5.6 Problems

1. Note that in Section refaw we did not assume equal rates of the local clocks, while in Section 5.2.3 we did make this assumption. Where exactly was this assumption used?

2. Write a code to implement the synchronizer in Section 5.2.3.

3. Write a code to implement the synchronizer in Section 5.2.3.

4. Write a code for the Frederickson & Lynch’s algorithm presented in section 5.3.1. Prove its correctness.

5. Improve the time complexity of Frederickson & Lynch’s algorithm presented in Section 5.3.1. Analyze the message complexity of your algorithm.

6. Prove the inequalities 5.14 and 5.16.
5.5 References


In [2] also the problem of inaccurate clocks is considered, and it is shown that in such case the synchronizers are still valid, though there is an additional overhead in time complexity. Improvement to the results in [2] are studied in [5]; one result there shows that it is possible to slow down the time by a factor of 2, without the need for an extra bit (as shown in section 5.2.3).
Example 5.2 For $b = 3$, we will thus ask whether the number is at most $f(1, 3) = 2$, $f(2, 3) = 4$, etc. Assume we find out that the smallest number is larger than $f(4, 3) = 15$ but not larger than $f(5, 3) = 26$. This means that it lies in the interval $[16, 17, \ldots, 26]$. But this interval contains $f(5, 3) - f(4, 3) = f(4, 2) = 11$ numbers, thus it can be searched by 4 questions and at most 2 overestimates, and we'll then find the minimum number within 9 questions and 3 overestimates.

### 5.4 Bibliographical notes

The algorithm presented in Section 5.3.2. For a network with $n$ processors, $e$ communication links, and a smallest identity $i$, a minimum finding algorithm that uses $O(e)$ bits and $O(ni)$ time was shown in [7].

The two basic synchronizers in Section 5.2.2 are from [1]. The synchronizers for the bounded delay in Section 5.2.3 network are from [2]. The time-messages trade-off in Section 5.3.2 is from [6].

In [1] the idea of synchronizer is further developed; actually, the main result of the paper is a synchronizers, that partitions the network into clusters, with a rooted spanning tree in each. Among each two neighboring clusters a preferred edge is chosen. The synchronizer then proceeds in two phases: At the first phase synchronizer $\beta$ is applied to each cluster separately, along its spanning tree. At the second phase synchronizer $\alpha$ is applied among the clusters, along the preferred edges.
(see Problem 6). By (5.12) we have

\[ m \leq f(h, m) \left( \frac{h + b}{b} \right), \]

and hence

\[ b \cdot m \leq (h + b)(h - b + 1) \cdots (h - 1). \]

Thus

\[ \left( b \cdot m \right)^{\frac{1}{b}} \leq [(h + b)(h - b + 1) \cdots (h - 1)]^{\frac{1}{b}} \]

\[ \leq \frac{1}{b} [(h + b) + (h - b + 1) + \cdots + (h - 1)] = h + \frac{b}{2} - \frac{1}{2}. \]

Therefore

\[ h \geq \left( b \cdot m \right)^{\frac{1}{b}} - \frac{b}{2} + \frac{1}{2}. \quad (5.17) \]

The lemma follows from (5.15) and (5.17).

\[ \square \]

Given only \( b \), we want now to have a procedure to find the minimum identity using the above procedure \textit{decide}, without an a-priori knowledge of its size. We show

**Theorem 5.4** Given \( b \), the minimum number \( i \) can be guessed within \( O(b^{\frac{1}{b}}) \) questions, and at most \( b \) overestimates.

**Proof:** We first extend the function \( f(t, b) \) to apply for any \( t \) and \( b \), subject to the recurrence relation (5.10). It can be easily shown that \( f(t, b) = 2^b \) for \( t < b \). This yields the numbers shown in Table 5.2.

The following procedure achieves the desired property. We first apply \textit{decide}(\( x \)), for \( x = f(1, b) \), \( x = f(2, b) \), etc. After exactly \( t = h(i, b) \) questions we’ll have a 'no' answer, and we’ll have to search for \( i \) in the interval \( (f(t - 1, b), \ldots, f(t, b)] \). But by (5.10) this means that we have to search in an interval of size \( f(t - 1, b - 1) \), which we can do using at most \( t - 1 \) questions and \( b - 1 \) overestimates. The overall procedure will thus use at most \( 2t - 1 \) questions and \( b \) overestimates.

By Stirling’s formula \( n! \approx \sqrt{2\pi n}(\frac{n}{e})^n \) (see [4]), and by (5.15), the lemma now follows.
Figure 5.1: Guessing game, with \( t=5 \) and \( b=2 \)

(see Problem 6). By (5.12) we have

\[
\binom{h}{b} \leq f(h-1, b) < m,
\]

and hence

\[ h(h-1) \cdots (h-b+1) < b! \cdot m. \]

Thus

\[ (h-b+1)^b < b! \cdot m, \]

and we have

\[ h < (b! \cdot m)^\frac{1}{b} + b - 1. \quad (5.15) \]

2. By induction on \((t, b)\), one can show that

\[ f(t, b) \leq \binom{t + b}{b} \quad (5.16) \]
Table 5.1: $f(t, b)$, for $1 \leq b \leq t$

Example 5.1 For $t = 5$ and $b = 2$, we have $f(5, 2) = \binom{5}{0} + \binom{5}{1} + \binom{5}{2} = 16$. The algorithm starts by guessing whether the minimum identity is larger than $f(4, 1) = 5$. The complete search tree resulted is shown in Figure 5.1.

Given $m$ and $b$, if we want to find the minimum number of question $t$ within which a minimum can be determined within the interval $[1, \cdots, m]$ with at most $b$ overestimates, we are assured that we can do it with $h = h(m, b)$ questions, where

$$h = \min \{ f(t, b) = \binom{t}{0} + \binom{t}{1} + \cdots + \binom{t}{b} \geq m \}. \quad (5.12)$$

By the previous discussion, this is clearly best possible. The next lemma gives an estimate for $h$, in terms of $m$ and $b$.

Lemma 5.2

$$h = (b! \cdot m)^{\frac{1}{b}} + \Theta(b) \quad (5.13)$$

Proof: 1. By induction on $(t, b)$, one can show that

$$\binom{t+1}{b} \leq f(t, b) \quad (5.14)$$
We consider several games, denoted by parameters \( N, t \) and \( b \), where \( 1, 2, \ldots, N \) denotes the interval where the number is known to lie, \( t \) is the total number of questions, and \( b \) is the total number of overestimates (no answers) allowed. In the following, a game is denoted by \(< N, t, b >\), while replacing any parameter by an * indicates that the parameter is unknown or that the goal of the game is to optimize it.

Consider first the game where we are given \( t \) and \( b \), and ask for the largest interval \([1, \cdots, f(t, b)]\) within which the smallest value can lie, and for an algorithm to to determine it.

**Theorem 5.3**

\[
f(t, b) = \left( \begin{array}{c} t \\ 0 \end{array} \right) + \left( \begin{array}{c} t \\ 1 \end{array} \right) + \cdots + \left( \begin{array}{c} t \\ b \end{array} \right).
\] (5.9)

Suppose our first question is \( Q(x) = \text{is the number } \geq x \text{?} \) if the answer is yes, the unknown number is greater than \( x \), and the player has to find it with \( t - 1 \) questions and \( b \) overestimates, and the largest interval that can be correctly searched is \([1, \cdots, x + f(t - 1, b)]\). If the answer is no, then the unknown number lies in the interval \([1, \cdots, x]\), to be searched for using \( t - 1 \) questions and at most \( b - 1 \) overestimates. Thus, the largest value of \( x \) which allows for a correct solution is \( f(t - 1, b - 1) \).

We therefore have

\[
f(t, b) = f(t - 1, b) + f(t - 1, b - 1).
\] (5.10)

Moreover,

\[
f(t, 1) = t + 1 \quad \text{and} \quad f(t, t) = 2t \quad \text{for every } t.
\] (5.11)

One can show now that the unique solution to (5.10), satisfying the boundary conditions (5.11), is the one given by (5.9).

The numbers \( f(t, b) \) are shown in the Table 5.1.

A correct interpretation proof yield the solution strategy. Given \( t \) and \( b \), the first question is \( f(t - 1, b - 1) \), and then we either move to a \(< *, t - 1, b - 1 >\) game or \(< *, t - 1, b >\) game. Note that if \( b = 1 \) then \( b - 1 \) is taken to be 1, and when \( t = b \) we have a complete binary tree.
$x_2 < x_1$, depending on the outcome of the $\text{decide}(x_1)$. The process will continue, until the smallest identity is determined. Thus, this technique allows us to reformulate the minimum finding distributed protocol in terms of a number guessing game, as follows:

A guessing game

- the network is a player.
- the minimum value in the network is a number, previously chosen and unknown to the player, which must be guessed.
- the player can ask questions of the type is the number greater than $x$, where each question corresponds to a simultaneous execution of $\text{decide}(x)$.
- situations 1 and 2 of Lemma 5.1 to a 'yes' and 'no' answer to the question, respectively.

Clearly each solution strategy to the guessing game corresponds to a solution algorithm to the minimum finding problem. As for the complexity of these algorithms, recall that, by Lemma 5.1 each execution of $\text{decide}$ (i.e., each question) requires $n$ time units, and that the number of bits transmitted is either 0 or at most $2n$, depending on whether the answer is yes or no, respectively. Therefore:

**Theorem 5.2** Let $S$ be a solution strategy to the number guessing game, which requires $b(x)$ overestimates and a total of $t(x)$ questions in the worst case, where $x$ is the unknown number. Assume $i$ is the smallest identity in the network, and that $n$ is known to all processors. Then

1. minimum finding can be performed in an anonymous synchronous network using at most $nt(i)$ time and $2eb(i)$ bits.

2. election in a synchronous network with distinct values can be performed using at most $nt(i)$ time and $2eb(i)$ bits.

3. a spanning tree can be constructed in a synchronous network with distinct values using at most $nt(i)$ time and $2e(b(i) + 2)$ bits.
by guessing that the smallest identity is, say, ‘10, and if this is the case then after $n$ time units all processors know it, and can move on to look for it in the set \{11, \ldots ,\}, thus saving $10n$ time units comparing to the previous procedure; however, if the number is in the set \{1, \ldots ,10\} then in this round $n$ bits will be used, and there will be at least one more such round, hence the bit complexity will be at least $2n$. A precise such trade-off is further explored in this section.

Suppose each processor in the network is performing the following procedure $\text{decide}(x)$, where the value $x$ is known to each processor:

$\text{decide}(x)$
\[
\text{clock} := 0; \\
\text{if} \ my\ id \leq x \ \text{then} \ [ \text{send yes} ; \\
\text{state} := \text{DECIDED} ] \\
\text{else state} := \text{UNDECIDED}; \\
\text{start counting}; \\
\text{if yes} \ is \ received \ with \ clock < n; \\
\text{if state} = \text{UNDECIDED} \ \text{then} \ [ \text{send the message}; \\
\text{state} := \text{DECIDED} ] \\
\text{else ignore the message};
\]

Lemma 5.1 On a unidirectional ring of size $n$, known to all processors, and given $x$, if all processors simultaneously start the execution of $\text{decide}(x)$, then at time $n$:

1. if all identities are greater than $x$, then all processors become UNDECIDED, and 0 bits have been sent.

2. if there exists at least one identity bounded by $x$, then all processors become DECIDED, and at most $2n$ bits have been sent.

Proof: Follows immediately from the algorithm and from the PI algorithm in Section 1.5.

Using this procedure, the minimum finding can be solved by applying a searching game, in which initially all processors will run $\text{decide}(x_1)$, and, based on the outcome, will run $\text{decide}(x_2)$, with $x_2 > x_1$ or
We sketch the main idea of the algorithm; its exact details are left as an exercise. Informally we will run LeLann’s algorithm, with the addition that identity $id$ will travel at a speed of $2^{id}$. To achieve this, a processor that wants to send a message holds it for $2^{id} - 1$ time units, and then sends it. It will take a message with smallest identity $\text{min}\_id$ a time $n \cdot 2^{\text{min}\_id}$ to complete a full cycle, using $n$ messages. By that time, the next smallest identity clearly makes at most $\frac{n}{2}$ steps, and, in general, the $k$'th smallest identity makes at most $\frac{n}{2^k}$ steps, which amounts to a total of $O(n)$ messages.

We thus have:

**Theorem 5.1** In a synchronous ring it is possible to elect a leader using at most $O(n)$ messages.

### 5.3.2 Unknown network size

In this section we show an algorithm, that finds the smallest identity in $O(kin^2)$ time and $O(kn)$ bits, for any integer $k > 0$, on a unidirectional ring of size $n$, that is unknown to the processors. Furthermore, this trade-off is optimal among the quite general type of algorithms considered. One consequence to this result that, in a synchronous ring with distinct identities, leader election can be performed with $O(kin^2)$ time and $O(kn)$ bits, for any integer $k > 0$. Extensions to general networks are also shown. The minimum finding protocol is reformulated as a combinatorial guessing game, which in turn has its uses beyond distributed computing.

Assuming that each processor in the ring is holding an integral identity, and that all processors start executing at time 0, the following procedure might be followed in order to find the minimum value: if a processor has a value 1, it sends a message to its neighbor; a processor that receives a message forwards it to its neighbor, unless it already sent it a message. If all the identities are larger than 1, then after $n$ units of time each processor knows that the smallest identity is larger than 1. The processors can then all start guessing whether the smallest identity is bounded by 2, and so on. If the smallest identity is $i$, this procedure will last exactly $i \cdot n$ time units, and use exactly $n$ bits. However, we can perform the procedure faster, using more bits; namely, we might start
Send-after-delay synchronizer

We now assume that a node sends its pulse \( m \) message \( \phi \) units of time after the start of its slot \( m \). Assume that \( P \) and \( Q \) are two neighboring processors, and that a message sent by \( P \) at pulse \( m \) is received by \( Q \) at time \( t \). Since it must arrive before the start of slot \( m + 1 \) and after the start of slot \( m \) at \( Q \) we have

\[
t_Q + m \tau \leq t_Q + (m + 1) \tau. \tag{5.6}
\]

We know from the delay bound and (5.1) that the following two inequalities hold:

\[
t < t_P + m \tau + 1 + \phi < t_Q + m \tau + 2 + \phi = t_Q + (m + 1) \tau + (2 - \tau) + \phi \tag{5.7}
\]

\[
t \geq t_P + m \tau + \phi > t_Q + m \tau - 1 + \phi = t_Q + (m - 1) \tau + (\tau - 1) + \phi. \tag{5.8}
\]

By choosing \( \phi = 1 \) and \( \tau = 3 \), we can satisfy both (5.7) and (5.8) (the actual code of the algorithm is left to the reader; see Problem 3).

