Chapter 2

Modelling I: Synchronous Network Model

This is the shortest chapter in the book. That is because all it has to accomplish is to present a simple computational model for synchronous network algorithms. We present the model separately so that you can use this chapter as a convenient reference while reading Chapters 3–7.

2.1 Synchronous Network Systems

A *synchronous network system* consists of a collection of computing elements located at the nodes of a directed network graph. In Chapter 1, we referred to these computing elements as "processors," which suggests that they are pieces of hardware. It is often useful to think of them instead as logical software "processes," running on (but not identical to) the actual hardware processors. The results that we present here make sense in either case. We will use the convention of calling the computing elements "processes" from now on in the book.

In order to define a synchronous network system formally, we start with a directed graph $G = (V, E)$. We use the letter n to denote $|V|$, the number of nodes in the network digraph. For each node i of G, we use the notation $out\text{-}nbrs_i$ to denote the "outgoing neighbors" of i, that is, those nodes to which there are edges from i in the digraph G, and $in\text{-}nbrs_i$ to denote the "incoming neighbors" of i, that is, those nodes from which there are edges to i in G. We let $distance(i, j)$ denote the length of the shortest directed path from i to j in G, if any exists; otherwise $distance(i, j) = \infty$. We define $diam$, the *diameter*, to be the maximum $distance(i, j)$, taken over all pairs (i, j). We also suppose that we

have some fixed message alphabet M, and we let *null* be a placeholder indicating the absence of a message.

Associated with each node $i \in V$, we have a *process*, which consists formally of the following components:

- $states_i$, a (not necessarily finite) set of *states*
- $start_i$, a nonempty subset of $states_i$ known as the *start states* or *initial states*
- $msgs_i$, a *message-generation function* mapping $states_i \times out\text{-}nbrs_i$ to elements of $M \cup \{null\}$
- $trans_i$, a *state-transition function* mapping $states_i$ and vectors (indexed by $in\text{-}nbrs_i$) of elements of $M \cup \{null\}$ to $states_i$

That is, each process has a set of states, among which is distinguished a subset of start states. The set of states need not be finite. This generality is important, since it permits us to model systems that include unbounded data structures such as counters. The message-generation function specifies, for each state and outgoing neighbor, the message (if any) that process i sends to the indicated neighbor, starting from the given state. The state-transition function specifies, for each state and collection of messages from all the incoming neighbors, the new state to which process i moves.

Associated with each edge (i, j) in G, there is a *channel*, also known as a *link*, which is just a location that can, at any time, hold at most a single message in M.

Execution of the entire system begins with all the processes in arbitrary start states, and all channels empty. Then the processes, in lock-step, repeatedly perform the following two steps:

1. Apply the message-generation function to the current state to generate the messages to be sent to all outgoing neighbors. Put these messages in the appropriate channels.
2. Apply the state-transition function to the current state and the incoming messages to obtain the new state. Remove all messages from the channels.

The combination of the two steps is called a *round*. Note that we do not, in general, place restrictions on the amount of computation a process does in order to compute the values of its message-generation and state-transition functions. Also note that the model presented here is deterministic, in the sense that the message-generation function and the state-transition function are (single-valued) functions. Thus, given a particular collection of start states, the computation unfolds in a unique way.

Halting. So far, we have not made any provision for *process halting*. It is easy, however, to distinguish some of the process states as *halting states*, and specify that no further activity can occur from these states. That is, no messages are generated and the only state transition is a self-loop. Note that these halting states do not play the same role in these systems as they do in traditional finite-state automata. There, they generally serve as *accepting states*, which are used to determine which strings are in the language computed by the machine. Here, they just serve to halt the process; what the process computes must be determined according to some other convention. The notion of accepting state is normally not used for distributed algorithms.

Variable start times. Occasionally, we will want to consider synchronous systems in which the processes might begin executing at different rounds. We model this situation by augmenting the network graph to include a special *environment node*, having edges to all the ordinary nodes. The job of the associated *environment process* is to send special *wakeup* messages to all the other processes. Each start state of each of the other processes is required to be *quiescent*, by which we mean that it does not cause any messages to be generated, and it can only change to a different state as the result of the receipt of a *wakeup* message from the environment or a non-*null* message from some other process. Thus, a process can be awakened either directly, by a *wakeup* message from the environment, or indirectly, by a non-*null* message from another, previously awakened, process.

Undirected graphs. Sometimes we will want to consider the case where the underlying network graph is undirected. We model this situation within the model we have already defined for directed graphs simply by considering a directed graph network with bidirectional edges between all pairs of neighbors. In this case, we will use the notation $nbrs_i$ to denote the neighbors of i in the graph.

2.2 Failures

We will consider various types of failures for synchronous systems, including both *process failures* and *link (channel) failures*.

A process can exhibit *stopping failure* simply by stopping somewhere in the middle of its execution. In terms of the model, the process might fail before or after performing some instance of Step 1 or Step 2 above; in addition, we allow it to fail somewhere in the middle of performing Step 1. This means that the process might succeed in putting only a subset of the messages it is supposed to produce into the message channels. We will assume that this can be *any* subset—

we do not think of the process as producing its messages sequentially and failing somewhere in the middle of the sequence.

A process can also exhibit *Byzantine failure*, by which we mean that it can generate its next messages and next state in some arbitrary way, without necessarily following the rules specified by its message-generation and state-transition functions.

A link can fail by losing messages. In terms of a model, a process might attempt to place a message in a channel during Step 1, but the faulty link might not record the message.

2.3 Inputs and Outputs

We still have not provided any facility for modelling inputs and outputs. We use the simple convention of encoding the inputs and outputs in the states. In particular, inputs are placed in designated input variables in the start states; the fact that a process can have multiple start states is important here, so that we can accommodate different possible inputs. In fact, we normally assume that the *only* source of multiplicity of start states is the possibility of different input values in the input variables. Outputs appear in designated output variables; each of these records the result of only the first write operation that is performed (i.e., it is a *write-once* variable). Output variables can be read any number of times, however.

2.4 Executions

In order to reason about the behavior of a synchronous network system, we need a formal notion of a system "execution."

A *state assignment* of a system is defined to be an assignment of a state to each process in the system. Also, a *message assignment* is an assignment of a (possibly *null*) message to each channel. An *execution* of the system is defined to be an infinite sequence

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \ldots ,$$

where each C_r is a state assignment and each M_r and N_r is a message assignment. C_r represents the system state after r rounds, while M_r and N_r represent the messages that are sent and received at round r, respectively. (These may be different because channels may lose messages.) We often refer to C_r as the state assignment that occurs at *time* r; that is, time r refers to the point just after r rounds have occurred.

If α and α' are two executions of a system, we say that α is *indistinguishable* from α' with respect to a process i, denoted $\alpha \overset{i}{\sim} \alpha'$, if i has the same sequence of states, the same sequence of outgoing messages, and the same sequence of incoming messages in α and α'. We also say that α and α' are *indistinguishable to process i through r rounds* if i has the same sequence of states, the same sequence of outgoing messages, and the same sequence of incoming messages up to the end of round r, in α and α'. We also extend these definitions to the situation where the executions being compared are executions of two different synchronous systems.

2.5 Proof Methods

The most important proof method for reasoning about synchronous systems involves proving *invariant assertions*. An invariant assertion is a property of the system state (in particular, of the states of all the processes) that is true in every execution, after every round. We allow the number of completed rounds to be mentioned in assertions, so that we can make claims about the state after each particular number r of rounds. Invariant assertions for synchronous systems are generally proved by induction on r, the number of completed rounds, starting with $r = 0$.

Another important method is that of *simulations*. Roughly speaking, the goal is to show that one synchronous algorithm A "implements" another synchronous algorithm B, in the sense of producing the same input/output behavior. The correspondence between A and B is expressed by an assertion relating the states of A and B, when the two algorithms are started on the same inputs and run with the same failure pattern for the same number of rounds. Such an assertion is known as a *simulation relation*. As for invariant assertions, simulation relationships are generally proved by induction on the number of completed rounds.

2.6 Complexity Measures

Two measures of complexity are usually considered for synchronous distributed algorithms: time complexity and communication complexity.

The *time complexity* of a synchronous system is measured in terms of the number of rounds until all the required outputs are produced, or until the processes all halt. If the system allows variable start times, the time complexity is measured from the first round in which a *wakeup* occurs, at any process.

The *communication complexity* is typically measured in terms of the total

number of non-*null* messages that are sent. Occasionally, we will also take into account the number of bits in the messages.

The time measure is the more important measure in practice, not only for synchronous distributed algorithms but for all distributed algorithms. The communication complexity is mainly significant if it causes enough congestion to slow down processing. This suggests that we might want to ignore it and just consider time complexity. However, the impact of the communication load on the time complexity is not just a function of an individual distributed algorithm. In a typical network, many distributed algorithms run simultaneously, sharing the same network bandwidth. The message load added to a link by any single algorithm gets added to the total message load on that link, and thus contributes to the congestion seen by all the algorithms. Since it is difficult to quantify the impact that any one algorithm's messages have on the time performance of other algorithms, we settle for simply analyzing (and attempting to minimize) the number of messages generated by individual algorithms.

2.7 Randomization

Instead of requiring the processes to be deterministic, it is sometimes useful to allow them to make random choices, based on some given probability distributions. Since the basic synchronous system model does not permit this, we augment the model by introducing a new *random function* in addition to the message-generation and transition functions, to represent the random choice steps. Formally, we add a $rand_i$ component to the automaton description for each node i; for each state s, $rand_i(s)$ is a probability distribution over some subset of $states_i$. Now in each round of execution, the random function $rand_i$ is first used to pick new states, and the $msgs_i$ and $trans_i$ functions are then applied as usual.

The formal notion of execution used in a randomized algorithm now includes not only state assignments and message assignments, but also information about random functions. Specifically, an *execution* of the system is defined to be an infinite sequence

$$C_0, D_1, M_1, N_1, C_1, D_2, M_2, N_2, C_2, \ldots ,$$

where each C_r and D_r is a *state assignment* and each M_r and N_r is a message assignment. D_r represents the new process states after the round r random choices.

Claims about what is computed by a randomized system are usually probabilistic, asserting that certain results are achieved with at least a certain probability. When such a claim is made, the intention is generally that it is supposed

to hold for all inputs and, in case of systems with failures, for all failure patterns. To model the inputs and failure patterns, a fictitious entity called an *adversary* is usually assumed to control the choices of inputs and occurrences of failures, and the probabilistic claim asserts that the system behaves well in competition with any allowable adversary. General treatment of these issues is beyond the scope of this book; we will just provide special case definitions as they are needed.

2.8 Bibliographic Notes

The general notion of a state machine model has its roots in the traditional finite-state automaton model. Basic material on finite-state machines appears in many undergraduate textbooks such as those of Lewis and Papadimitriou [195] and Martin [221]. The particular kind of state machine model defined here is extracted from numerous papers in distributed computing theory, for example, the Byzantine agreement paper by Fischer and Lynch [119].

The idea of invariant assertions seems to have been first proposed by Floyd [124] for sequential programs and generalized by Ashcroft [15] and by Lamport [175] for concurrent programs. Similar ideas have appeared in many other places. The idea of simulations also has numerous sources. One of the most important is the early work on data abstraction in sequential programs embodied, for example, in Liskov's programming language CLU [198] and in work of Milner [228] and Hoare [158]. Later work that extended the notion to concurrent programs includes papers by Park [236], Lamport [177], Lynch [203], Lynch and Tuttle [218], and Jonsson [165].

This Page Intentionally Left Blank

Chapter 3

Leader Election in a Synchronous Ring

In this chapter, we present the first problem to be solved using the synchronous model of Chapter 2: the problem of *electing a unique leader* process from among the processes in a network. For starters, we consider the simple case where the network digraph is a ring.

This problem originally arose in the study of local area *token ring* networks. In such a network, a single "token" circulates around the network, giving its current owner the sole right to initiate communication. (If two nodes in the network were to attempt simultaneously to communicate, the communications could interfere with one another.) Sometimes, however, the token may be lost, and it becomes necessary for the processes to execute an algorithm to regenerate the lost token. This regeneration procedure amounts to electing a leader.

3.1 The Problem

We assume that the network digraph G is a ring consisting of n nodes, numbered 1 to n in the clockwise direction (see Figure 3.1). We often count mod n, allowing 0 to be another name for process n, $n+1$ another name for process 1, and so on. The processes associated with the nodes of G do not know their indices, nor those of their neighbors; we assume that the message-generation and transition functions are defined in terms of local, relative names for the neighbors. However, we do assume that each process is able to distinguish its clockwise neighbor from its counterclockwise neighbor. The requirement is that, eventually, exactly one process should output the decision that it is the leader, say by changing a special *status* component of its state to the value *leader*. There are several versions of

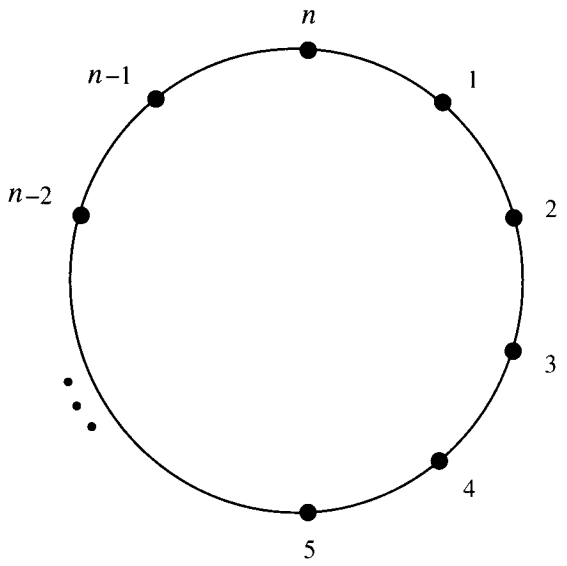


Figure 3.1: A ring of processes.

the problem:

1. It might also be required that all non-leader processes eventually output the fact that they are not the leader, say by changing their *status* components to the value *non-leader*.

2. The ring can be either *unidirectional* or *bidirectional*. If it is unidirectional, then each edge is directed from a process to its clockwise neighbor, that is, messages can only be sent in a clockwise direction.

3. The number n of nodes in the ring can be either known or unknown to the processes. If it is known, it means that the processes only need to work correctly in rings of size n, and thus they can use the value n in their programs. If it is unknown, it means that the processes are supposed to work in rings of different sizes. Therefore, they cannot use information about the ring size.

4. Processes can be identical or can be distinguished by each starting with a *unique identifier* (*UID*) chosen from some large totally ordered space of identifiers such as the positive integers, $\mathbb{N}^+$. We assume that each process's UID is different from each other's in the ring, but that there is no constraint on which UIDs actually appear in the ring. (For instance, they do not have to be consecutive integers.) These identifiers can be restricted to be

manipulated only by certain operations, such as comparisons, or they can admit unrestricted operations.

3.2 Impossibility Result for Identical Processes

A first easy observation is that if all the processes are identical, then this problem cannot be solved at all in the given model. This is so even if the ring is bidirectional and the ring size is known to the processes.

