10

Graph Algorithms

10.1. Introduction

The topology of a distributed system is represented by a graph where the nodes represent processes, and the links represent communication channels. Accordingly, distributed algorithms for various graph theoretic problems have numerous applications in communication and networking. Here are some motivating examples.

Our first example deals with routing in a communication network. When a message is sent from node $i$ to a non-neighboring node $j$, the intermediate nodes route the message based on the information stored in the local routing table, so that eventually the message reaches the destination node. This is called destination-based routing. An important problem is to compute these routing tables and maintain them, so that messages reach their destinations in smallest number of hops or with minimum delay. The minimum hop routing is equivalent to computing the shortest path between a pair of nodes using locally available information.

Our second example focuses on the amount of space required to store a routing table in a network. Without any optimization, the space requirement is $O(N)$, where $N$ is the number of nodes. But with the explosive growth of the Internet, $N$ is increasing at a steep rate, and the space requirement of the routing table is a matter of concern. This leads to the question: Can we reduce the size of the routing table? Given the value of $N$, what is the smallest amount of information that each individual node must store in their routing tables, so that every message eventually reaches its final destination?

Our third example visits the problem of broadcasting in a network whose topology is represented by a connected graph. Uncontrolled transmission of messages leads to flooding, which wastes communication bandwidth. To conserve bandwidth, messages should be transmitted along the edges of a spanning tree of the graph. How to compute the spanning tree of a graph? How to maintain a spanning tree when the topology changes?

Our fourth example addresses the classical problem of computing of maximum flow between a pair of nodes in a connected network. Here the flow represents the movement of a certain commodity, like a bunch of packets, from one node to another. Each edge of the network has a certain capacity that defines the upper limit of the flow through that edge.
This problem, known as the maxflow problem, is of fundamental importance in networking and operations research.

An important issue in distributed algorithms for graphs is that of static vs. dynamic topology. The topology is called static when it does not change. A topology that is not static is called dynamic – it is the result of spontaneous addition and deletion of nodes and edges. Clearly, algorithms for dynamic topologies are more robust than those that work on static topologies only. In this chapter, we will study distributed algorithms for solving a few graph theoretic problems.

10.2. Routing

Routing is a fundamental problem in networks. The major issue is to discover and maintain an acyclic path from the source of a message to its destination. The relevant information is stored in a routing table that is updated as and when the topology changes. Whenever a packet arrives at a node that is not its destination, the routing table determines how to route the packet so that it reaches its destination. A path can have many attributes. In some cases, the path consists of a minimum number of hops. Sometimes, there is a cost associated with a link, so routing via the path of least cost is required. This is known as shortest path routing. As an example, in multimedia applications, delay is a major factor. Due to congestion, the delay across certain links can be large, so sometimes routing via a path of minimum delay becomes important. In this section, we will discuss algorithms related to routing.

10.2.1. Computation of Shortest Path

Let \( G = (V,E) \), an undirected graph where \( V \) represents a set of \( N \) nodes \( 0..N-1 \) representing processes, and \( E \) represents a set of edges representing links. The topology is static. Each edge \((i,j)\) has a weight \( w(i,j) \) that represents the cost of communication through that edge. A simple path between a source and a destination node is called a shortest path, when the sum of all the weights in the path between them is the smallest of all such paths. The weight \( w(i,j) \) is application dependent. For computing the path with minimum number of hops, \( w(i,j) = 1 \). However, when \( w(i,j) \) denotes the delay in message propagation through link (which depends on the degree of congestion), the shortest path computation can be regarded as the fastest path computation. To keep things simple, assume that \( w(i,j) \geq 0 \). Our goal is to present an asynchronous message passing algorithm using which each node \( i \) \((1 \leq i < N-1)\) can compute the shortest path to a designated node \( 0 \) (called the source or the root node) from itself.
The algorithm to be presented here is due to Chandy and Misra [CM82], and designed to work with a single initiator node 0. This is a refinement of the well-known Bellman-Ford algorithm used to compute routes in the ARPANET during 1969-1979. Each node knows the weights of all edges incident on it. Let $D(i)$ denote the best knowledge of node $i$ about its distance to node 0 via the shortest path. Clearly $D(0) = 0$. Initially $\forall i: i > 0$: $D(i) = \infty$. As the computation progresses, the value of $D(i)$ approaches the true value of its shortest distance from node 0. The initiator node 0 initiates the algorithm by sending out $(D(0) + w(0,j), 0)$ to each neighbor $j$. Every node $j$, after receiving a message $(D(i) + w(i,j), i)$ from a neighbor $i$, does the following. If $D(i) + w(i,j) < D(j)$, then $j$ assigns $D(i) + w(i,j)$ to $D(j)$, recognizes $i$ as its parent, sends a message reporting the new shortest distance to its non-parent neighbors, and sends an acknowledgement to $i$. If however $D(i) + w(i,j) \geq D(j)$ then no new shortest path is discovered, and $j$ only sends an acknowledgement to $i$. When the computation terminates, the path $j, \text{parent}(j), \text{parent}(\text{parent}(j)) \ldots 0$ defines the shortest path from $j$ to 0. Each node has the following three variables:

