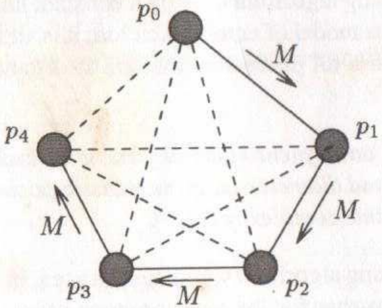

Figure 2.6: A non-BFS tree.

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, in order 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; since 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 is constructing 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 2.2, which attempts to add all the nodes at the same level of the tree concurrently.

The pseudocode for depth-first search is in Algorithm 2.3. To slightly simplify the code, the algorithm assumes that the root, p_r, has neighbors and is not the only node in the system; simple modifications to Lines 4–6 can be made to remove this assumption.

The proof of the next lemma is similar to the proof of the corresponding lemma for Algorithm 2.2, and is left as an exercise.

Algorithm 2.3 Depth-first search spanning tree algorithm for a specified root: code for processor p_i, $0 \le i \le n-1$.

```
Initially parent equals nil, children is empty
and unexplored includes all the neighbors of p_i

1:  upon receiving no message:
2:      if i = r and parent is nil then
3:          parent := i
4:          let p_j be a processor in unexplored
5:          remove p_j from unexplored
6:          send M to p_j

7:  upon receiving M from neighbor p_j:
8:      if parent is nil then                       // p_i has not received M before
9:          parent := j
10:         remove p_j from unexplored
11:         if unexplored ≠ ∅ then
12:             let p_k be a processor in unexplored
13:             remove p_k from unexplored
14:             send M to p_k
15:         else send ⟨parent⟩ to parent
16:     else send ⟨reject⟩ to p_j

17: upon receiving ⟨parent⟩ or ⟨reject⟩ from neighbor p_j:
18:     if received ⟨parent⟩ then add j to children
19:     if unexplored = ∅ then
20:         if parent ≠ i then send ⟨parent⟩ to parent
21:         terminate                               // DFS sub-tree rooted at p_i has been built
22:     else
23:         let p_k be a processor in unexplored
24:         remove p_k from unexplored
25:         send M to p_k
```

Lemma 2.10 *In every admissible execution in the asynchronous model, Algorithm 2.3 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 ⟨*reject*⟩ or ⟨*parent*⟩) in response to receiving M on each of its adjacent edges. Therefore, at most $4m$ messages are sent by Algorithm 2.3. 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 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 2.2 and Algorithm 2.3 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, the nodes have unique identifiers, which are natural numbers; as we shall see later, in Section 3.2, this assumption is necessary.

To build a spanning tree, each processor which wakes up spontaneously, attempts to build a DFS tree with itself as the root. 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 2.4. To implement the above idea, each node keeps the maximal identifier it has seen so far in a variable *leader*; initially, $leader_i$ contains p_i's own identifier, if it wakes up spontaneously, and 0 otherwise.

When a node wakes up spontaneously, it sends a DFS message carrying its identifier. When a node receives a DFS message of some processor 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*, sets its *parent* variable and continues the DFS with identifier y. 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 back, 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. Otherwise, $y =$ *leader*, and the node already belongs to this spanning 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 2.1.

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 id_m be p_m's identifier.

Messages carrying id_m are always propagated; moreover, once a node handles the messages carrying id_m, all messages originating at nodes other than p_m are ignored. Since messages carrying id_m are handled as in the DFS algorithm (Algorithm 2.3), *parent* and *children* variables are set correctly. Thus, we can prove:

Algorithm 2.4 Spanning tree construction: code for processor p_i, $0 \le i \le n-1$.

Initially *parent* equals nil, *leader* is 0, *children* is empty, and *unexplored* includes all the neighbors of p_i

```
upon receiving no message:
    if parent = nil then                                  // wake up spontaneously
        leader := id ; parent := i
        let p_j be a processor in unexplored
        remove p_j from unexplored
        send ⟨leader⟩ to p_j

upon receiving ⟨new-id⟩ from neighbor p_j:
    if leader < new-id then                               // switch to new tree
        leader := new-id ; parent := j
        unexplored := all the neighbors of p_i except p_j  // reset unexplored
        if unexplored ≠ ∅ then
            let p_k be a processor in unexplored
            remove p_k from unexplored
            send ⟨leader⟩ to p_k
        else send ⟨parent⟩ to parent
    else if leader = new-id then send ⟨already⟩ to p_j  // already in same tree
        // otherwise, leader > new-id and the DFS for new-id is stalled

upon receiving ⟨parent⟩ or ⟨already⟩ from neighbor p_j:
    if received ⟨parent⟩ then add j to children
    if unexplored = ∅ then
        if parent ≠ i then send ⟨parent⟩ to parent
        else terminate as root of the spanning tree
    else
        let p_j be a processor in unexplored
        remove p_j from unexplored
        send leader to p_j
```

Lemma 2.12 *Let p_m be the node with maximal identifier. In every admissible execution in the asynchronous model, Algorithm 2.4 constructs a* DFS *tree of the network rooted at p_m.*

This implies that a DFS tree rooted at p_m is constructed when the algorithm terminates at p_m; p_m can broadcast a termination message on this tree to let all nodes know (using Algorithm 2.1).

Even without the use of termination messages, the messages of all nodes other than p_m are eventually stalled by reaching a node holding a larger identifier (e.g.,