5.3 Communication-time trade off

5.3.1 Known network size

Assume our unidirectional ring is synchronous, and that all identities are non-negative integers, and all processors start at time 0. It is crucial to note that the value of \( n \) is not known to any processor (see also Section 5.3.2). By time 1 all messages have been delivered to their destinations, at time 1 all processors take a step; in general, for every \( t \), by time \( t \) all time \( t - 1 \) messages have been delivered to their destinations, and at time \( t \) all processors take a step. As observed in [3], a better algorithm can be designed in this case.
By the nature of the PI algorithm, we have

\[ t_P - 1 < t_Q < t_P + 1 \]  \hspace{1cm} (5.1)

for every two neighboring nodes \( P \) and \( Q \).

**Send-on-start synchronizer**

First assume that each node sends its message at the beginning of a time slot (we term it the *send-on-start synchronizer*); that is, messages of time \( m \geq 0 \) (in the given synchronous algorithm) are sent at local time \( m\tau \). Assume that \( P \) and \( Q \) are two neighboring processors, that a message sent by \( P \) at pulse \( m \) is received by \( Q \) at time \( t \). Since it must arrive before the start of slot \( m + 1 \) at \( Q \) we have

\[ t_P + m\tau \leq t. \]  \hspace{1cm} (5.2)

Since this message has to arrive before \( Q \) sends its time \( m + 1 \) messages, we require that

\[ t < t_Q + (m + 1)\tau. \]  \hspace{1cm} (5.3)

However, by (5.1) and the bound on the delay we have

\[ t < t_P + m\tau + 1 < t_Q + m\tau + 2 = t_Q + (m + 1)\tau + (2 - \tau). \]  \hspace{1cm} (5.4)

Thus, by choosing \( \tau = 2 \), inequality (5.3) will follow from (5.4), and we are guaranteed that the synchronizer works correctly, in the sense that each processor will get all messages sent to it at time \( m \) before sending its time \( m + 1 \) messages, thus a simulation of the synchronous algorithm on the asynchronous network will be possible.

However, note that by (5.1) and (5.3) we can only guarantee

\[ t \geq t_P + m\tau > t_Q + m\tau - 1 = t_Q + (m - 1)\tau + (\tau - 1). \]  \hspace{1cm} (5.5)

For our choice \( (\tau = 2) \) this implies that messages for phase \( m \) will be received by all neighbors after the start of slot \( m - 1 \) (as well as before the start of slot \( m + 1 \)). Therefore we must add one bit to each message, to distinguish between these two cases. (The algorithm is left as an exercise; see Problem 2).
parent, while an internal node waits to know that it is safe and that all of its children are safe, and only then it sends a safe message to its parent. When the root learns that all the nodes are safe, it will again broadcast a message down the tree, by which all nodes will generate next pulse, and so on.

This synchronizer has the property of being expensive in time - it takes $O(|V|)$ units of time for generating each pulse - and cheap in messages - only $O(|V|)$ messages are sent per each pulse, on a network $G = (V, E)$.

### 5.2.3 Bounded delay networks

While it is true that most systems lack a common clocking mechanism, they do often guarantee message delivery within a fixed time bound. In addition, the nodes in these systems have highly accurate timers, that are not synchronized with each other, but operate at equal rates.

We now show that, given a bounded-delay network, the implementation of a synchronizer is relatively simple. We show two synchronization algorithms. In each the initialization phase takes $|E|$ messages. The first algorithm requires an additional bit in each message, and increases the time complexity by a factor of 2, while the other does not require any extra bit, but increases the time complexity by a factor of 3.

We assume that there is one vertex $r$ that starts the algorithm. We first broadcast an $\text{init}$ message, using the $PI$ algorithm (see Section 1.5). Each node, upon receiving the first $\text{init}$ message, resets its timer to 0 and starts counting time. Its time slot $m$ will begin at time $m\tau$ and end at time $(m + 1)\tau$, for $m \geq 0$ ($\tau$ will be determined later). Given a synchronous algorithm, the synchronizer will in the above way generate the pulses needed to run it on our bounded-delay network. We present one synchronizer, in which messages of pulse $m$ are sent at the beginning of time slot $m$, and another in which they are sent with a certain fixed delay delay.

Denote by $t_P$ the global time when node $P$ set its timer to 0. Note that the actual values of $t_P$ is not known to the processors, but used only in our discussion.
an acknowledgement to it, and then proceeds with its other operation. Note that this will double the overall message complexity of the algorithm; however, since in many protocols a form of acknowledgements already exists - implicitly or explicitly - (see, e.g., PIF in Section 1.5), the increase is practically lower. In this way, eventually a node knows when all of its messages, sent in its current pulse \( t \), have been received by its neighbors. We term such a node \textit{safe}. In order to start pulse \( t + 1 \), it is not sufficient for a node to be safe; it must make sure that all the messages sent to it by its neighbors at their pulse \( t \) have been received. However, if a node learns that all of its neighbors are safe, it can move to its next pulse. It thus remains to ensure a mechanism that will let a node know that all of its neighbors are safe. We describe two such synchronizers. The first one is local, highly-distributed, where pulses are generated fast, but cost a lot of messages. The second one is global, centralized, where pulses are generated slowly, which results in saving of messages.

\textbf{Synchronizer } \( \alpha \)

When a node knows that all of its messages of the current pulse have been received, it notifies all of its neighbors that it is safe. When a node receives \textit{safe} messages from all of its neighbors, it moves to its next pulse.

This synchronizer has the property of being expensive in messages - it uses \( O(|E|) \) messages per pulse - and cheap in time - clearly \( O(1) \), on a network \( G = (V, E) \).

\textbf{Synchronizer } \( \beta \)

Assume we have a leader in the network, together with a spanning tree rooted at it. Each node knows its parent and children in the tree. Initially the root will send a message, telling all of its children to generate the first pulse; each child in response will notify its children, etc. In this way the first pulse messages are being sent throughout the network. These messages are being acknowledged, as discussed above. When a node learns that it is safe and all of its children are safe, it sends a \textit{safe} message to its parent in the tree. This process starts at the leaves of the tree; a leaf waits to get acknowledgements to all its messages, then detects that it is safe and sends a \textit{safe} message to its
model, in which messages arrive after finite but otherwise arbitrary delays, and on the other one we find the fully synchronized models, where we assume that all the network work synchronously in pulses.

Though both assumptions are far from reality, they have their benefits. Designing of a protocol for a full asynchrony network results in a protocol that will work in stronger models, while providing a lower bound or impossibility result for the fully synchronous network model will result in the same bound for the asynchronous network model. Intermediate models are more realistic, and an example is shown in Section 5.2.3.

In the synchronous network model that we consider we assume that:

- There exists a clock at each processor, that generates pulses. (The rates of the clocks are immaterial.)

- Message sent from processor $P$ to processor $Q$ at pulse $t$ (of $P$) arrives at $Q$ before its $t+1$st pulse.

The synchronous model thus assumes a common clocking system for all nodes, and a bounded message delivery delay. Time thus can be thought of as being partitioned into slots, message transmission occurs at the beginning of each slot, and a message transmitted at slot $t$ is received and processed by the start of slot $t+1$.

Suppose we are given a protocol designed for such synchronous network. In order to run the protocol on an asynchronous network, we need a mechanism, that will generate pulses at each processor. This mechanism is termed *synchronizer*. Designing a protocol in such way might result in quite an efficient algorithm. In any case, its correctness proof is usually much easier than the one for a protocol for asynchronous network: one has to prove the correctness of the synchronous protocol, which is usually much easier task the for asynchronous network, and to prove correctness of the synchronizer. A synchronizer is, in other words, a distributed algorithm, which enables each node to define time slots and to detect the start and end of each slot.

### 5.2.2 General networks

Given a synchronous protocol, we first add to it a mechanism of acknowledgements: whenever $Q$ receives a message from $P$, it first sends
Chapter 5
SYNCHRONOUS NETWORKS

5.1 Introduction

In this chapter we study the issue of synchronization. We first present the notion of a synchronizer, that enables us to design an algorithm for a synchronous network, and run it on an asynchronous network. We present two simple such synchronizers, and present simple synchronizers for a network in which there is a known bound on the delay within which a message is guaranteed to be delivered.

Another interesting aspect of dealing with synchronous systems is due to the fact that in such systems, that operate in rounds, even not receiving a message in a certain round might bear some information, which thus enables us to save messages at the expense of having a longer execution (in terms of time). This issue of communication-time trade-off is presented by two examples; the discussion uses the ring network, for sake of simplicity. First we show that, assuming synchrony, one can get below the lower bound of Section 3.2, in a ring in whose size is not known to the processors. We then discuss the same problem, where the ring size is known to the processors; this small difference results in a completely different trade-off, with applications beyond the area distributed computing.

5.2 Synchronizers

5.2.1 The notion of a synchronizer

In studying protocols for distributed networks, we have in mind two extremes of networks’ models: on one side we have fully asynchronous
4.7 Problems

1. Modify the proof of the lower bound, to apply also for synchronous networks.

2. What is the number of messages sent in Humblet’s algorithm, if 1 processor starts the algorithm? What is the expected number of messages sent in Humblet’s algorithm, if 2 processor starts the algorithm? (assume any ‘reasonable’ probability distribution).

3. Prove that the complexity of leader election in a complete network of \( n \) processors, with exactly \( k \) that start the algorithm, is \( \Theta(n \log k) \).

4. Show that if, in Humblet’s algorithm, processor \( B \) will be allowed to forward more than one message to its master, than the message complexity of the algorithm will not be of \( O(n \log n) \).
4.6 References


Otherwise, by making at this point an exchange as described above the adversary creates a spanning tree of cost 0, which proves the algorithm to be incorrect. (Alternatively, the incorrectness of the algorithm follows from the fact that the same execution is possible for the new graph, which results in a spanning tree that is not of a minimum cost.) Thus, the algorithm must use a set of edges that intersects all such cycles before it stops.

It is easy to see that the \( \binom{n}{2} \) squares in the set

\[
B = \{ [(x_i, x_j), (x_j, y_i), (y_i, y_j), (y_j, x_i)] | 1 \leq i < j \leq n \}
\]

are disjoint.

This completes the proof of the theorem.

Remark: In the case where all the edge-weights are distinct, and arbitrarily long messages are allowed, a minimum spanning tree can be constructed in the following way: Choose a leader using the algorithm in [2] \( O(n \log n) \) messages. Each node transfers to the leader the weights of the edges adjacent to it \( O(n) \) "long" messages. The leader then reconstructs the weighted graph by assigning to the edge \((i, j)\) the unique weight that was transferred to it by both \(i\) and \(j\), finds the minimum spanning tree of \(G\), and transfer to each node the weights of the minimum spanning tree edges adjacent to it \( O(n) \) messages.

This above remark can be generalized to show that if we have, in a given graph \(G\), \(k\) edges of non-distinct weights, then a minimum spanning tree of \(G\) can be constructed in \( O(\max(k, n \log n)) \) "long" messages.

### 4.5 Bibliographical notes

The first \( O(n \log n) \) leader election algorithm for complete network was given in [5]. Simpler algorithms were subsequently presented in [3, 1, 7]. The algorithm in Section 4.2.1 follows [3]. A more general approach in [4] give another simple algorithm as a by-product. The lower bound in Section 4.2.2 are from [5], and the lower bounds in Sections 4.3 4.4 are from [6].
We take $G$ to be the complete graph of vertex set $V = X \cup Y$, where $X \cap Y = \emptyset$. We assume that $|V|$ is even and $|X| = |Y| = n \geq 2$. The proof is essentially the same for $|V|$ odd. At the beginning the adversary assumes that all the edges within $X$ and within $Y$ have a cost of 0, and all the edges connecting $X$ and $Y$ have a cost of 1. It is important to note that this initial structure of the graph $G$ is known only to the adversary; as far as each single vertex is concerned, it only knows that there are $n$ edges of cost equal 1 and $n - 1$ edges of cost equal 0 adjacent to itself.

The adversary can exchange costs of edges as described above. In fact, the adversary will make at most one exchange of costs of the type depicted in Figure 4.2 (where $x_1, x_2 \in X$ and $y_1, y_2 \in Y$) at the end of the algorithm. Therefore any minimum spanning tree in the resulting graph will have a cost of 0, and will use the edge $(x_1, y_1)$ or the edge $(x_2, y_2)$ or both.

When the algorithm starts, some vertex eventually asks for an (unused) edge of cost 0 or 1, and the adversary chooses such an edge at random from the initial structure above and gives it to this vertex. This process is repeated each time a vertex asks for an unused edge.

The minimum spanning tree produced by it uses at least one edge of cost 1 (connecting $X$ and $Y$). When the algorithm stops the set of unused edges does not contain a cycle of four edges like the one shown in Figure 4.2(a), called $XY$-square (where $x_1, x_2 \in X$ and $y_1, y_2 \in Y$).
4.4 Minimum spanning tree

We show that each algorithm for finding a minimum spanning tree of a complete network must use at least $\Omega(n^2)$ messages. Since it is obvious how to get an algorithm that will not use more than $O(n^2)$ messages for any task on complete network, this lower bound is clearly optimal.