1. $D$, the current shortest distance of node 0 to itself. Initially $D(0) = 0$, $D(i: i > 0) = \infty$
2. A node called parent: Initially parent($j$) = $j$
3. A variable deficit, representing the number of unacknowledged messages. Initially deficit = 0.

The computation terminates, when the values of deficit at every node is 0. At this time, the value of $D$ at each node represents the distance of the shortest path between 0 and that node. The edge connecting the parent node can be used to route a message towards 0 via the shortest path. The program for process $i$ ($i > 0$) is as follows:

```plaintext
program shortest path {for process i > 0};
define D, S : distance;
{S denotes the shortest distance received through a message}
parent : process
deficit : integer;
N: set of neighbors of process i;
initially D= \infty, parent = i, deficit = 0

{for process 0}
send (w(0,i), 0) to each neighbor i;
```
deficit := |N(0)|;
\textbf{do} \text{ack} \quad \text{deficit} := \text{deficit} - 1 \textbf{od};
\{\text{deficit} = 0 \text{ signals termination}\}

\{\text{for process } i > 0\}\textbf{do} 

\text{message} = (S,k) \quad \text{if parent} \neq k \text{ or } i \neq \text{send ack to parent}\;
\text{parent} := k; \text{D} := S;
\text{send} (D + w(i,j), i) \text{ to each neighbor } j \neq \text{parent};
\text{deficit} := \text{deficit} + |N(i)| - 1

\text{message} (S,k) \quad \text{if message} \geq D \quad \text{send ack to sender}
\text{ack} \quad \text{deficit} := \text{deficit} - 1
\text{deficit} = 0 \quad \text{if parent} \neq i \quad \text{send ack to the parent}

\textbf{od}

\textbf{Note.} The acknowledgments are technically unnecessary, but can be used by the initiator to detect termination and initiate a subsequent round of computation. The original algorithm due to Bellman and Ford did not use acknowledgments. Chandy and Misra later added acknowledgements in the style of [DS80] to help with termination detection. They also proposed a modification that allowed the edge weights to be negative, thus adding a new twist to the problem.

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{graph.png}
\caption{A graph with each edge labeled by its weight. For each node \(i: i > 0\) the directed edge points to its parent node.}
\end{figure}

The termination of the shortest path algorithm follows from the Dijkstra-Scholten termination detection algorithm presented in the previous chapter. Here, we will only examine why the algorithm converges to the desired goal state.
Fig 10.1 contains a sample graph. As the computation progresses, each node discovers shorter paths between the initiator and itself. If there is a cyclic path (like 0 1 2 3 4 2) then the length of a path containing the cycle must be greater than an acyclic sub-path (like 0 1 2) of it. Therefore, traversing a cyclic path does not make the distance any shorter. Counting acyclic paths only, between any two nodes there are only a finite number of such paths. Therefore, $D$ can decrease a finite number of times - reaching the lowest possible value when the algorithm terminates.

**Lemma 10.1.** When the algorithm terminates, let $k = \text{parent} (i)$. If $D(k)$ is the distance of the shortest path from $k$ to 0, then $D(i) = D(k) + w(k, i)$ is the distance of the shortest path from $i$ to 0 and the shortest path includes $k$.

**Proof.** Suppose this is false. Then the shortest path from 0 to $i$ is via a neighbor $j$ of $i$, where $j \neq k$. If $D(j) + w(j, i) < D(k) + w(k, i)$ then at some point, $i$ must have received a message from $j$, and since $D(i) > D(j) + w(j, i)$, $i$ would have set $D(i)$ to $D(j) + w(j, i)$ and $\text{parent}(i)$ to $j$. Even if the message from $k$ was received later, $i$ would not have changed $D(i)$ any further, since $D(i) < D(k) + w(k, i)$ will hold. This contradicts the statement of the lemma. 

