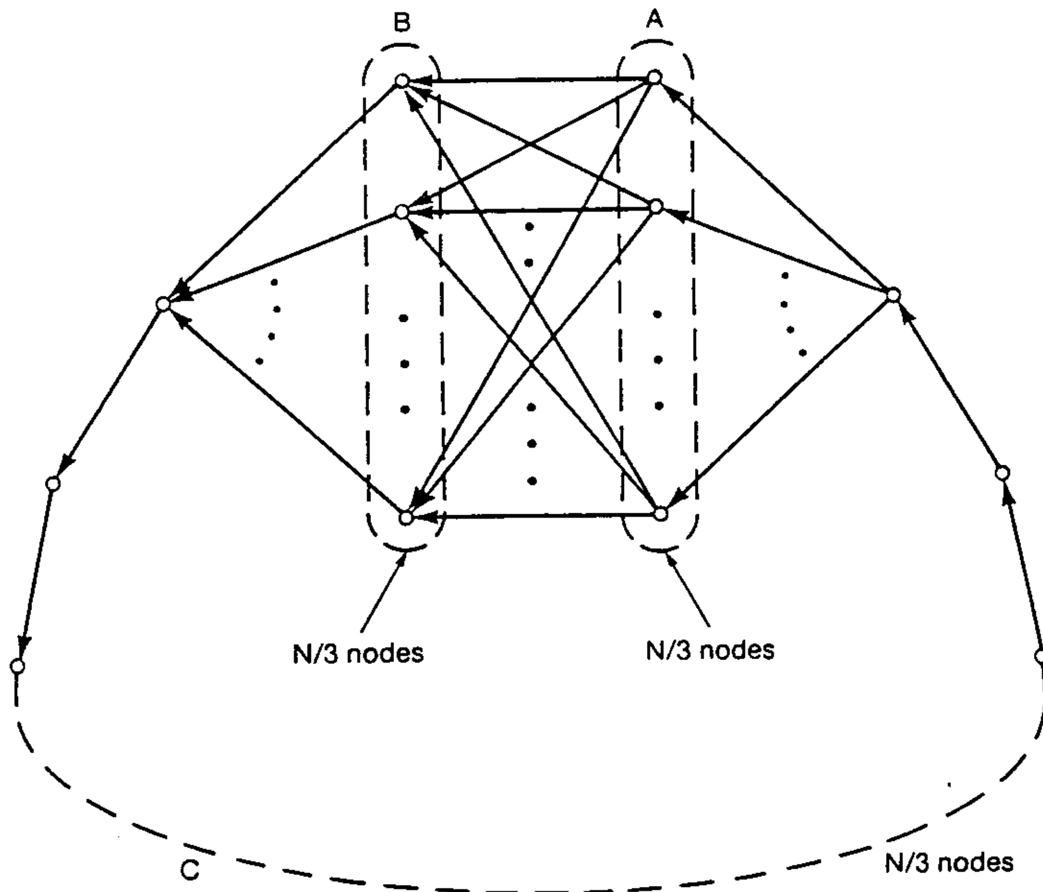


Figure 4.9: A network for the $\Omega(n \cdot |E|)$ lower bound
from A to B must be followed by a traversal of the path C. ■

Lemma 4.9 proves that our traversal algorithm is optimal for dense networks (in which $|E| = \Omega(n \cdot \log n)$) and under the restriction that the traversal have at most one outstanding message at a time. Furthermore, it suggests that the algorithm is optimal in the general case.

4.6. Applications

Traversal-2 and -3, which are different implementations of the same algorithm, can be modified to produce a useful structure, called *infrastructure*, on the network. The infrastructure is the combination of an in-directed spanning tree and

an out-directed spanning tree. The in-tree construction was detailed in section 4.3.3. Here we will describe how an out-tree may be produced by Traversal-2. Note that the defined infrastructure is a strongly connected subnetwork which spans the network and has at most $2n$ links. The infrastructure proves to have several practical applications.

4.6.1. Producing a spanning out-tree

An *out-directed tree* (or, out-tree) is a subnetwork in which every node, except one, called the *root*, has exactly one in-coming link and the underlying undirected graph is a tree. Since every node in the out-tree has exactly one incoming link, there is a unique path from the root to every node in the out-tree. An *out-directed spanning tree* is an out-tree which spans the network. An example of an out-tree is given in figure 4.10.

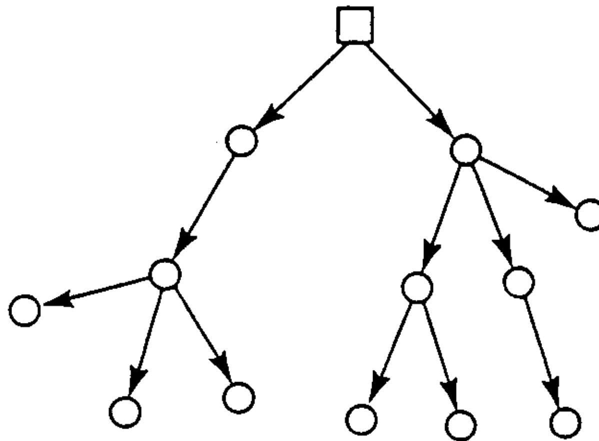


Figure 4.10: An example of an out-tree

We now explain how every node in the network marks some of its out-going links *outtree* during the traversal algorithm such that, when the algorithm terminates the collection of outtree marked links constitute a spanning out-tree of the network rooted at the traversal initiating node.

To construct an out-tree we make the following observation: The collection of father links constitute an out-tree. To prove it note the following: (1) Every node, except the root, has exactly one father incoming link, and (2) Going backward on the path defined by the father links from any node v we always arrive at the root. Note (2) follows the fact that the incoming father link of any node v connects v to a node which was explored before v thus, we cannot close a cycle and we must arrive at the root.

To detect the father out-going links of node v we observe the following: The traversing process leaves node v twice or more through link l (in Traversal-2) while v is *active* if and only if l is a father link. Thus, every *active* node counts whether or not each of its out-going links has been on a backtracking cycle more than once. If a link participated in a backtracking cycle more than once while v is active then it is a father link and is marked outtree.

4.6.2. Applications of the traversal algorithm

In this section we show how the traversal algorithm and its resulting infrastructure can be used to solve other problems. In particular we apply the traversal algorithm to perform broadcasting, route messages, and to systematically emulate any bidirectional algorithm on a unidirectional network. Each of the applications can be solved by a traversal; However, after executing the traversal once, the application problems can be solved more efficiently by using the infrastructure produced

by the traversal.