We prove:

**Theorem 4.7** Any distributed algorithm to find a minimum spanning tree in a complete weighted graph $G = (V,E)$ with $n$ vertices, uses in the worst case at least $\Omega(n^2)$ edges of the graph.

**Proof:** We make use of the fact that a vertex cannot distinguish between unused edges of equal costs adjacent to itself (since it does not know the identities of the neighbors across these edges).

The proof of this theorem is described as a game between the network and an outside adversary performed on the graph $G$. Whenever a vertex wants to send a message along an unused edge of a certain cost, the adversary chooses such an edge for it (if more than one exists). The goal of the adversary is to demonstrate that the algorithm is not correct if it stops before using $\left( \frac{m}{2} \right)$ edges, where $m = \lfloor \frac{|V|}{2} \rfloor$.

The adversary may change the terminals of unused edges as long as the sets of costs of the edges adjacent to each vertex do not change, since such a change cannot be detected by any processor. Equivalently, this game may be viewed as follows: at the beginning of the execution the adversary cuts all the edges; then, whenever a processor wants to send a message along a previously unused edge, the adversary makes a connection according to some strategy, as long as construction of the network can be consistently completed. For example, if we have as part of the graph the subgraph shown in Figure 4.2(a), where the shown edges are unused, then the adversary can change the costs of the edges to these (or, alternatively, cut the edges and reconnect them to the form) shown in Figure 4.2(b); no vertex should be able to tell that any change was made in the network, since its local view of the network remained unchanged. This entails that the algorithm can now complete its task if and only if it could have done it before this exchange was made.
matching, then $\sigma_{i+1}$ is an extension of $\sigma_i$ by a message $(v, e)$ where $e = (v, j)$ is chosen with smallest possible $j$ (we use here axiom1, axiom3 and an appropriate variant of axiom4 for matching-type algorithms).

By the assumption that $A$ is a matching-type algorithm, a sequence $\sigma$ in $EX(A, G)$ that does contain a maximum matching is eventually constructed. Let this matching be $\{(u_i, v_i)|1 \leq u_i < v_i \leq n, du_i < u_{i+1}\}$.

Let $n_i$ be the number of messages in $\sigma$ which use an edge that connects $u_i$ or $v_i$ to some $j < u_i$. By the construction of $\sigma$ $n_i \geq u_i - 1 \geq i - 1$. Thus the length of $\sigma$ is greater than

$$0 + 1 + \ldots (\lfloor \frac{n}{2} \rfloor - 1) = \frac{n^2}{8} + \Omega(n).$$

(Note that we did not count the edges $(u_i, v_i)$ of the matching). This completes the proof of the theorem.

From this theorem it follows that

**Theorem 4.6** Let $A$ be a matching-type algorithm acting on a complete graph $G$ with $n$ nodes. Then the message complexity $m(A)$ of $A$ is $\Omega(n^2)$.

Note that Theorems 4.5 and 4.6 are independent of the number of initiators, which is not the case for Theorems 4.3 and 4.4.

In [2] it was noted that global algorithms in general graphs require $|E|$ messages when the number of vertices is unknown. We conclude this section by observing that even when the numbers of nodes and edges are known - and in fact the graph is almost complete and known up to isomorphism - then $|E| - 1$ messages may be requires in the worst case. To see this, consider a complete graph of $n$ nodes to which a new vertex $v$ is added on some unknown edge (the resulting graph has $n + 1$ vertices and $\left(\frac{n}{2}\right) + 1$ edges). Apply the algorithm on such a graph with $v$ asleep, and as long as there are unused edges, assume that $v$ is on one of them. Thus $|E| - 1$ edges must be used in order to wake the vertex $v$. 
Note 1: The lower bounds in Theorems 4.3 and 4.4 hold even in the case when every node knows the identities of all other nodes (but cannot tell which edge leads to which node). This note applies to global algorithms, but certainly not to the problem of leader election.

Note 2: This bound will apply also for synchronous networks, for arbitrary starting times (see Problem 1).

Note 3: In the example constructed in the proof of Theorem 1 the number of processors which initialize the algorithm is $O(n)$ (it equals $\frac{n+1}{2}$ for $n = 2^i - 1$). In fact, $O(n)$ initiators are essential for any such example, since $O(n \log k)$ algorithms for global algorithms, where $k$ is the number of the initiators of the algorithm, do exist (see Problem 3).

### 4.3 Matching-Type Algorithms

The above theorems imply that algorithms for tasks like constructing a spanning tree, finding the maximum identity, finding a leader, constructing a Hamiltonian path or constructing a maximum matching $^2$ have a lower bound of $\Omega(n \log n)$ edges (and messages); however, for the last two cases we show even a stronger result. Let a matching-type algorithm be an algorithm that is guaranteed to cover a maximum matching (that is, to induce a graph which contains a matching of size $\left\lfloor \frac{n}{2} \right\rfloor$). We show that each algorithm must use at least $\Omega(n^2)$ messages. Since it is obvious how to get an algorithm that will not use more than $O(n^2)$ messages for any task on complete network, this lower bound is clearly optimal.

**Theorem 4.5** Let $A$ be a matching-type algorithm acting on a complete graph $G$ with $n$ nodes. Then the edge complexity $e(A)$ of $A$ is $\Omega(n^2)$.

**Proof:** Let $A$ be a matching-type algorithm. We construct a sequence in $EX(A, G)$ of length $O(n^2)$. Arbitrarily number the vertices from 1 to $n$. We construct the sequence $\sigma$ in the following manner: Let $\sigma_0$ be the empty sequence. For $i \geq 0$ if $G(\sigma_i)$ does not contain a maximum

---

$^2$It is not hard to see that an algorithm that is guaranteed to construct a maximum matching must be global for complete graphs of $n$ vertices for even $n$, and to induce connected graphs of at least $n-1$ vertices for odd $n$. 


The theorem will follow from the inequality
\[ \epsilon(2k + 1) \geq 2\epsilon(k) + k + 1 \quad (k < \frac{n}{2}). \]

Let \( U \) be a disjoint union of \( U_1, U_2 \) and \( v \), such that \( |U_1| = |U_2| = k \), and \( \epsilon(U) = \epsilon(2k + 1) \). We denote \( C = U_1 \cup U_2 \). Let \( \sigma_1 \) and \( \sigma_2 \) be sequences in \( EX(A, G) \) of lengths \( \epsilon(U_1), \epsilon(U_2) \) inducing subgraphs \( G_1, G_2 \) that have one connected component with vertex set \( U_1, U_2 \) (and all other vertices are isolated), respectively. These two sequences do not interfere with each other, and therefore - by axiom 2 - their concatenation \( \sigma = \sigma_1 \circ \sigma_2 \) is also in \( EX(A, G) \). The proper subgraph \( C \) of \( G(\sigma) \) satisfies the assumptions of axiom 4. Note that each node in \( C \) has at least \( k \) adjacent unused edges within \( C \). By axiom 4 there is an extension of \( \sigma \) by a message \( (v, e) \), where \( v \in C \). By axiom 3 we may choose the edge \( e \) to connect two vertices in \( C \). This process can be repeated until at least one vertex in \( C \) saturates all its edges to other vertices in \( C \). This requires at least \( k \) messages along previously unused edges. One more application of axiom 4 and axiom 3 results in a message from some node in \( C \) to the vertex \( v \). The resulting sequence \( \sigma' \) induces a graph that contains one connected component on the set of vertices \( U \) and isolated vertices otherwise. Thus we have
\[ \epsilon(2k + 1) = \epsilon(U) \geq \epsilon(U_1) + \epsilon(U_2) + k + 1 \geq 2\epsilon(k) + k + 1. \]

The above inequality implies that for \( n = 2^i - 1 \) and the initial condition \( \epsilon(1) = 0 \) we have
\[ \epsilon(n) \geq \frac{n + 1}{2} \log\left(\frac{n + 1}{2}\right). \]

Since it is obvious that \( \epsilon(m) \geq \epsilon(n) \) for \( m > n \), this implies the theorem.

\[ \square \]

From this theorem it follows that

**Theorem 4.4** Let \( A \) be a global algorithm acting on a complete graph \( G \) with \( n \) nodes. Then the message complexity \( m(A) \) of \( A \) is \( \Omega(n \log n) \).
**Definition 4.1**  The edge complexity \( e(A) \) of an algorithm \( A \) acting on a graph \( G \) is the maximal length of a sequence new over all executions of \( A \).

The message complexity \( m(A) \) of an algorithm \( A \) acting on a graph \( G \) is the maximal length of a sequence send over all executions of \( A \).

Clearly \( m(A) \geq e(A) \).

The following lemma is needed in the sequel:

**Lemma 4.4**  Let \( A \) be a global algorithm acting on a complete graph \( G = (V, E) \), and let \( U \subseteq V \). Then there exists a sequence of messages \( \sigma \in EX(A, G) \) such that \( G(\sigma) \) has one connected component whose set of vertices is \( U \) and the vertices in \( V - U \) are isolated.

**Proof:** A desired sequence \( \sigma \) can be constructed in the following way. Start with the empty sequence (using axiom1). Then add a message along a new edge that starts in a vertex in \( U \) (axiom4) and that does not leave \( U \) (axiom3 and the completeness of \( G \)). This is repeated until a graph having the desired properties is constructed.

\[ \square \]

We now prove the lower bound for global algorithms.

**Theorem 4.3**  Let \( A \) be a global algorithm acting on a complete graph \( G \) with \( n \) nodes. Then the edge complexity \( e(A) \) of \( A \) is \( \Omega(n \log n) \).

**Proof:** For a subset \( U \) of \( V \) we define \( e(U) \) to be the maximal length of a sequence \( \sigma \) in \( EX(A, G) \) which induces a graph that has a connected component whose set of vertices is \( U \) and isolated vertices otherwise (such a sequence exists by lemma 1). Define \( e(k) \), \( 1 \leq k \leq n \), by

\[
e(k) = \min\{e(U) | U \subseteq V, |U| = k\}.
\]

Note that \( e(n) \) is a lower bound on the edge complexity of the algorithm \( A \).

\[ ^1 \text{In general, one expects } e(k) = e(U) \text{ for any subset } U \text{ of } k \text{ vertices. However, the reader may construct simple algorithms for which } e(U_1) \neq e(U_2) \text{ for two distinct subsets } U_1 \text{ and } U_2 \text{ of equal cardinality. It is clear that such an algorithm must use the actual identities of the processors in the network.} \]
\( c(send_j) \neq c(send_i) \) for all \( j < i \). \( new(t) \) denotes the prefix of size \( t \) of the sequence \( new \).

Define the graph \( G(new(t)) = (V, E(new(t))) \), where \( E(new(t)) \) is the set of edges used in \( new(t) \), and call it the graph induced by the sequence \( new(t) \). If for every execution of the algorithm \( A \) the corresponding graph \( G(new) \) is connected then we term this algorithm \( \text{global} \). Note that all the graphs \( G(new) \) above have a fixed set \( V \) of vertices (some of which may be isolated).

For each algorithm \( A \) and graph \( G \) we define the \textit{exhaustive set of} \( A \) \textit{with respect to} \( G \), denoted by \( EX(A,G) \) (or \( EX(A) \) when \( G \) is clear from the context), as the set of all the sequences \( \sigma = new(t) \) corresponding to possible executions of \( A \).

From the model used in this paper the following facts - defined below as axioms - hold for every algorithm \( A \) and every graph \( G \) (These axioms reflect only some properties of distributed algorithms which are needed here.)

\textbf{axiom1} : the empty sequence is in \( EX(A,G) \).

\textbf{axiom2} : if two sequences \( \sigma_1 \) and \( \sigma_2 \), which do not interfere with each other, are in \( EX(A,G) \), then so is also their concatenation \( \sigma_1 \circ \sigma_2 \). \( \sigma_1 \) and \( \sigma_2 \) do not interfere with each other if no two edges \( e_1 \) and \( e_2 \) that occur in \( \sigma_1 \) and \( \sigma_2 \) respectively have a common end point; this means that the corresponding partial executions of \( A \) do not affect each other and hence any of their synchronous merges corresponds to a legal execution of \( A \).

\textbf{axiom3} : if \( \sigma \) is a sequence in \( EX(A,G) \) with a last element \( (v, e, m) \), and if \( e' \) is an unused edge adjacent to \( v \), then the sequence obtained from \( \sigma \) by replacing \( e \) by \( e' \) is also in \( EX(A,G) \). (This reflects the fact that a node cannot distinguish between its unused edges.)

Note that these three facts do not imply that \( EX(A,G) \) contains any non-empty sequence. However, if the algorithm \( A \) is global then the following fact holds as well:

\textbf{axiom4} : if \( \sigma \) is in \( EX(A,G) \) and \( C \) is a proper subset of \( V \) containing all the non-isolated nodes in \( G(\sigma) \), then there is an extension \( \sigma' \) of \( \sigma \) in which the first message \( (v, e) \) in \( \sigma' \) but not in \( \sigma \) satisfies \( v \in C \). (This reflects the facts that some unused edge will eventually carry a message and that isolated nodes in \( G(\sigma) \) may remain asleep until some message from an already awakened nodes will reach them).
**Proof:** Let the sizes of the domains of the processors, after the algorithm has terminated, be $s_1 = n \geq s_2 \geq \cdots \geq s_n$. By Lemma 4.2, $s_2 \leq \frac{n}{2}$ (since otherwise the two domains, of sizes $s_1$ and $s_2$, will not be disjoint. Similarly, $s_k \leq \frac{n}{k}$ for every $k$. Since the number of messages sent by each processor is bounded by four times its domain, we get that the total number of messages sent is bounded by

$$4 \cdot (s_1 + s_2 + \cdots + s_n) \leq 4 \cdot n \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) = O(n \log n),$$

and this completes the proof.

\[\square\]

### 4.2.2 $\Omega(n \log n)$ Lower Bound

Let $A$ be a distributed algorithm acting on a graph $G = (V, E)$. An execution of $A$ consists of atomic steps, each being receiving a message, doing some local computation and then sending messages to neighbors. With each execution we can associate a sequence $send = \langle send_1, send_2, \ldots, send_k \rangle$ that includes all the sent messages, in their order of occurrence (if there are no such events then $send$ is the empty sequence). In the case that two or more messages are sent at the same time, order them randomly (thus, in such case many sequences $send$ may correspond to the same execution). Each event $send_i$ we identify with the triple $(v(send_i), e(send_i), m_i)$, containing the message sent $m_i$, the node sending the message $v(send_i)$, and the edge used by it $e(send_i)$. We assume that $send_1$ occurred at time 0, and $send_i$ at time $\tau_i \geq 0$.