Theorem 3.1 *Let A be a system of n processes, $n > 1$, arranged in a bidirectional ring. If all the processes in A are identical, then A does not solve the leader-election problem.*

Proof. Suppose that there is such a system A that solves the leader-election problem. We obtain a contradiction. We can assume without any loss of generality that each process of A has exactly one start state. This is so because if each process has more than one start state, we could simply choose any one of the start states and obtain a new solution in which each process has only one start state. With this assumption, A has exactly one execution.

So consider the (unique) execution of A. It is straightforward to verify, by induction on the number r of rounds that have been executed, that all the processes are in identical states immediately after r rounds. Therefore, if any process ever reaches a state where its *status* is *leader*, then all the processes in A also reach such a state at the same time. But this violates the uniqueness requirement. □

Theorem 3.1 implies that the only way to solve the leader-election problem is to break the symmetry somehow. A reasonable assumption derived from what is usually done in practice is that the processes are identical except for a UID. This is the assumption we make in the rest of this chapter.

3.3 A Basic Algorithm

The first solution we present is a fairly obvious one, which we call the *LCR algorithm* in honor of Le Lann, Chang, and Roberts, from whose papers this algorithm is extracted. The algorithm uses only unidirectional communication and does not rely on knowledge of the size of the ring. Only the leader performs an output. The algorithm uses only comparison operations on the UIDs. Below is an informal description of the *LCR* algorithm.

LCR algorithm (informal):

Each process sends its identifier around the ring. When a process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own, the process declares itself the leader.

In this algorithm, the process with the largest UID is the only one that outputs *leader*. In order to make this intuition precise, we give a more careful description of the algorithm in terms of the model of Chapter 2.

LCR algorithm (formal):

The message alphabet M is exactly the set of UIDs.

For each i, the states in $states_i$ consist of the following components:

u, a UID, initially i's UID
send, a UID or *null*, initially i's UID
status, with values in {*unknown*, *leader*}, initially *unknown*

The set of start states $start_i$ consists of the single state defined by the given initializations.

For each i, the message-generation function $msgs_i$ is defined as follows:

send the current value of *send* to process $i+1$

Actually, process i would use a relative name for process $i+1$, for example, "clockwise neighbor"; we write $i+1$ because it is simpler. Recall from Chapter 2 that we use the *null* value as a placeholder indicating the absence of a message. So if the value of the *send* component is *null*, this msg_i function does not actually send any message.

For each i, the transition function $trans_i$ is defined by the following pseudocode:

```
send := null
if the incoming message is v, a UID, then
    case
        v > u: send := v
        v = u: status := leader
        v < u: do nothing
    endcase
```

The first line of the transition function definition just cleans up the state from the effects of the preceding message delivery (if any). The rest of the code contains the interesting work—the decision about whether to pass on or discard the incoming UID, or to accept it as permission to become the leader.

This description is written in what should be a reasonably readable programming language, but note that it has a direct translation into a process state machine in the model in Chapter 2. In this translation, each process state consists of a value for each of the variables, and the transitions are describable in terms of changes to the variables. Note that the entire block of code written for the $trans_i$ function is supposed to be executed indivisibly, as part of the processing for a single round.

How do we go about proving formally that the algorithm is correct? Correctness means that exactly one process eventually performs a *leader* output. Let $i_{\max}$ denote the index of the process with the maximum UID, and let $u_{\max}$ denote its UID. It is enough to show that (1) process $i_{\max}$ outputs *leader* by the end of round n, and (2) no other process ever performs such an output. We prove these two properties, respectively, in Lemmas 3.2 and 3.3.

Here and in many other places in the book, we attach the subscript i to a state component name to indicate the instance of that state component belonging to process i. For example, we use the notation u_i to denote the value of state component u of process i. We generally omit the subscripts when writing the process code, however.

Lemma 3.2 *Process i_{max} outputs leader by the end of round n.*

Proof. Note that $u_{\max}$ is the initial value of variable $u_{i_{\max}}$, the variable u at process $i_{\max}$, by the initialization. Also note that the values of the u variables never change (by the code), that they are all distinct (by assumption), and that $i_{\max}$ has the largest u value (by definition of $i_{\max}$). By the code, it suffices to show the following invariant assertion:

Assertion 3.3.1 *After n rounds, $status_{i_{max}} = leader$.*

The normal way to try to prove an invariant such as this one is by induction on the number of rounds. But in order to do this, we need a preliminary invariant that says something about the situation after smaller numbers of rounds. We add the following assertion:

Assertion 3.3.2 *For $0 \leq r \leq n - 1$, after r rounds, $send_{i_{max}+r} = u_{max}$.*

(Recall that addition is modulo n.) This assertion says that the maximum value appears in the *send* component at the position in the ring at distance r from $i_{\max}$.

It is straightforward to prove Assertion 3.3.2 by induction on r. For $r = 0$, the initialization says that $send_{i_{\max}} = u_{\max}$ after 0 rounds, which is just what is needed. The inductive step is based on the fact that every node other than $i_{\max}$ accepts the maximum value and places it into its *send* component, since $u_{\max}$ is greater than all the other values.

Having proved Assertion 3.3.2, we use its special case for $r = n - 1$ and one more argument about what happens in a single round to show Assertion 3.3.1. The key fact here is that process $i_{\max}$ accepts $u_{\max}$ as a signal to set its *status* to *leader*. □

Lemma 3.3 *No process other than i_{max} ever outputs the value leader.*

Proof. It is enough to show that all other processes always have *status* = *unknown*. Again, it helps to state a stronger invariant. If i and j are any two processes in the ring, $i \neq j$, define $[i, j)$ to be the set of indices $\{i, i+1, \ldots, j-1\}$, where addition is modulo n. That is, $[i, j)$ is the set of processes starting with i and moving clockwise around the ring up to and including j's counterclockwise neighbor. The following invariant asserts that no UID v can reach any *send* variable in any position between $i_{\max}$ and v's original home i:

> **Assertion 3.3.3** *For any r and any i, j, the following holds. After r rounds, if $i \neq i_{max}$ and $j \in [i_{max}, i)$ then $send_j \neq u_i$.*

Again, it is straightforward to prove the assertion by induction; now the key fact used in the proof is that a non-maximum value does not get past $i_{\max}$. This is because $i_{\max}$ compares the incoming value with $u_{\max}$, and $u_{\max}$ is greater than all the other UIDs.

Finally, Assertion 3.3.3 can be used to show that only process $i_{\max}$ can receive its own UID in a message, and hence only process $i_{\max}$ can output *leader*. □

Lemmas 3.2 and 3.3 together imply the following:

Theorem 3.4 *LCR solves the leader-election problem.*

Halting and *non-leader* outputs. As written, the *LCR* algorithm never finishes its work, in the sense of all the processes reaching a halting state. We can augment each process to include halting states, as described in Section 2.1. Then we can modify the algorithm by allowing the elected leader to initiate a special *report* message to be sent around the ring. Any process that receives the *report* message can halt, after passing it on. This strategy not only allows processes to halt, but could also be used to allow the non-leader processes to output *non-leader*. Furthermore, by attaching the leader's index to the *report* message, this

strategy could also allow all the participating processes to output the identity of the leader. Note that it is also possible for each non-leader node to output *non-leader* immediately after it sees a UID greater than its own; however, this does not tell the non-leader nodes when to halt.

In general, halting is an important property for a distributed algorithm to satisfy; however, it cannot always be achieved as easily as in this case.

Complexity analysis. The time complexity of the basic *LCR* algorithm is n rounds until a leader is announced, and the communication complexity is $O(n^2)$ messages in the worst case. In the halting version of the algorithm, the time complexity is $2n$ and the communication complexity is still $O(n^2)$. The extra time needed for halting and for the *non-leader* announcements is only n rounds, and the extra communication is only n messages.

Transformation. The preceding two remarks describe and analyze a general transformation, from any leader-election algorithm in which only the leader provides output and no process ever halts, to one in which the leader and the non-leaders all provide output and all processes halt. The extra cost of obtaining the extra outputs and the halting is only n rounds and n messages. This transformation works for any combination of our other assumptions.

Variable start times. Note that the *LCR* algorithm works without modification in the version of the synchronous model with variable start times. See Section 2.1 for a description of this version of the model.

Breaking symmetry. In the problem of electing a leader in a ring, the key difficulty is breaking symmetry. Symmetry-breaking is also an important part of many other problems that need to be solved in distributed systems, including *resource-allocation* problems (see Chapters 10–11 and 20) and *consensus* problems (see Chapters 5–7, 12, 21, and 25).

3.4 An Algorithm with $O(n \log n)$ Communication Complexity

Although the time complexity of the *LCR* algorithm is low, the number of messages used by the algorithm seems somewhat high, a total of $O(n^2)$. This might not seem significant, because there is never more than one message on any link at any time. However, in Chapter 2, we discussed why the number of messages is an interesting measure to try to minimize; this is because of the possible network

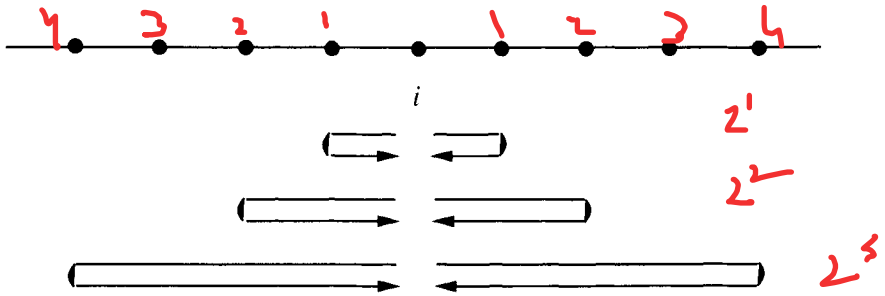


Figure 3.2: Trajectories of successive tokens originating at process i in the *HS* algorithm.

congestion that can result from the total communication load of many concurrently running distributed algorithms. In this section, we present an algorithm that lowers the communication complexity to $O(n \log n)$.

The first published algorithm to reduce the worst-case complexity to $O(n \log n)$ was that of Hirschberg and Sinclair, so we call this algorithm the *HS algorithm*. Again, we assume that only the leader needs to perform an output, though the transformation at the end of Section 3.3 implies that this restriction is not important. Again, we assume that the ring size is unknown, but now we allow bidirectional communication.

As does the *LCR* algorithm, the *HS* algorithm elects the process with the maximum UID. Now every process, instead of sending its UID all the way around the ring as in the *LCR* algorithm, sends it so that it travels some distance away, then turns around and comes back to the originating process. It does this repeatedly, to successively greater distances. The *HS* algorithm proceeds as follows.

***HS* algorithm (informal):**

Each process i operates in phases $0, 1, 2, \ldots$. In each phase l, process i sends out "tokens" containing its UID u_i in both directions. These are intended to travel distance 2^l, then return to their origin i (see Figure 3.2). If both tokens make it back safely, process i continues with the following phase. However, the tokens might not make it back safely. While a u_i token is proceeding in the outbound direction, each other process j on u_i's path compares u_i with its own UID u_j. If $u_i < u_j$, then j simply discards the token, whereas if $u_i > u_j$, then j relays u_i. If $u_i = u_j$, then it means that process j has received its own UID before the token has turned around, so process j elects itself as the leader.

All processes always relay all tokens in the inbound direction.

Now we describe the algorithm more formally. This time, the formalization requires some bookkeeping to ensure that tokens follow the proper trajectories.

For instance, flags are carried by the tokens indicating whether they are travelling outbound or inbound. Also, hop counts are carried with the tokens to keep track of the distances they must travel in the outbound direction; this allows the processes to figure out when the directions of the tokens should be reversed. Once the algorithm is formalized in this way, a correctness argument of the sort given for *LCR* can be provided.

_HS_ algorithm (formal):

The message alphabet M is the set of triples consisting of a UID, a *flag* value in $\{out, in\}$, and a positive integer *hop-count*.

For each i, the states in $states_i$ consist of the following components:

u, of type UID, initially i's UID
send+, containing either an element of M or *null*,
    initially the triple consisting of i's UID, *out*, and 1
send−, containing either an element of M or *null*,
    initially the triple consisting of i's UID, *out*, and 1
status, with values in $\{unknown, leader\}$, initially *unknown*
phase, a nonnegative integer, initially 0

The set of start states $start_i$ consists of the single state defined by the given initializations.

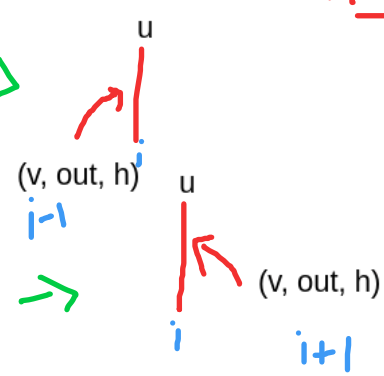
Bidirectional Message M transfer

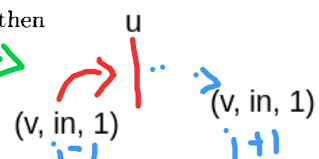
For each i, the message-generation function $msgs_i$ is defined as follows:

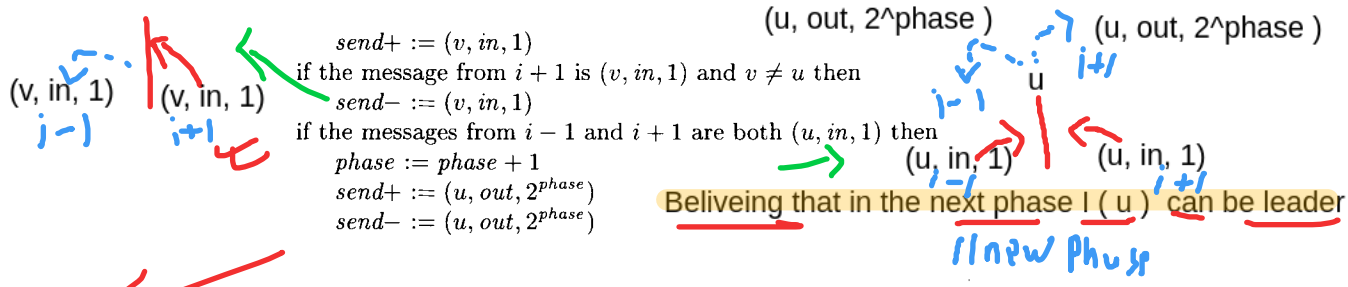
send the current value of *send+* to process $i+1$
send the current value of *send−* to process $i-1$

For each i, the transition function $trans_i$ is defined by the following pseudocode:

```
send+ := null
send− := null
if the message from i − 1 is (v, out, h) then
    case
        v > u and h > 1: send+ := (v, out, h − 1)
        v > u and h = 1: send− := (v, in, 1)
        v = u: status := leader
    endcase
if the message from i + 1 is (v, out, h) then
    case
        v > u and h > 1: send− := (v, out, h − 1)
        v > u and h = 1: send+ := (v, in, 1)
        v = u: status := leader
    endcase
if the message from i − 1 is (v, in, 1) and v ≠ u then
```







As before, the first two lines just clean up the state. The next two pieces of code describe the handling of outbound tokens: tokens with UIDs that are greater than u_i are either relayed or turned around, depending on the *hop-count*, and the receipt of u_i causes i to elect itself the leader. The next two pieces of code describe the handling of inbound tokens: they are simply relayed. (A trivial *hop-count* of 1 is used for inbound tokens.) If process i receives both of its own tokens back, then it goes on to the next phase.

