Formally, the definition of an execution for the synchronous case is further constrained over the definition from the asynchronous case as follows. The sequence of alternating configurations and events can be partitioned into disjoint rounds. A *round* consists of a deliver event for every message in an *outbuf* variable, until all *outbuf* variables are empty, followed by one computation event for every processor. Thus a round consists of delivering all pending messages and then having every processor take an internal computation step to process all the delivered messages.

An execution is *admissible* for the synchronous model if it is infinite. Because of the round structure, this implies that every processor takes an infinite number of computation steps and every message sent is eventually delivered. As in the asynchronous case, assuming that admissible executions are infinite is a technical convenience; termination of an algorithm can be handled as in the asynchronous case.

Note that in a synchronous system with no failures, once the algorithm is fixed, the only relevant aspect of executions that can differ is the initial configuration. In an asynchronous system, there can be many different executions of the same algorithm, even with the same initial configuration and no failures, because the interleaving of processor steps and the message delays are not fixed.

2.1.2 Complexity Measures

We will be interested in two complexity measures, the number of messages and the amount of time, required by distributed algorithms. For now, we will concentrate on worst-case performance; later in the book we will sometimes be concerned with expected-case performance.

To define these measures, we need a notion of the algorithm terminating. We assume that each processor's state set includes a subset of *terminated* states and each processor's transition function maps terminated states only to terminated states. We say that the system (algorithm) has *terminated* when all processors are in terminated states and no messages are in transit. Note that an admissible execution must still be infinite, but once a processor has entered a terminated state, it stays in that state, taking "dummy" steps.

The *message complexity* of an algorithm for either a synchronous or an asynchronous message-passing system is the maximum, over all admissible executions of the algorithm, of the total number of messages sent.

The natural way to measure time in synchronous systems is simply to count the number of rounds until termination. Thus the *time complexity* of an algorithm for a synchronous message-passing system is the maximum number of rounds, in any admissible execution of the algorithm, until the algorithm has terminated.

Measuring time in an asynchronous system is less straightforward. A common approach, and the one we will adopt, is to assume that the maximum message delay in any execution is one unit of time and then calculate the running time until termination. To make this approach precise, we must introduce the notion of time into executions.

A *timed* execution is an execution that has a nonnegative real number associated with each event, the *time* at which that event occurs. The times must start at 0, must be nondecreasing, must be strictly increasing for each individual processor[1], and must increase without bound if the execution is infinite. Thus events in the execution are ordered according to the times at which they occur, several events can happen at the same time as long as they do not occur at the same processor, and only a finite number of events can occur before any finite time.

We define the *delay* of a message to be the time that elapses between the computation event that sends the message and the computation event that processes the message. In other words, it consists of the amount of time that the message waits in the sender's *outbuf* together with the amount of time that the message waits in the recipient's *inbuf*.

The *time complexity* of an asynchronous algorithm is the maximum time until termination among all timed admissible executions in which every message delay is at most one. This measure still allows arbitrary interleavings of events, because no lower bound is imposed on how closely events occur. It can be viewed as taking any execution of the algorithm and normalizing it so that the longest message delay becomes one unit of time.

2.1.3 Pseudocode Conventions

In the formal model just presented, an algorithm would be described in terms of state transitions. However, we will seldom do this, because state transitions tend to be more difficult for people to understand; in particular, flow of control must be coded in a rather contrived way in many cases.

Instead, we will describe algorithms at two different levels of detail. Simple algorithms will be described in prose. Algorithms that are more involved will also be presented in pseudocode. We now describe the pseudocode conventions we will use for synchronous and asynchronous message-passing algorithms.

Asynchronous algorithms will be described in an interrupt-driven fashion for each processor. In the formal model, each computation event processes all the messages waiting in the processor's *inbuf* variables at once. For clarity, however, we will generally describe the effect of each message individually. This is equivalent to the processor handling the pending messages one by one in some arbitrary order; if more than one message is generated for the same recipient during this process, they can be bundled together into one big message. It is also possible for the processor to take some action even if no message is received. Events that cause no message to be sent and no state change will not be listed.

The local computation done within a computation event will be described in a style consistent with typical pseudocode for sequential algorithms. We use the reserved word "terminate" to indicate that the processor enters a terminated state.

An asynchronous algorithm will also work in a synchronous system, because a synchronous system is a special case of an asynchronous system. However, we will often be considering algorithms that are specifically designed for synchronous

[1]$comp(i)$ is considered to occur at p_i and $del(i, j, m)$ at both p_i and p_j.

systems. These synchronous algorithms will be described on a round-by-round basis for each processor. For each round we will specify what messages are to be sent by the processor and what actions it is to take based on the messages just received. (Note that the messages to be sent in the first round are those that are initially in the *outbuf* variables.) The local computation done within a round will be described in a style consistent with typical pseudocode for sequential algorithms. Termination will be implicitly indicated when no more rounds are specified.

In the pseudocode, the local state variables of processor p_i will not be subscripted with i; in discussion and proof, subscripts will be added when necessary to avoid ambiguity.

Comments will begin with //.

In the next sections we will give several examples of describing algorithms in prose, in pseudocode, and as state transitions.