Let $send(t)$ be the prefix of length $t$ of the sequence $send$, namely $send(t) = \langle send_1, ..., send_i \rangle$ ($send(0)$ is the empty sequence). If $t < t'$ then we say that $send(t')$ is an extension of $send(t)$, and we denote $send(t) < send(t')$. $send$ is called a completion of $send(t)$.

Note that a completion of a sequence is not necessarily unique.

Let $new = new(send)$ be the subsequence $< new_1, new_2, \ldots, new_r >$ of the sequence $send$ that consists of all the events in $send$ that use previously unused edges. (An edge is used if a message has been already sent along it from either side.) This means that the message $send_i = (v(send_i), e(send_i), m_i)$ belongs to $new$ if and only if
message to processor \( Q \); if \( Q \) discovers that his \((\text{size}, \text{id})\) is smaller than \( P \)'s, he will send a *loose* message, thus \( P \) will receive a *win* message, and its size will increase, contradicting the assumption. Therefore, we conclude that \( Q \)'s \((\text{size}, \text{id})\) is larger than \( P \)'s, a contradiction.

\[\square\]

To show that no more than one leader exist we show the following lemma, which will also prove useful in the complexity analysis of the algorithm. We number the atomic steps in the execution by \( 1, 2, ..., \) and say that step \( i \) occurred at time \( i \). Let \( \text{domain}_t(P) \) denote the set of vertices that have already sent a *win* message to \( P \) by time \( t \).

**Lemma 4.3** If \( |\text{domain}_t(P)| = |\text{domain}_t(P')|, \) then \( \text{domain}_t(P) \cap \text{domain}_t(P') \neq \emptyset. \)

**Proof:** Assume that \( Q \in \text{domain}_t(P) \cap \text{domain}_t(P') \), and, wlog, that \( Q \) is the first node in the domain of \( P \) that joins the domain of \( P' \), and that this is the smallest time \( t' \) when this happens. Since the domain of \( P \) does not change after \( t' \), and since the domain of \( P' \) became larger than that of \( P \) at the moment \( t' \) when \( Q \) has joined the domain of \( P' \), therefore it is not possible that \( |\text{domain}_t(P)| = |\text{domain}_t(P')| \), a contradiction.

\[\square\]

**Theorem 4.1** In every execution of Algorithm A, exactly one processor is elected as a leader.

**Proof:** Each execution terminates, with at most one leader, by Lemma 4.1. If there were two leaders, each of their domains will be the whole set of processors, thus violating Lemma 4.3.

\[\square\]

The following deals with the message complexity of the algorithm:

**Theorem 4.2** The message complexity of Algorithm A is \( O(n \log n) \).
Lemma 4.1 Each execution of the algorithm terminates.

Proof: The relation between the messages is depicted in Figure 4.1; An edge $a \rightarrow b$ in this figure means that in a response to receiving message of type $a$ a processor can send a message of type $b$, and that no other messages can be sent except or the ones shown in the figure.

Figure 4.1: The relation between the messages used in the algorithm

In every execution of the algorithm, win messages are received at most $n$ times by each processor. Hence they are also sent only finite number of times. Hence we have also finite number of try1 and back messages. It can also be shown that there are also a finite number of loose messages, and of try2 messages.

\\

Lemma 4.2 In each execution of the algorithm, at least one processor is elected as a leader.

Proof: Assume that upon termination (which occurs, according to Lemma 4.1) we have no leader (in other words, that the algorithm reached a deadlock situation, in which no more messages are sent, but no leader exists in the network).

Let $P$ be the processor with largest domain $s$ upon termination; if there is more than one processor with a domain that size, we take $P$ as the one with largest identity. When $P$ received its last win message, it first increased its size to $s$. Since we assumed $s \neq n$, this means that $P$ sent a try1 message. This try1 message is eventually forwards as a try2
Algorithm A

\[\text{state} := \text{CANDIDATE};\]
\[\text{edge}_\text{to}\text{\_master} := 0;\]
\[\text{size} := 0;\]
\[\text{waiting} := \text{FALSE};\]
\[\text{unused}_\text{edges} := \{1, \ldots, n - 1\};\]
\[\text{repeat} \quad \text{receive}(m);\]
\[\text{case m of}\]
\[\text{(win)}: \quad [\text{size} := \text{size} + 1;\]
\[\quad \text{if size} = n\]
\[\qquad \text{then state} := \text{LEADER};\]
\[\qquad \text{else send(try1, size, id, size)};\]
\[\text{(try1, size, id, e)}: \quad \text{if not waiting}\]
\[\qquad \text{then [ send(try2, size, id, e, edge}_\text{to}\text{\_master});\]
\[\qquad \text{waiting} := \text{TRUE} ];\]
\[\qquad \text{else push(m)};\]
\[\text{(try2, size, id, e, e')} : \quad \text{if (size, id) < (size, id)}\]
\[\qquad \text{then send(loose, e')};\]
\[\qquad \text{else [ state} := \text{NO\_CANDIDATE};\]
\[\qquad \quad \text{send(back, e, e')];}\]
\[\text{(loose, e)}: \quad \text{waiting} := \text{FALSE};\]
\[\quad \text{if queue} \neq \emptyset\]
\[\qquad \text{then [ pop(try1, size, id, e)};\]
\[\qquad \quad \text{waiting} := \text{TRUE};\]
\[\qquad \quad \text{send(try2, size, id, e, edge}_\text{to}\text{\_master});\]
\[\text{(back, e, e')} : \quad \text{edge}_\text{to}\text{\_master} := e;\]
\[\quad \text{waiting} := \text{FALSE};\]
\[\quad \text{send(win, e)};\]
\[\quad \text{if queue} \neq \emptyset \text{ then [ pop(try1, size, id, e)};\]
\[\qquad \text{waiting} := \text{TRUE};\]
\[\qquad \text{send(try2, size, id, e, edge}_\text{to}\text{\_master});\]
\[\text{until state} = \text{LEADER}.\]
4.2. LEADER ELECTION

(\textit{try2}, size, id, e, e') – a message forwarded along edge $e'$ by a processor to its master.

(\textit{loose}, e) – a message sent back along edge $e$ from the master, when it won a capturing attempt.

(\textit{back}, e, e') – a message sent back along edge $e'$ from a master, when it lost to a capturing attempt.

Any processor that receives a message for the first time, starts to execute the code. This includes a non-empty set of processors that start the algorithm by receiving a message (\textit{win}, 0) message. The edge on which a message is received was written as the last coordinate of the message; thus, (\textit{try1}, size, id, e) is sent on edge $e$. 
Let $size_P$ and $id_P$ denote the size of the domain and the identity of processor $P$. The capturing process proceeds as follows. As long as a processor is a candidate, it sends a message, trying to capture another node. Suppose node $A$ sends a message to node $B$. $B$ transfers the message to its master $C$ (might be $B$ itself). Now, $C$ compares $(size_A, id_A)$ with $(size_C, id_C)$. If $(size_A, id_A) > (size_C, id_C)$ - that is, either $size_A > size_C$ or $size_A = size_C$ and $id_A > id_C$ - it stops being a candidate, informs so to $B$, who joins $A$'s domain, and then $A$ continues its capturing process. If $(size_A, id_A) < (size_C, id_C)$ then $C$ informs to $B$, but $B$ does not inform $A$ anything. In order to keep the low number of messages (see Problem 4), $B$ is forwarding only one message at a time to its master; only after hearing from it it will transfer other messages.

We assume that each processor arbitrarily labels its adjacent edges with the numbers $1, 2, \cdots, n - 1$, and that edge 0 is a self-loop. The algorithm in processor $P$ is using the following data structures:

- $unused\_edges$ - the set of unused edges out of $\{1, 2, \cdots, n - 1\}$.
- $state \in \{\text{CANDIDATE, NO\_CANDIDATE, LEADER}\}$.
- $size_P$ - denotes the number of processors in $P$'s domain.
- $edge\_to\_master \in \{1, 2, \cdots, n\}$, denotes the edge connecting $P$ to its current master, including 0 which connects $P$ to itself.
- $waiting$ - a Boolean variable, indicating whether a node waits for a response from its master, or it is free to forward messages to it.
- $queue$ - contains messages that are waiting to be forwarded to the master.
- $pop(m)$, $push(m)$ - procedures for maintaining this $queue$.

The following messages are used:

- $(win, e)$ - a message sent along edge $e$ by a processor who is joining $P$'s domain.
- $(try1, size, id, e)$ - a message sent along edge $e$ by a processor, carrying its size and identity, in an attempt to capture a new processor.
Chapter 4
COMPLETE NETWORKS

4.1 Introduction

In this chapter we study the leader election and other problems in a complete network of processors, in which each pair of processors is connected by a communication line. In contrast with the ring network, in which the diameter is large, but each processor knows exactly its neighborhood, here we have the other extreme, where the diameter is the smallest possible (1), but each processor has no knowledge of its environment.

We first study the leader election problem, for which we present an $O(n \log n)$ algorithm, and show a matching $\Omega(n \log n)$ lower bound. The lower bound is actually shown for a wider class of problems, in which the messages have to go on edges that form a spanning subgraph.

We then show $\Omega(n^2)$ lower bounds for the number of messages needed to construct a minimum spanning tree, a maximal matching or a Hamiltonian circuit. As each problem can be easily solved within $O(n^2)$ messages in a complete network, these bounds are optimal.

4.2 Leader election

4.2.1 $O(n \log n)$ Algorithm

In the algorithm, nodes attempt to capture other nodes, thus increasing their domains. A node that captures all other nodes will be the leader. A captured node keeps its domain, without trying to augment it. A node can be part of more than one domain, but it will remember its current domain, by having a designated edge pointing to its master.


3.5 Problems

1. Show that the time complexity of Peterson’s 1st algorithm is bounded by $2n - 1$, and give an example that achieves this bound.

2. Prove that each execution of Peterson’s 2nd algorithm terminates, with exactly one processor being determined as the leader.

3. Modify Peterson’s 2nd algorithm so that the processor with maximal identity will be determined as the leader (not by letting the elected one - see previous example - search for the leader).
3.4 References


3.3 Bibliographical notes

The algorithms in Section 3.1 is from [8] (the algorithm in Section 3.1.2 appears also in [2]). That the algorithm in Section 3.1.3 does not necessarily find the maximum identity was noted in [6] (from where Example 3.3 is taken). The lower bound in Section 3.2 is from [9]).

The best algorithm for unidirectional rings appears in [2], and uses at most $1.365n \log n$ messages. For bidirectional rings without sense of direction, the best algorithms known have the same bound $1.44n \log n$ as the algorithm in Section 3.1.3. One appears in [5], where an algorithm is first developed for the half-duplex model (in which messages can be sent on each edge in each direction, but not simultaneously), and are then extended to the usual bidirectional model. The other appears in [7], where processors are attempting to run the algorithm in Section 3.1.3, and, in case there is an inconsistency in the directions, a 'collision' occurs, which calls for a certain action to be taken.

The first $\Omega(n \log n)$ lower bound result for leader election on rings was shown in [1]; it states that, for bidirectional rings of unknown size, the worst-case number of messages is at least $\frac{1}{4}n \log n + O(n)$. The same result is also shown in [9], where also a bound of $\frac{1}{5}n \log n + O(n)$ for the average-case number of messages is shown. For unidirectional rings of known size, a bound of $0.51n \log n + O(n)$ was shown (for infinitely many rings of sizes that are powers of 5), and for bidirectional rings of known size, a bound of $0.25n \log n + O(n)$ was shown (for infinitely many rings of sizes that are powers of 2). In [3] a lower bound was shown for every $n$, and for synchronous rings.

In [10] it is shown that the Chang and Roberts' algorithm requires $\Omega(n \log n)$ messages with probability tending to 1, and that this algorithm compares favorably with best-known ones, and should be considered in many cases to be the unidirectional algorithm of choice, due to its simplicity. A bidirectional version of it is also developed, for a ring without sense of direction, with an average performance bounded by $\frac{3}{4}n (H_n + \frac{3}{2} - \frac{1}{3n})$ for $n > 3$. 
in each (see Example 3.6). But, by the Cyclic permutation property, at least one of these must be counted in our summation, thus

\[
\frac{1}{n!} \sum_{k=1}^{n} \frac{n \cdot n!}{k} = n H_n \approx 0.69 n \log n.
\]

\[\square\]

**Example 3.6** To exemplify the last part of the proof, let \( n = 4, \ k = 3, \ s = (1, 3, 5, 6) \), and consider the 24 permutations of \( s \). There are \( 44! = 96 \) subsequences of size 3 in these permutations. We can group them as follows fix 3 positions on the ring, say positions 3, 4 and 1, and fix there inputs, say \((3, 5, 1)\). Then we can look at the rings \((1, 6, 3, 5), \ (5, 6, 1, 3) \) and \((3, 6, 5, 1)\). They contribute \((1, 3, 5), \ (5, 1, 3) \) and \((3, 5, 1)\) in these locations. In the same manner we can group all the 96 triples into such groups that contain 3 cyclic permutations each.