Complexity analysis. We first analyze the communication complexity. Every process sends out a token in phase 0; this is a total of $4n$ messages for the token to go out and return, in both directions. For $l > 0$, a process sends a token in phase l exactly if it receives both its phase $l-1$ tokens back. This is exactly if it has not been "defeated" by another process within distance 2^{l-1} in either direction along the ring. This implies that within any group of $2^{l-1}+1$ consecutive processes, at most one goes on to initiate tokens at phase l. This can be used to show that at most

$$\left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor$$

processes altogether initiate tokens at phase l. Then the total number of messages sent out at phase l is bounded by

$$4\left(2^l \cdot \left\lfloor \frac{n}{2^{l-1}+1} \right\rfloor\right) \leq 8n.$$

This is because phase l tokens travel distance 2^l. Again, the factor of 4 is derived from the fact that the token is sent out in both directions—clockwise and counterclockwise—and that each outbound token must turn around and return.

The total number of phases that are executed before a leader is elected and all communication stops is at most $1 + \lceil \log n \rceil$ (including phase 0), so the total number of messages is at most $8n(1 + \lceil \log n \rceil)$, which is $O(n \log n)$, with a constant factor of approximately 8.

The time complexity for this algorithm is just $O(n)$. This can be seen by noting that the time for each phase l is $2 \cdot 2^l = 2^{l+1}$ (for the tokens to go out and

return). The final phase takes time n—it is an incomplete phase, with tokens only travelling outbound. The next-to-last phase is phase $l = \lceil \log n \rceil - 1$, and its time complexity is at least as great as the total time complexity of all the preceding phases. Thus, the total time complexity of all but the final phase is at most

$$2 \cdot 2^{\lceil \log n \rceil}.$$

It follows that the total time complexity is at most $3n$ if n is a power of 2, and $5n$ otherwise. The rest of the details are left as an exercise.

Variable start times. The *HS* algorithm works without modification in the version of the synchronous model with variable start times.

3.5 Non-Comparison-Based Algorithms

We next consider the question of whether it is possible to elect a leader with fewer than $O(n \log n)$ messages. The answer to this problem, as we shall demonstrate shortly with an impossibility result—a lower bound of $\Omega(n \log n)$—is negative. That result, however, is valid only in the case of algorithms that manipulate the UIDs using comparisons only. (*Comparison-based algorithms* are defined in Section 3.6 below.)

In this section, we allow the UIDs to be positive integers and permit them to be manipulated by general arithmetic operations. For this case, we give two algorithms, the *TimeSlice algorithm* and the *VariableSpeeds algorithm*, each of which uses only $O(n)$ messages. The existence of these algorithms implies that the lower bound of $\Omega(n \log n)$ cannot be proved for the general case.

3.5.1 The *TimeSlice* Algorithm

The first of these algorithms uses the strong assumption that the ring size n is known to all the processes, but only assumes unidirectional communication. In this setting, the following simple algorithm, which we call the *TimeSlice algorithm*, works. It elects the process with the minimum UID.

Note that this algorithm uses synchrony in a deeper way than do the *LCR* and *HS* algorithms. Namely, it uses the non-arrival of messages (i.e., the arrival of *null* messages) at certain rounds to convey information.

***TimeSlice* algorithm:**

Computation proceeds in phases 1, 2, ..., where each phase consists of n consecutive rounds. Each phase is devoted to the possible circulation, all the way around the ring, of a token carrying a particular UID. More

specifically, in phase v, which consists of rounds $(v-1)n+1, \ldots, vn$, only a token carrying UID v is permitted to circulate.

If a process i with UID v exists, and round $(v-1)n+1$ is reached without i having previously received any non-*null* messages, then process i elects itself the leader and sends a token carrying its UID around the ring. As this token travels, all the other processes note that they have received it, which prevents them from electing themselves as leader or initiating the sending of a token at any later phase.

With this algorithm, the minimum UID $u_{\min}$ eventually gets all the way around, which causes its originating process to become elected. No messages are sent before round $(u_{\min}-1)n+1$, and no messages are sent after round $u_{\min} \cdot n$. The total number of messages sent is just n. If we prefer to elect the process with the maximum UID rather than the process with the minimum, we can simply let the minimum send a special message around after it is discovered in order to determine the maximum. The communication complexity is still $O(n)$.

The good property of the *TimeSlice* algorithm is that the total number of messages is n. Unfortunately, the time complexity is about $n \cdot u_{\min}$, which is an unbounded number, even in a fixed-size ring. This time complexity limits the practicality of the algorithm; it is only useful in practice for small ring networks in which UIDs are assigned from among the small positive integers.

3.5.2 The *VariableSpeeds* Algorithm

The *TimeSlice* algorithm shows that $O(n)$ messages are sufficient in the case of rings in which processes know n, the size of the ring. But what if n is unknown? It turns out that in this case, also, there is an $O(n)$ message algorithm, which we call the *VariableSpeeds algorithm* for reasons that will become apparent in a moment. The *VariableSpeeds* algorithm uses only unidirectional communication.

Unfortunately, the time complexity of the *VariableSpeeds* algorithm is even worse than that of the *TimeSlice* algorithm: $O(n \cdot 2^{u_{\min}})$. Clearly, no one would even think of using this algorithm in practice! The *VariableSpeeds* algorithm is what we call a *counterexample algorithm*. A *counterexample algorithm* is one whose main purpose is to show that a conjectured impossibility result is false. Such an algorithm is generally not interesting by itself—it is neither practical nor particularly elegant from a mathematical viewpoint. However, it does serve to show that an impossibility result cannot be proved.

Here is the algorithm.

***VariableSpeeds* algorithm:**

Each process i initiates a token, which travels around the ring, carrying

the UID u_i of the originating process i. Different tokens travel at different rates. In particular, a token carrying UID v travels at the rate of 1 message transmission every 2^v rounds, that is, each process along its path waits 2^v rounds after receiving the token before sending it out.

Meanwhile, each process keeps track of the smallest UID it has seen and simply discards any token carrying an identifier that is larger than this smallest one.

If a token returns to its originator, the originator is elected.

As for the *TimeSlice* algorithm, the *VariableSpeeds* algorithm guarantees that the process with the minimum UID is elected.

Complexity analysis. The *VariableSpeeds* algorithm guarantees that by the time the token carrying the smallest identifier $u_{\min}$ gets all the way around the ring, the second smallest identifier could only get at most halfway around, the third smallest could only get at most a quarter of the way around, and in general, the kth smallest could only get at most $\frac{1}{2^{k-1}}$ of the way around. Therefore, up to the time of election, the token carrying $u_{\min}$ uses more messages than all the others combined. Since $u_{\min}$ uses exactly n messages, the total number of messages sent, up to the time of election, is less than $2n$.

But also, note that by the time $u_{\min}$ gets all the way around the ring, all nodes know about this value, and so will refuse to send out any other tokens. It follows that $2n$ is an upper bound on the total number of messages that are *ever* sent by the algorithm (including the time after the *leader* output).

The time complexity, as mentioned above, is $n \cdot 2^{u_{\min}}$, since each node delays the token carrying UID $u_{\min}$ for $2^{u_{\min}}$ time units.

Variable start times. Unlike the *LCR* and *HS* algorithms, the *VariableSpeeds* algorithms cannot be used "as is" in the version of the synchronous model with variable start times. However, a modification of the algorithm works:

Modified *VariableSpeeds* algorithm:

Define a process to be a *starter* if it receives a *wakeup* message strictly before (i.e., at an earlier round than) receiving any ordinary (non-*null*) messages.

Each starter i initiates a token to travel around the ring, carrying its UID u_i; non-starters never initiate tokens. Initially, this token travels "fast," at the rate of one transmission per round, getting passed along by all the non-starters that are awakened by the arrival of the token, just until it first arrives at a starter. (This could be a different starter, or i itself.) After the

token arrives at a starter, the token continues its journey, but from now on at the "slow" rate of one transmission every 2^{u_i} rounds.

Meanwhile, each process keeps track of the smallest starter's UID that it has seen and discards any token carrying an identifier that is larger than this smallest one. If a token returns to its originator, the originator is elected.

The modified *VariableSpeeds* algorithm ensures that the starter process with the minimum UID is elected. Let $i_{\text{min-start}}$ denote this process.

Complexity analysis. We count the messages in three classes.

1. The messages involved in the initial fast transmission of tokens. There are just n of these.

2. The messages involved in the slow transmission of tokens, up to the time when $i_{\text{min-start}}$'s token first reaches a starter. This takes at most n rounds from when the first process awakens. During this time, a token carrying UID v could use at most $\frac{n}{2^v}$ messages, for a total of at most $\sum_{v=1}^{n} \frac{n}{2^v} < n$ messages.

3. The messages involved in the slow transmission of tokens, after the time when $i_{\text{min-start}}$'s token first reaches a starter. This analysis is similar to that for the basic *VariableSpeeds* algorithm. By the time the winning token gets all the way around the ring, the kth smallest starter's identifier could only get at most $\frac{1}{2^{k-1}}$ of the way around. Therefore, the total number of messages sent, up to the time of election, is less than $2n$. But by the time the winning token gets all the way around the ring, all nodes know about its value, and so will refuse to send out any other tokens; thus, $2n$ is an upper bound on the number of messages in this class.

Thus, the total communication complexity is at most $4n$.

The time complexity is $n + n \cdot 2^{u_{\text{min-start}}}$.

3.6 Lower Bound for Comparison-Based Algorithms

So far, we have presented several algorithms for leader election on a synchronous ring. The *LCR* and *HS* algorithms are comparison based, and the latter achieves a communication complexity bound of $O(n \log n)$ messages and a time bound of $O(n)$. The *TimeSlice* and *VariableSpeeds* algorithms, on the other hand, are not comparison based, and use $O(n)$ messages, but have a huge running time. In

this section, we show a lower bound of $\Omega(n \log n)$ messages for comparison-based algorithms. This lower bound holds even if we assume that communication is bidirectional and the ring size n is known to the processes. In the next section, we show a similar lower bound of $\Omega(n \log n)$ messages for non-comparison-based algorithms with bounded time complexity.

The result of this section is based on the difficulty of *breaking symmetry*. Recall the impossibility result in Theorem 3.1, which says that, because of symmetry, it is impossible to elect a leader in the absence of distinguishing information such as UIDs. The main idea in the following argument is that a certain amount of symmetry can arise even in the presence of UIDs. In this case, the UIDs allow symmetry to be broken, but it might require a large amount of communication to do so.

Recall that we are assuming throughout this chapter that the processes in the ring are all identical except for their UIDs. Thus, the start states of the processes are identical except for designated components that contain the process UID. In general, we have not imposed any constraints on how the message-generation and transition functions can use the UID information.

We assume for the rest of this chapter (this section and the next) that there is only one start state containing each UID. (As in the proof of Theorem 3.1, this assumption does not cause any loss of generality.) The advantage of this assumption is that it implies that the system (with a fixed assignment of UIDs) has exactly one execution.

A comparison-based algorithm obeys certain additional constraints, expressed by the following slightly informal definition. A UID-based ring algorithm is *comparison based* if the only ways that the processes manipulate the UIDs are by copying them, by sending and receiving them in messages, and by comparing them for $\{<, >, =\}$.

This definition allows a process, for example, to store any of the various UIDs that it has encountered and to send them out in messages, possibly combined with other information. A process can also compare the stored UIDs and use the results of these comparisons to make choices in the message-generation and state-transition functions. These choices could involve, for example, whether or not to send a message to each of its neighbors, whether or not to elect itself the leader, whether or not to keep the stored UIDs, and so on. The important fact is that all of the activity of a process depends only on the relative ranks of the UIDs it has encountered, rather than on their particular values.

The following formal notion is used to describe the kind of symmetry that can exist, even with UIDs. Let $U = (u_1, u_2, \ldots, u_k)$ and $V = (v_1, v_2, \ldots, v_k)$ be two sequences of UIDs, both of the same length k. We say that U is *order equivalent* to V if, for all $i, j, 1 \leq i, j \leq k$, we have $u_i \leq u_j$ if and only if $v_i \leq v_j$.

Example 3.6.1 Order equivalence

The sequences $(5, 3, 7, 0)$, $(4, 2, 6, 1)$, and $(5, 3, 6, 1)$ are all order equivalent if the UID set is the natural numbers with the usual ordering.

Notice that two sequences of UIDs are order equivalent if and only if the corresponding sequences of relative ranks of their UIDs are identical. Two technical definitions follow. A round of an execution is said to be *active* if at least one (non-null) message is sent in it. The *k-neighborhood* of process i in ring R of size n, where $0 \leq k < \lfloor n/2 \rfloor$, is defined to consist of the $2k + 1$ processes $i - k, \ldots, i + k$, that is, those that are within distance at most k from process i (including i itself).

Finally, we need a definition of what it means for process states to be the same, except for the particular choices of UIDs they contain. We say that two process states s and t *correspond* with respect to sequences $U = (u_1, u_2, \ldots, u_k)$ and $V = (v_1, v_2, \ldots, v_k)$ of UIDs provided that the following hold: all the UIDs in s are chosen from U, all the UIDs in t are chosen from V, and t is identical to s except that each occurrence of u_i in s is replaced by an occurrence of v_i in t, for all i, $1 \leq i \leq k$. *Corresponding messages* are defined analogously.

We can now prove the key lemma for our lower bound, Lemma 3.5. It says that processes that have order-equivalent k-neighborhoods behave in essentially the same way, until information has had a chance to propagate to the processes from outside the k-neighborhoods.

Lemma 3.5 *Let A be a comparison-based algorithm executing in a ring R of size n and let k be an integer, $0 \leq k < \lfloor n/2 \rfloor$. Let i and j be two processes in A that have order-equivalent sequences of UIDs in their k-neighborhoods. Then, at any point after at most k active rounds, processes i and j are in corresponding states, with respect to the UID sequences in their k-neighborhoods.*

Example 3.6.2 Corresponding states

Suppose that the sequence of UIDs in process i's 3-neighborhood is $(1, 6, 3, 8, 4, 10, 7)$ (where process i's UID is 8), and the sequence in process j's 3-neighborhood is $(4, 10, 7, 12, 9, 13, 11)$ (where process j's UID is 12). Since these two sequences are order equivalent, Lemma 3.5 implies that processes i and j remain in corresponding states with respect to their 3-neighborhoods, as long as no more than three active rounds have occurred. Roughly speaking, the reason this is so is that if there are only three active rounds, there has not been any opportunity for information from outside the order-equivalent 3-neighborhoods to reach i and j.