4.6.2.1. Broadcast with Echo

The problem of broadcasting with echo can be solved on a unidirectional network by a traversal algorithm. In the broadcast with echo problem one node, the root, has a piece of information which it sends to all the nodes in the network, and the root gets a positive acknowledgment that all the nodes have received the information.

A straightforward solution to the problem will use a traversing process to carry the information on it. The message complexity of this solution is the message complexity of the traversal algorithm, $O(n \cdot |E|)$.

After one traversal, the next broadcast with echo can be more efficiently performed by traversing only the infrastructure links. Since the infrastructure defines a strongly connected network any node (not only the root) may start a traversal for this purpose. The complexity of this traversal is $O(n^2)$ since the number of links in the infrastructure is at most $2n - 2$.

After one traversal was performed, a further improvement can be achieved as follows: Every node which wants to start a broadcast with echo first sends the information of the broadcast to the root node along the outtree marked links, and then the root node starts a broadcast with echo as described below. After receiving the echo, the root node will broadcast an echo on the out-tree links.

The infrastructure can be used for an efficient broadcast with echo from the root as follows: The root sends the broadcast message on all its out-going links in the infrastructure. Upon receiving the broadcast message for the first time, every

other node sends the message to all its out-going neighbors in the infrastructure. Any other copy of the broadcast message is discarded. This implements the broadcast part of the algorithm. To notify the root that all the nodes in the network have received the broadcast message, every node v sends an echo as follows: After sending the copy of the broadcast message, v sends an echo over all of its infrastructure outgoing links except the link marked intree. Node v sends an echo over the intree marked outgoing link only after an echo has been received on all of its infrastructure incoming links. When the root has received the echo over all of its infrastructure incoming links, the notification has been completed, and a message to this effect is sent on the outtree marked links.

The average message complexity of the resulting broadcast with echo algorithm is $6n$ (averaging out the first broadcast, which is a regular traversal). Sending the broadcast message from the initiating node to the root costs at most $n-1$ messages. The broadcast message is then transmitted once over each infrastructure link which adds at most $2n-2$ messages to the complexity. The echo message is also sent once over each infrastructure link, hence another $2n-2$ messages. Then, to echo the initiating node, another $n-1$ messages are transmitted on the outtree marked links.

4.6.2.2. Messages sending

The in- and out-trees which result from the traversal algorithm enable us to efficiently send a message from every node to every other node. To pass a message from node v to node u, node v sends the message along the intree marked links to the root which then broadcast the message on the out-tree. Thus, at most $2n-2$ messages are sent in the routing mechanism, in order to send a message from any node to any other node.

4.6.2.3. Emulating bi-directional distributed algorithms

The above routing mechanism can be used as a means to emulate any bidirectional distributed algorithm on a strongly connected unidirectional network. Whenever a node has to send a message on an incoming link, it will use the above message passing mechanism. Thus, if a problem has a bit complexity $O(P(n))$ on a bidirectional network, then its complexity on the unidirectional network is upper bounded by $O(n \cdot P(n) + n^2 \cdot \log n + n \cdot |E|)$ (the last two terms are entailed by the construction of the infrastructure).

CHAPTER 5.

ELECTION IN UNIDIRECTIONAL NETWORKS

In this chapter we present a distributed algorithm for election in strongly connected unidirectional networks. The algorithm requires $O(\log n)$ bits of memory in each processor and its communication complexity is $O(n \cdot |E| + n^2 \log n)$ bits.

5.1. Introduction

The strongly connected unidirectional network is the most general network model, in the sense that every network topology, bidirectional or unidirectional, can be modeled as a unidirectional network by replacing any bidirectional link with two anti-parallel unidirectional links. Hence, any distributed algorithm for strongly connected unidirectional networks is also an algorithm for any other network model.

To design an election algorithm for strongly connected unidirectional networks the traversal algorithms of Chapter 4 can be used in various ways. First, in a straight forward approach, every initiator starts a traversal. When ever two traversals meet, the lower id one is destroyed. The worst case communication complexity of this algorithm is $O((n \cdot |E| + n^2 \cdot \log n) \cdot n)$ bits, as $O(n)$ traversals could be initiated such that each spends $O(n \cdot |E| + n^2 \cdot \log n)$ bits. Second, the modular technique of Korach et al. [Kor85] could be used to economically eliminate traversals. Using their technique the communication complexity is reduced to $O((n \cdot |E| + n^2 \cdot \log n) \cdot \log n)$ bits.

No algorithm for election in unidirectional networks has come to our attention prior to the one presented here. However, two bidirectional algorithms, the shortest path algorithm of Gallager [Gal76] and the connectivity checking algorithm of Segall [Seg83] can easily be modified into a unidirectional election algorithm. In [Seg83], Segall presents a connectivity checking algorithm upon whose termination every node knows the ids of all the other nodes connected to it. The shortest path algorithm in [Gal76] exhibits the same property when it terminates. The communication complexity of the two algorithms is $O(n \cdot |E| \cdot \log n)$ bits, and each node is assumed to have $O(n \log n)$ bits of memory.

The unidirectional variation of the two algorithms proceeds in two phases: In the first phase, every node acquires the ids of its incoming neighbors; in the second, it acquires the ids of all the other nodes in the network.

Let an *incoming* neighbor of node v be a node at the other end of an incoming link of v, and let *in–neighbors* of v be the set of all the incoming neighbors of v. Let the *record* of node v be a two-field data structure, of which the first contains the id of v and the second the ids of v's in-neighbors. In the first phase, every node transmits its id on all its incident outgoing links. In the second phase, every node broadcasts its record to all the other nodes in the network. For this purpose, each node v maintains two sets of ids, the *received* set and the *known* set. The received set contains the ids of the nodes whose records were already received by v. The known set contains ids which appeared in a record of at least one node from the received set, i.e., ids of nodes whose existence is known to v. Initially, the received set of node v contains the id of v, and the known set contains the ids of v and of v's in-neighbors. Clearly, when the two sets in a node are identical, they contain the ids of all the nodes in the network (which can easily be proved by induction).

The communication complexity of the algorithm thus described is $O(|E|^2 \cdot \log n)$ bits; however, assuming that messages sent over one link are received in the order transmitted, the communication complexity can be reduced to $O(n \cdot |E| \cdot \log n)$ bits by avoiding repeated transmission of the same id over the same link. Note, that for these algorithms each node is assumed to have $O(n \cdot \log n)$ bits of memory.