We now recall that the algorithm presented in Section 2.3, uses exactly \( n H_n \) messages on the average case, and this is clearly best possible by the above theorem; namely,

**Theorem 3.7** In a unidirectional ring whose size is unknown to the processors, the Chang & Roberts' algorithm has an optimal average-case performance of \( n H_n \) messages.
For the lower bound, we assume that, given a set of $n$ distinct identities $I$, all the $n!$ permutations of its elements are equally probable. (Note that we assume here $n$ distinct locations on the ring; assuming only $(n-1)!$ permutations yields similar results; this modification is not discussed here.) Let $\text{ave}_A(I)$ and $\text{worst}_A(I)$ denote the average-case and worst-case message complexities of a given algorithm $A$ for a set of initial identities $I$. Following Theorem 3.4, we get:

**Theorem 3.5** For a given algorithm $A$ for maximum-finding in unidirectional rings, and set of $n$ identities $I$, we have

$$\text{ave}_A(I) \geq \frac{1}{n!} \sum_{s \in \text{perm}(I)} N(s, E(A))$$

(3.7)

and

$$\text{worst}_A(I) \geq \max_{s \in \text{perm}(I)} N(s, E(A)).$$

(3.8)

**Theorem 3.6** For a given algorithm $A$ for maximum-finding in unidirectional rings, and set of $n$ identities $I$, we have

$$\text{ave}_A(I) \geq \Omega(n \log n).$$

**Proof:** By 3.7

$$\text{ave}_A(I) \geq \frac{1}{n!} \sum_{s \in \text{perm}(I)} N(s, E(A))$$

$$= \frac{1}{n!} \sum_{s \in \text{perm}(I)} \sum_{k=1}^{n} N_k(s, E(A))$$

$$= \frac{1}{n!} \sum_{k=1}^{n} \sum_{s \in \text{perm}(I)} N_k(s, E(A))$$

There are $n \cdot n!$ prefixes of size $k$ among all the permutations in $\text{perm}(I)$. We can group them into $\frac{n!}{k!}$ groups, with exactly $k$ cyclic permutations
Lemma 3.6 Let $s, t, u \in D$, such that $u$ is a subsequence of both $s$ and $t$, and let $A$ be a maximum-finding algorithm. If in the execution of $A$ on ring $s$ a message $u$ is sent, then in the execution of $A$ on ring $t$ a message $u$ is sent.

Proof: By induction on the length of $u$, and the assumptions of the message-driven model.

The essence of the proof is that a processor can tell about its environment only by the messages it receives; hence, if in two scenarios it gets the same messages, it must react identically. This idea is widely used in lower bounds arguments in distributed computing (see XXX).

Theorem 3.4 Given a maximum-finding algorithm $A$ for unidirectional rings, there exists an exhaustive set $E(A)$, such that, for every ring $s$, $A$ sends at least $N(s, E(A))$ messages on $s$.

Proof: Define

$$E(A) = \{ s \in D \mid \text{a message } s \text{ is sent when } A \text{ executes on ring } s \}. \quad (3.6)$$

1. Prefix property: if $tu \in E(A)$, $\text{length}(t) \geq 1$, then message $tu$ was sent on the ring labeled $tu$ by 3.6, hence by Lemma 3.6 $t$ was sent on the ring $t$, hence $t \in E(A)$.

2. Cyclic permutation property: for a ring $s$, by the above assumption, at least one processor must send a message in $t \in C(s)$, thus $t \in E(A)$.

3. Let $t \in E(A)$ and let $t$ be a subsequence of $s$. A message $t$ was sent on ring $t$ by 3.6, hence by Lemma 3.6 the message $t$ was also sent on $s$. Thus, by 3.1, at least $N(s, E(A))$ messages were sent on $s$.

This completes the proof of the Theorem.
For $s \in D$, $E \subseteq D$, $|E| < \infty$, let
\[ N(s, E) = \{|t \in E| \text{ is a prefix of some } r \in C(s)\}, \] (3.1)
and
\[ N_k(s, E) = \{|t \in E| \text{ is a prefix of some } r \in C(s) \\land \text{length}(t) = k\}. \] (3.2)
Clearly
\[ N(s, E) = \sum_{k=1}^{\text{length}(s)} N_k(s, E). \] (3.3)

**Example 3.4** Let $s = (1, 3, 5)$, and $E = \{(4, 5), (3, 5), (5, 4, 1), (3, 6, 2), (5, 4)\}$. length($s$) = 4, $(1, 3, 5)$ is a prefix of $s$, $(1, 3)$ is a subsequence of $s$, and $C(s) = \{(4, 1, 3, 5), (1, 3, 5, 4), (3, 5, 4, 1), (5, 4, 1, 3)\}$. $N(s, E) = 4$, since there are 4 elements of $E$ that are prefixes of some cyclic permutation of $s$ (namely, $(3, 5), (5, 4, 1)$, $(3)$ and $(5, 4)$). Thus, $N_1(s, E) = 1$ (due to $(3)$), $N_2(s, E) = 2$ (due to $(3, 5)$ and $(5, 4)$), and $N_3(s, E) = 1$ (due to $(5, 4, 1)$). Also, $N_k(s, E) = 0$ for every $k \geq 4$.

**Definition 3.1** A set $E \subseteq D$ is exhaustive if it has the following two properties:

\begin{align*}
\text{Prefix property: if } s \in E \text{ then} & \quad (3.4) \\
\text{any prefix of } s \text{ is also in } E. & \\
\text{Cyclic permutation property: } \forall s \in D, C(s) \cap E \neq \emptyset. & \quad (3.5)
\end{align*}

**Example 3.5** The set $E = \{(s_1, s_2, \ldots, s_k | s_1 = \max_{1 \leq j \leq k} s_j\}$ is exhaustive. The messages sent by the algorithm in Section 2.3 are clearly of this type. Moreover, given a ring with identities $id_0, id_1, \ldots, id_{n-1}$, all the messages sent are exactly those corresponding to subsequences of identities $(id_j, id_{j+1}, \ldots, id_k)$ in which $id_k$ is the largest. Specifically, in Example 3.4, $E\{(4), (4, 1), (4, 1, 3), (1), (3), (5), (5, 4), (5, 4, 1), (5, 4, 1, 3)\}$, and these are exactly the messages sent by Chang & Roberts’ algorithm on the ring $s = (4, 1, 3, 5)$. 
3.2 Lower bound

We show now that on a unidirectional asynchronous ring, whose size $n$ is unknown to the processors, any algorithm for asynchronous ring requires on the average at least $\Omega(n \log n)$ messages.

Assuming the processors $P_0, P_1, \ldots, P_{n-1}$ have identities $id_0, id_1, \ldots, id_{n-1}$. Being in a completely asynchronous environment, we assume that the algorithm is message-driven; that is, we assume that any non-empty subset of the processors start the algorithm, that includes atomic steps of receiving a message, doing local computation and sending (may be more than one message) to neighbors (in the unidirectional ring there is exactly one neighbor to each processor); no other messages are sent; specifically, no 'spontaneous' messages are generated in the systems. Being on a unidirectional ring, we can assume that the messages sent during each execution of a given algorithm are the same specific execution. Also, we make the assumption that each message contains only the identity of the sender, concatenated to the message it received. Thus, each message is of the form $(id_i, id_{i+1}, \ldots, id_j)$, by which we mean that message $(id_i)$ was initiated by $P_i$ to $P_{i+1}$, who, as a response, sent the message $(id_i, id_{i+1})$ to $P_{i+2}$, etc., and eventually $P_j$ sent the message $(id_i, id_{i+1}, \ldots, id_j)$ to $P_{j+1}$. With each execution of the algorithm we can thus associate a set of such messages.

Each ring we associate with the sequence of identities of its processors. For a sequence $s = (s_1, s_2, \ldots, s_t)$, whose length $\text{length}(s)$ is $t$, any sequence $s = (s_1, s_2, \ldots, s_r, 1 \leq r < t$ is termed a prefix. For two sequences $s$ and $r$, their concatenation $sr$ is the sequence obtained by writing the elements of $s$ followed by those of $r$. If $s = rtu$, then we say that $u$ is a subsequence of $s$. $C(s)$ is the set of all cyclic permutations of $s$. Clearly $|C(s)| = \text{length}(s)$.

Another important assumption is that in any execution of a maximum-finding algorithm $A$, at least one processor must see its own value; in other words, in a ring labeled $s$, at least one message in $C(s)$ is sent by $A$.

Let $D$ denotes the set of all finite sequences of distinct integers; that is, if $Z$ denotes the set of integers, then

$$D = \{(s_1, s_2, \ldots, s_k) | k \geq 1, \forall i \ s_i \in Z, \forall i \neq j \ s_i \neq s_j\}.$$
3.1. UNIDIRECTIONAL RING

**Proof:** Case 1: \( t \) is even. Let \( \text{tid}_i \) and \( \text{tid}_j \) be the \( \text{tid} \)'s of \( P_i \) and \( P_j \) after phase \( t - 1 \). In phase \( t \) \( P_j \) has received the value \( \text{tid}_i \), hence \( \text{tid}_i < \text{tid}_j \). At the start of phase \( t - 1 \), \( P_i \) had a tid \( x \) s.t. \( x < \text{tid}_i \).

If there were no candidate processor between \( P_i \) and \( P_j \), at the start of the previous \( t - 1 \) phase, then in phase \( t - 1 \) \( P_j \) would have received the value \( x < \text{tid}_i \), a contradiction.

Case 2: \( t \) is odd. \( P_j \) received in phase \( t \) \( \text{tid}_i > \text{tid}_j \). If there were no candidate processor between \( P_i \) and \( P_j \) at the start of the previous \( t - 1 \) phase, then in phase \( t - 1 \) \( P_j \) would have received \( \text{tid}_i > \text{tid}_j \), and would have become a relay.

\[ \square \]

**Theorem 3.3** The message complexity of Peterson’s 2nd algorithm is bounded by \( 1.44... n \log n \).

**Proof:** Denote the phases by \( p, p - 1, ..., 1 \) (1 being the last phase). Let \( t_k \) denote the number of processors that remain candidates after phase \( k \), which is equal to the number of processors that start phase \( k - 1 \). Clearly \( t_{p+1} = n, t_1 = 1, \) and \( t_2 \geq 2 \). By Lemma 3.5 \( t_k \) is bounded by the number of processors that became relay during phase \( k + 1 \). This implies

\[ t_k \leq t_{k+2} - t_{k+1} . \]

Hence

\[ t_k \geq F_{k+1} , \]

where \( F_i \)'s are the Fibonacci numbers, defined by the recurrence relation

\[ F_1 = 1, \quad F_2 = 1, \]

\[ F_{k+1} = F_k + F_{k-1} . \]

Since \( F_i = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^i + O(1) \) (see [1]), we get

\[ n = t_{p+1} \geq F_{p+2} = \frac{1}{\sqrt{5}} \left( \frac{1 + \sqrt{5}}{2} \right)^{p+2} + O(1) \]

hence \( p \leq 1.44... \log n \), and therefore the total number of messages sent by the algorithm is bounded by \( 1.44... n \log n \).

\[ \square \]
of \( P_6 \), while processor \( P_7 \)'s identity resides as the tid of \( P_5 \), with \( P_7 \), \( P_8 \), becoming relays. After the third (odd) phase, processor \( P_5 \) becomes relay. In the fourth (even) phase, \( P_6 \) sends the identity 8, which is seen by \( P_8 \). The algorithm used a total of 30 messages.

It can be shown, in a manner similar to the previous discussion, that the algorithm terminates, with exactly one processor determining itself as a leader (see Problem 2). However, it might be the case that this processor will not be the one with the maximal identity, as shown in the following example.

**Example 3.3** Consider the ring in Figure 3.2. After the first phase processors \( P_5 \), \( P_6 \), \( P_7 \) and \( P_8 \) remain active. After the second phase only processors \( P_7 \) and \( P_5 \) remain active. During the third phase, \( P_7 \) detects itself as a leader.

![Figure 3.2: An example for a ring network](image)

The complexity analysis is based on the following lemma.

**Lemma 3.5** If processors \( P_i \) and \( P_j \), \( i < j \), are two consecutive candidate processors at the start of phase \( t \), \( t > 1 \), that remain active at its end, then there exists \( k \), \( i < k < j \), such that processor \( P_k \) became relay during phase \( t - 1 \).
Peterson’s 2nd Algorithm

\[
\begin{align*}
\text{state} & := \text{CANDIDATE;} \\
\text{tid} & := \text{id;} \\
\text{phase\_parity} & := 0; \\
\text{while state} \neq \text{RELAY} \text{ do} \\
\text{begin} \\
\text{case phase\_parity of} \\
1 & : /* \text{start odd phase */} \\
\text{begin} \\
& \text{send(tid);} \\
& \text{receive(ntid);} \\
& \text{if ntid} = \text{id} \text{ then state} := \text{LEADER;} \\
& \text{if tid} < \text{ntid} \text{ then state} := \text{RELAY;} \\
\text{end} \\
0 & : /* \text{start even phase */} \\
\text{begin} \\
& \text{send(tid);} \\
& \text{receive(ntid);} \\
& \text{if ntid} = \text{id} \text{ then state} := \text{LEADER;} \\
& \text{if tid} > \text{ntid} \text{ then state} := \text{RELAY;} \\
& \text{else tid} := \text{ntid;} \\
\text{end} \\
\text{end} \\
\text{end} \\
\text{(now state} = \text{RELAY}) \\
\text{while state} \neq \text{LEADER do} \\
\text{begin} \\
& \text{receive(tid);} \\
& \text{if tid} = \text{id} \text{ then state} := \text{LEADER} \\
& \text{else send(tid);} \\
\text{end}
\end{align*}
\]

Example 3.2 We apply the algorithm to the ring in Figure 2.3. After the first (odd) phase, processors \( P_1, P_2, P_3 \text{ and } P_4 \) become relays. After the second (even) phase, processor \( P_5 \)’s identity resides as the tid
3.1.3 Peterson’s 2nd algorithm

The following modification is now incorporated into the algorithm. Consider again the algorithm in section 2.4.2. Instead of comparing its identity simultaneously with both neighbors, in an attempt to find out whether the processor’s identity is local maximum, a processor first compares it with its left neighbor, then with its right neighbor. The crucial point is that, while comparing with the right neighbor, may be the previous right neighbor is not active any more, and this is where we gain on the number of messages. Formally, the algorithm looks as follows:
The correctness proof is now immediate:

**Theorem 3.1** Every execution of Peterson’s 1st algorithm terminates, with the processor with largest identity being a leader.