2.2 BROADCAST AND CONVERGECAST ON A SPANNING TREE

We now present several examples to help the reader gain a better understanding of the model, pseudocode, correctness arguments, and complexity measures for distributed algorithms. These algorithms solve basic tasks of collecting and dispersing information and computing spanning trees for the underlying communication network. They serve as important building blocks in many other algorithms.

Broadcast

We start with a simple algorithm for the *(single message) broadcast* problem, assuming a spanning tree of the network is given. A distinguished processor, p_r, has some information, namely, a message $\langle M \rangle$, it wishes to send to all other processors. Copies of the message are to be sent along a tree that is rooted at p_r and spans all the processors in the network. The spanning tree rooted at p_r is maintained in a distributed fashion: Each processor has a distinguished channel that leads to its *parent* in the tree as well as a set of channels that lead to its *children* in the tree.

Here is the prose description of the algorithm. Figure 2.2 shows a sample asynchronous execution of the algorithm; solid lines depict channels in the spanning tree, dashed lines depict channels not in the spanning tree, and shaded nodes indicate processors that have received $\langle M \rangle$ already. The root, p_r, sends the message $\langle M \rangle$ on all the channels leading to its children (see Fig. 2.2(a)). When a processor receives the message $\langle M \rangle$ on the channel from its parent, it sends $\langle M \rangle$ on all the channels leading to its children (see Fig. 2.2(b)).

The pseudocode for this algorithm is in Algorithm 1; there is no pseudocode for a computation step in which no messages are received and no state change is made.

Finally, we describe the algorithm at the level of state transitions: The state of each processor p_i contains:

- A variable $parent_i$, which holds either a processor index or nil

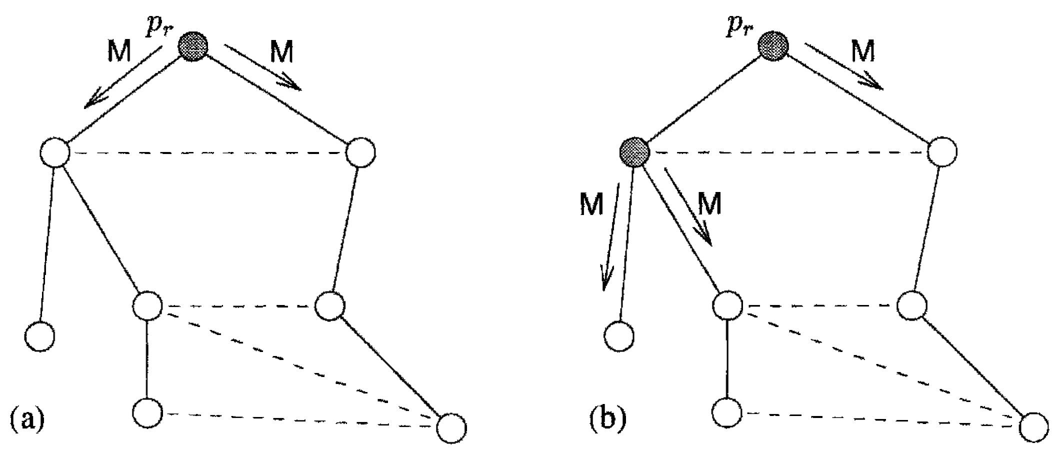

Fig. 2.2 Two steps in an execution of the broadcast algorithm.

- A variable *children*$_i$, which holds a set of processor indices
- A Boolean *terminated*$_i$, which indicates whether p_i is in a terminated state

Initially, the values of the *parent* and *children* variables are such that they form a spanning tree rooted at p_r of the topology graph. Initially, all *terminated* variables are false. Initially, *outbuf*$_r[j]$ holds ⟨M⟩ for each j in *children*$_r$;[2] all other *outbuf* variables are empty. The result of *comp*(i) is that, if ⟨M⟩ is in an *inbuf*$_i[k]$ for some k, then ⟨M⟩ is placed in *outbuf*$_i[j]$, for each j in *children*$_i$, and p_i enters a terminated state by setting *terminated*$_i$ to true. If $i = r$ and *terminated*$_r$ is false, then *terminated*$_r$ is set to true. Otherwise, nothing is done.

Note that this algorithm is correct whether the system is synchronous or asynchronous. Furthermore, as we discuss now, the message and time complexities of the algorithm are the same in both models.

What is the message complexity of the algorithm? Clearly, the message ⟨M⟩ is sent exactly once on each channel that belongs to the spanning tree (from the parent to the child) in both the synchronous and asynchronous cases. That is, the total number of messages sent during the algorithm is exactly the number of edges in the spanning tree rooted at p_r. Recall that a spanning tree of n nodes has exactly $n - 1$ edges; therefore, exactly $n - 1$ messages are sent during the algorithm.

Let us now analyze the time complexity of the algorithm. It is easier to perform this analysis when communication is synchronous and time is measured in rounds.

The following lemma shows that by the end of round t, the message ⟨M⟩ reaches all processors at distance t (or less) from p_r in the spanning tree. This is a simple claim, with a simple proof, but we present it in detail to help the reader gain facility with the model and proofs about distributed algorithms. Later in the book we will leave such simple proofs to the reader.

[2]Here we are using the convention that *inbuf* and *outbuf* variables are indexed by the neighbors' indices instead of by channel labels.