In this chapter we present an election algorithm for general strongly connected unidirectional networks, whose communication complexity is $O(n \cdot |E| + n^2 \cdot \log n)$-*bits*, using $O(\log n)$ bits of memory in each node. The algorithm yields two directed spanning trees, both rooted at the leader; one is an incoming tree, and the other is an outgoing tree. The algorithm is thus an improvement on the algorithms of Gallager and Segall both in terms of communication complexity and in terms of the number of memory bits required at each node. Furthermore, unlike our algorithm, neither Segall's nor Gallager's algorithm provides the spanning trees.

The election algorithm presented here is a generalization of the traversal algorithm from the previous chapter. On one hand, the unidirectional traversal algorithm with a predefined in-tree (Section 4.3.2.) is a building block of the election algorithm. On the other hand, the traversal algorithm with the in-tree construction (traversal-2) can be seen as a special case of the election algorithm. The traversal algorithm is the election algorithm, under the constraint that only one node starts the algorithm. The resulting traversal algorithm incurs the same communication cost as the election algorithm.

5.2. A Unidirectional Election Algorithm

In this section we present a recursive distributed algorithm for election in unidirectional strongly connected networks.

5.2.1. Definitions and Outline

The election algorithm is based on the following recursive properties of strongly connected directed multigraphs:

1. The set of links, defined by selecting one outgoing link from every node, contains a nonempty set of disjoint directed cycles.
2. The subgraph, obtained from G by contracting any of the cycles defined above into one node, results in a strongly connected multigraph.
3. Repeated application of the operations in 1 and 2 contract G into a single node.

The distributed algorithm proceeds in conceptual phases, which follow the above contraction process. When a cycle is detected, its nodes are grouped into a cluster. Similar phases are used in [Hum83]. Initially we consider each node to be a single node cluster. The phases of the algorithm are: selection of an outgoing link from each cluster, called a *selected* link; detection of cycles among clusters; and contraction of cycles of clusters.

A cluster is recursively defined as follows:

1. A single node is a cluster.
2. A set of clusters that are joined in a ring by their selected outgoing links is a

cluster.

Recursively we assume that every cluster satisfies the following 4 properties (see figure 5.1):

1. A unique node in the cluster is distinguished as the *cluster head*.

2. All the nodes in the cluster know the id of the cluster head which is also the *id of the cluster*.

3. Each node in the cluster, except the cluster-head, has one outgoing link marked as *intree* link. The collection of intree links forms a directed incoming tree, spanning the cluster and rooted at the cluster-head.

4. A strongly connected subnetwork which spans the cluster, called the *infrastructure* of the cluster, is defined on the cluster.

Clearly, a single node cluster satisfies the inductive assumptions. It is: the cluster-head of itself; a single node in-tree; and a strongly connected subnetwork. To describe the algorithm we will describe the inductive step, i.e., we assume that a set of clusters which satisfy the assumptions already exists and explain how a bigger cluster which satisfies the inductive assumptions is composed out of this set.

To select a cluster outgoing link, each cluster head initiates a Depth First Traversal (DFT) process. The traversal process is used to search for a link which is potentially outgoing from the cluster. For the depth first traversal algorithm we use the traversal algorithm which was developed in section 4.3.2.

To detect a cycle, we use a simple algorithm for election on a unidirectional ring. Each cluster forwards the largest id it has seen. When a cluster receives the

same id twice it has detected a cycle.

The contraction of a cycle is accomplished by first electing one of the cluster-heads on the cycle to be the cluster-head of the expanded cluster. Then, the newly elected cluster-head synchronizes the cluster by broadcasting the new cluster-id to all the nodes and, constructing all the inductive requirements on the new cluster.

The most costly phase is the DFT in a search for a cluster outgoing link. The reason for this is that in the contraction phase we lose the DFT information accumulated by all the clusters around the cycle except one. When the cluster-head initiates a DFT in the next phase, the search will have to spend much effort regaining all the lost DFT information. However, by selecting the cluster-head of the largest cluster on the ring to be the new cluster-head, we minimize the amount of information lost. Thus, we limit the rate of information loss to the rate of cluster growth. (i.e. if a large cluster were contracted with a small one, the amount of information lost is proportional to the size of the small one). In fact, we are able to show that the cost of all the DFT's conducted during the algorithm, is within a constant factor of the cost of a single DFT. Since this point is critical in the complexity calculation, but rather minor to the description of the algorithm, we postpone a detailed discussion of it until the complexity section.

After a cluster is formed, its nodes are synchronized to search for an untraversed link outgoing from the cluster. To achieve this synchronization, the in-tree rooted at the cluster-head is used. When a cycle of clusters is contracted into a bigger cluster, all their in-trees are merged into one in-tree, spanning the new cluster. The operations of merging in-trees and searching clusters utilize each other alternately. The structures left by the DFT's are used to modify and merge the separate

in-trees around the cycle into one in-tree. In turn, the in-tree in a cluster is used for routing purposes, by its DFT process (see section 4.3.2.

In the next three subsections we present the three phases of the algorithm starting with the cluster outgoing link selection (see figure 5.1). During the algorithm, links are in one of three states: *new*, *elementary* or *killed*. A *new* outgoing link is one which has not yet been traversed. An *elementary* link is a link which was a cluster selected outgoing link during one of the previous stages. The set of elementary links within one cluster forms the infrastructure of the cluster. A *killed* link is a nonelementary link already traversed during the algorithm (i.e., an intra cluster nonelementary link).

5.2.2. Selection of a Cluster-Outgoing Link

Once a cluster is formed, its head node initiates a DFT algorithm to search the cluster's infrastructure for a node with an untraversed outgoing link. The first such link to be found is selected as the cluster's outgoing link. If it turns out to be an intra cluster link, the DFT continues where it was stopped in the search of another untraversed outgoing link. If no cluster outgoing link is found, the cluster contains all the nodes of the network, and the algorithm terminates.

For the completeness of the algorithm description we review the essential details of the DFT from section 4.3.2.