**Proof:** By Lemma 3.4, we eventually start a phase \( p \) where \( t_p = 2 \) or \( t_p = 1 \), and the rest follows from Lemma 3.3.

From Lemma 3.4 it follows that the number of phases is bounded by \( \lceil \log_2 n \rceil \). As long as \( t_i > 3 \), exactly \( 2n \) messages are sent in phase \( i \), and in the cases \( t_i = 2 \) or \( t_i = 1 \), the number is bounded by \( 2n \) (see Lemma 3.3), hence we get:

**Theorem 3.2** The message complexity of Peterson’s 1st algorithm is bounded by \( 2n \log n \).

It can be shown (see Problem 1) that the time complexity is bounded by \( 2n - 1 \).
3. The maximal identity $\text{id}_{\max}$ resides as a tid of one candidate processor.

**Proof:** By induction on the phase $p$. The lemma certainly holds before the start of phase $p = 1$. Assume it holds before the start of phase $p$. Consider the $t_p$ candidate processors. Their tids are distinct, by the inductive hypothesis. By the algorithm and the inductive hypothesis, no processor will see its own identity during phase $p$. By the algorithm, the induction hypothesis and Lemma 3.1, the tids at the end of this phase will be determined from among these $t_p$ values only. This proves (1).

Assume that in the start of phase $p$, the identity $id_i$ resides in processor $P_k$, that the preceding candidate processor is $P'_i$, and the subsequent one is $P''_i$. By the induction hypothesis, $P_i$ lies between $P'_i$ and $P''_i$. All these three processors are distinct, since $t_i \geq 3$. The first message sent by $P_k$ is $id_i$, it will be received by $P''_i$, and thus will be sent by it (and be received by the next candidate processor). Moreover, processor $P_k$ receives as a first message an identity that is smaller than its tid, hence it becomes relay in this phase. This proves (2).

By the induction hypothesis, $id_{\max}$ is one of the $t_p$ tids at the start of phase $p$. It will be sent by the processor that holds it, and then forwarded and will become the tid of the next candidate. (Note that, by the inductive hypothesis and the assumption $t_p \geq 3$, $P_{\max}$ will not see itself during this phase.) This completes the proof of (3).

\[\square\]

By the algorithm and Lemma 3.2, we can now show:

**Lemma 3.3** If $t_p = 2$ or $t_p = 1$, then the algorithm terminates after at most $2n$ messages, with $P_{\max}$ being a leader.

Using Lemma 3.2, one can now easily show:

**Lemma 3.4** For every $p$, if $t_p \geq 3$, then $t_{p+1} \leq \frac{t_p}{2}$.

**Proof:** Following the lines of proof of Lemma 3.2, out of two consecutive candidate processors at the beginning of phase $p$, at least one will become relay during this phase. This clearly implies $t_{p+1} \leq \frac{t_p}{2}$. 

else state := RELAY;
end

( now state = RELAY)
while state ≠ LEADER do
    begin
    receive(tid);
    if tid = id then state := LEADER;
    send(tid);
end

Example 3.1 We apply the algorithm to the ring in Figure 2.3. After the first phase processors \( P_4, P_3 \) and \( P_1 \) find out that their predecessors are local maxima, hence they stay to next phase, with tid’s 8, 3 and 5, respectively. After the second phase only \( P_7 \) remains, and updates its tid to 8. It then sends 8, which is discovered by \( P_8 \), who is elected as leader. The algorithm used a total of 36 messages.

To prove correctness of the algorithm, we denote by \( P_{\text{max}} \) the processor that holds the maximal identity \( id_{\text{max}} \). We can assume that the algorithm proceeds in phases, starting at phase 1. We denote by \( t_p \) the number of non-relay processors, that start phase \( p \). First we observe the following.

Lemma 3.1 During the execution of the algorithm, a candidate processor that becomes a relay will never return to a candidate state.

Proof: Clear from the code.

Lemma 3.2 Assume that \( t_p \geq 3 \). Then at the start of phase \( p \) the following holds:

1. Each identity resides as a tid of at most one candidate processor.

2. If the identity \( id_i \) of processor \( P_i \) resides as the tid of processor \( P_k \), then all the processors \( P_i, P_{i+1}, \ldots, P_{k-1} \) are relays.
3.1.2 Peterson’s 1st algorithm

The first algorithm is a modification of the algorithm in 2.4.2 for unidirectional networks. Recall that in the first phase of the algorithm, processor $P_i$ sent and received messages from both of its neighbors $P_{i-1}$ and $P_{i+1}$, and stayed to the next phase only if it found out that it was the largest among the three. To simulate this on a unidirectional network, each processor sends its identity and receives an identity, then sends the identity it just received and receives a second identity. In this way, $oP_{i+1}$ will learn about the identities of $P_{i-1}$ and $P_i$, and will stay to the next phase only if it finds out that $P_i$ is the largest among the three. It then assumes the identity of $P_i$, and moves on to the next phase, otherwise it becomes a relay. A relay only forwards messages. In every state, a processor that sees its own identity becomes a leader. Now, since this algorithm will not terminate correctly in case of $n = 2$, the following slight change is incorporated into it: after receiving an identity as a first message received in a phase, a processor sends as a second identity the maximum between its identity and the one just received.

A formal code for the algorithm follows. $id$ is the identity of the processor. $tid$, $ntid$ and $nntid$ are variables, containing the temporary identity of the processor, the first identity received in a phase and the second one received in a phase, correspondingly.

Peterson’s 1st Algorithm

```
state := CANDIDATE;
tid := id;
while state ≠ RELAY do
    begin /* start phase */
        send(tid);
        receive(ntid);
        if ntid = id then state := LEADER;
        if tid > ntid then send(tid)
            else send(ntid)
        receive(nntid);
        if nntid = id then state := LEADER;
        if ntid ≥ max(tid, nntid) then tid := ntid
    end
```
Chapter 3
LEADER IN A RING - II

In this chapter we further explore the properties of the leader election problem on ring networks. We first present two algorithms for unidirectional networks, with message complexities of $2n \log n + O(n)$ and $1.44...n \log n + O(n)$, respectively. We then present a lower bound of $\Omega(n \log n)$ for the number of messages, required by any algorithm for leader election on unidirectional rings.

3.1 Unidirectional ring

3.1.1 Basic assumptions

In this chapter only unidirectional rings are considered. Since we consider message-driven algorithms, we assume w.l.o.g. that all processors start the execution. Thus, given a nalgorithm, each processor will send the same first message in any execution. It thus will receive the same first message in each execution, and thus it will send the same second message in each execution, and so on. All executions of the algorithm will have the same set of messages, with a possible change in order, due to the various delays on the communication lines. In other words, we can view any algorithm to work in phases, though the network is asynchronous.

We specify the leader election problem by requiring that exactly one processor will enter a leader state. It can then send a message around the ring, informing each processor about its value and causing him to terminate his part in the algorithm. In order to simplify the presentation, this additional round of messages is omitted from the discussion.
Hirschberg & Sinclair’s algorithm

\[
\begin{align*}
& state := \text{CANDIDATE}; \\
& maxnum := 1; \\
& \textbf{while} state = \text{CANDIDATE} \textbf{ do} \\
& \hspace{1em} \text{send}(id, 0, maxnum) \text{ to both neighbors;} \\
& \hspace{1em} \text{wait} \text{ both replies (but react to other messages);} \\
& \hspace{1em} \text{if either reply is ‘no’ then} \\
& \hspace{2em} state = \text{NO-LEADER} \\
& \hspace{1em} maxnum := 2 \cdot maxnum; \\
& \text{on receiving a message (value, num, maxnum):} \\
& \hspace{1em} \text{if value < id then send\_backward(no, value);} \\
& \hspace{1em} \text{if value > id then} \\
& \hspace{2em} state = \text{NO-LEADER} \\
& \hspace{1em} num := num + 1; \\
& \hspace{1em} \text{if num < maxnum then send\_forward(value, num, maxnum)} \\
& \hspace{1em} \text{else send\_backward(ok, value)} \\
& \hspace{1em} \text{if value = id then} \\
& \hspace{2em} state = \text{LEADER} \\
& \text{on receiving a message (no, value) or (ok, value)} \\
& \hspace{1em} \text{if value \neq id then send\_forward the message} \\
& \hspace{1em} \text{else this is the reply you are waiting.}
\end{align*}
\]

12. Complete the details in the analysis of Hirschberg and Sinclair’s algorithm, proving that the message complexity is bounded by \(8n \log_2 n + 8n\) and that the time complexity is \(O(n)\).

13. A center in a graph is a vertex, such that the maximal distance from it to any other vertex is as small as possible. Prove that in a tree there is either one center, or two centers, that are adjacent. Design a sequential algorithm to find a center in a tree. Prove its correctness, and analyze its time complexity.

14. Design a distributed algorithm to find a center in a network, whose underlying graph is a tree. Prove its correctness, and analyze its message and time complexity.
2.7 Problems

1. Given a leader election protocol, it is possible to transform it to a maximum finding protocol.

2. Given a leader election protocol, it is possible to transform it to a spanning tree construction protocol.

3. Given a maximum finding protocol, it is possible to transform it to a leader election protocol.

4. Given a maximum finding protocol, it is possible to transform it to a spanning tree construction protocol.

5. Given a spanning tree construction protocol, show that it is possible to transform it to a leader election protocol.

6. Given a spanning tree construction protocol, show that it is possible to transform it to a maximum finding protocol.

7. Write a formal correctness proof for LeLann’s algorithm.

8. Write a formal correctness proof for Chang & Roberts’ algorithm.

9. Complete the details in the first proof of the average performance of Chang & Roberts’ algorithm.

10. Modify Chang & Roberts’ algorithm so that each processor will be aware of termination and of the maximum identity.

11. The following code is taken from [6]. Prove its correctness. Also, note that in the last phase the two messages are sent all over the ring, while it certainly suffices to have it sent just once; modify the algorithm by incorporating this change into it. (send_forward by processor \(P_i\) means that if a message was received from processor \(P_{i-1}\), it is sent to \(P_{i+1}\) and send_backward means that if a message was received from processor \(P\) it is sent back to \(P\)).
2.6 References


Example 2.3 We apply the algorithm to the ring in Figure 2.3. After the first phase processors with identities 5, 7, and 8 discover that they are local maxima, hence staying to next phase. In this second phase only 8 discovers that it is local maximum. It then starts the next phase, in which it sees its own identity, with a total number of 40 messages.

2.5 Bibliographical notes

The notion of sense of direction in Section 2.1 was introduced in [9]. The algorithms in Section 2.3 are from [7] and [3], and the one in Section 2.4.1 is from 2.4.1, and the one in Section 2.4.2 is from [5]. This idea was also observed by Peterson ([8]) and Dolev, Klawe & Rodeh ([2]), who used it as a starting point for their leader election algorithms for unidirectional rings (see Sections 3.1 and 3.3).

The algorithm of Section 2.4.1 was improved in [1]; the main difference is that in [6] messages are sent in both directions at the same time during a given phase, while in [1] they are sent alternately to one direction. This yields an upper bound of $6n \log n + 4n$ messages, and an upper bound of $3n \log n + n$ if $n$ is a power of 2 (like in [1], $n$ is not known to the processors).
It follows that the number of messages is bounded by

$$4(1 \cdot n + 2 \cdot \left\lfloor \frac{n}{2^1} \right\rfloor + \ldots + 2^t \cdot \left\lfloor \frac{n}{2^{t-1}+1} \right\rfloor)$$

where $t$ is the maximal phase (clearly $t < 1 + \log_2 n$). Hence we get a bound of $8n \log_2 n + 8n$ of messages. The time complexity is clearly of $O(n)$.

**Example 2.2** We apply the algorithm to the ring in Figure 2.3. In the first phase the identity of each processor makes 4 steps, after which processors $P_5, P_7$ and $P_8$ stay to next phase. In the next phase, these three processors send messages to distance 2 (total of 24 messages), after which $P_7$ and $P_8$ remain to the next phase. They send messages to distance 4, after which only $P_8$ remains active. In the last phase it sends two messages to distance 8, and then terminate the algorithm. This amounts to a total of 88 messages.

### 2.4.2 Franklin’s algorithm

Pursuing the idea suggested by the previous algorithm, note what happens on a ring with identities sorted $1, 2, \ldots, n$ (assume processor $p_i$ has identity $i$): after the first phase only processor $p_n$ stays active to the second phase, but it then send messages to distances 2, 4, $\ldots$, until he reaches its $\log n$th phase.

Observing this, Franklin ([5]) suggested a better and simpler algorithm: each processor sends a message to each direction. Upon receipt of its two neighbor’s identities, it determines whether it is the maximum of the three, and then stays active, and repeats this process, otherwise it becomes relay. A relay processor only transmits messages. Clearly in each phase the number of messages sent is exactly $2n$ (each edge carries exactly two messages). If we denote the number of active processors that start phase $k$ by $active_k$, then we clearly have

$$active_{k+1} \leq \left\lfloor \frac{active_k}{2} \right\rfloor.$$

Therefore, the number of phases is bounded by $\lfloor \log_2 n \rfloor$, and thus the message complexity of the algorithm is bounded by $2n(\log n + 1)$, and its time complexity is $O(n)$. 
Note: Following is an alternative proof: We count the total number of messages sent over all $n!$ possible rings, and then divide it by $n!$ to get the average behavior. Observe a given processor $P_i$ of the ring; its identity certainly makes one step. This counts for a total of $n!$ steps. In order for $P_i$'s identity to make its second step, it must be the largest among $P_i$'s and $P_{i+1}$'st identities, which happens in exactly $\frac{n!}{2}$ cases. In order for $P_i$'s identity to make its $k$'th step, it must be the largest among the $k$ processors $P_i, P_{i+1}, \ldots, P_{i+k-1}$, which happens in exactly $\frac{n!}{k}$ cases. We thus get a total of

$$n! \left(1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}\right)$$

messages, as in (2.1).