Proof (of Lemma 3.5). Without loss of generality, we may assume that $i \neq j$. We proceed by induction on the number r of rounds that have been performed in the execution. For each r, we prove the lemma for all k.

Basis: $r = 0$. By the definition of a comparison-based algorithm, the initial states of i and j are identical except for their own UIDs, and hence they are in corresponding initial states, with respect to their k-neighborhoods (for any k).

Inductive step: Assume that the lemma holds for all $r' < r$. Fix k such that i and j have order-equivalent k-neighborhoods, and suppose that the first r rounds include at most k active rounds.

If neither i nor j receives a message at round r, then by induction (for $r-1$ and k), i and j are in corresponding states just after $r-1$ rounds, with respect to their k-neighborhoods. Since i and j have no new input, they make corresponding transitions and end up in corresponding states after round r.

So assume that either i or j receives a message at round r. Then, round r is active, so the first $r-1$ rounds include at most $k-1$ active rounds. Note that i and j have order-equivalent $(k-1)$-neighborhoods, and likewise for $i-1$ and $j-1$ and for $i+1$ and $j+1$. Therefore, by induction (for $r-1$ and $k-1$), we have that i and j are in corresponding states after $r-1$ rounds, with respect to their $(k-1)$-neighborhoods, and similarly for $i-1$ and $j-1$, and for $i+1$ and $j+1$.

We proceed by case analysis.

1. At round r, neither $i-1$ nor $i+1$ sends a message to i.

   Then, since $i-1$ and $j-1$ are in corresponding states after $r-1$ rounds, and likewise for $i+1$ and $j+1$, we have that neither $j-1$ nor $j+1$ sends a message to j at round r. But this contradicts the assumption that either i or j receives a message at round r.

2. At round r, $i-1$ sends a message to i but $i+1$ does not.

   Then, since $i-1$ and $j-1$ are in corresponding states after $r-1$ rounds, $j-1$ also sends a message to j at round r, and that message corresponds to the message sent by $i-1$ to i, with respect to the $(k-1)$-neighborhoods of $i-1$ and $j-1$, and hence with respect to the k-neighborhoods of i and j. For similar reasons, $j+1$ sends no message to j at round r. Since i and j are in corresponding states after round $r-1$, and receive corresponding messages, they remain in corresponding states, this time with respect to their k-neighborhoods.

3. At round r, $i+1$ sends a message to i but $i-1$ does not.

   Analogous to the previous case.

4. At round r, both $i-1$ and $i+1$ send messages to i.

   A similar argument. □

Lemma 3.5 tells us that many active rounds are necessary to break symmetry if there are large order-equivalent neighborhoods. We now define particular rings with the special property that they have many order-equivalent neighborhoods of various sizes. Let $c, 0 \leq c \leq 1$, be a constant, and let R be a ring of size n. Then R is said to be *c-symmetric* if for every l, $\sqrt{n} \leq l \leq n$, and for every segment S of R of length l, there are at least $\lfloor \frac{cn}{l} \rfloor$ segments in R that are order equivalent to S (counting S itself).[1]

If n is a power of 2, then it is easy to construct a ring that is $\frac{1}{2}$-symmetric. Specifically, we define the *bit-reversal ring* of size n as follows. Suppose that $n = 2^k$. Then we assign to each process i the integer in the range $[0, n-1]$ whose k-bit binary representation is the reverse of the k-bit binary representation of i (we use 0^k as the k-bit binary representation of n, identifying n with 0).

Example 3.6.3 Bit-reversal ring

For $n = 8$, we have $k = 3$, and the assignment is as in Figure 3.3.

Lemma 3.6 *Any bit-reversal ring is $\frac{1}{2}$-symmetric.*

Proof. Left as an exercise.[2] □

For values of n that are not powers of 2, there also always exist c-symmetric rings, though the general case requires a smaller constant c.

Theorem 3.7 *There exists a constant c such that, for all $n \in \mathbb{N}^+$, there is a c-symmetric ring of size n.*

The proof of Theorem 3.7 involves a fairly complicated recursive construction.[3] It is not possible to produce the needed ring simply, say by starting with the bit-reversal ring for the next smaller power of 2 and just adding some extra processes; these extra processes would destroy the symmetry.

So we can assume, for any n, that we have a c-symmetric ring R of size n. The following lemma states that if such a ring elects a leader, then it must have many active rounds.

[1] Try to ignore the square root lower bound condition—it is just a technicality.

[2] Note that for the bit-reversal ring, there is no need for the square root lower bound condition.

[3] This is where the square root lower bound condition arises.

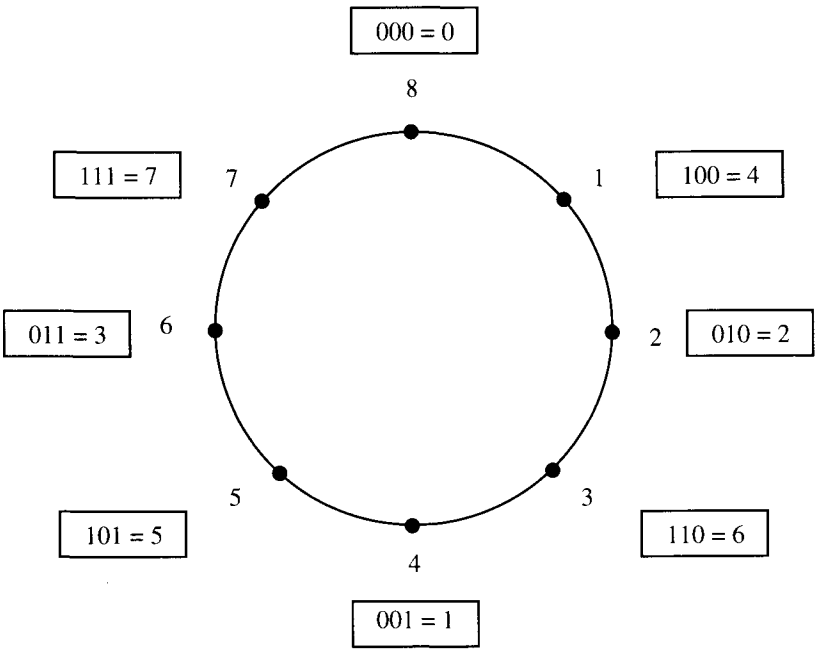


Figure 3.3: Bit-reversal ring of size 8.

Lemma 3.8 *Let A be a comparison-based algorithm executing in a c-symmetric ring of size n, and suppose that A elects a leader. Suppose that k is an integer such that $\sqrt{n} \leq 2k+1$ and $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$. Then A has more than k active rounds.*

Proof. We proceed by contradiction. Suppose that A elects a leader, say process i, in at most k active rounds. Let S be the k-neighborhood of i; S is a segment of length $2k+1$. Since the ring is c-symmetric, there must be at least $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$ segments in the ring that are order equivalent to S, counting S itself. Thus, there is at least one other segment that is order equivalent to S; let j be the midpoint of that segment. Now, by Lemma 3.5, i and j remain in equivalent states throughout the execution, up to the election point. We conclude that j also gets elected, a contradiction. □

Now we can prove the lower bound.

Theorem 3.9 *Let A be a comparison-based algorithm that elects a leader in rings of size n. Then there is an execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected.*[4]

[4]The $\Omega(n \log n)$ expression hides a fixed constant, independent of n.

Proof. Fix c to be the constant whose existence is asserted by Theorem 3.7, and use that theorem to obtain a c-symmetric ring R of size n. We consider executions of the algorithm in ring R.

Define $k = \left\lfloor \frac{cn-2}{4} \right\rfloor$. Then $\sqrt{n} \leq 2k+1$ (provided n is sufficiently large), and $\left\lfloor \frac{cn}{2k+1} \right\rfloor \geq 2$. It follows by Lemma 3.8 that there are more than k active rounds, that is, that there are at least $k+1$ active rounds.

Consider the rth active round, where $\sqrt{n}+1 \leq r \leq k+1$. Since the round is active, there is some process i that sends a message in round r. Let S be the $(r-1)$-neighborhood of i. Since R is c-symmetric, there are at least $\left\lfloor \frac{cn}{2r-1} \right\rfloor$ segments in R that are equivalent to S. By Lemma 3.5, at the point just before the rth active round, the midpoints of all these segments are in corresponding states, so they all send messages.

Now let $r_1 = \lceil \sqrt{n} \rceil + 1$ and $r_2 = k+1 = \left\lfloor \frac{cn-2}{4} \right\rfloor + 1$. The argument above implies that the total number of messages is at least

$$\sum_{r=r_1}^{r_2} \left\lfloor \frac{cn}{2r-1} \right\rfloor \geq \sum_{r=r_1}^{r_2} \frac{cn}{2r-1} - r_2.$$

The second term is $O(n)$, so it suffices to show that the first term is $\Omega(n \log n)$. We have

$$\begin{aligned} \sum_{r=r_1}^{r_2} \frac{cn}{2r-1} &= \Omega\left(n \sum_{r=r_1}^{r_2} \frac{1}{r} \right) \\ &= \Omega\left(n \left(\ln r_2 - \ln r_1 \right) \right) \end{aligned}$$

by an integral approximation of the sum,

$$\begin{aligned} &= \Omega\left(n \left(\ln \left(\left\lfloor \frac{cn-2}{4} \right\rfloor + 1 \right) - \ln(\lceil \sqrt{n} \rceil + 1) \right) \right) \\ &= \Omega(n \log n). \end{aligned}$$

This is as needed. □

3.7 Lower Bound for Non-Comparison-Based Algorithms*

Can we describe any lower bounds on the number of messages for the case of non-comparison-based algorithms? Although the $\Omega(n \log n)$ barrier can be broken in this case, it is possible to show that this can only happen at the cost of large time complexity. For example, suppose that the time until leader election is bounded

by t. Then, if the total number of UIDs in the space of identifiers is sufficiently large—say, greater than some particular fast-growing function $f(n,t)$—then there is a subset U of the identifiers on which it is possible to show that the algorithm behaves "like a comparison-based algorithm," at least through t rounds. This implies that the lower bound for comparison carries over to the time-bounded algorithm using identifiers in U.

We give somewhat more detail, but our presentation is still just a sketch. We will define the fast-growing function $f(n,t)$ using *Ramsey's Theorem*, which is a kind of generalized Pigeonhole Principle. In the statement of the theorem, an n-subset is just a subset with n elements, and a coloring just assigns a color to each set.

Theorem 3.10 (Ramsey's Theorem) *For all integers n, m, and c, there exists an integer $g(n,m,c)$ with the following property. For every set S of size at least $g(n,m,c)$, and any coloring of the n-subsets of S with at most c colors, there is some subset C of S of size m that has all of its n-subsets colored the same color.*

We begin by putting each algorithm into a *normal form*, in which each state simply records, in LISP S-expression format, the initial UID plus all the messages ever received, and each non-*null* message contains the complete state of its sender. Certain of these S-expressions are then designated as *election* states, in which the process is identified as having been elected as the leader. If the original algorithm is a correct leader-election algorithm, then the new one (with the modified output convention) is also, and the communication complexity is the same.

Our lower bound theorem is as follows.

Theorem 3.11 *For all integers n and t, there exists an integer $f(n,t)$ with the following property. Let A be any (not necessarily comparison-based) algorithm that elects a leader in rings of size n within time t and uses a UID space of size at least $f(n,t)$. Then there is an execution of A in which $\Omega(n \log n)$ messages are sent by the time the leader is elected.*

Proof Sketch. Fix n and t. Without loss of generality, we only consider algorithms in normal form. Since the algorithms involve only n processes and proceed for only t rounds, all the S-expressions that arise have at most n distinct arguments and at most t parenthesis depth.

Now for each algorithm A, we define an equivalence relation $\equiv_A$ on n-sets (i.e., sets of size n) of UIDs; roughly speaking, two n-sets will be said to be equivalent if they give rise to the same behavior for algorithm A. More precisely, if V and V' are two n-sets of UIDs, then we say that $V \equiv_A V'$ if, for every

S-expression of depth at most t over V, the corresponding S-expression over V' (generated by replacing each element of V with the same rank element within V') give rise to the same decisions, in algorithm A, about whether to send a message in each direction and about whether or not the process is elected as leader.

Because the S-expressions in the definition of the equivalence relation have at most n arguments and at most t depth, there are only finitely many $\equiv_A$ equivalence classes; in fact, there is an upper bound on the number of classes that does not depend on the algorithm A, but only on n and t. Let $c(n,t)$ be such an upper bound.

Now fix algorithm A. We describe a way of coloring n-sets of UIDs, so we can apply Ramsey's Theorem. Namely, we just associate a color with each $\equiv_A$ equivalence class of n-sets, and color all the n-sets in that class by that color.

Now define $f(n,t) = g(n, 2n, c(n,t))$, where g is the function in Theorem 3.10, and consider any UID space containing at least $f(n,t)$ identifiers. Then, Theorem 3.10 implies the existence of a subset C of the UID space, containing at least $2n$ elements, such that all n-subsets of C are colored the same color. Take U to be the set consisting of the n smallest elements of C.

Then we claim that the algorithm behaves exactly like a comparison algorithm through t rounds, when UIDs are chosen from U. That is, every decision made by any process, about whether to send a message in either direction or about whether the process is a leader, depends only on the relative order of the arguments contained in the current state. To see why this is so, fix any two subsets W and W' of U, of the same size—say m. Suppose that S is an S-expression of depth at most t with UIDs chosen from W, and S' is the corresponding S-expression over W' (generated by replacing each element of W with the same rank element within W'). Then W and W' can be extended to sets V and V', each of size exactly n, by including the $n-m$ largest elements of C. Since V and V' are colored the same color, the two S-expressions give rise to the same decisions about whether to send a message in each direction and about whether or not the process is elected as leader.

Since the algorithm behaves exactly like a comparison algorithm through t rounds, when UIDs are chosen from U, Theorem 3.9 yields the lower bound. □

3.8 Bibliographic Notes

The impossibility result of Section 3.2 seems to be a part of the ancient folklore of this area; one version of this result, for a different model, appears in a paper by Angluin [13]. The *LCR* algorithm is derived from one developed by Le Lann

[191], with optimizations due to Chang and Roberts [71]. The *HS* algorithm is due to Hirschberg and Sinclair [156].

There have been a series of improvements in the constant in the $O(n \log n)$ upper bound, culminating in a bound of approximately $1.271n \log n + O(n)$, by Higham and Przytycka [155]; this bound works for the unidirectional case. Peterson [239] and Dolev, Klawe, and Rodeh [97] have given $O(n \log n)$ algorithms for the unidirectional case.

The *TimeSlice* algorithm also seems to be folklore, but is similar to the election strategy used in the MIT token ring network. The *VariableSpeeds* algorithm was developed by Frederickson and Lynch [127], and simultaneously by Vitanyi [282].

The lower bound results, both for comparison-based and for non-comparison-based algorithms, are due to Frederickson and Lynch [127]. Another construction of c-symmetric rings is carried out by Attiya, Snir, and Warmuth [27]. Ramsey's Theorem is a standard result of combinatorial theory, and is presented, for example, in the graph theory book of Berge [47].