5.2.2.1. Distributed Depth First Traversal of Unidirectional Networks

A building block of the election algorithm is the distributed Depth First Traversal (DFT) of unidirectional networks in which an intree is defined. The root node of the intree initiates the DFT by spawning a process which visits all the nodes

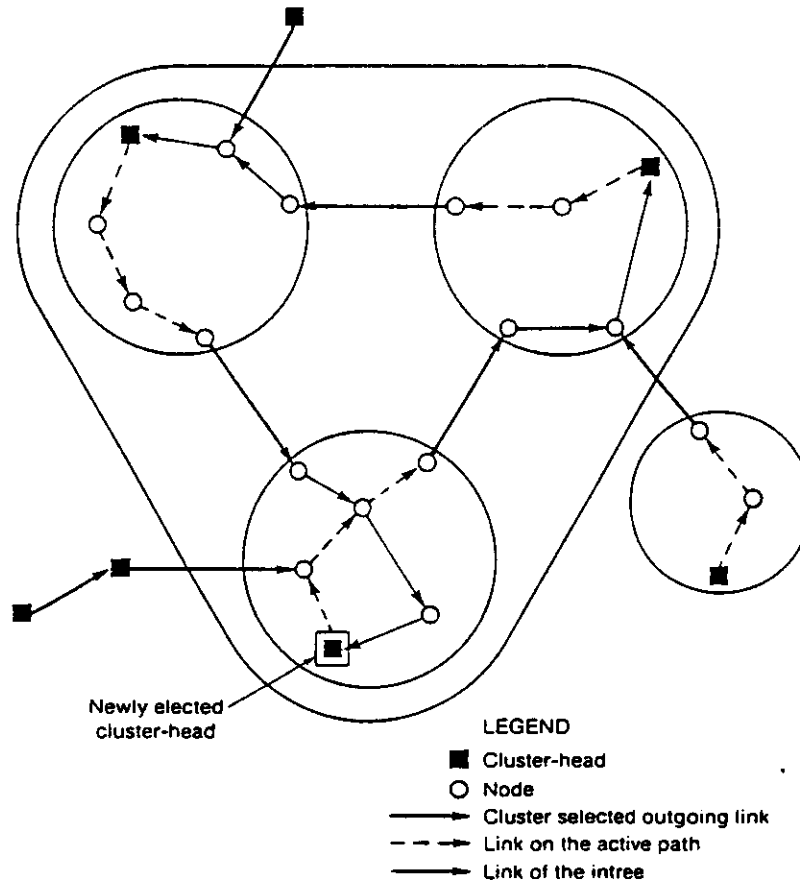


Figure 5.1: Clusters in the Election Algorithm

in the network. Upon arriving at a node for the first time, the process marks the node and recursively spawns one new child process, which sequentially visits each of the node's outgoing neighbors. If a process arrives at an already marked node, it backtracks to its father. After its child process has backtracked from all the outgoing neighbors, it is killed, and the process backtracks to its father. The traversal terminates when the child process of the root node is killed.

To perform the backtracking in a unidirectional network, we use the given intree and note two observations. Nodes on which live processes are located form a simple directed path. In the sequel we call this path an *active path* and its links *active links*. The active outgoing link of each process leads to its child process.

The first node on the active path is the root of the intree. All backtrackings are from the last node on the active path to its father. Thus, to accomplish backtracking, a process follows the unique path of the predefined intree, from its location to the root. From the root, the process follows the active path to its father (whose identity each process remembers).

Each backtracking performed requires at most $2n$ messages of $\log n$ bits each. Since there are $|E|$ backtrackings, the total communication cost of the DFT is $O(n \cdot |E| \cdot \log n)$ bits.

Note that at any given time, all but one of the live processes are waiting for their child processes to terminate. The last process, which has no child and is actively visiting nodes, is called the *focal point* of the DFT. The focal point is analogous to the stack pointer in the centralized Depth First Search algorithm [Tar72].

We can view the DFT as a token traversal scheme in which the token location is the focal point of the DFT. In the algorithm each cluster will employ a DFT to choose one outgoing link as the cluster's selected outgoing link.

5.2.2.2. Selecting a cluster outgoing link

The cluster outgoing link selection phase begins after the synchronization of the cluster has terminated. The head node of the new cluster initiates a DFT token on the cluster's infrastructure. The token carries the id of the cluster which created it and the maximum cluster-id observed so far (maximum-id). The cluster-id is used to distinguish between inter- and intra-cluster links, and the maximum-id is used to detect a cycle of clusters. The DFT token searches the cluster for a new link, i.e., a link never used by the algorithm. Doing so, the DFT token leaves behind a trail of

active links, which leads from the cluster-head to the token location (which is the traversal focal point).

Upon finding a new outgoing link, l, the token traverses link l to node v on the other end of l. If v belongs to the same cluster, the token returns to l's tail via the in-tree and the active path. Link l is then marked killed, and it will never again be traversed. If, on the other hand, the token arrives at a different cluster, the information it carries and the newly selected link it has established enable the cycle detection to continue as described below.

5.2.3. Cycle Detection

For the sake of simplicity, we have selected an inefficient algorithm for cycle detection. Since its complexity is, in general, not the bottleneck, we have avoided discussing an improved mechanism for cycle detection (The improvements are a generalization of [Pet82, Dol82], with which our algorithm will perform optimally on rings).

After each cluster selects an outgoing link, the network contains at least one cycle which consists of two or more clusters (see figure 5.1). Let each cluster send its id on the cluster outgoing link. A cluster forwards another cluster-id only if it is larger than all the cluster-ids it has received in the past. Eventually, one and only one cluster in each cycle will receive the same cluster-id twice, thus detecting a cycle. The cluster-head that detected the cycle synchronizes the new cluster.

The operation of cycle detection is carried out by the cluster-heads. To implement it, each node receiving a message from a different cluster forwards it to its cluster-head through the cluster's in-tree. To forward a maximum-id from a cluster-head to the next cluster, the cluster-head broadcasts the maximum-id over the

infrastructure of its cluster. All nodes retain a maximum-id variable, which is updated by the maximum-id of the broadcast message. The DFT token updates its maximum-id to the largest it encounters along its way. If a cluster outgoing link has been selected, the broadcast message is simply forwarded over the outgoing link to the next cluster.

When a cluster-head receives a maximum-id which is equal to its own, it has detected a cycle and it is elected to start the synchronization phase.