Algorithm 1 Spanning tree broadcast algorithm.

```
Initially $\langle \mathsf{M} \rangle$ is in transit from $p_r$ to all its children in the spanning tree.

Code for $p_r$:
1: upon receiving no message:                    // first computation event by $p_r$
2:     terminate

Code for $p_i$, $0 \le i \le n-1, i \neq r$:
3: upon receiving $\langle \mathsf{M} \rangle$ from parent:
4:     send $\langle \mathsf{M} \rangle$ to all children
5:     terminate
```

Lemma 2.1 *In every admissible execution of the broadcast algorithm in the synchronous model, every processor at distance t from p_r in the spanning tree receives the message $\langle \mathsf{M} \rangle$ in round t.*

Proof. The proof proceeds by induction on the distance t of a processor from p_r.

The basis is $t = 1$. From the description of the algorithm, each child of p_r receives $\langle \mathsf{M} \rangle$ from p_r in the first round.

We now assume that every processor at distance $t - 1 \geq 1$ from p_r in the spanning tree receives the message $\langle \mathsf{M} \rangle$ in round $t - 1$.

We must show that every processor p_i at distance t from p_r in the spanning tree receives $\langle \mathsf{M} \rangle$ in round t. Let p_j be the parent of p_i in the spanning tree. Since p_j is at distance $t - 1$ from p_r, by the inductive hypothesis, p_j receives $\langle \mathsf{M} \rangle$ in round $t - 1$. By the description of the algorithm, p_j then sends $\langle \mathsf{M} \rangle$ to p_i in the next round. □

By Lemma 2.1, the time complexity of the algorithm is d, where d is the depth of the spanning tree. Recall that d is at most $n - 1$, when the spanning tree is a chain. Thus we have:

Theorem 2.2 *There is a synchronous broadcast algorithm with message complexity $n - 1$ and time complexity d, when a rooted spanning tree with depth d is known in advance.*

A similar analysis applies when communication is asynchronous. Once again, the key is to prove that by time t, the message $\langle \mathsf{M} \rangle$ reaches all processors at distance t (or less) from p_r in the spanning tree. This implies that the time complexity of the algorithm is also d when communication is asynchronous. We now analyze this situation more carefully.

Lemma 2.3 *In every admissible execution of the broadcast algorithm in an asynchronous system, every processor at distance t from p_r in the spanning tree receives message $\langle \mathsf{M} \rangle$ by time t.*

Proof. The proof is by induction on the distance t of a processor from p_r.

The basis is $t = 1$. From the description of the algorithm, $\langle\mathsf{M}\rangle$ is initially in transit to each processor p_i at distance 1 from p_r. By the definition of time complexity for the asynchronous model, p_i receives $\langle\mathsf{M}\rangle$ by time 1.

We must show that every processor p_i at distance t from p_r in the spanning tree receives $\langle\mathsf{M}\rangle$ in round t. Let p_j be the parent of p_i in the spanning tree. Since p_j is at distance $t - 1$ from p_r, by the inductive hypothesis, p_j receives $\langle\mathsf{M}\rangle$ by time $t - 1$. By the description of the algorithm, p_j sends $\langle\mathsf{M}\rangle$ to p_i when it receives $\langle\mathsf{M}\rangle$, that is, by time $t - 1$. By the definition of time complexity for the asynchronous model, p_i receives $\langle\mathsf{M}\rangle$ by time t. □

Thus we have:

Theorem 2.4 *There is an asynchronous broadcast algorithm with message complexity $n - 1$ and time complexity d, when a rooted spanning tree with depth d is known in advance.*

Convergecast

The broadcast problem requires one-way communication, from the root, p_r, to all the nodes of the tree. Consider now the complementary problem, called *convergecast*, of collecting information from the nodes of the tree to the root. For simplicity, we consider a specific variant of the problem in which each processor p_i starts with a value x_i and we wish to forward the maximum value among these values to the root p_r. (Exercise 2.3 concerns a general convergecast algorithm that collects all the information in the network.)

Once again, we assume that a spanning tree is maintained in a distributed fashion, as in the broadcast problem. Whereas the broadcast algorithm is initiated by the root, the convergecast algorithm is initiated by the *leaves*. Note that a leaf of the spanning tree can be easily distinguished, because it has no children.

Conceptually, the algorithm is recursive and requires each processor to compute the maximum value in the subtree rooted at it. Starting at the leaves, each processor p_i computes the maximum value in the subtree rooted at it, which we denote by v_i, and sends v_i to its parent. The parent collects these values from all its children, computes the maximum value in its subtree, and sends the maximum value to its parent.

In more detail, the algorithm proceeds as follows. If a node p_i is a leaf, then it starts the algorithm by sending its value x_i to its parent (see Fig. 2.3(a)). A non-leaf node, p_j, with k children, waits to receive messages containing $v_{i_1}, \ldots, v_{i_k}$ from its children $p_{i_1}, \ldots, p_{i_k}$. Then it computes $v_j = \max\{x_j, v_{i_1}, \ldots, v_{i_k}\}$ and sends v_j to its parent. (See Figure 2.3(b).)

The analyses of the message and time complexities of the convergecast algorithm are very much like those of the broadcast algorithm. (Exercise 2.2 indicates how to analyze the time complexity of the convergecast algorithm.)