The paper by Attiya, Snir, and Warmuth [27] contains other results about limitations of computing power in synchronous rings, using proof techniques similar to those used in Section 3.6.

3.9 Exercises

3.1. Fill in more of the details for the inductive proof of correctness of the *LCR* algorithm.

3.2. For the *LCR* algorithm,

(a) Give a UID assignment for which $\Omega(n^2)$ messages are sent.

(b) Give a UID assignment for which only $O(n)$ messages are sent.

(c) Show that the average number of messages sent is $O(n \log n)$, where this average is taken over all the possible orderings of the processes on the ring, each assumed to be equally likely.

3.3. Modify the *LCR* algorithm so that it also allows all the non-leader processes to output *non-leader*, and so that all the processes eventually halt. Present the modified algorithm using the same style of "code" that we used for the *LCR* algorithm.

3.4. Show that the *LCR* algorithm still works correctly in the version of the synchronous model allowing variable start times. (You might have to modify the code slightly.)

3.5. Carry out a careful proof of correctness for the *HS* leader-election algorithm, using the invariant assertion style used for *LCR*.

3.6. Show that the *HS* algorithm still works correctly in the version of the synchronous model allowing variable start times. (You might have to modify the code slightly.)

3.7. Suppose that the *HS* leader-election algorithm is modified so that successive powers of k are used for path lengths, $k > 2$, instead of successive powers of 2. Analyze the time and communication complexity of the modified algorithm, similarly to the way the original *HS* algorithm is analyzed in the book. Compare the results to those for the original algorithm.

3.8. Consider modifying the *HS* algorithm so that the processes only send tokens in one direction rather than both.

(a) Show that the most straightforward modification to the algorithm in the text does not yield $O(n \log n)$ communication complexity. What is an upper bound for the communication complexity?

(b) Add a little more cleverness to the algorithm in order to restore the $O(n \log n)$ complexity bound.

3.9. Design a unidirectional leader-election algorithm that works with unknown ring size, and only uses $O(n \log n)$ messages in the worst case. Your algorithm should manipulate the UIDs using comparisons only.

3.10. Give code for a state machine to express the *TimeSlice* leader-election algorithm.

3.11. Describe a variant of the *TimeSlice* algorithm that saves time at the expense of additional messages, by allowing some number k of UIDs instead of just one to circulate in each phase. Prove the correctness of your algorithm and analyze its complexity.

3.12. Give code for a state machine to express the *VariableSpeeds* leader-election algorithm.

3.13. Show that the unmodified *VariableSpeeds* algorithm does not necessarily have the desired $O(n)$ communication complexity if processes can wake up at different times.

3.14. Prove the best *lower* bound you can for the number of rounds required, in the worst case, to elect a leader in a ring of size n. Be sure to state your assumptions carefully.

3.15. Give an explicit description of the bit-reversal ring for $n = 16$.

3.16. Prove that the bit-reversal ring of size $n = 2^k$ is $\frac{1}{2}$-symmetric, for any $k \in \mathbb{N}^+$.

3.17. Design a c-symmetric ring for non-powers of 2, for some value of $c > 0$.

3.18. Consider the problem of electing a leader in a synchronous ring of size n, where n is known to all the processes and the processes have no UIDs. Devise a *randomized* leader-election algorithm, that is, one in which the processes can make random choices in addition to just following their code deterministically. State carefully the properties that your algorithm satisfies. For example, is it absolutely guaranteed to elect a unique leader, or is there a small probability that it will fail to do this? What are the expected time and message complexities of your algorithm?

3.19. Consider a synchronous bidirectional ring of unknown size n, in which processes have UIDs. Give upper and lower bounds on the number of messages required by a comparison-based algorithm in which all the processes compute n mod 2.

This Page Intentionally Left Blank

Chapter 4

Algorithms in General Synchronous Networks

In Chapter 3, we presented algorithms and lower bounds for the problem of leader election in very simple synchronous networks—unidirectional and bidirectional rings. In this chapter, we consider a larger collection of problems in a larger class of synchronous networks. In particular, we present algorithms for *leader election*, *breadth-first search (BFS)*, finding *shortest paths*, finding a *minimum spanning tree (MST)*, and finding a *maximal independent set (MIS)*, in networks based on arbitrary graphs and digraphs.

The problem of leader election arises when a process must be selected to "take charge" of a network computation. The problems of breadth-first search, finding shortest paths, and finding a minimum spanning tree are motivated by the need to build structures suitable for supporting efficient communication. The problem of finding a maximal independent set arises from the problem of network resource allocation. (We will revisit many of these problems and algorithms later, in Chapter 15, in the context of asynchronous networks.)

In this chapter, we consider an arbitrary, strongly connected network digraph $G = (V, E)$ having n nodes. (Sometimes we will restrict attention to the case where all edges are bidirectional, i.e., where the graph is undirected.) We assume, as usual for synchronous systems, that the processes communicate only over the directed edges of the digraph. In order to name the nodes, we assign them the indices $1, \ldots, n$, but, unlike the ring's indices, these have no special connection to the nodes' positions in the graph. The processes do not know their indices, nor those of their neighbors, but refer to their neighbors by local names. We do assume that if a process i has the same process j for both an incoming and outgoing neighbor, then i knows that the two processes are the same.

4.1 Leader Election in a General Network

We start by reconsidering the problem of leader election, this time in a network based on an arbitrary strongly connected digraph.

4.1.1 The Problem

We assume that the processes have unique identifiers (UIDs), chosen from some totally ordered space of identifiers; each process's UID is different from each other's in the network, but there is no constraint on which UIDs actually appear. As in Chapter 3, the requirement is that, eventually, exactly one process should elect itself the leader, by changing a special *status* component of its state to the value *leader*. As in Chapter 3, there are several versions of the problem:

1. It might also be required that all non-leader processes eventually output the fact that they are not the leader, by changing their *status* components to *non-leader*.

2. The number n of nodes and the diameter, *diam*, can be either known or unknown to the processes. Or, an upper bound on these quantities might be known.

4.1.2 A Simple Flooding Algorithm

We give a simple algorithm that causes both leaders and non-leaders to identify themselves. The algorithm requires that the processes know *diam*. The algorithm just floods the maximum UID throughout the network, so we call it the *FloodMax algorithm*.

> ***FloodMax* algorithm (informal):**
>
> Every process maintains a record of the maximum UID it has seen so far (initially its own). At each round, each process propagates this maximum on all of its outgoing edges. After *diam* rounds, if the maximum value seen is the process's own UID, the process elects itself the leader; otherwise, it is a non-leader.

The code for process i follows.

> ***FloodMax* algorithm (formal):**
>
> The message alphabet is the set of UIDs.
>
> ***states*$_i$ consists of components:**
>
> u, a UID, initially i's UID
> *max-uid*, a UID, initially i's UID

status $\in \{$*unknown*, *leader*, *non-leader*$\}$, initially *unknown*
rounds, an integer, initially 0

msgs$_i$**:**
if *rounds* $<$ *diam* then
  send *max-uid* to all $j \in$ *out-nbrs*

trans$_i$**:**
rounds := *rounds* + 1
let U be the set of UIDs that arrive from processes in *in-nbrs*
max-uid := max$(\{$*max-uid*$\} \cup U)$
if *rounds* = *diam* then
  if *max-uid* = u then *status* := *leader*
  else *status* := *non-leader*

It is easy to see that *FloodMax* elects the process with the maximum UID. More specifically, define $i_{\max}$ to be the index of the process with the maximum UID, and $u_{\max}$ to be that UID. We show the following:

Theorem 4.1 *In the FloodMax algorithm, process i_{max} outputs leader and each other process outputs non-leader, within diam rounds.*

Proof. It is enough to prove the following assertion:

Assertion 4.1.1 *After diam rounds, status$_{i_{max}}$ = leader and status$_j$ = non-leader for every $j \neq i_{max}$.*

The key to the proof of Assertion 4.1.1 is the fact that after r rounds, the maximum UID has reached every process that is within distance r of $i_{\max}$, as measured along directed paths in G. This condition is captured by the invariant:

Assertion 4.1.2 *For $0 \leq r \leq diam$ and for every j, after r rounds, if the distance from i_{max} to j is at most r, then max-uid$_j$ = u_{max}.*

In particular, in view of the definition of the diameter of the graph, Assertion 4.1.2 implies that every process has the maximum UID by the end of *diam* rounds. To prove Assertion 4.1.2, it is useful to have the following additional auxiliary invariants:

Assertion 4.1.3 *For every r and j, after r rounds, rounds$_j$ = r.*

Assertion 4.1.4 *For every r and j, after r rounds, max-uid$_j$ $\leq$ u_{max}.*

Assertions 4.1.2, 4.1.3, and 4.1.4, specialized to $r = diam - 1$, plus an argument about what happens at round *diam*, imply Assertion 4.1.1 and therefore the result. □

The *FloodMax* algorithm can be regarded as a kind of generalization of the *LCR* algorithm of Section 3.3, because the *LCR* algorithm also floods the maximum value throughout the (ring) network. However, note that the *LCR* algorithm does not require any special knowledge about the network, such as its diameter. In *LCR*, a process is elected simply when it receives its own UID in a message, rather than after a specified number of rounds as in *FloodMax*. This strategy is particular to ring networks and does not work in general digraphs.

Complexity analysis. It is easy to see that the time until the leader is elected (and all other processes know that they are not the leader) is *diam* rounds. The number of messages is $diam \cdot |E|$, where $|E|$ is the number of directed edges in the digraph, because a message is sent on every directed edge for each of the first *diam* rounds.

Upper bound on the diameter. Note that the algorithm also works correctly if the processes all know an upper bound d on the diameter rather than the diameter itself. The complexity measures then increase so that they depend on d rather than *diam*.

4.1.3 Reducing the Communication Complexity

There is a simple optimization[1] that can be used to decrease the communication complexity in many cases, although it does not decrease the order of magnitude in the worst case. Namely, processes can send their *max-uid* values only when they first learn about them, not at every round. We call this algorithm *OptFloodMax*. The modification to the code for *FloodMax* is as follows.

***OptFloodMax* algorithm:**

***state*$_i$ has an additional component:**
new-info, a Boolean, initially *true*

***msgs*$_i$:**
if *rounds* $<$ *diam* and *new-info* $=$ *true* then
  send *max-uid* to all $j \in$ *out-nbrs*

[1] "Optimization" is not really the appropriate word to use here. "Improvement" would be better, but "optimization" is standard usage.

***trans*$_i$:**
rounds := *rounds* + 1
let U be the set of UIDs that arrive from processes in *in-nbrs*
if max (U) > *max-uid* then *new-info* := *true* else *new-info* := *false*
max-uid := max $(\{max\text{-}uid\} \cup U)$
if *rounds* = *diam* then
  if *max-uid* = u then *status* := *leader* else *status* := *non-leader*

It is easy to believe that this modification yields a correct algorithm. How can we prove this formally? One way is to carry out another invariant assertion proof similar to the one for *FloodMax*. However, this would involve repeating a lot of the work we have already done for the earlier proof. Instead of starting from scratch, we give a proof based on relating the *OptFloodMax* algorithm formally to the *FloodMax* algorithm. This is a simple example of the use of the *simulation* method for verifying the correctness of distributed algorithms.

Theorem 4.2 *In the OptFloodMax algorithm, process i_{max} outputs leader and each other process outputs non-leader, within diam rounds.*

Proof. It is enough to prove the following assertion, analogous to Assertion 4.1.1 in the proof for *FloodMax*.

> **Assertion 4.1.5** *After diam rounds, $status_{i_{max}}$ = leader and $status_j$ = non-leader for every $j \neq i_{max}$.*

We start by proving a preliminary invariant that says that a process's *new-info* flag is always set to *true* whenever there is new information that the process is supposed to send at the next round. More specifically, it says that if any outgoing neighbor of i does not know a UID at least as great as the maximum UID known by i, then i's *new-info* flag must be *true*.

> **Assertion 4.1.6** *For any r, $0 \leq r \leq diam$, and any i, j, where $j \in out\text{-}nbrs_i$, the following holds: after r rounds, if $max\text{-}uid_j < max\text{-}uid_i$ then $new\text{-}info_i$ = true.*

Assertion 4.1.6 is proved by induction on r. The basis case, $r = 0$, is true because all the *new-info* flags are initialized to *true*. For the inductive step, consider any particular processes i and j, where $j \in out\text{-}nbrs_i$. If $max\text{-}uid_i$ increases in round r, then $new\text{-}info_i$ gets set to *true*, which suffices. On the other hand, if $max\text{-}uid_i$ does not increase, then the inductive hypothesis implies that either $max\text{-}uid_j$ was already sufficiently large, or else $new\text{-}info_i$ = *true* just before round r. In the former case, $max\text{-}uid_j$ remains sufficiently large because

the value never decreases. In the latter case, the new information is sent from i to j at round r, which causes $max\text{-}uid_j$ to become sufficiently large.

Now, to prove that *OptFloodMax* is correct, we imagine running it side by side with *FloodMax*, starting with the same UID assignment. The heart of the proof is a *simulation relation*, which is just an invariant assertion that involves the states of both algorithms after the same number of rounds.

> **Assertion 4.1.7** *For any r, $0 \leq r \leq diam$, after r rounds, the values of the u, max-uid, status, and rounds components are the same in the states of both algorithms.*

The proof of the simulation assertion, Assertion 4.1.7, is carried out by induction on r, just as for the usual sorts of assertions involving only a single algorithm. The interesting part of the inductive step is showing that the *max-uid* values remain identical.

So consider any i, j, where $j \in out\text{-}nbrs_i$. If $new\text{-}info_i = true$ before round r, then i sends the same information to j in round r in *OptFloodMax* as it does in *FloodMax*. On the other hand, if $new\text{-}info_i = false$ before round r, then i sends nothing to j in round r in *OptFloodMax*, but sends $max\text{-}uid_i$ to j in round r in *FloodMax*. However, Assertion 4.1.6 implies that, in this case, $max\text{-}uid_j \geq max\text{-}uid_i$ before round r, so the message has no effect in *FloodMax*. It follows that i has the same effect on $max\text{-}uid_j$ in both algorithms. Since this is true for all i and j, it follows that the *max-uid* values remain identical in both algorithms.

Assertions 4.1.7 and 4.1.1 together imply Assertion 4.1.5, as needed. □

The method we just used to prove the correctness of *OptFloodMax* is often useful for proving the correctness of "optimized" versions of distributed algorithms. First, an inefficient but simple version of the algorithm is proved correct. Then a more efficient but more complicated version of the algorithm is verified by proving a formal relationship between it and the simple algorithm. For synchronous network algorithms, this relationship generally takes the form used above—an invariant involving the states of both algorithms after the same number of rounds.

Another improvement. It is possible to reduce the number of messages in the *FloodMax* algorithm slightly further. Namely, if a process i receives a new maximum from a process j that is both an incoming neighbor and an outgoing neighbor, that is, with which it has bidirectional communication, then i need not send a message in the direction of j at the next round.