2.4 Bidirectional rings

2.4.1 Hirschberg & Sinclair’s algorithm

The first $O(n \log n)$ algorithm for bidirectional rings is due to Hirschberg and Sinclair. We only sketch it here, since a better a simpler one will be presented shortly. It is designed for rings with no sense of direction.

In their algorithm, each processor starts as a candidate. It sends a message to distance 1. This message is returned to it from both sides, and the processor thus learns if it is the local maximum of these 3 processors. Each local maximum goes on to the next phase, where messages are sent to distance 2 to both sides, and the processor thus learns if it is the local maximum of these 5 processors. In general, each local maximum in phase $k$ goes on to phase $k + 1$, where messages are sent to distance $2^k$ to both sides, and the processor thus learns if it is the local maximum of these $2^{k+1} + 1$ processors. (We leave a formal description of the algorithm to the problem section.) One can easily prove the following lemma:

Lemma 2.4 At most $\left\lfloor \frac{n}{2^k+1} \right\rfloor$ processors initiate a message to distance $2^k$. 
**Proof**: Assume a ring with \( n \) designated positions, and a set of identities \( \{1, 2, \ldots, n\} \), s.t. the \( n! \) permutations are equally probable. Let \( P(i, k) \) denote the probability that a message carrying the identity \( i \) will be sent exactly \( k \) times. This equals the probability that the first \( k - 1 \) neighbors of the processor are holding identity \( i \) will have identities smaller than \( i \), and the \( k' \)th neighbor is holding an identity greater than \( i \) (see Figure 2.4).

![Figure 2.4: Calculation of \( P(i, k) \)](image)

Hence

\[
P(i, k) = \frac{(i - 1)}{(k - i)} \cdot \frac{n - i}{n - k}.
\]

Therefore, the expected number \( \alpha \) of messages is

\[
\alpha = n + \sum_{i=1}^{n-1} \sum_{k=1}^{i} k \cdot P(i, k) =
\]

and, by changing order of summation, we can derive (for more details, see [2])

\[
= n + \sum_{k=1}^{n-1} \frac{n}{k + 1} = n \left(1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}\right) = n \cdot H_n \approx 0.69 \ldots n \log n + O(1).
\]
Chang & Roberts’ Algorithm

\[
\begin{align*}
\text{state} & := \text{CANDIDATE}; \\
\text{send}(id); \\
\text{repeat} \\
\quad & \text{receive}(nid); \\
\quad & \text{if} \ nid > id \ \text{then} \ \text{send}(nid); \\
\quad & \text{state} := \text{NO\_LEADER}; \\
\text{until} \ nid = id; \\
\text{state} & := \text{LEADER};
\end{align*}
\]

One can easily show by induction that each processor eventually sends its first message, and eventually receives its first message, and, in a similar manner, that each processor that receives an identity larger than its own eventually sends its \(i\)’th message, and eventually receives its \(i\)’th message, for every \(2 \leq i \leq n\). Due to the fifo ordering, it follows that each processor will eventually receive the maximal identity, and thus will become \text{NO\_LEADER}, the processor who holds this identity will receive it and terminate the algorithm. Note that each other processor will not be aware of termination, though it will reach a stage where no more messages will be sent or received (as observed from the outside).

We have thus proved that

**Example 2.1** Consider the ring in Figure 2.3, with processors \(P_1, \cdots, P_n\), where processor \(i\) has identity \(i\). Applying LeLann’s algorithm will result in using 64 messages, while applying Chang and Roberts’ one will result in a total of 20 messages.

**Theorem 2.3** Every execution of Chang \\& Roberts’ algorithm terminates, with only the processor with maximal identity being aware of termination, and s.t. exactly one processor stays in a \text{LEADER} state.

**Lemma 2.2** The message complexity of Chang \\& Roberts’ algorithm is \(O(n^2)\), and its time complexity is \(n\).

**Lemma 2.3** The average message complexity of Chang \\& Roberts’ algorithm is \(O(n\log n)\).
if state = CANDIDATE then state := LEADER.

One can easily show by induction that each processor eventually sends its first message, and eventually receives its first message, and, in a similar manner, that each processor eventually sends its \( i \)'th message, and eventually receives its \( i \)'th message, for every \( 2 \leq i \leq n \). Due to the fifo ordering, upon receiving the \( n \)'th message, a processor clearly terminates its algorithm. Moreover, the maximal identity clearly will get each processor into a \texttt{NO_LEADER} state, except for the processor who holds this identity, that will enter a \texttt{leader} state. We have thus proved that

**Theorem 2.2** Every execution of LeLann’s algorithm terminates, and exactly one processor stays in a \texttt{leader} state upon termination.

**Lemma 2.1** The message complexity of LeLann’s algorithm is \( n^2 \), and its time complexity is \( n \).

![Figure 2.3: An example for a ring network](image)

The following improvement directly comes to mind while discussing the previous algorithm: there is no point in letting smaller identity being forwarded; instead, let the big guys fight. This was suggested first by Chang and Roberts ([3]). This algorithm, though still being \( O(n^2) \) in the worst-case, will prove to be the best possible, in terms of average performance.
2.3 Unidirectional rings - $O(n^2)$ algorithms

Suppose we are given a protocol that works in a unidirectional ring. The protocol is message driven, that is, it is composed of atomic steps. In such a step, one processor receives a message, does local computation, and then it might send a message. Also, we assume that each message is eventually delivered. So, all processors will eventually execute their first step, which will be identical in all possible executions. Using induction, it can be shown that all processors will also execute their 2nd step, which will be identical in all executions, and so on (complete the proof!). In other words, in all executions of a protocol in a unidirectional ring, all messages sent are the same; in particular, all executions have the same number of messages.

The first algorithm, for this problem is by LeLann ([7]); it was motivated by a situation of a ring, in which a token circulates. Some processors detect that the token might have got lost, and initiate a new candidate token. These candidate tokens now have to choose a leader among themselves. They circulate in the ring, with each processor recording the highest candidate token it has seen so far. A processor that receives its own candidate token, and is the largest among these candidate tokens, emits a new token to the ring. Schematically, this algorithm looks as follows, at each processor. $id$ is a constant, holding the identity of the processor, $nid$ is a variable, containing identities received by the processor, and $state$ is a variable, that will initially be set to $candidate$ by all processors, and eventually will be $leader$ in one processor and $no\_leader$ in all others.

**LeLann's Algorithm**

\[
\begin{align*}
\text{state} & := \text{CANDIDATE}; \\
nid & := id; \\
\text{repeat} \\
& \quad \text{send}(nid); \\
& \quad \text{receive}(nid); \\
& \quad \text{if } nid > id \text{ then } state := \text{NO\_LEADER}; \\
\text{until } nid = id;
\end{align*}
\]
2.2. LEADER ELECTION AND RELATED PROBLEMS

Figure 2.2: A bidirectional ring

with sense of direction  without sense of direction

When dealing with ring networks, we will assume that all processors start the protocol, by receiving an awake message from the outside. The modification of each protocol to the case when only a subset of the processors receives these initial messages is left to the reader. Note that, from complexity point of view, assuming that all processors are starting the protocol results in a worst-case situation, thus this assumption can be made without loss of generality.
CHAPTER 2. LEADER IN A RING - I

We assume that the processors are \( P_0, P_1, \ldots, P_{n-1} \), and thus the neighbors of processor \( P_i \) are \( P_{i-1} \) and \( P_{i+1} \), where arithmetic is modulo \( n \).

While studying the ring topology, it is customary to deal with unidirectional rings or bidirectional rings. In a unidirectional ring, each processor has one incoming edge and one outgoing edge. The flow of messages follows the directions of the communication lines, in the obvious way. In a bidirectional ring, each processor is connected to its two neighbors, by bidirectional lines. One of these neighbors is termed \( \text{left}(p) \), the other \( \text{right}(p) \). If these directions are globally consistent - that is, for every processor \( p \) it is the case that \( \text{right}(\text{left}(p)) = p \) - we term this a bidirectional ring with sense of direction, otherwise it is a bidirectional ring without sense of direction. These three situations are depicted in Figures 2.1 and 2.2.

![diagram](image)

Figure 2.1: A unidirectional ring

2.2 Leader election and related problems

The three problems of leader election, maximum finding and spanning tree construction are closely related, though not identical. Moreover, in certain models it might be the case that one problem will not be solvable, while others will be solvable; for example, if we assume that the identities are not necessarily distinct, then, with any subset of ini-
Chapter 2
LEADER IN A RING - I

2.1 Introduction

Since the beginning of this research area of distributed algorithms, symmetry breaking was consider to be one of the most important problems. In addition of its importance on its own right, its simplicity enables us to also demonstrate many of the fundamental issues of distributed computing, dealing with the design and analysis of algorithms, lower bound proofs, synchronization issues, topological considerations, and probabilistic issues. Some of these issues will be addressed in this chapter, and some in subsequent chapters.

In this chapter we concentrate on ring topologies. Though physical ring has clearly many disadvantages, due to its low connectivity and long distances between most pairs of vertices, still this topology is widely used. Two widely used examples are

- Protocols for radio networks, in which a token circulates in a (virtual) ring, and each processor gets its right to broadcast when it gets the token.

- Mutual exclusion protocols in operating systems environment (or, in general, any shared memory environment), where a set of processors want to enter a critical section, and the entry to this section is guaranteed by a token, that circulates between the processes.

In addition, the simplicity of the ring topology enables us to better understand various parameters, especially in study of fundamental issues. The problems mentioned in the first paragraph above have all been first addressed for this topology.
6. The following question relates to a network with one initiator. If, during an execution of a distributed algorithm, at each step a processor can send a message to at most one neighbor, we call it a token algorithm, otherwise we term it a chaotic algorithm. Show how to transform any chaotic algorithm to a token algorithm, and show the cost for doing it, in terms of the factor of increase in the number of messages. Try to get as low factor as possible.
1.6 References


1.7 Problems

1. Write a code for the PI algorithm, and prove its correctness. Assume that any subset of processors may start the algorithm.

2. What is the time complexity of the PIF algorithm?

3. Write a code for algorithm PIF, and prove its correctness.

4. Modify the PI and the PIF algorithms, to work when a subset of processors have the message $m$, that they want to broadcast to every processor in the network.

5. Prove that in our general model, for any network with $|E|$ edges, each broadcast algorithm must send at least $|E|$ messages. Show that this claim does not hold if the originator $a$ knows the total number of processors in the network.
Proof: (1) follows from Theorem 1.1.
(2) follows from the fact that, if we consider each message of (1) as directed from the sender to the receiver, then we get a directed graph, with $|V|$ vertices and $|V| - 1$ edges, with one vertex with incoming degree 0 and all others with incoming degree of 1, and (see [2]) we thus have a spanning tree rooted at $a$.
For (3), observe an edge $(p, q)$ that is not on the spanning tree of (2). By Theorem 1.1 $p$ received the message, but not from $q$, and thus sent it to $q$ before stopping; similarly, $q$ sent the message to $p$.
(4) follows from (1) and (3).

Algorithm PIF

One drawback of the PI algorithm is that $a$ does not know whether and when the broadcast has terminated. Using the above theorem, we can incorporate a feedback, in the following way: Each leaf in the tree (see Theorem 1.2) eventually receives messages from all of its neighbors; upon receiving such last message, it sends it to its parent in the tree. Each internal node in the tree waits until it has received messages from all of its children in the tree, and then sends a message to its parent. It can be proved that this process terminates with the originator $a$ receiving the message $m$ from all of its children, and then it knows that all of them have received it.

This algorithm is known as PIF, short for Propagation of Information with Feedback in [3], or echo algorithm in [1].
The following algorithm bears the nature of breadth-first search, but combines it with lack of clocks; PI is short for Propagation of Information (see [3] and [1]).

The algorithm is as follows: messages are sent from $a$ to each neighbor. Each processor, upon receiving a message along edge $e$, sends it to all neighbors except for along the edge $e$, and then stops the algorithm.

**Theorem 1.1** Each execution of the algorithm PI terminates, with each processor receiving the message.

**Proof:** The proof is by induction on the distance $d$ of a processor from the source $a$. The theorem clearly holds for $d = 0$. Assuming it holds for all processors at distance $< d$, and let $p$ be a processor at distance $d$ from $a$. Consider a shortest path $a - q_1 - q_2 - \cdots - q_{d-1} - p$ between $a$ and $p$. Since processor $q_{d-1}$ is at distance $d - 1$ from $a$, then by the induction hypothesis it received the message $m$. If he received it from $p$, then this means that $p$ has already received $m$, and the proof is completed. Otherwise, it received it from a different neighbor, and it then sends it to all its other neighbors, including $p$. Eventually $p$ will receive this message as its first message, or it will not receive it because it received an earlier message; in any case we conclude that $p$ will eventually receive the message. This completes the proof.

\[\square\]

**Theorem 1.2** In each execution of the PI algorithm:

1. There are exactly $|V| - 1$ messages to which responses are made (rather than 'stop').

2. These $|V| - 1$ messages are carried along edges that form a spanning tree rooted at $a$.

3. Each of the other $|E| - |V| + 1$ edges carries exactly one message in each direction.

4. The total amount of messages sent is exactly $2|E| - |V| + 1$. 
edge to send it to. When this is done, each processor in the network was visited, and the token traversed each edge exactly once in each direction. The reader is encouraged to write a code for the algorithm, to be executed at each node. A possible execution of the algorithm is depicted in Figure 1.2.

The proof of correctness follows immediately from the one for sequential algorithms (see [2]). Note that this contains both the facts that the algorithm terminates, and the fact that it results in meeting the specification of the broadcast problem.

The algorithm certainly requires exactly $2|E|$ number of messages, and, assuming each taking one unit of time, it takes exactly $2|E| - 1$ units of time.