5.2.4. Cycle Contraction and Cluster Synchronization

In the previous phase a new cluster with an elected cluster-head was found. In this phase the elected cluster-head satisfies the remaining three inductive assumptions on the new cluster. The elected cluster-head thus; (1) notifies all the nodes in the clusters around the cycle of their new cluster-id, (2) It combines the in-trees of all the clusters into one in-tree which spans the new cluster and is rooted at the elected cluster-head, and (3) It combines the infrastructures into one infrastructure spanning the new cluster. After receiving a positive acknowledgment that all the nodes have completed these constructions, the elected cluster-head starts the next phase of cluster outgoing link selection. The synchronization phase is carried out by a broadcast with echo mechanism on the cluster new infrastructure.

Upon receiving the first copy of the broadcast message, every node performs the following five operations in the following order: (1) updates its own cluster-id; (2) It marks its cluster-outgoing link (if it has one) as elementary; (3) It updates (if necessary) its intree mark; (4) It forwards copies of the broadcast message over its elementary outgoing links; and (5) It acknowledges the reception of the message through the in-tree. Any duplicate copies of the message are discarded.

The second operation above, has combined the infrastructures into one infrastructure which spans the new cluster.

5.2.4.1. Merging the in-trees

To merge all the in-trees around the cycle, we use the active paths in each cluster which lead from the cluster-heads to the head nodes of the selected outgoing links. The active paths are constructed during the DFT in the clusters outgoing link selection phase.

We notice that, if each node of a cluster that has an active outgoing link puts its intree mark on the active link, then the incoming spanning tree is rerooted to the *head* node of the cluster outgoing link (see figure 5.2). (Note that the head node of the cluster outgoing link belongs to the next cluster on the cycle of clusters.) To obtain a directed in-tree which spans all the clusters around the cycle, we perform this operation in all the clusters around the cycle except for the elected cluster. Thus, the in-tree formed is rooted at the elected cluster-head. This in-tree is used by the nodes to notify their new cluster-head of the contraction termination.

5.2.4.2. Acknowledging the broadcast

To notify the cluster-head that all the nodes in the new cluster are aware of their new cluster-id, every node sends an acknowledgment as follows: After receiving the first copy and making all the necessary updates, every node sends an acknowledgment over all the elementary outgoing links aside from the intree marked link. An acknowledgment is sent over the intree marked outgoing link only after an acknowledgment has been received on all the elementary incoming links.

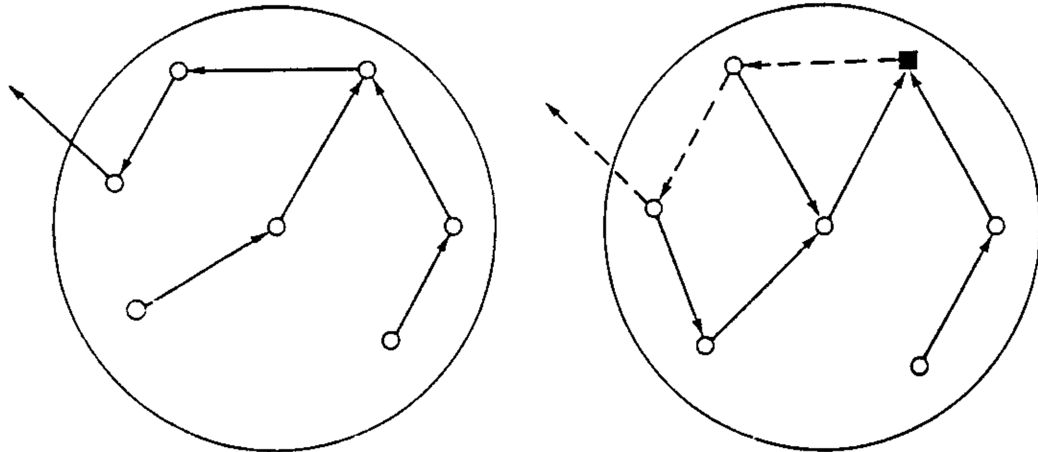


Figure 5.2: Rerooting an in-tree

When the elected cluster-head has received the acknowledgment message over all of its elementary incoming links, the contraction has terminated, and a new phase of cluster outgoing link selection is started.

5.2.5. Termination

The algorithm terminates when a cluster fails to select a cluster-outgoing link. This cluster spans the whole network, its cluster-head is the elected node and its maximum-id is the largest id.

5.3. Complexity of the Election Algorithm

Communication complexity analysis involves counting the total number of bits transmitted over all the network links. The communication cost of the algorithm has three components: the cost of cluster synchronizations, the cost of cycle detections and the cost of cluster-outgoing link selections.

We observe the following two facts.

Fact 1: The number of cycle detections is, at most, $n-1$.

Fact 2 : In a cluster of size k, there are, at most $2k-1$ *elementary* links.

Fact 1 holds because the contraction of a cycle strictly reduces the network size. Fact 2 follows from fact 1 and the observation that there exists a one-to-one correspondence between clusters and elementary links.

5.3.1. Cluster Synchronization Cost

Lemma 5.1: The total cost of synchronizing the clusters is at most $O(n^2 \log n)$ bits.

proof: According to fact 1 there are, at most, $n-1$ cluster synchronizations. The synchronization messages propagate on the infrastructure of a cluster which contains, at most, $2n-1$ links. In each synchronization one broadcast message and one echo message are transmitted on each elementary link. Since each message is of size $O(\log n)$ bits the result follows. ■

5.3.2. Cycle Detection Cost

Lemma 5.2: The total cost of all cycle detections is at most $O(n^2 \log n)$ bits.

proof: Each time a new cluster head is elected a new maximum-id is sent around a cycle of clusters. Thus, by fact 2, there are at most $2n-2$ maximum-id initiations. Each such initiation is sent over the infrastructure of some cluster. The same maximum-id is forwarded at most three times on the same link in a particular cluster; Once when it enters the cluster and the link is an in-tree link on which it was forwarded to the cluster-head; Once when the cluster-head broadcasts the maximum-id;

and once on its return to that cluster-head (which then detects a cycle). Since, a link forwards the same maximum-id at most three times for each cluster it belongs to, the result follows. ■

5.3.3. The Cost of Cluster Outgoing Link Selection

The cost involved in selecting outgoing links consists of the cost of killing intercluster links and the cost of the DFT's.

To kill link l, the algorithm transmits one message of $O(\log n)$ bits over l and a kill message of size O(1) bits over a path from the head of l to its tail. The kill message goes down the in-tree to the cluster-head and then along the active path to the tail of l. This node is distinguished from other nodes on the active path since it is the focal point of the DFT. Thus, the killing of one link costs $O(n)$ bits. Since the algorithm kills, at most, $|E|$ links, the killing of intercluster links adds up to, at most, $O(n \cdot |E|)$ bits.