It is possible to elect a leader in a general digraph network with UIDs, but in which no information about n or *diam* is available to the processes. We suggest that you stop here and try to construct an algorithm to do this. One possibility is to introduce an auxiliary protocol that allows each process to calculate the diameter of the network. Ideas presented later in this chapter might also be useful.

4.2 Breadth-First Search

The next problem we consider is that of performing a breadth-first search (BFS) in a network based on an arbitrary strongly connected directed graph having a distinguished source node. More precisely, we consider how to establish a *breadth-first spanning tree* for the digraph. The motivation for constructing such a tree comes from the desire to have a convenient structure to use as a basis for broadcast communication. The BFS tree minimizes the maximum communication time from the process at the distinguished node to all other processes in the network (under the simplifying assumption that it takes the same amount of time for a message to traverse each communication channel).

The BFS problem and its solutions are somewhat simpler in the case where all pairs of neighbors have bidirectional communication, that is, where the network graph is undirected. We will indicate the simplifications for this case.

4.2.1 The Problem

We define a *directed spanning tree* of a directed graph $G = (V, E)$ to be a rooted tree that consists entirely of directed edges in E, all edges directed from parents to children in the tree, and that contains every vertex of G. A directed spanning tree of G with root node i is *breadth-first* provided that each node at distance d from i in G appears at depth d in the tree (that is, at distance d from i in the tree). Every strongly connected digraph has a breadth-first directed spanning tree.

For the BFS problem, we suppose that the network is strongly connected and that there is a distinguished *source node* i_0. The algorithm is supposed to output the structure of a breadth-first directed spanning tree of the network graph, rooted at i_0. The output should appear in a distributed fashion: each process other than i_0 should have a *parent* component that gets set to indicate the node that is its parent in the tree.

As usual, processes only communicate over directed edges. Processes are assumed to have UIDs but to have no knowledge of the size or diameter of the network.

4.2.2 A Basic Breadth-First Search Algorithm

The basic idea for this algorithm is the same as for the standard sequential breadth-first search algorithm. We call this algorithm *SynchBFS*.

***SynchBFS* algorithm:**

At any point during execution, there is some set of processes that is "marked," initially just i_0. Process i_0 sends out a *search* message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a *search* message, it marks itself and chooses one of the processes from which the *search* has arrived as its parent. At the first round after a process gets marked, it sends a *search* message to all of its outgoing neighbors.

It is not hard to see that the *SynchBFS* algorithm produces a BFS tree. To show this formally, we can prove the invariant that after r rounds, every process at distance d from i_0 in the graph, $1 \leq d \leq r$, has its *parent* pointer defined; moreover, each such pointer points to a node at distance $d-1$ from i_0. This invariant can, as usual, be proved by induction on the number of rounds.

Complexity analysis. The time complexity is at most *diam* rounds. (Actually, this analysis can be refined a little, to the maximum distance from the particular node i_0 to any other node.) The number of messages is just $|E|$—a *search* message is transmitted exactly once on each directed edge.

Reducing the communication complexity. As for the *FloodMax* algorithm, it is possible to reduce the number of messages slightly: a newly marked process need not send a *search* message in the direction of any process from which it has already received such a message.

Message broadcast. The *SynchBFS* algorithm can easily be augmented to implement message broadcast. If a process has a message m that it wants to communicate to all of the processes in the network, it can simply initiate an execution of *SynchBFS* with itself as the root, *piggybacking* message m on the *search* message it sends in round 1. Other processes continue to piggyback m on all their *search* messages as well. Since the tree eventually spans all the nodes, message m is eventually delivered to all the processes.

Child pointers. In an important variant of the BFS problem, it is required that each process learn not only who its parent in the tree is, but also who all of its children are. In this case, it is necessary for each process receiving a *search*

message to respond to that message with a *parent* or *non-parent* message, telling the sender whether or not it has been chosen by the recipient as the parent.

If bidirectional communication is allowed between all pairs of neighbors, that is, if the network graph is undirected, then there is no difficulty—and little extra cost—in adding this extra communication. However, since we are allowing pairs of neighbors with only unidirectional communication, some of the *parent* and *non-parent* messages may need to be sent via indirect routes. For example, a *parent* or *non-parent* message could be sent via a new execution of *SynchBFS*, using piggybacking as above. In order for such a message to be recognized by its intended recipient, the message should also carry the UID of the intended recipient (plus a local name by which the recipient knows the sender), which should therefore itself be piggybacked on the original *search* message. Note that many executions of these *SynchBFS* "subroutines" can go on in parallel. In order to fit our formal model, in which at most one message can be sent on each link at each round, it may be necessary to combine many messages into one.

For a directed graph with unidirectional communication on some edges, in addition to outputting parent and child pointers, it may also be useful to have processes output information about the shortest routes from children to their parents. Such information could be produced, for example, using additional executions of *SynchBFS*.

Complexity analysis. If the graph is undirected, then the total time to compute a BFS tree, including child pointers, is $O(diam)$, and the communication complexity is $O(|E|)$.

Even if some of the pairs of neighbors have unidirectional communication, the time to compute the tree plus child pointers is still only $O(diam)$, because the extra BFS executions can all go on in parallel. In this case, the total number of messages is $O(diam|E|)$, because at most $|E|$ messages can be sent at each of the $O(diam)$ rounds. However, because a message might contain information from up to $|E|$ concurrent BFS executions, there might be as many as $|E|b$ bits in a message, where b is the maximum number of bits needed to represent a single UID. This yields a total of $O(diam|E|^2b)$ bits of communication. A smaller bound on the total number of bits can be obtained by noting that each of the (at most $|E|$) concurrent BFS executions uses at most $|E|$ messages, each having at most b bits. So the total number of communication bits is at most $O(|E|^2b)$.

Termination. How can the source process i_0 tell when the construction of the tree has been completed? If each *search* message is answered with either a *parent* or *non-parent* message, then after any process has received responses for all of its *search* messages, it knows who all its children in the BFS tree are and knows

that they have all been marked. So, starting from the leaves of the BFS tree, notification of completion can be "fanned in" to the source: each process can send notification of completion to its parent in the tree as soon as (a) it has received responses for all its *search* messages (so that it knows who its children are and knows that they have been marked), and (b) it has received notification of completion from all its children. This type of procedure is called a *convergecast*.

If the graph is undirected, then the total time to compute a BFS tree, including child pointers, and to propagate notification of completion back to the source is $O(diam)$ and the communication complexity is only $O(|E|)$. If unidirectional communication is allowed, then the total time, including notification of completion, is $O\left(diam^2\right)$. The reason the behavior is quadratic is that the notification has to proceed sequentially, one level at a time in the tree. The total number of messages is $O\left(diam^2|E|\right)$ and the total number of communication bits is at most $O(|E|^2 b)$.

4.2.3 Applications

Breadth-first search is one of the most basic building blocks for distributed algorithms. We give some examples here of how the *SynchBFS* algorithm can be used or augmented to help in performing other tasks.

Broadcast. As we mentioned earlier, a message broadcast can be implemented along with the establishment of a BFS tree. Another idea is first to produce a BFS tree with child pointers, as described above, and then to use the tree to conduct the broadcast. The message need only be propagated along edges from parents to their children. This allows the work of constructing the BFS tree to be reused, because many messages can be sent on the same tree. Once the BFS tree has been constructed, the additional time to broadcast a single message is only $O(diam)$, and the number of messages is only $O(n)$.

Global computation. Another application of BFS trees is the collection of information from throughout the network or, more generally, the computation of a function based on distributed inputs. For example, consider the problem in which each process has a nonnegative integer input value and we want to find the sum of all the inputs in the network. Using a BFS tree, this can be done easily (and efficiently) as follows. Starting from the leaves, "fan in" the results in a convergecast procedure, as follows. Each leaf sends its value to its parent; each parent waits until it gets the values from all its children, adds them to its own input value, and then sends the sum to its own parent. The sum calculated by the root of the BFS tree is the final answer.

Assuming that the BFS tree has already been constructed, and assuming bidirectional communication on all tree edges, this scheme requires $O(diam)$ time and $O(n)$ messages. The same scheme can be used to compute many other functions, for example, the maximum or minimum of the integer inputs. (What is required is that the function be associative and commutative.)

Electing a leader. Using *SynchBFS*, an algorithm can be designed to elect a leader in a network with UIDs, even when the processes have no knowledge of n or *diam*. Namely, all the processes can initiate breadth-first searches in parallel. Each process i uses the tree thereby constructed and the global computation procedure just described to determine the maximum UID of any process in the network. The process with the maximum UID then declares itself to be the leader, and all others announce that they are not the leader. If the graph is undirected, the time is $O(diam)$ and the number of messages is $O(diam|E|)$, again because at most $|E|$ messages can be sent at each of the *diam* rounds. The number of bits is at most $O(n|E|b)$, where b is the maximum number of bits used to represent a single UID.

Computing the diameter. The diameter of the network can be computed by having all processes initiate breadth-first searches in parallel. Each process i uses the tree thereby constructed to determine $max\text{-}dist_i$, defined to be the maximum distance from i to any other process in the network. Each process i then reuses its breadth-first tree for a global computation to discover the maximum of the *max-dist* values. If the graph is undirected, the time is $O(diam)$ and the number of messages is $O(diam|E|)$, the number of bits is $O(n|E|b)$. The diameter thus computed could be used, for example, in the leader-election algorithm *FloodMax*.

4.3 Shortest Paths

Now we examine a generalization of the BFS problem. Again, we consider a strongly connected directed graph, with the possibility of unidirectional communication between some pairs of neighbors. This time, however, we assume that each directed edge $e = (i, j)$ has an associated nonnegative real-valued *weight*, which we denote by $weight(e)$ or $weight_{i,j}$. The weight of a path is defined to be the sum of the weights on its edges. The problem is to find a shortest path from a distinguished source node i_0 in the digraph to each other node in the digraph, where a *shortest path* is defined to be a path with minimum weight.[2] A collection of shortest paths from i_0 to all the other nodes in the digraph constitutes a subtree of the digraph, all of whose edges are oriented from parent to child.

[2]The mixture of measures of weight and distance is unfortunate, but traditional.

As for breadth-first search, the motivation for constructing such a tree comes from the desire to have a convenient structure to use for broadcast communication. The weights represent costs that may be associated with the traversal of edges, for instance, communication delay or a monetary charge. A shortest paths tree minimizes the maximum worst-case cost of communicating with any process in the network.

We assume that every process initially knows the weight of all its incident edges, or, more precisely, that the weight of an edge appears in special *weight* variables at both its endpoint processes. We also assume that each process knows the number n of nodes in the digraph. We require that each process should determine its parent in a particular shortest paths tree, and also its distance (i.e., the total weight of its shortest path) from i_0.

If all edges are of equal weight, then a BFS tree is also a shortest paths tree. Thus, in this case, a trivial modification of the simple *SynchBFS* tree construction can be made to produce the distance information as well as the *parent* pointers.

The case where weights can be unequal is more interesting. One way to solve the problem is by the following algorithm—a version of the Bellman-Ford sequential shortest paths algorithm.

***BellmanFord* algorithm:**

Each process i keeps track of *dist*, the shortest distance from i_0 it knows so far, together with *parent*, the incoming neighbor that precedes i in a path whose weight is *dist*. Initially, $dist_{i_0} = 0$, $dist_i = \infty$ for $i \neq i_0$, and the *parent* components are undefined. At each round, each process sends its *dist* to all its outgoing neighbors. Then each process i updates its *dist* by a "relaxation step," in which it takes the minimum of its previous *dist* value and all the values $dist_j + weight_{j,i}$, where j is an incoming neighbor. If *dist* is changed, the *parent* component is also updated accordingly. After $n - 1$ rounds, *dist* contains the shortest distance, and *parent* the parent in the shortest paths tree.

It is not hard to see that, after $n - 1$ rounds, the *dist* values converge to the correct distances. One way to argue the correctness of *BellmanFord* is to show (by induction on r) that the following is true after r rounds: Every process i has its *dist* and *parent* components corresponding to a shortest path among those paths from i_0 to i consisting of at most r edges. (If there are no such paths, then $dist = \infty$ and *parent* is undefined.) We leave the details for an exercise.

Complexity analysis. The time complexity of the *BellmanFord* algorithm is $n - 1$, and the number of messages is $(n - 1)|E|$.

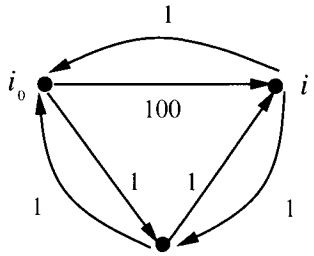


Figure 4.1: Shortest paths stabilize only after 2 rounds, though $diam = 1$.

Example 4.3.1 Time complexity of *BellmanFord*

You might suspect that by analogy with *SynchBFS*, the time complexity of *BellmanFord* is actually *diam*. An example that indicates why this is not the case is shown in Figure 4.1. In this example, it takes 2 rounds for the correct distance, 2, from i_0 to i to stabilize, since the path along which that distance is realized has two edges. However, the diameter is only 1.

The *BellmanFord* algorithm also works using an upper bound on n in place of n itself. If no such bound is known, it is possible to use techniques such as those described in Section 4.2 to discover one.

4.4 Minimum Spanning Tree

The next problem we consider is that of finding a minimum, or minimum-weight, spanning tree (MST) in an undirected graph network with weighted edges. Again, the main use for such a tree is as a basis for broadcast communication. A minimum-weight spanning tree minimizes the total cost for any source process to communicate with all the other processes in the network.

4.4.1 The Problem

A *spanning forest* of an undirected graph $G = (V, E)$ is a forest (i.e., a graph that is acyclic but not necessarily connected) that consists entirely of undirected edges in E and that contains every vertex of G. A *spanning tree* of an undirected graph G is a spanning forest of G that is connected. If there are weights associated with the undirected edges in E, then the *weight* of any subgraph of G (such as a spanning tree or spanning forest of G) is defined to be the sum of the weights of its edges.

Recall that we formalize the underlying undirected graph within our directed graph model as a directed graph having bidirectional edges between all pairs of neighbors. As in Section 4.3, we assume that each directed edge $e = (i, j)$ has an associated nonnegative real-valued *weight*, $weight(e) = weight_{i,j}$, only this time, we assume that for all i and j, $weight_{i,j} = weight_{j,i}$. We assume that every process initially knows the weight of all its incident edges, or, more precisely, that the weight of an edge appears in *weight* variables at both its endpoint processes. We assume that the processes have UIDs and know n. The problem is to find a minimum-weight (undirected) spanning tree for the entire network; specifically, each process is required to decide which of its incident edges are and which are not part of the minimum spanning tree.

4.4.2 Basic Theory

All known MST algorithms, sequential as well as concurrent, are based on the same simple theory, which we describe in this subsection. The basic strategy for constructing a minimum spanning tree involves starting with the trivial spanning forest consisting of n single nodes and repeatedly merging components along connecting edges until a spanning tree is produced. In order to end up with a minimum spanning tree, it is important that the merging occur only along certain selected edges—namely, those that are minimum-weight outgoing edges of some component. Justification for this method of selection is provided by the following lemma.