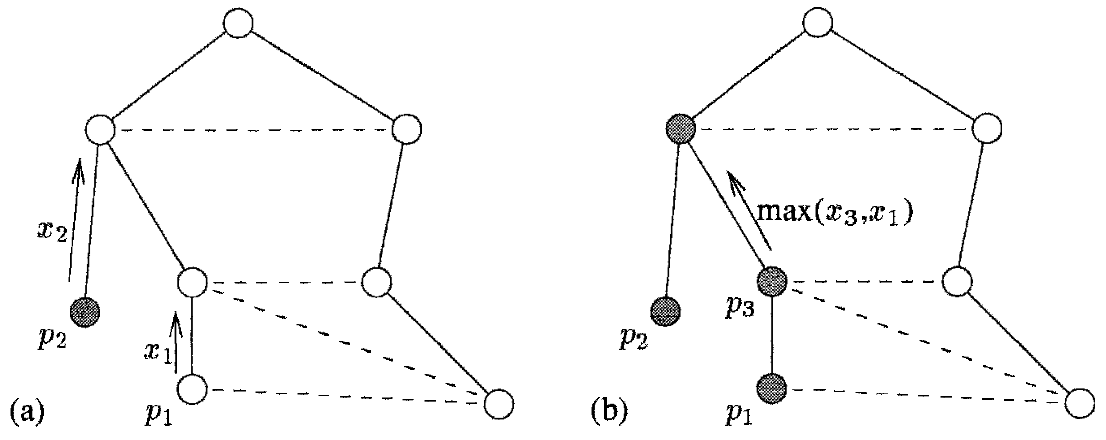

Fig. 2.3 Two steps in an execution of the convergecast algorithm.

Theorem 2.5 *There is an asynchronous convergecast algorithm with message complexity $n-1$ and time complexity d, when a rooted spanning tree with depth d is known in advance.*

It is sometimes useful to combine the broadcast and convergecast algorithms. For instance, the root initiates a request for some information, which is distributed with the broadcast, and then the responses are funneled back to the root with the convergecast.

2.3 FLOODING AND BUILDING A SPANNING TREE

The broadcast and convergecast algorithms presented in Section 2.2 assumed the existence of a spanning tree for the communication network, rooted at a particular processor. Let us now consider the slightly more complicated problem of broadcast without a preexisting spanning tree, starting from a distinguished processor p_r. First we consider an asynchronous system.

The algorithm, called *flooding*, starts from p_r, which sends the message ⟨M⟩ to all its neighbors, that is, on all its communication channels. When processor p_i receives ⟨M⟩ for the first time, from some neighboring processor p_j, it sends ⟨M⟩ to all its neighbors except p_j (see Figure 2.4).

Clearly, a processor will not send ⟨M⟩ more than once on any communication channel. Thus ⟨M⟩ is sent at most twice on each communication channel (once by each processor using this channel); note that there are executions in which the message ⟨M⟩ is sent twice on all communication channels, except those on which ⟨M⟩ is received for the first time (see Exercise 2.6). Thus it is possible that $2m-(n-1)$ messages are sent, where m is the number of communication channels in the system, which can be as high as $\frac{n(n-1)}{2}$.

We will discuss the time complexity of the flooding algorithm shortly.

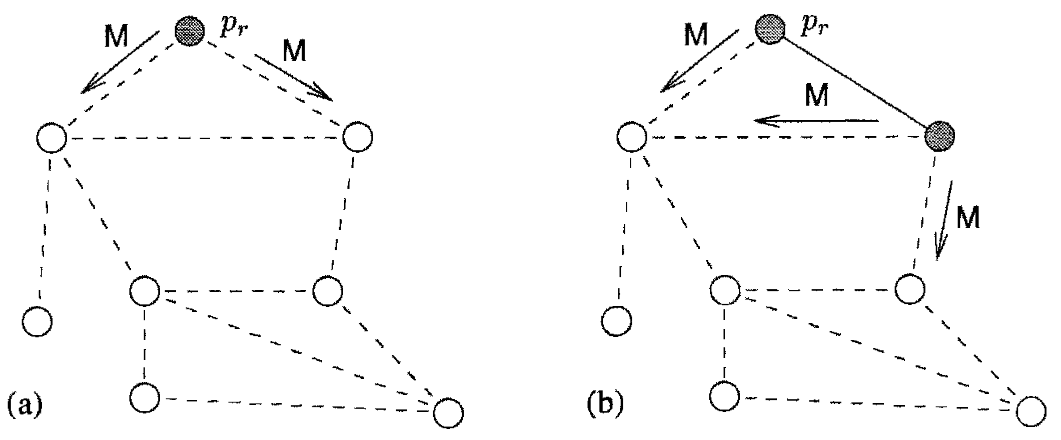

Fig. 2.4 Two steps in an execution of the flooding algorithm; solid lines indicate channels that are in the spanning tree at this point in the execution.

Effectively, the flooding algorithm induces a spanning tree, with the root at p_r, and the parent of a processor p_i being the processor from which p_i received ⟨M⟩ for the first time. It is possible that p_i received ⟨M⟩ concurrently from several processors, because a *comp* event processes all messages that have been delivered since the last *comp* event by that processor; in this case, p_i's parent is chosen arbitrarily among them.