As mentioned, the cost of one DFT on a network with n nodes and $|E|$ links is $O(n \cdot |E| \cdot \log n)$ since there is one backtracking for each link of the network, and each backtracking costs $O(n \log n)$. The DFT operation is employed in the election algorithm to search the infrastructures of different clusters. Since there are at most twice as many links as nodes in an infrastructure, the cost incurred by each DFT of a cluster with k nodes is $O(k^2 \log n)$. As the DFT could be used $n-1$ times by the algorithm, the total cost of all DFT's might be $O(n^3 \log n)$.

To reduce the total cost of all DFT's from $O(n^3 \log n)$ to $O(n^2 \log n)$ bits, a special *cluster head election* phase is added after the cycle detection and before the synchronization phases. In this phase the cluster head which is elected in the cycle detection phase synchronizes the election of the cluster-head of the largest size

cluster around the cycle. The new cluster-head then proceeds with the cluster synchronization phase as described before. After synchronizing the cluster, the cluster-head resumes its DFT process from the previous stage, i.e., from the head node of its former cluster outgoing link, thus avoiding re-searching nodes that were already searched. The head node of its former cluster outgoing link is now the last node on the active path.

5.3.3.1. Cluster-Head Election

To elect a cluster-head of a largest size cluster (ties are resolved by cluster-ids), the cluster-head detecting the cycle sends an elect-message around the alternating sequence of active paths and in-trees which form a directed cycle (see figure 3.1). On its way, the elect-message finds out which cluster has the greatest number of nodes and what the total number of nodes in all the clusters around the cycle is. This information is updated on the elect-message by the cluster-heads along the directed cycle.

Once the elect message returns to the originating cluster-head, the control of cluster synchronization is passed to the newly elected cluster-head. Any new and larger maximum-id which arrives at the cluster-head detecting the cycle during the election and the synchronization phase is held back by this node. It is forwarded to the new cluster-head upon the reception of the synchronization broadcast.

After being elected, the new cluster-head resumes its DFT of the previous cluster outgoing link selection phase. Doing so it uses the active path and the node marks which its DFT had left. The algorithm thus can be viewed as a process in which all the cluster-heads are candidates for leadership. When clusters owned by different candidates form a cycle, the candidate which owns the largest size cluster

eliminates all the other candidates. In doing so the candidate merely has enlarged its cluster to include the clusters of the candidates it has eliminated. The DFT structures it had previously are then extended to search the enlarged cluster. Henceforth, we refer to the clusters which did not change their cluster-id as one cluster which has consumed other clusters during the algorithm.

The above scheme is similar in principle to the capturing rule of algorithm A in chapter 2 (Section 2.5). There we enabled the largest candidate, in terms of captured nodes, to kill and take nodes from a smaller candidate. To see that the above scheme does not add more than a constant factor to the complexity of a single DFT, we will use a lemma similar to Lemma 2.2 and which was introduced in a similar context by Gallager: [Gal77]

Lemma 5.3: For any given k , the number of clusters that own n/k nodes or more is, at most, k.

Proof: Let C_1 and C_2 be any two clusters which had size n/k at some point of time. We shall show that each of C_1 and C_2 must have had n/k nodes disjointly. If they have never consumed each other, we are done, since the clusters were certainly disjoint. If, w.l.o.g., C_1 has consumed C_2, then C_1 must have already had n/k nodes at the time of eliminating C_2. ■

Corollary : The largest cluster to be consumed by another cluster owns at most $n/2$ nodes, the next largest at most $n/3$, etc.

Thus, we arrive at the following:

Lemma 5.4: The total cost of the DFT's is $O(n^2 \log\ n)$.

proof: The cost of traversing a cluster of size k is at most $k^2 \log n$ bits. Hence, the

total cost is bounded by $\sum_{i=1}^{n} \left\lceil \frac{n}{i} \right\rceil^2$ messages. But, $\sum_{i=1}^{n} \left\lceil \frac{n}{i} \right\rceil^2 \leq \sum_{i=0}^{\lfloor \log n \rfloor} 2^i \cdot \left\lceil \frac{n}{2^i} \right\rceil^2$ $= \sum_{i=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n^2}{2^i} \right\rceil \leq 2\,n^2$. Hence, the bit complexity of the depth first traversals is $O(n^2 \cdot \log n)$ bits. ■

Adding the bit costs of all three components, we arrive at a total communication complexity of $O(n^2 \log n + n \cdot |E|)$ bits for the whole algorithm. By the arguments presented in section 4.5 we conjecture that this is also the lower bound on the communication complexity of the election problem on arbitrary topology strongly connected unidirectional networks.

5.4. The Traversal Algorithm as a Special Case of the Election Algorithm

In this section we show how Traversal-2 and -3 can be derived as a special case of the election algorithm. Imagine the behavior of the election algorithm when it is started by a single node. In this case, the initiator initiates a process which visits all the nodes in the network and terminates. In the first subsection we show that the process can be modified to behave exactly as the process in Traversal-2. In subsection 5.4.2 we show how Traversal-3 can be derived from the election algorithm.

The process of deriving the traversal algorithms from the election algorithm provides a constructive proof of lemma 4.6.

5.4.1. Deriving Traversal-2 from the election algorithm

Assuming that only one node initiates the election algorithm, we make the following observations:

First, at any given time, at most one cluster is searched, i.e., no messages are exchanged in any of the other clusters.

Second, all the clusters and their selected outgoing links form a simple directed path in which each cluster is a node and each link is a selected outgoing link. This path, called the *clusters active path*, occasionally closes on itself (see figure 5.3). When the path closes either a cycle of clusters is formed, or the last link on the path is an intra cluster link (in the last cluster on the path, see figure 5.3, cases 2 and 1, respectively).

Third, consider the cluster's active path when it does not close on itself. Then, the nodes at which the cluster's active path enters clusters are the first nodes to be explored in each cluster. Therefore, if these nodes were selected as the cluster-heads of their clusters, the active paths of all the clusters would form a single contiguous path at all times.