Lemma 4.3 *Let $G = (V, E)$ be a weighted undirected graph, and let $\{(V_i, E_i) : 1 \leq i \leq k\}$ be any spanning forest for G, where $k > 1$. Fix any i, $1 \leq i \leq k$. Let e be an edge of smallest weight in the set*

$$\{e' : e' \text{ has exactly one endpoint in } V_i\}.$$

Then there is a spanning tree for G that includes $\bigcup_j E_j$ and e, and this tree is of minimum weight among all the spanning trees for G that include $\bigcup_j E_j$.

Proof. By contradiction. Suppose the claim is false—that is, that there exists a spanning tree T that contains $\bigcup_j E_j$, does not contain e, and is of strictly smaller weight than any other spanning tree that contains $\bigcup_j E_j$ and e. Now consider the graph T' obtained by adding e to T. Clearly, T' contains a cycle, which has another edge $e' \neq e$ that is outgoing from V_i.

By the choice of e, we know that $weight(e') \geq weight(e)$. Now, consider the graph T'' constructed by deleting e' from T'. Then T'' is a spanning tree for G, it contains $\bigcup_j E_j$ and e, and its weight is no greater than that of T. But this contradicts the claimed property of T. □

Lemma 4.3 provides the justification for the following general strategy for constructing an MST.

> **General strategy for MST:**
> Start with the trivial spanning forest that consists of n individual nodes and no edges. Then repeatedly do the following: Select an arbitrary component C in the forest and an arbitrary outgoing edge e of C having minimum weight among the outgoing edges of C. Combine C with the component at the other end of e, including edge e in the new combined component. Stop when the forest has a single component.

Lemma 4.3 can be used in an inductive proof to show that, at any stage in the construction, the forest is a subgraph of an MST. Several well-known sequential MST algorithms are special cases of this general strategy. For example, the *Prim-Dijkstra algorithm* begins by distinguishing one of the initial single-node components and repeatedly adds the minimum-weight outgoing edge from the current component, each time attaching a single new node until a complete spanning tree is obtained. For another example, the *Kruskal algorithm* repeatedly adds the minimum-weight edge among all the edges that join two separate components in the current spanning forest, thus combining components until there is only one component, which is the final spanning tree.

In order to use this general strategy in a distributed setting, it would be nice to be able to extend the forest with several edges determined concurrently. That is, each of several components could determine its minimum-weight outgoing edge independently, and then all of the determined edges could be added to the forest, thereby causing several combinations of components to occur all at once. But Lemma 4.3 does not guarantee the correctness of this parallel strategy. In fact, the strategy is not correct, in general.

Example 4.4.1 Cycle creation in parallel MST algorithm

> Consider the graph in Figure 4.2. The dots represent components in the spanning forest. The three edges with weight 1 are the only outgoing edges. If the components choose their minimum-weight outgoing edges as depicted by the arrows, a cycle would be created.

The cycle problem is avoidable, however, in the special case where all the edges have distinct weights. This is because of the following lemma.

Lemma 4.4 *If all edges of a graph G have distinct weights, then there is exactly one MST for G.*

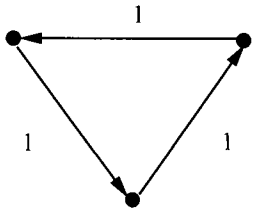


Figure 4.2: A cycle created by concurrent choices of minimum-weight outgoing edges.

Proof. The proof is similar to the proof of Lemma 4.3. Suppose that there are two distinct minimum-weight spanning trees, T and T', and let e be the minimum-weight edge that appears in only one of the two trees. Suppose (without loss of generality) that $e \in T$. Then the graph T'' obtained by adding e to T' contains a cycle, and at least one other edge in that cycle, e', is not in T. Since the edge weights are all distinct and since e' is in only one of the two trees, we must have $weight(e') > weight(e)$, by our choice of e. Then removing e' from T'' yields a spanning tree with a smaller weight than T', which is a contradiction. □

Now reconsider the general strategy for the case where the graph has distinct edge weights, and so, by Lemma 4.4, there is a unique MST. In this case, at any stage of the construction, any component in the forest has exactly one minimum-weight outgoing edge (which we abbreviate, unpronounceably, as MWOE). Lemma 4.3 implies that if we begin the stage with a forest, all of whose edges are in the unique MST, then all of the MWOEs, for all components, are also in the unique MST. So we can add them all at once, without any danger of creating a cycle.

4.4.3 The Algorithm

We present a distributed algorithm for constructing an MST in an arbitrary weighted undirected graph, following the general strategy described in the previous subsection. Since components will be allowed to combine concurrently, we assume that edge weights are all distinct; near the end of this subsection, we will say how this assumption can be removed. We call the algorithm *SynchGHS* because it is based on an asynchronous algorithm developed by Gallager, Humblet, and Spira. (We will present the asynchronous algorithm, called simply *GHS*, in Section 15.5.)

***SynchGHS* algorithm:**

The algorithm builds the components in "levels." For each k, the level k components constitute a spanning forest, where each level k component consists of a tree that is a subgraph of the MST. Each level k component has at least 2^k nodes. Every component, at every level, has a distinguished leader node. The processes allow a fixed number of rounds, which is $O(n)$, to complete each level.

The algorithm starts with level 0 components consisting of individual nodes and no edges. Suppose inductively that the level k components have been determined (along with their leaders). More specifically, suppose that each process knows the UID of the leader of its component; this UID is used as an identifier for the entire component. Each process also knows which of its incident edges are in the component's tree.

To get the level $k+1$ components, each level k component conducts a search along its spanning tree edges for the MWOE of the component. The leader broadcasts search requests along tree edges, using the message broadcast strategy described in Section 4.2. Each process finds, among its incident edges, the one of minimum weight that is outgoing from the component (if there is any such edge); it does this by sending *test* messages along all non-tree edges, asking whether or not the other end is in the same component. (This determination is made by comparing the component identifiers.) Then the processes convergecast this local minimum-weight edge information toward the leader, taking minima along the way. The minimum obtained by the leader is the MWOE of the entire component.

When all level k components have found their MWOEs, the components are combined along all these MWOEs to form the level $k+1$ components. This involves the leader of each level k component communicating with the component process adjacent to the MWOE, to tell it to mark the edge as being in the new tree; the process at the other end of the edge is also told to do the same thing.

Then a new leader is chosen for each level $k+1$ component, as follows. It can be shown that for each group of level k components that get combined into a single level $k+1$ component, there is a unique edge e that is the common MWOE of *two* of the level k components in the group. (We argue this below.) We let the new leader be the endpoint of e having the larger UID. Note that this new leader can identify itself using only information available locally.

Finally, the UID of the new leader is propagated throughout the new component, using a broadcast.

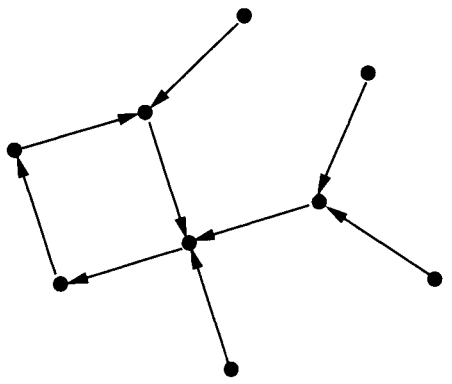


Figure 4.3: A graph in which each node has exactly one outgoing edge. Note the unique cycle.

Eventually, after some number of levels, the spanning forest consists of only a single component containing all the nodes in the network. Then a new attempt to find a MWOE will fail, because no process will find an outgoing edge. When the leader learns this, it broadcasts a message saying that the algorithm is completed.

A key to the algorithm is the fact that, among each group of level k components that get combined, there is a unique (undirected) edge that is the common MWOE of both endpoint components. In order to see why this is so, consider the *component digraph* G', whose nodes are the level k components that combine to form one level $k+1$ component and whose edges are the MWOEs. G' is a weakly connected digraph in which every node has exactly one outgoing edge. (A digraph is *weakly connected* if its undirected version, obtained by ignoring the directions of all the edges, is connected.) So we can use the following property:

Lemma 4.5 *Let G be a weakly connected digraph in which each node has exactly one outgoing edge. Then G contains exactly one cycle.*

Proof. The proof is left as an exercise. □

Example 4.4.2 Graph with one outgoing edge per node

Figure 4.3 shows an example of a graph in which each node has exactly one outgoing edge.

We apply Lemma 4.5 to the component digraph G' to obtain the unique cycle of components. Because of the way G' was constructed, successive edges

in the cycle must have nonincreasing weights; therefore, the length of this cycle cannot be greater than 2. So the length of the unique cycle is exactly 2. But this corresponds to an edge that is the common MWOE of both adjacent components.

In the *SynchGHS* algorithm, it is crucial that the levels be kept synchronized. This is needed to ensure that when a process i tries to determine whether or not the other endpoint j of a candidate edge is in the same component, both i and j have up-to-date component UIDs. If the UID at j is observed to be different from that at i, we would like to be certain that i and j really are in different components, not just that they haven't yet received their component UIDs from their leaders. In order to execute the levels synchronously, processes allow a predetermined number of rounds for each level. To be certain that all the computation for the round has completed, this number will be $O(n)$; note that $O(diam)$ rounds are not always sufficient. The need to count this number of rounds is the only reason that the nodes need to know n. (In Section 15.5, when we revisit this algorithm in the asynchronous network setting, we will use a different strategy for synchronizing the components.)

Complexity analysis. Note first that the number of nodes in each level k component is at least 2^k. This can be shown by induction, using the fact that at each level, each component is combined with at least one other component at the same level. Therefore, the number of levels is at most $\log n$. Since each level takes time $O(n)$, it follows that the time complexity of *SynchGHS* is $O(n \log n)$. The communication complexity is $O((n + |E|) \cdot \log n)$, since at each level, $O(n)$ messages are sent in total along all the tree edges, and $O(|E|)$ additional messages are required for finding the local minimum-weight edges.

Reducing the communication. It is possible to reduce the number of messages to $O(n \log n + |E|)$ by using a more careful strategy to find local minimum-weight edges. This improvement causes an increase in the time complexity, although it does not increase its order of magnitude. The idea is as follows.

Each process marks its incident edges as "rejected" when they are found to lead to a node in the same component; thereafter, there is no need to test them again. Also, at each level, the remaining candidate edges are tested one at a time, in order of increasing weight, just until the first one is found that leads outside the component (or until the candidate edges are exhausted).

With this improvement, the number of messages sent over tree edges is, as before, $O(n \log n)$. We carry out an amortized analysis of the number of messages used for finding local minimum-weight edges. Each edge gets tested and rejected at most once, for a total of $O(|E|)$. An edge that is tested and is found to be the local minimum-weight edge, but not the MWOE for the entire

component, may be tested again. However, there is at most one such exploration originating at each node at each level, adding up to a total of $O(n \log n)$. The total communication complexity is thus $O(n \log n + |E|)$.

The strategy just described has another advantage. Since each node marks both its incident edges that are in the MST and those that are not in the tree, there is no need for the final phase in which the leader notifies everyone that the algorithm is completed. Each node can simply output the information about its adjacent edges as it is discovered.

Non-unique edge weights. Now consider the MST problem for a graph whose edge weights are not necessarily distinct. In this case, the *SynchGHS* algorithm can be used, with a small modification. Note first that the *SynchGHS* algorithm only manipulates the weights using $\{<, >, =\}$ comparisons.

Given arbitrary edge weights, we can derive a set of distinct *edge identifiers* using the UIDs. The identifer of an edge (i, j) is the triple $(weight_{i,j}, v, v')$, where v and v' are the UIDs of i and j, with $v < v'$. (Thus, (i, j) and (j, i) have the same edge identifier.) A total ordering is defined among the edge identifiers, based on lexicographic order among the triples.

Since *SynchGHS* manipulates the weights using comparisons only, we can run it using the edge identifiers in place of the (real-valued) weights; the resulting execution will be the same as if *SynchGHS* were running with a set of unique weights satisfying the same ordering relationships. A tree is thus produced. We leave as an exercise the task of showing that this tree is in fact an MST for the original graph.

Leader election. Once an MST (or any spanning tree) is known for a network based on an undirected graph, it is easy to elect a unique leader, provided UIDs are available. Namely, the leaves of the spanning tree begin a convergecast along the paths of the tree; each internal node waits to hear from all but one of its neighbors before sending a message to its remaining neighbor. If a node hears from all its neighbors without having itself sent out a message, it declares itself the leader. Also, if two neighboring nodes get messages from each other at the same round, then one of them, say, the one with the larger UID, declares itself the leader. The total additional complexity of this leader-election procedure (after the MST is constructed) is just $O(n)$ time and $O(n)$ messages.

Combining this with the MST complexity analysis, we see that, starting with a weighted undirected graph in which the nodes know n (but not *diam*), a leader can be elected in time $O(n \log n)$, with $O(n \log n + |E|)$ communication.

4.5 Maximal Independent Set

The final problem we consider in this chapter is that of finding a *maximal independent set* (*MIS*) of the nodes of an undirected graph. A set of nodes is called an *independent set* if it contains no pair of neighboring nodes, and an independent set is said to be *maximal* if it cannot be increased to form a larger independent set by the addition of any other nodes. Note that an undirected graph can have many different maximal independent sets. We do not require the largest possible maximal independent set—any will do.

The MIS problem can be motivated by problems of allocating shared resources to processes in a network. The neighbors in the graph G might represent processes than cannot simultaneously perform some activity involving shared resources (for example, database access or radio broadcast). We might wish to select a set of processes that could be allowed to act simultaneously; in order to avoid conflict, these processes should comprise an independent set in G. Furthermore, for performance reasons, it is undesirable to block a process if none of its neighbors is active; thus, the chosen set of processes should be maximal.

4.5.1 The Problem

Let $G = (V, E)$ be an undirected graph. A set $I \subseteq V$ of nodes is said to be *independent* if for all nodes $i, j \in I$, $(i, j) \notin E$. An independent set I is *maximal* if any set I' that strictly contains I is not independent. The goal is to compute a maximal independent set of G. More specifically, each process whose index is in I should eventually output *winner*, that is, should set a special *status* component of its state to the value *winner*, and each process whose index is not in I should output *loser*.

We assume that n, the number of nodes, is known to all the processes. (We could, alternatively, use an upper bound on n.) We do not assume the existence of UIDs.

4.5.2 A Randomized Algorithm

It is not hard to show that in some graphs, the MIS problem cannot be solved if the processes are required to be deterministic. The argument is similar to the one in the proof of Theorem 3.1. In this section, we present a simple solution that uses *randomization* to overcome this inherent limitation of deterministic systems. To be precise, we note that the randomized algorithm actually solves a weaker problem than the one that is stated above, in that it will have a (probability zero) possibility of never terminating. We call this algorithm *LubyMIS*, after Luby, its discoverer.