The flooding algorithm can be modified to explicitly construct this spanning tree, as follows: First, p_r sends ⟨M⟩ to all its neighbors. As mentioned above, it is possible that a processor p_i receives ⟨M⟩ for the first time from several processors. When this happens, p_i picks one of the neighboring processors that sent ⟨M⟩ to it, say, p_j, denotes it as its parent and sends a ⟨parent⟩ message to it. To all other processors, and to any other processor from which ⟨M⟩ is received later on, p_i sends an ⟨already⟩ message, indicating that p_i is already in the tree. After sending ⟨M⟩ to all its other neighbors (from which ⟨M⟩ was not previously received), p_i waits for a response from each of them, either a ⟨parent⟩ message or an ⟨already⟩ message. Those who respond with ⟨parent⟩ messages are denoted as p_i's children. Once all recipients of p_i's ⟨M⟩ message have responded, either with ⟨parent⟩ or ⟨already⟩, p_i terminates (see Figure 2.5).

The pseudocode for the modified flooding algorithm is in Algorithm 2.

Lemma 2.6 *In every admissible execution in the asynchronous model, Algorithm 2 constructs a spanning tree of the network rooted at* p_r.

Proof. Inspecting the code reveals two important facts about the algorithm. First, once a processor sets its *parent* variable, it is never changed (and it has only one parent). Second, the set of children of a processor never decreases. Thus, eventually, the graph structure induced by *parent* and *children* variables is static, and the *parent* and *children* variables at different nodes are consistent, that is, if p_j is a child of p_i, then p_i is p_j's parent. We show that the resulting graph, call it G, is a directed spanning tree rooted at p_r.

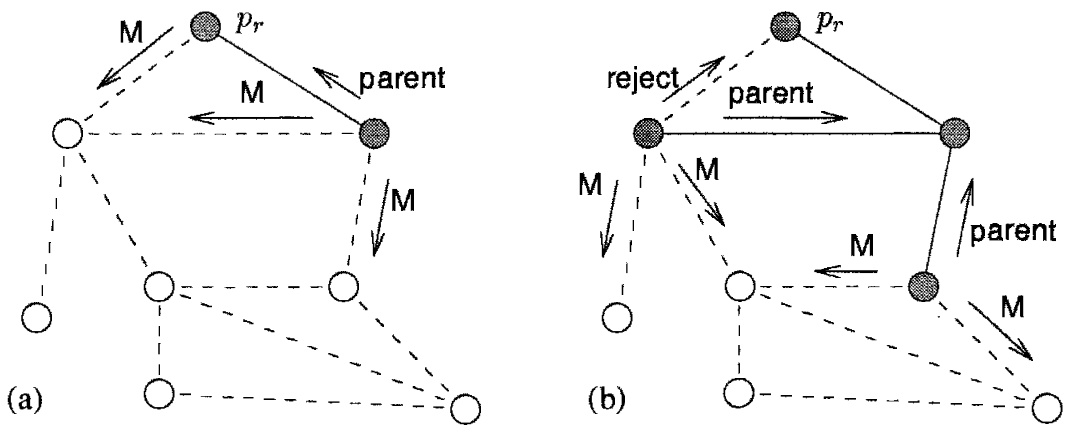

Fig. 2.5 Two steps in the construction of the spanning tree.

Why is every node reachable from the root? Suppose in contradiction some node is not reachable from p_r in G. Since the network is connected, there exist two processors, p_i and p_j, with a channel between them such that p_j is reachable from p_r in G but p_i is not. Exercise 2.4 asks you to verify that a processor is reachable from p_r in G if and only if it ever sets its *parent* variable. Thus p_i's *parent* variable remains nil throughout the execution, and p_j sets its *parent* variable at some point. Thus p_j sends ⟨M⟩ to p_i in Line 9. Since the execution is admissible, the message is eventually received by p_i, causing p_i to set its *parent* variable. This is a contradiction.

Why is there no cycle? Suppose in contradiction there is a cycle, say, $p_{i_1}, p_{i_2}, \ldots, p_{i_k}, p_{i_1}$. Note that if p_i is a child of p_j, then p_i receives ⟨M⟩ for the first time after p_j does. Since each processor is the parent of the next processor in the cycle, that would mean that p_{i_1} receives ⟨M⟩ for the first time before p_{i_1} (itself) does, a contradiction. □

Clearly, the modification to construct a spanning tree increases the message complexity of the flooding algorithm only by a constant multiplicative factor.

In the asynchronous model of communication, it is simple to see that by time t, the message ⟨M⟩ reaches all processors that are at distance t (or less) from p_r. Therefore:

Theorem 2.7 *There is an asynchronous algorithm to find a spanning tree of a network with m edges and diameter D, given a distinguished node, with message complexity $O(m)$ and time complexity $O(D)$.*

The modified flooding algorithm works, unchanged, in the synchronous case. Its analysis is similar to that for the asynchronous case. However, in the synchronous case, unlike the asynchronous, the spanning tree constructed is guaranteed to be a *breadth-first search* (BFS) tree:

Lemma 2.8 *In every admissible execution in the synchronous model, Algorithm 2 constructs a* BFS *tree of the network rooted at p_r.*