![Figure 1.2: A DFS broadcast](image)

**Algorithm BFS**

Another possibility is to employ a BFS (Breadth-First Search) strategy (see [2]). A straightforward implementation of it will result in a too-costly algorithm: due to the lack of synchronization, one way to implement this algorithm is to send messages to processors at distance 1 from $a$, then to acknowledge $a$, then to issue messages to neighbors at distance 2, and to acknowledge $a$ again, and so forth. We note that more efficient BFS algorithms exist, but they will be mentioned later.

**Algorithm PI**
Note that the execution of a given algorithm on a given synchronous network is unique. (We will also consider some intermediate models, but we defer their presentation to the appropriate section).

**Definition 1.5** The message complexity of a given algorithm $A$ on a given (asynchronous or synchronous) network $N$ is the longest possible execution of $A$ on $N$.

**Definition 1.6** The time complexity of a given algorithm $A$ on a given synchronous network $N$ is the time of the execution of $A$ on $N$.

There are several ways to define the time complexity of a distributed protocol for asynchronous networks. The two common ones follow the following:

**Definition 1.7** The chain time complexity (or simply time complexity) of a given algorithm $A$ on a given asynchronous network $N$ is the longest chain of messages, taken over all possible runs of $A$ on $N$.

**Definition 1.8** The synchronous time complexity of a given algorithm $A$ on a given asynchronous network $N$ is the time it takes the algorithm to run on the network synchronously.

### 1.5 An example: the broadcast problem

Suppose a processor $a$ has a message $m$, that it wants to broadcast to all other nodes in the network. We assume that the network is connected, that is, there is a path connecting every two nodes in the network.

**Algorithm DFS**

As a first attempt, we can easily modify the DFS (Depth First Search) algorithm to our needs. Processor $a$ initiates a message, which is commonly referred to as a token, to one of its neighbor, who then forwards along a new edge, and so on. A token that arrives in a processor that already has been visited by it, returns immediately (backtracks) to its sender along the same edge along which it was sent. This process continues until the originator $a$ receives the token, and has no new
duplication or any other changes. Messages sent on one communication line are queued at the receiving end - one queue for each directed line, two queues for an undirected line - until processed. Each message is guaranteed to arrive at the corresponding queue, after a finite but arbitrary delay. (Alternatively, we can assume that when a message is sent, it is instantaneously put in the queue, and the underlying scheduler only has to grant permission to the processors to process their queues. We do not assume any form of fairness of the scheduler; the only assumption is that, if there are messages in the queues of processor in \( Q \subset P, Q \neq P \), it will grant permission to one processor in \( Q \) within a finite number of steps (that is, if there are messages in queues, then the scheduler cannot ignore them indefinitely. This assumption is clearly necessary, otherwise the situation of a deadlock is unavoidable.

**Definition 1.2** An execution of an algorithm is a sequence of atomic steps, taken according to it (if two steps occur simultaneously, we order them arbitrarily). The length of an execution is the number of these atomic steps.

We deal with message passing systems, thus each message \( m_k \) is sent as a response to receiving message \( m_{k-1} \) by its sender, who sent it as a response to receiving message \( m_{k-2} \) by \( m_{k-1} \)'s sender, etc. This sequence started by a message \( m_0 \), sent by a processor that thus has initiated the chain of messages \( m_0, m_1, \ldots, m_k \). Thus, each execution can be described as a set of chains of messages.

**Definition 1.3** An execution of an algorithm is a sequence of atomic steps, taken according to it (if two steps occur simultaneously, we order them arbitrarily). The length of an execution is the number of these atomic steps.

**Definition 1.4** A synchronous system is an asynchronous system, in which processors also have local clocks. Algorithms are executed in rounds, such that initially \((t = 0 \text{ at each processor})\) all processors are executing their 0'th round; subsequently, for every \( t > 0 \), before time \( t \) each processor \( P \) receives from its neighbor \( Q \) the message sent to it at time \( t - 1 \) of \( Q \) (in case such a message was sent), processes it, and will send its round \( t \) messages at its time \( t \).
nunication lines (or channels), along which they can exchange messages, which is their only means of communication.

**Definition 1.1** An asynchronous system $\mathcal{G}$ is composed of $n$ processors $P = \{P_1, P_2, \ldots, P_n\}$, connected by communication lines. It is specified by its (underlying) communication graph $G = (P, E)$, where the edge $(P_1, P_2) \in E$ iff the processors $P_1$ and $P_2$ are connected by a communication line. The graph $G$ can be directed or undirected; if $G$ is undirected, the existence of an edge $(P_1, P_2) \in E$ means that messages can be sent from $P_1$ to $P_2$ and vice versa, even simultaneously, whereas if $G$ is directed, an edge $(P_1, P_2) \in E$ can be used only in order to send messages from $(P_1$ to $P_2$. We usually identify a distributed system $\mathcal{G}$ with its communication graph $G$.

Communication is done by only sending and receiving messages along the communication lines. The length of a message is bounded, so that it can contain one entity in the network, and an additional fixed number of control bits. In particular, a message cannot contain an information that is proportional in size to that of the information in the entire network.

There are no clocks in the system.

Each processor knows nothing about the network, except for its own identity, its algorithm, and the number of its neighbors.

Each processor is executing a **distributed algorithm** (or simply an *algorithm*), that is composed of *atomic steps*, each - except for the first one - consisting of receiving one message, doing some local computations, and sending any number of messages (including the possibility of not sending any message). The first step of a processor may also be receiving of a *wake-up* message from the outside, and in response it acts as above.

We note that the term *protocol* is also used in the literature, to describe a distributed algorithm; in some places an algorithm is defined to be the program to be followed by each individual processor, while the term *protocol* is referred to the collection of these algorithms, to be executed by the distributed system. We will use these terms alternately, without making this distinction.

We assume an underlying mechanism, that ensures the proper delivery of messages, in the order there were sent (FIFO), without loss,
these two areas of parallel and distributed computing - if such a line exists - based on these two notions.

On the other hand, the following difference between these two areas is valid, and seems to remain valid for the foreseeable future: most studies in parallel computing deal with optimal ways to solve a given problem, like the sorting networks, Boolean circuits, or VLSI networks (that we also view as part of parallel computing). On the other hand, people studying networks often concern with solving problem on a given network. Routing, leader election, sorting, are all performed on a given network. On top of it, also the complexity measures, and henceforth the techniques applied, are quite distinct. Though there is a lot of common interest, still the communities that study these two wide topics are quite disjoint.

1.3 Type of problems studied

In the general model described, we will study specific algorithm (leader election, sorting), or general algorithms, designed for a network with some underlying computation (termination, deadlock detection).

We will usually be interested in their message complexity, but we will also address the issues of time complexity and bit complexity.

On the negative side, we will investigate the issues of lower bounds and impossibility results. The lower bounds show us the limitation of our efforts, in terms of how optimal our algorithms are, while impossibility results mark certain problems as unsolvable by any algorithm, which means that in order to solve them one is either required to change the model or to change the problem.

1.4 The models

1.4.1 General

In this book we will consider several models of computation.

The commonly used model in this book will be the asynchronous message-passing model. It contains processors, connected by commu-
of direction or any other knowledge. For example, processor e knows only that it is connected to three other nodes in the network; it knows neither their identities, nor the total number of nodes in the network.

![Figure 1.1: A distributed network](image)

In subsequent chapters we will address this problem of topological awareness, as well as other related topics like networks with identical identities (anonymous networks).

Another point to mention is that we will not be concerned here with lower-level protocol, that are supposed to guarantee our assumptions (like FIFO ordering of messages, like eventual transmission of each message, like handling queues of messages on communication lines or at processors, etc.). These topics are usually covered in Communication Network courses.

### 1.2 Parallel vs. distributed computing

We would like to point out our view of the relation between parallel and distributed computing. At the beginning of studies of distributed algorithms in the late 70’s, studies concentrated on asynchronous, message-passing systems. Today the area shifted, and distributed computing covers many topics that deal with shared memory, and with certain degrees of synchronization. So, it is impossible to draw the line between
data, in case of faulty processors or faulty communication lines); we also expect cases where processors will not have sufficient resources, like memory, to handle a network’s problem by itself, thus a cooperation with other processors is necessary in order to perform a given task.

The means of communication is either message passing or shared memory, or any combination of them. In the message-passing model, processors can only communicate by sending and receiving messages along the communication lines. In the general asynchronous model, where no clocks are around, the only means of a processor to act is as a response to another processor’s message (except for an initialization phase, where a non-empty subset of processors gets a starting signal from outside the system. In the shared-memory model, subsets of processors may share variables, to which they can write and from which they can read, following certain assumptions. It is usually assumed that each message gets to its destination after a finite but unbounded delay.

We will mostly assume asynchronous, message-passing model; In this model, we assume a given network, where processors, having distinct identities, are connected via communication lines, and assume that each processor is having a program, that contains operations of sending and receiving messages along its communication lines, and doing local computations. A network will be given by its underlying (undirected) graph, $G=(V,E)$, whose set of vertices $V$ represents the processors, and whose set of edges $E$ represents the bidirectional communication lines. In case of bidirectional communication lines, the underlying graph will be directed.

An execution of this algorithm will be determined by a finite set of operations, where each processor follows its program. In some literature, the program that resides in each processor is terms algorithm, while the set of these algorithms - each residing in one of the networks’ nodes - is termed protocol. We will use both the terms distributed algorithm - or just algorithm - and protocol to denote this notion of non-sequential algorithm.

**Example 1.1** A network is shown in Figure 1.1, with six processors $a, b, c, d, e$ and $f$. Unless otherwise assumed, nodes do not have sense
Chapter 1
DISTRIBUTED SYSTEMS

1.1 Introduction

We study selected topics in principles of distributed computing. These topics include algorithmic issues, dealing with the design and analysis of algorithms, as well as lower bounds and impossibility results. Various network models are studied, both fault-free and faulty ones, both synchronous and (mainly) asynchronous, both shared memory and (mainly) message-passing models.

Studies of principles of distributed computing have their effect on a wide variety of applications, in the areas of data-base systems and communication systems. These networks, once abstracted, share common underlying structure, and the topics presented here have their effect on each of these particular areas.

Various terms have been around, describing algorithms which are non-sequential, that is, in which it is possible - and usually the case - that more than one activity is being done in the system in certain time intervals. Such concurrent systems can work synchronously, partly synchronously, or asynchronously, they can work with message-passing mechanism of exchanging information between the different parts of the network, with shared-memory between certain groups of users, or any combination of the above, but they all have one characteristic in common: there is not just one cpu, that is executing the algorithm sequentially. The term decentralized algorithm was also suggested to present this issue.

The advantages one has in using distributed systems are conceptually obvious: we expect them to increase efficiency of an algorithm, or increase reliability (like, designing algorithms to run on duplicated
4 COMPLETE NETWORKS  
4.1 Introduction ........................................ 45
4.2 Leader election ..................................... 45
  4.2.1 $O(n \log n)$ Algorithm ....................... 45
  4.2.2 $\Omega(n \log n)$ Lower Bound ................. 51
4.3 Matching-Type Algorithms ....................... 55
4.4 Minimum spanning tree ............................ 57
4.5 Bibliographical notes ............................ 59
4.7 Problems .......................................... 62

5 SYNCHRONOUS NETWORKS  
5.1 Introduction ....................................... 63
5.2 Synchronizers ..................................... 63
  5.2.1 The notion of a synchronizer ................. 63
  5.2.2 General networks ............................... 64
  5.2.3 Bounded delay networks ....................... 66
5.3 Communication-time trade off ................... 68
  5.3.1 Known network size ........................... 68
  5.3.2 Unknown network size ......................... 69
5.4 Bibliographical notes ........................... 76
5.6 Problems .......................................... 80
# Contents

1 DISTRIBUTED SYSTEMS .............................................. 1
  1.1 Introduction .................................................. 1
  1.2 Parallel vs. distributed computing .......................... 3
  1.3 Type of problems studied ...................................... 4
  1.4 The models .................................................... 4
    1.4.1 General .................................................. 4
  1.5 An example: the broadcast problem .......................... 7
  1.7 Problems ..................................................... 11

2 LEADER IN A RING - I .............................................. 13
  2.1 Introduction .................................................. 13
  2.2 Leader election and related problems ........................ 14
  2.3 Unidirectional rings - $O(n^2)$ algorithms ................. 16
  2.4 Bidirectional rings ........................................... 20
    2.4.1 Hirschberg & Sinclair’s algorithm ....................... 20
    2.4.2 Franklin’s algorithm .................................... 21
  2.5 Bibliographical notes ....................................... 22
  2.7 Problems ..................................................... 24

3 LEADER IN A RING - II ............................................ 27
  3.1 Unidirectional ring .......................................... 27
    3.1.1 Basic assumptions ....................................... 27
    3.1.2 Peterson’s 1st algorithm ................................ 28
    3.1.3 Peterson’s 2nd algorithm ................................ 32
  3.2 Lower bound .................................................. 36
  3.3 Bibliographical notes ....................................... 41
  3.5 Problems ..................................................... 44
Preface

This book contains selected topics in the theoretical study of distributed systems. It is based on a course that I have been teaching in the Department of Computer Science at the Technion, Israel, since 1985, in the ALCOM Summer School on Algorithms in Aarhus University, Denmark, 1991, and in the Summer School on Distributed Algorithms in Siena, Italy, 1994.

The material is intended to be covered in a one semester course for graduate students, or undergraduate students in their final year; it certainly can also serve as an introductory reading for researchers who seek their way into the area.

The study of asynchronous networks is relatively new, and started in the late 70's. However, being quite non-coherent in terms of the many models and problems studied, I found it impossible to cover all the results in this area in a one semester course. I thus chose typical issues, in an attempt to give the reader a first taste of this fascinating and new research area. Further results and suggested reading for each subject can be found in the Bibliographical Notes section of each Chapter - though the list of references is not always complete as in a survey article, and my apology in advance for all authors whose relevant works are not cited - as well as in its Problems section.

Shmuel Zaks

Copyright © 1989

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of the author.
TOPICS IN DISTRIBUTED COMPUTING

SHMUEL ZAKS

Department of Computer Science
Technion, Haifa, Israel