LubyMIS is based on the following iterative scheme, in which an arbitrary nonempty independent set is selected from the given graph G, the nodes in this set and all of their neighbors are removed from the graph, and the process is repeated. If W is a subset of the nodes of a graph, then we use $nbrs(W)$ to denote the set of neighbors of nodes in W.

Let *graph* be a record with fields *nodes*, *edges*, and *nbrs*, initialized to
the indicated components of the original graph G.
Let I be a set of nodes, initially empty.

while $graph.nodes \neq \phi$ do
  choose a nonempty set $I' \subseteq graph.nodes$ that is independent in *graph*
  $I := I \cup I'$
  graph := the induced subgraph[3] of *graph* on $graph.nodes - I' - graph.nbrs(I')$
end while

It is not hard to see that this scheme always produces a ~~maximal~~ independent set. To see why it is independent, note that at each stage, the selected set I' is independent, and we explicitly discard from the remaining graph all neighbors of vertices that are put into I. To see why it is maximal, note that the only nodes that are removed from consideration are neighbors of nodes that are put into I.

The key question in implementing this general scheme in a distributed network is how to choose the set I' at each iteration. Here is where randomization is used. In each stage, each process i chooses an integer val_i in the range $\{1, \ldots, n^4\}$ at random, using the uniform distribution. The reason for the use of n^4 as a bound is that it is sufficiently large so that, with high probability, all processes in the graph will choose distinct values. (We do not carry out this calculation in this book, but instead refer you to Luby's research paper.) Once the processes have chosen these values, we define I' to consist of all the nodes i that are local *winners*, that is, those nodes i such that $val_i > val_j$ for all neighbors j of i. This obviously yields an independent set, since two neighbors cannot simultaneously defeat each other.

In this implementation it is possible, if the random choices are unlucky, that the set I' might be empty at some stages; those stages will be "useless," accomplishing nothing. Provided the algorithm does not reach a point after which it keeps performing useless stages forever, we can simply ignore the useless stages and assert that *LubyMIS* correctly follows the general scheme. We will, how-

[3] The *induced subgraph* of a graph G on a subset W of its nodes is defined to be the subgraph (W, E'), where E' is the set of edges of G that connect nodes in W.

ever, have to take the useless stages into account in the analysis. The algorithm follows.

***LubyMIS* algorithm (informal):**

The algorithm works in *stages*, each consisting of three rounds.

Round 1: In the first round of a stage, the processes choose their respective *vals* and send them to their neighbors. By the end of round 1, when all the *val* messages have been received, the winners—that is, the processes in I'—know who they are.

Round 2: In the second round, the winners notify their neighbors. By the end of round 2, the losers—that is, the processes having neighbors in I'—know who they are.

Round 3: In the third round, each loser notifies its neighbors. Then all the involved processes—the winners, the losers, and the losers' neighbors—remove the appropriate nodes and edges from the graph. More precisely, this means the winners and losers discontinue participation after this stage, and the losers' neighbors remove all the edges that are incident on the newly removed nodes.

We now describe the algorithm more formally in our model. As described in Section 2.7, each process uses a special random function $rand_i$, which it applies at each round prior to applying the $msgs_i$ and $trans_i$ functions. Here, we use *random* to indicate a random choice from $\{1, \ldots, n^4\}$, using the uniform distribution.

***LubyMIS* algorithm (formal):**

***states*$_i$:**
round $\in \{1, 2, 3\}$, initially 1
val $\in \{1, \ldots, n^4\}$, initially arbitrary
awake, a Boolean, initially *true*
rem-nbrs, a set of vertices, initially the neighbors in the original graph G
status $\in \{unknown, winner, loser\}$, initially *unknown*

***rand*$_i$:**
if *awake* and *round* $= 1$ then *val* := *random*

***msgs*$_i$:**
if *awake* then
  case
    round $= 1$:
      send *val* to all nodes in *rem-nbrs*
    round $= 2$:
      if *status* $=$ *winner* then

```
        send winner to all nodes in rem-nbrs
    round = 3:
        if status = loser then
            send loser to all nodes in rem-nbrs
endcase
```

In the following code, we identify 3 with 0, modulo 3.

```
trans_i:
if awake then
    case
        round = 1:
            if val > v for all incoming values v then status := winner
        round = 2:
            if a winner message arrives then status := loser
        round = 3:
            if status ∈ {winner, loser} then awake := false
            rem-nbrs := rem-nbrs − {j : a loser message arrives from j}
    endcase
    round := (round + 1 mod 3)
```

Note that *LubyMIS* still works correctly if, at some stages, some neighboring processes choose the same random values.

4.5.3 Analysis*

We have already argued that, provided that *LubyMIS* does not stall, executing useless stages forever, it will produce an MIS. Now we claim that with probability one, the algorithm in fact does not stall. More specifically, we claim that at any stage of the algorithm, the expected number of edges removed from the remaining graph is at least a constant fraction of the total number of remaining edges; this implies that there is a constant probability that at least a constant fraction of the edges is removed. In turn, this implies that the expected number of rounds until termination is $O(\log n)$. It also implies that, with probability one, the algorithm does in fact terminate.

The complete analysis of *LubyMIS* can be found in Luby's original paper; it involves substantial counting arguments about graphs. We just state the main technical lemma without proof, and indicate how it is used to obtain the needed results. For the next three lemmas, fix $G = (V, E)$ and, for an arbitrary node $i \in V$, define

$$sum(i) = \sum_{j \in nbrs_i} \frac{1}{d(j)} ,$$

where $d(j)$ is the degree of j in G. Here is the technical lemma:

Lemma 4.6 *Let I' be defined as in one stage of the LubyMIS algorithm. Then, for every i in the graph just before the stage,*

$$\Pr[i \in nbrs(I')] \geq \frac{1}{4} \min\left(\frac{sum(i)}{2}, 1\right).$$

Using Lemma 4.6, we obtain the bound on the expected number of edges removed from the graph:

Lemma 4.7 *The expected number of edges removed from G in a single stage of LubyMIS is at least $\frac{|E|}{8}$.*

Proof. The algorithm ensures that every edge with at least one endpoint in $nbrs(I')$ is removed. It follows that the expected number of edges removed is at least

$$\frac{1}{2} \sum_{i \in V} d(i) \cdot \Pr[i \in nbrs(I')].$$

This is because each vertex i has the indicated probability of having a neighbor in I'; if this is the case, then i is removed, which causes the deletion of all of its $d(i)$ incident edges. The factor of $\frac{1}{2}$ is included to compensate for possible overcounting of removed edges, since each edge has two endpoints that could cause its deletion.

We next plug in the bound from Lemma 4.6, concluding that the expected number of removed edges is at least

$$\frac{1}{8} \sum_{i \in V} d(i) \cdot \min\left(\frac{sum(i)}{2}, 1\right).$$

Breaking this up according to which term of the *min* is less, this is equal to

$$\frac{1}{8}\left(\frac{1}{2} \sum_{i:sum(i)<2} d(i) \cdot sum(i) + \sum_{i:sum(i)\geq 2} d(i)\right).$$

Now we expand the definition of $sum(i)$ and also write $d(i)$ as a trivial sum, obtaining

$$\frac{1}{8}\left(\frac{1}{2} \sum_{i:sum(i)<2} \sum_{j \in nbrs_i} \frac{d(i)}{d(j)} + \sum_{i:sum(i)\geq 2} \sum_{j \in nbrs_i} 1\right).$$

Note that each undirected edge (i, j) contributes two summation terms to the expression in parentheses, one for each direction; in each case, the sum of these two terms is greater than 1. So the total is at least $\frac{|E|}{8}$. □

Lemma 4.7 can be used to conclude

Lemma 4.8 *With probability at least $\frac{1}{16}$, the number of edges removed from G in a single stage of LubyMIS is at least $\frac{|E|}{16}$.*

Using both Lemmas 4.7 and 4.8, we conclude:

Theorem 4.9 *With probability one, LubyMIS eventually terminates. Moreover, the expected number of rounds until termination is $O(\log n)$.*

Randomized algorithms. The technique of randomization is used frequently in distributed algorithms. Its main use is to break symmetry. For example, the leader-election and MIS problems cannot be solved in general graphs by deterministic processes without UIDs because of the impossibility of breaking symmetry. In contrast, these problems can be solved using randomization. Even when there are UIDs, randomization may allow symmetry to be broken faster.

One problem with randomized algorithms, however, is that their guarantees of correctness and/or performance might only hold with high probability, not with certainty. In designing such algorithms, it is important to make sure that the crucial properties of the algorithm are guaranteed with certainty, not probabilistically. For example, any execution of *LubyMIS* is guaranteed to produce an independent set, regardless of the outcomes of the random choices. The performance, however, depends on the luckiness of the random choices. There is even a (probability zero) possibility that all processes will repeatedly choose the same value, thereby stalling progress forever. Whether or not these are serious drawbacks to the algorithm depends on the application for which it is used.

4.6 Bibliographic Notes

The *FloodMax* and *OptFloodMax* algorithms appear to be folklore. Afek and Gafni [6] have developed complexity bounds for leader election in complete synchronous networks. The *SynchBFS* algorithm is based on the standard sequential breadth-first search algorithm appearing, for example, in [83]. The *BellmanFord* algorithm is a distributed version of a sequential algorithm developed (separately) by Bellman and Ford [43, 125].

The *SynchGHS* algorithm is a synchronized (and therefore considerably simplified) version of the well-known asynchronous MST algorithm developed by Gallager, Humblet, and Spira. The *LubyMIS* algorithm and its analysis appear in a paper by Luby [200].

An example of a (probability zero) execution of a randomized algorithm in which the processes keep making the same choice appears in [271].

4.7 Exercises

4.1. Fill in more details in the correctness proof for the *FloodMax* algorithm.

4.2. In terms of n, the number $diam|E|$ of messages used in the *FloodMax* algorithm is easily seen to be $O(n^3)$. Either produce a class of digraphs in which the product $diam|E|$ really is $\Omega(n^3)$ or show that no such class of digraphs exists.

4.3. For the *OptFloodMax* algorithm, either prove a smaller upper bound than $O(n^3)$ on the number of messages or exhibit a class of digraphs and corresponding UID assignments in which the number of messages is $\Omega(n^3)$.

4.4. Consider the "further optimized" version of *OptFloodMax* described at the end of Section 4.1.3, which prevents processes from resending *max-uid* information to processes from which they have previously received the same information.

(a) Give code for this algorithm, in the same style as the other code in this chapter.

(b) Prove the correctness of your algorithm by relating it to *OptFloodMax*, using the same sort of simulation strategy used in the proof of correctness for *OptFloodMax* itself (i.e., in the proof of Theorem 4.2).

4.5. (a) Write the code for the *SynchBFS* algorithm.

(b) Prove the correctness of your algorithm using invariant assertions.

(c) Do the same—parts (a) and (b)—for the *SynchBFS* algorithm with child pointers.

(d) Do the same—parts (a) and (b)—for the *SynchBFS* algorithm with child pointers and notification of completion.

4.6. Consider the optimized version of *SynchBFS* described in Section 4.2.2, which prevents processes from sending *search* messages to processes from which they have previously received such messages.

(a) Give code for this algorithm.

(b) Prove the correctness of your algorithm by relating it to *SynchBFS*, using the same sort of simulation strategy used in the proof of correctness for *OptFloodMax* (i.e., in the proof of Theorem 4.2).

4.7. Describe in detail an algorithm that extends *SynchBFS* to produce not only child pointers, but also information about shortest routes from children in the BFS tree to their parents. This information should be distributed along those paths so that each process on a path knows the next process along the path. Analyze the time and communication complexity.

4.8. Describe in detail an algorithm that extends *SynchBFS* to allow the source process i_0 to broadcast a message to all other processes and obtain an acknowledgment that all processes have received it. Your algorithm should use $O(|E|)$ messages and $O(diam)$ time. You may assume that the network graph is undirected.

4.9. Analyze the time and communication complexity of the global computation scheme, the leader-election scheme and the diameter computation scheme at the end of Section 4.2, assuming that communication is allowed to be unidirectional between some pairs of neighbors.

4.10. Devise the most efficient leader-election algorithm you can, for a strongly connected directed network in which the processes have UIDs but do not have any knowledge of the number of nodes in or diameter of the network.

(a) Do this assuming that communication is bidirectional between every pair of neighbors, that is, that the network graph is undirected.

(b) Do this without making this assumption.

Analyze.

4.11. Develop the most efficient algorithm you can for finding the total number of nodes in a strongly connected directed network in which the processes have UIDs.

(a) Do this assuming that communication is bidirectional between every pair of neighbors, that is, that the network graph is undirected.

(b) Do this without making this assumption.

Analyze.

4.12. Develop the most efficient algorithm you can for finding the total number of edges in a strongly connected directed network in which the processes have UIDs.

(a) Do this assuming that communication is bidirectional between every pair of neighbors, that is, that the network graph is undirected.

(b) Do this without making this assumption.

Analyze.

4.13. Develop the most efficient algorithm you can, for an arbitrary undirected graph network, to determine a minimum-height rooted spanning tree. You may assume the processes have UIDs, but there is no distinguished leader node.

4.14. (a) Give code for the *BellmanFord* shortest paths algorithm.

(b) Prove its correctness using invariant assertions.

4.15. Give code for the *SynchGHS* algorithm.

4.16. Prove Lemma 4.5.

4.17. In the *SynchGHS* algorithm, show that it is not the case that $O(diam)$ rounds are always sufficient to complete each level of the computation.

4.18. Show that the version of *SynchGHS* that uses edge identifiers in place of edge weights (described near the end of Section 4.4) in fact produces an MST.

4.19. *Research Question:* Come up with a better synchronous minimum spanning tree algorithm than *SynchGHS*—better in terms of the time complexity, the communication complexity, or both.

4.20. Give code for the convergecast algorithm outlined at the end of Section 4.4, which elects a leader given an arbitrary spanning tree of an undirected graph network.

4.21. Give the best upper and lower bounds you can for the problem of establishing an arbitrary spanning tree in an undirected graph network. You may assume UIDs, but no weights. State carefully what assumptions you use about the processes' knowledge of the graph.

4.22. Consider a *line network*, that is, a linear collection of n processes $1, \ldots, n$, where each process is bidirectionally connected to its neighbors. Assume that each process i can distinguish its left from its right and knows whether or not it is an endpoint.

Assume that each process i initially has a very large integer value v_i and that it can hold in memory only a constant number of such values at any time. Design an algorithm to sort the values among the processes, that is, to cause each process i to return one output value o_i, where the multiset of outputs is equal to the multiset of inputs and $o_1 \leq \ldots \leq o_n$. Try to design the most efficient algorithm you can both in terms of the number of messages and the number of rounds. Prove your claims.

4.23. Prove that, under the assumptions given in Section 4.5, but assuming that the processes are deterministic rather than probabilistic, there are some graphs in which it is impossible to solve the MIS problem. Find the largest class of graphs you can for which your impossibility result holds.

4.24. Suppose that *LubyMIS* is executed in a ring of size n. Estimate the probability that any particular edge is removed from the graph in one iteration of the algorithm.