Algorithm 2 Modified flooding algorithm to construct a spanning tree: code for processor p_i, $0 \le i \le n-1$.

```
Initially $parent = \bot$, $children = \emptyset$, and $other = \emptyset$.

1:  upon receiving no message:
2:      if $p_i = p_r$ and $parent = \bot$ then          // root has not yet sent ⟨M⟩
3:          send ⟨M⟩ to all neighbors
4:          $parent := p_i$

5:  upon receiving ⟨M⟩ from neighbor $p_j$:
6:      if $parent = \bot$ then                          // $p_i$ has not received ⟨M⟩ before
7:          $parent := p_j$
8:          send ⟨parent⟩ to $p_j$
9:          send ⟨M⟩ to all neighbors except $p_j$
10:     else send ⟨already⟩ to $p_j$

11: upon receiving ⟨parent⟩ from neighbor $p_j$:
12:     add $p_j$ to children
13:     if $children \cup other$ contains all neighbors except parent then
14:         terminate

15: upon receiving ⟨already⟩ from neighbor $p_j$:
16:     add $p_j$ to other
17:     if $children \cup other$ contains all neighbors except parent then
18:         terminate
```

Proof. We show by induction on t that at the beginning of round t, (1) the graph constructed so far according to the *parent* variables is a BFS tree consisting of all nodes at distance at most $t-1$ from p_r, and (2) ⟨M⟩ messages are in transit only from nodes at distance exactly $t-1$ from p_r.

The basis is $t = 1$. Initially, all *parent* variables are nil, and ⟨M⟩ messages are outgoing from p_r and no other node.

Suppose the claim is true for round $t-1 \ge 1$. During round $t-1$, the ⟨M⟩ messages in transit from nodes at distance $t-2$ are received. Any node that receives ⟨M⟩ is at distance $t-1$ or less from p_r. A recipient node with a non-nil *parent* variable, namely, a node at distance $t-2$ or less from p_r, does not change its *parent* variable or send out an ⟨M⟩ message. Every node at distance $t-1$ from p_r receives an ⟨M⟩ message in round $t-1$ and, because its *parent* variable is nil, it sets it to an appropriate parent and sends out an ⟨M⟩ message. Nodes not at distance $t-1$ do not receive an ⟨M⟩ message and thus do not send any. □

Therefore:

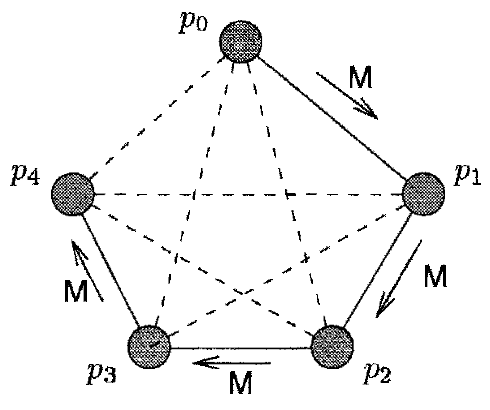

Fig. 2.6 A non-BFS tree.

Theorem 2.9 *There is a synchronous algorithm to find a* BFS *tree of a network with m edges and diameter D, given a distinguished node, with message complexity $O(m)$ and time complexity $O(D)$.*

In an asynchronous system, it is possible that the modified flooding algorithm does not construct a BFS tree. Consider a fully connected network with five nodes, p_0 through p_4, in which p_0 is the root (see Fig. 2.6). Suppose the ⟨M⟩ messages quickly propagate in the order p_0 to p_1, p_1 to p_2, p_2 to p_3, and p_3 to p_4, while the other ⟨M⟩ messages are very slow. The resulting spanning tree is the chain p_0 through p_4, which is not a BFS tree. Furthermore, the spanning tree has depth 4, although the diameter is only 1. Note that the running time of the algorithm is proportional to the diameter, not the number of nodes. Exercise 2.5 asks you to generalize these observations for graphs with n nodes.

The modified flooding algorithm can be combined with the convergecast algorithm described above, to request and collect information. The combined algorithm works in either synchronous or asynchronous systems. However, the time complexity of the combined algorithm is different in the two models; because we do not necessarily get a BFS tree in the asynchronous model, it is possible that the convergecast will be applied on a tree with depth $n - 1$. However, in the synchronous case, the convergecast will always be applied on a tree whose depth is at most the diameter of the network.

2.4 CONSTRUCTING A DEPTH-FIRST SEARCH SPANNING TREE FOR A SPECIFIED ROOT

Another basic algorithm constructs a *depth-first search* (DFS) tree of the communication network, rooted at a particular node. A DFS tree is constructed by adding one node at a time, more gradually than the spanning tree constructed by Algorithm 2, which attempts to add all the nodes at the same level of the tree concurrently.

The pseudocode for depth-first search is in Algorithm 3.