We now modify the election algorithm according to the above observations namely, in each cluster the first node to be explored is elected as the cluster-head. Cycle detection (case 2 in figure 5.3) occurs when the token leaves one cluster C_1 (by traversing its selected outgoing link for the first time) and arrives at another, previously explored, cluster C_2. Recalling the third observation, the cluster-head of C_2 is the "oldest" node on the newly formed cycle of clusters. This cluster-head (h) synchronizes the cycle of clusters and, is elected to be the cluster-head of the new cluster. Also, (from the third observation) the active paths around the cycle form a single contiguous path. If the synchronization phase is modified to leave all the active paths intact, the distinct DFT's may be considered as one DFT. Thus the newly elected cluster-head, h resumes a DFT which already has an active path going through all the clusters around the cycle. As a result, no node in the network is

searched more than once during the whole algorithm. The cluster-head resumes the DFT at the node in which the cycle was detected (the LOOP node in figure 5.3, case 2).

The above variation of the election algorithm can be viewed as a traversal process. One node spawns the process which terminates at the initiator after visiting all the nodes in the network, one at a time. This traversal process can be further modified to work on a unidirectional network of finite automata, as we show in the next section.

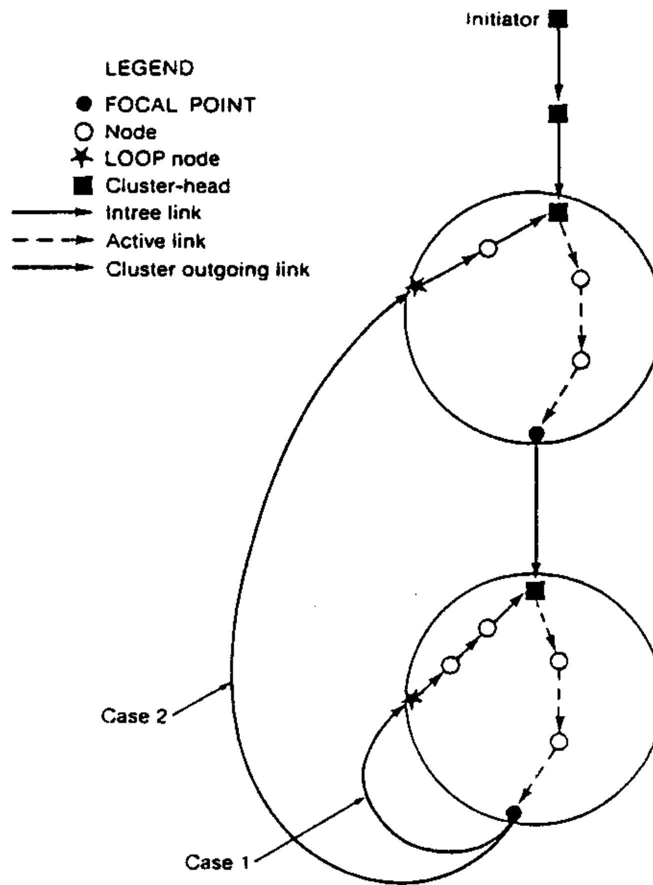


Figure 5.3: Clusters in the Traversal Algorithm

5.4.2. Deriving Traversal-3 from the election algorithm

Two operations in the preceding traversal algorithm require $O(\log n)$ bits memory in each node. The first is distinguishing an intra cluster from an intercluster link. The second (which occurs during the DFT of the infrastructure) is backtracking from the last node on the active path.

In the first operation, the $O(\log n)$ bits are used to distinguish between the case that a newly traversed link, l, is an intra cluster link and the case that l closed a cycle of clusters. This operation was accomplished in the preceding algorithm by comparing the id carried by the token with the cluster-id of the head node of l. To perform the operation without ids, we note that, in both cases, l closed a directed cycle of nodes and links (composed of an active path followed by a path in an in-tree, see figure 5.3). In the first case, exactly one cluster-head resides on the cycle while, in the second case, at least two cluster-heads reside on the cycle (see figure 5.3).

We now explain how the token decides whether there is more than one cluster-head on the cycle. The last node on the active path in each cluster is marked focal point (it is the focal point of that cluster's DFT, see section 3.2). The head node of a newly traversed link l is marked LOOP if it is an already explored node. Upon arriving at a LOOP node, the token is sent around the cycle to find out whether there is more than one cluster-head on it. Since there is exactly one LOOP node on such a cycle, the walk around it utilizes a finite number of bits on the token. If exactly one cluster-head was found, the LOOP mark is removed. The token then walks around to the focal point, kills l and resumes the DFT. If on the other hand, more than one cluster-head was found, a cycle of clusters was identified. The token then makes another trip around the cycle in order to synchronize its clusters. On the

second round, the token first erases all the focal point marks. Second, it erases all the cluster-head marks, aside from the first. Third, the token modifies the in-trees of all clusters, aside from the first one, to be rerooted at the LOOP node (in the same way as was done in the election algorithm). Notice that the active path of the first cluster lies between the first and the second cluster-head. Arriving at the LOOP node for the second time, the synchronization phase terminates. The LOOP mark is removed, the focal point mark is put on, and the DFT is resumed.

The backtracking in the DFT is performed without using node ids by using the same solution that was suggested in section 4.4.2.

The communication cost of the cycle detection and synchronization phases does not change in the modified algorithm. The cost of the cluster outgoing link selection phases is $O(n^2 \log n)$ bits since the DFT searching for outgoing links requires one backtracking for each link of the infrastructure. Thus, the communication complexity of the traversal algorithm is the same as that of the election algorithm.

5.5. Concluding Remarks

The election algorithm can be modified to produce an out-tree in the same way that Traversal-2 and -3 were modified in the previous chapter. The combined structure of the in-tree and out-tree can be used in many different ways as was suggested in the previous chapter.

As shown in [Gaf84], any algorithm which requires common knowledge is equivalent to an election algorithm. Therefore, we expect the election and traversal algorithms to serve as building blocks in many unidirectional network algorithms. An example of such an application, which involves termination detection, can be

found in [Mis83].

An interesting observation is that the amount of communication in the unidirectional election algorithm, $O(n \cdot |E| + n^2 \cdot \log n)$ bits, is n times the number of messages in the optimal bidirectional algorithm. We would obtain the same cost if we were to simulate the bidirectional algorithm, [Gal83] with each acknowledgment charged as n bits, on a unidirectional network. Together with lemma 4.9, this leads to the conjecture that our election algorithm is as efficient as possible in terms of communication cost.

References

[Aho83] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms,* Addison-Wesley Publishing Company (January 1983).

[Als76] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources," in *Proceedings 2nd Int. Conf. Software Eng.*, San Francisco (October 1976).

[Arj83] E. Arjomandi, M. J. Fischer, and N. A. Lynch, "Efficiency of Synchronous Versus Asynchronous Distributed Systems," *JACM* 30(3), pp.449-456 (July 1983).