Algorithm 3 Depth-first search spanning tree algorithm for a specified root: code for processor p_i, $0 \le i \le n-1$.

```
Initially parent = ⊥, children = ∅, unexplored = all neighbors of p_i

1:  upon receiving no message:
2:      if $p_i = p_r$ and $parent = \perp$ then                // root wakes up
3:          $parent := p_i$
4:          explore()

5:  upon receiving ⟨M⟩ from $p_j$:
6:      if $parent = \perp$ then                  // $p_i$ has not received ⟨M⟩ before
8:          $parent := p_j$
9:          remove $p_j$ from unexplored
10:         explore()
11:     else
12:         send ⟨already⟩ to $p_j$                            // already in tree
13:         remove $p_j$ from unexplored
14: upon receiving ⟨already⟩ from $p_j$:
15:     explore()

16: upon receiving ⟨parent⟩ from $p_j$:
17:     add $p_j$ to children
18:     explore()

19: procedure explore():
20:     if $unexplored \neq \emptyset$ then
21:         let $p_k$ be a processor in unexplored
22:         remove $p_k$ from unexplored
23:         send ⟨M⟩ to $p_k$
24:     else
25:         if $parent \neq p_i$ then send ⟨parent⟩ to parent
26:         terminate               // DFS subtree rooted at $p_i$ has been built
```

The correctness of Algorithm 3 essentially follows from the correctness of the sequential DFS algorithm, because there is no concurrency in the execution of this algorithm. A careful proof of the next lemma is left as an exercise.

Lemma 2.10 *In every admissible execution in the asynchronous model, Algorithm 3 constructs a* DFS *tree of the network rooted at* p_r.

To calculate the message complexity of the algorithm, note that each processor sends ⟨M⟩ at most once on each of its adjacent edges; also, each processor generates at most one message (either ⟨already⟩ or ⟨parent⟩) in response to receiving ⟨M⟩ on each of its adjacent edges. Therefore, at most $4m$ messages are sent by Algorithm 3.

Showing that the time complexity of the algorithm is $O(m)$ is left as an exercise for the reader. We summarize:

Theorem 2.11 *There is an asynchronous algorithm to find a depth-first search spanning tree of a network with m edges and n nodes, given a distinguished node, with message complexity $O(m)$ and time complexity $O(m)$.*

2.5 CONSTRUCTING A DEPTH-FIRST SEARCH SPANNING TREE WITHOUT A SPECIFIED ROOT

Algorithm 2 and Algorithm 3 build a spanning tree for the communication network, with reasonable message and time complexities. However, both of them require the existence of a distinguished node, from which the construction starts. In this section, we discuss how to build a spanning tree when there is no distinguished node. We assume, however, that the nodes have unique identifiers, which are natural numbers; as we shall see in Section 3.2, this assumption is necessary.

To build a spanning tree, each processor that wakes up spontaneously attempts to build a DFS tree with itself as the root, using a separate copy of Algorithm 3. If two DFS trees try to connect to the same node (not necessarily at the same time), the node will join the DFS tree whose root has the higher identifier.

The pseudocode appears in Algorithm 4. To implement the above idea, each node keeps the maximal identifier it has seen so far in a variable *leader*, which is initialized to a value smaller than any identifier.

When a node wakes up spontaneously, it sets its *leader* to its own identifier and sends a DFS message carrying its identifier. When a node receives a DFS message with identifier y, it compares y and *leader*. If $y >$ *leader*, then this might be the DFS of the processor with maximal identifier; in this case, the node changes *leader* to be y, sets its *parent* variable to be the node from which this message was received, and continues the DFS with identifier y. If $y =$ *leader*, then the node already belongs to this spanning tree. If $y <$ *leader*, then this DFS belongs to a node whose identifier is smaller than the maximal identifier seen so far; in this case, no message is sent, which stalls the DFS tree construction with identifier y. Eventually, a DFS message carrying the identifier *leader* (or a larger identifier) will arrive at the node with identifier y, and connect it to its tree.

Only the root of the spanning tree constructed explicitly terminates; other nodes do not terminate and keep waiting for messages. It is possible to modify the algorithm so that the root sends a termination message using Algorithm 1.

Proving correctness of the algorithm is more involved than previous algorithms in this chapter; we only outline the arguments here. Consider the nodes that wake up spontaneously, and let p_m be the node with the maximal identifier among them; let m be p_m's identifier.

First observe that ⟨leader⟩ messages with leader id m are never dropped because of discovering a larger leader id, by definition of m.

Algorithm 4 Spanning tree construction: code for processor p_i, $0 \leq i \leq n-1$.

```
Initially $parent = \bot$, $leader = -1$, $children = \emptyset$, $unexplored$ = all neighbors of $p_i$

1:  upon receiving no message:
2:      if $parent = \bot$ then                                  // wake up spontaneously
3:          $leader := id$
4:          $parent := p_i$
5:          explore()

6:  upon receiving ⟨leader,$new\text{-}id$⟩ from $p_j$:
7:      if $leader < new\text{-}id$ then                           // switch to new tree
8:          $leader := new\text{-}id$
9:          $parent := p_j$
10:         $children := \emptyset$
11:         $unexplored$ := all neighbors of $p_i$ except $p_j$
12:         explore()
13:     else if $leader = new\text{-}id$ then
14:         send ⟨already,$leader$⟩ to $p_j$                        // already in same tree
                  // otherwise, $leader > new\text{-}id$ and the DFS for $new\text{-}id$ is stalled

15: upon receiving ⟨already,$new\text{-}id$⟩ from $p_j$:
16:     if $new\text{-}id = leader$ then explore()

17: upon receiving ⟨parent,$new\text{-}id$⟩ from $p_j$:
18:     if $new\text{-}id = leader$ then                         // otherwise ignore message
19:         add $p_j$ to $children$
20:         explore()

21: procedure explore():
22:     if $unexplored \neq \emptyset$ then
23:         let $p_k$ be a processor in $unexplored$
24:         remove $p_k$ from $unexplored$
25:         send ⟨leader,$leader$⟩ to $p_k$
26:     else
27:         if $parent \neq p_i$ then send ⟨parent,$leader$⟩ to $parent$
28:         else terminate as root of spanning tree
```