[Bur80] J. E. Burns, "A Formal Model for Message Passing Systems," TR-91, Indiana Univ., Bloomington (May 1980).

[Cha79] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Communication of the ACM* 22(5), pp.281-283 (May 1979).

[Dal77] Y. Dalal, "Broadcast Protocols in Packet Switched Computer Networks," Ph. D. Dissertation, Electrical Engineering, Stanford University (April 1977).

[Dol82] D. Dolev, M. Klawe, and M. Rodeh, "An O(n log n) Unidirectional Algorithm for Extrema Finding in a Circle," *Journal of Algorithm* 3, pp.245-260 (1982).

[Eve79] S. Even, *Graph Algorithms,* Computer Science Press, Rockville Maryland (1979).

[Fre84] G. N. Frederickson and N. A. Lynch, "The Impact of Synchronous Communication on the Problem of Electing a Leader in a Ring," pp. 493-503 in *Proceedings of the 16th Ann. ACM Symp. on Theory of Computing*, Washington, D.C. (1984).

[Gaf84a] E. Gafni and W. Korfhage, "Distributed Election in Unidirectional Eulerian Networks," in *Proceedings Twenty-Second Annual Allerton Conference on Communication, Control, and Computing*, Allerton, IL (October 3-5, 1984).

[Gaf84b] E. Gafni, M. Loui, P. Tawiri, D. West, and S. Zaks, "Lower Bounds on Common Knowledge in Distributed Algorithms, with Applications," Preprint, Univ. of Illinois (January 1984).

[Gaf85] E. Gafni, "Improvements in the Time Complexity of Two Message-Optimal Election Algorithms," in *Proceedings of the ACM Symp. on Principles of Distributed Computing*, Minacki Ontario (August 1985). also UCLA CSD-85001 January 1985.

[Gal76] R. G. Gallager, "A Shortest Path Routing Algorithm with Automatic Resynch," , M.I.T. (March 1976). Unpublished note.

[Gal77] R. G. Gallager, "Finding a Leader in a Network with O(E) + O(N log N) Messages," , M.I.T. (1977). Unpublished Note.

[Gal83] R. G. Gallager, P. A. Humblet, and P. M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees," *ACM Trans. Program. Lang. Syst.* **5**, pp.66-77 (Jan 1983).

[Hir80] D. S. Hirschberg and J. B. Sinclair, "Decentralized extrema finding in circular configurations of processors," *Comm. ACM* **23**, pp.627-628 (November 1980).

[Hop73] J. Hopcroft and R. Tarjan, "Algorithm 447: Efficient Algorithms for Graph Manipulation," *Comm. ACM* **16**, pp.372-378 (1973).

[Hum83] P. A. Humblet, "A Distributed Algorithm for Minimum Weight Directed Spanning Trees," *IEEE Trans. Comm.* **COM-31**(6), pp.756-762 (June 1983).

[Hum84] P. A. Humblet, "Selecting a Leader in a Clique in O(N log N) Messages," pp. 1139-1140 in *Proceedings of 23rd Conference on Decision and Control*, Las Vegas, Nevada (December 1984).

[Kah78] R. E. Kahn, S. A. Gronemeyer, J. Burchfiel, and R. C. Kunzelman, "Advances in Packet Radio Technology," *IEEE Proceedings*, pp.1468-1496 (November 1978).

[Kor84] E. Korach, S. Moran, and S. Zaks, "Tight Lower and Upper Bounds for Some Distributed Algorithms for a Complete Network of Processors," in *Proceedings of the ACM Symp. on Principles of Distributed Computing*, Vancouver BC (August 1984).

[Kor85] E. Korach, S. Kutten, and S. Moran, "A Modular Technique for the Design of Efficient Distributed Leader Finding Algorithms," in *Proceedings of the ACM Symp. on Principles of Distributed Computing*, Minacki Ontario (August 1985).

[LeL77] G. LeLann, "Distributed systems - towards a formal approach," *Information Processing 77*, pp.155-160 (1977).

[Lyn81] N.A. Lynch and M.J. Fischer, "On describing the behavior and implementation of distributed systems," *Theoretical Computer Science* **13**, pp.17-43 (1981).

[Men78] D. A. Menasce, R. Muntz, and J. Pokek, "A Locking Protocol for Resource Coordination in Distributed Databases," *Proceedings ACM SIGMOD* (June 1978).

[Mis83] J. Misra, "Detecting Termination of Distributed Computations Using Markers," pp. 290-295 in *Proceedings of the ACM Symp. on Principles of Distributed Computing*, Montreal (August 1983).

[Pac82] J. Pachl, E. Korach, and D. Rotem, "A Technique for Proving Lower Bounds for Distributed Maximum-Finding Algorithms," pp. 378-382 in *Proceedings of the 14th Ann. ACM Symp. on Theory of Computing*, San Francisco CA (1982).

[Pet82] G. L. Peterson, "An O(n log n) Unidirectional Algorithm for the Circular Extrema Problem," *ACM Trans. Program. Lang. Syst.* **4**(4), pp.758-762 (October 1982).

[Pet84] G. L. Peterson, "Efficient Algorithms for Elections in Meshes and Complete Networks," TR 140, Dep. of Computer Science, Univ. of Rochester, Rochester, New York (August 1984).

[Rin77] J. Rinde, "Routing and control in a centrally directed network," in *Proceedings AFIPS* (June 1977). Volume 46.

[Seg83] A. Segall, "Distributed Network Protocols," *IEEE Transactions on Information Theory* **IT-29**(1) (January 1983).

[Sei85] C. L. Seitz, "The Cosmic Cube," *Communications of the ACM* **28**(1), pp.22-33 (January 1985).

[Spi77] P. M. Spira, "Communication Complexity of Distributed Minimum Spanning Tree Algorithms," pp. 236-244 in *Proceedings Second Berkeley Workshop on Distributed Data Managment and Computer Networks* (May 25-27, 1977).

[Tar72] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.* **1**, pp.146-160 (1972).

[Tym71] L. R. Tymes, "TYMNET-A terminal oriented communication neteork," in *Proceedings AFIPS* (spring 1971). Volume 38.

[Vit84] P. Vitanyi, "Distributed election in an Archimedean ring of processors," pp. 542-547 in *Proceedings 16th ACM Symp. on Theory of Computing*, Washington, D.C. (1984).