Second, ⟨already⟩ messages with leader id m are never dropped because they have the wrong leader id. Why? Suppose p_i receives an ⟨already⟩ message from p_j with leader id m. The reason p_j sent this message to p_i is that it received a ⟨leader⟩ message from p_i with leader id m. Once p_i sets its leader id to m, it never resets it, because m is the largest leader id in the system. Thus when p_i receives p_j's ⟨already⟩

message with leader id m, p_i still has its leader id as m, the message is accepted, and the exploration proceeds.

Third, ⟨parent⟩ messages with leader id m are never dropped because they have the wrong leader id. The argument is the same as for ⟨already⟩ messages.

Finally, messages with leader id m are never dropped because the recipient has terminated. Suppose in contradiction that some p_i has terminated before receiving a message with leader id m. Then p_i thinks it is the leader, but its id, say i, is less than m. The copy of Algorithm 3 with leader id i must have reached every node in the graph, including p_m. But p_m would not have responded to the leader message, so this copy of Algorithm 3 could not have completed, a contradiction.

Thus the copy of Algorithm 3 for leader id m completes, and correctness of Algorithm 3 implies correctness of Algorithm 4.

A simple analysis of the algorithm uses the fact that, in the worst case, each processor tries to construct a DFS tree. Therefore, the message complexity of Algorithm 4 is at most n times the message complexity of Algorithm 3, that is, $O(nm)$. The time complexity is similar to the time complexity of Algorithm 3, that is, $O(m)$.

Theorem 2.12 *Algorithm 4 finds a spanning tree of a network with m edges and n nodes, with message complexity $O(n \cdot m)$ and time complexity $O(m)$.*

Exercises

2.1 Code one of the simple algorithms in state transitions.

2.2 Analyze the time complexity of the convergecast algorithm of Section 2.2 when communication is synchronous and when communication is asynchronous.

Hint: For the synchronous case, prove that during round $t + 1$, a processor at height t sends a message to its parent. For the asynchronous case, prove that by time t, a processor at height t has sent a message to its parent.

2.3 Generalize the convergecast algorithm of Section 2.2 to collect all the information. That is, when the algorithm terminates, the root should have the input values of all the processors. Analyze the bit complexity, that is, the total number of bits that are sent over the communication channels.

2.4 Prove the claim used in the proof of Lemma 2.6 that a processor is reachable from p_r in G if and only if it ever sets its *parent* variable.

2.5 Describe an execution of the modified flooding algorithm (Algorithm 2) in an asynchronous system with n nodes that does not construct a BFS tree.

2.6 Describe an execution of Algorithm 2 in some asynchronous system, where the message is sent twice on communication channels that do not connect a parent and its children in the spanning tree.

2.7 Perform a precise analysis of the time complexity of the modified flooding algorithm (Algorithm 2), for the synchronous and the asynchronous models.

2.8 Explain how to eliminate the ⟨*already*⟩ messages from the modified flooding algorithm (Algorithm 2) in the synchronous case and still have a correct algorithm. What is the message complexity of the resulting algorithm?

2.9 Do the broadcast and convergecast algorithms rely on knowledge of the number of nodes in the system?

2.10 Modify Algorithm 3 so that it handles correctly the case where the distinguished node has no neighbors.

2.11 Modify Algorithm 3 so that all nodes terminate.

2.12 Prove that Algorithm 3 constructs a DFS tree of the network rooted at p_r.

2.13 Prove that the time complexity of Algorithm 3 is $O(m)$.

2.14 Modify Algorithm 3 so it constructs a DFS numbering of the nodes, indicating the order in which the message ⟨M⟩ arrives at the nodes.

2.15 Modify Algorithm 3 to obtain an algorithm that constructs a DFS tree with $O(n)$ time complexity.

Hint: When a node receives the message ⟨M⟩ for the first time, it notifies all its neighbors but passes the message only to one of them.

2.16 Prove Theorem 2.12.

2.17 Show that in Algorithm 4 if the *leader* variable is not included in the ⟨parent⟩ message and the test in Line 18 is not performed, then the algorithm is incorrect.

Chapter Notes

The first part of this chapter introduced our formal model of a distributed message-passing system; this model is closely based on that used by Attiya, Dwork, Lynch, and Stockmeyer [27], although many papers in the literature on distributed algorithms have used similar models.

Modeling each processor in a distributed algorithm as a state machine is an idea that goes back at least to Lynch and Fischer [176]. Two early papers that explicitly represent an execution of a distributed system as a sequence of state transitions are by Owicki and Lamport [204] and by Lynch and Fischer [176]. The same idea is, more implicitly, present in the paper by Owicki and Gries [203].

A number of researchers (e.g., Fischer, Lynch, and Paterson [110]) have used the term "admissible" to distinguish those executions that satisfy additional constraints from all executions: That is, the term "execution" refers to sequences that satisfy