**Theorem 10.2.** $D(i)$ computed by the Chandy-Misra algorithm is the distance of the shortest path from $i$ to 0.

**Proof (by induction)** The basis is $D(0) = 0$. The inductive step follows from Lemma 10.1.

When the weight of each edge is 1, the shortest path computation leads to the formation of a *Breadth First Search* (BFS) spanning tree. Every node with shortest distance $D$ from the root node 0 has a parent whose distance from the root node is $D-1$. The set of all nodes and the edges joining the each node with its parent defines the BFS spanning tree. Furthermore, when the acknowledgements are taken out, the algorithm essentially reduces to the classical Bellman-Ford algorithm. To cope with a change in topology, the initiator has to repeatedly run the algorithm – the system will be reinitialized before each new run begins.

The worst-case message complexity of the shortest path algorithm is exponential in terms of the number of nodes. This proof of this is left as an exercise to the reader.
10.2.2. Link State Routing

This is a popular form of shortest path routing used in networks, and it adapts to changes in the network topology. Each node $i$ periodically broadcasts the weights of all edges $(i,j)$ incident on it (this is the link state) to all its neighbors – this helps each node to eventually compute the topology of the network, and independently determine the correct route to any destination node. The technique is known as reliable flooding.

From time to time, links and nodes may go down and come back, and new link states must replace the old ones. The links may not be FIFO, so to distinguish between the old and the new link states, each link state contains a sequence number $seq$. The sequence numbers increase monotonically, so a new link-state packet with a larger sequence number reflects that it is more recent, and replaces an old packet with a smaller sequence number. Only the most recent link state packet is forwarded to the neighbors.

The protocol is as follows:

```plaintext
program link state {for node i}

type LSP: record
  sender: process
  state: set of (j, w(i,j) for each edge (i,j))
  seq : integer
end

define L: LSP;
s: integer (initially 0);
local: array[0..N-1] of LSP
{the $k^{th}$ element is the LSP from node $k$, and is initialized to $k, \emptyset, 0$}

do topology change detected []
  local[i] := (i, state, s);
  send local[i];
  s := s +1;

L received []
  if L.sender = i [] discard L
  L.sender \neq i [] L.seq > local[sender]. seq []
     local[sender] := L; send L
  fi

od
```
The sequence number increases monotonically. The algorithm guarantees that eventually every node receives the local state of every other node, and stores them in the array \textit{local}. From this, each node can independently compute the topology of the graph, and apply a sequential algorithm for computing the shortest path to any node of the graph.

When failures are not taken into consideration the correctness follows trivially. The total number of link state packets is \(N \times E\).

The failure (or temporary unavailability) of links and nodes can make the algorithm more complicated. Link failures are detected by the nodes on which the link is incident – these nodes appropriately update their local states before the next broadcast. The failure of a node is detected by a neighbor, which marks the link to the faulty node as unavailable. When a node goes down, the LSP’s stored in it are lost, so it has to reconstruct the topology from the newer packets. After the node resumes operation, it reinitializes its sequence \(s\) to 0. As a consequence, the newer packets from \(i\) will be discarded in favor of older packets transmitted in the past, until the current value of \(s\) exceeds the last value of \(s\) in the LSP’s transmitted before node \(i\) went down. Since such an anomalous behavior can continue for a long time, each link state packet also contains a time-to-live field (TTL), which is an \textit{estimate} of the time after which a packet should be considered stale, and discarded. Every node decrements the TTL field of all its LSP’s a steady rate\(^1\). Furthermore, every time a node forwards a stored link state packet, it decrements its TTL. When the TTL of a packet becomes 0, the packet is discarded.

With a 64-bit field to represent \textit{seq}, the unbounded growth of the sequence numbers does not pose a problem in real life since no node is reliable enough to outlive the overflow a 64-bit counter at a rate permitted by technologies in the foreseeable future. Of course transient failures can corrupt \textit{seq} in an unpredictable manner and challenge the protocol. Corrupt LSP entries are eventually flushed out using the TTL field.

\subsection*{10.2.3 Interval Routing}

Consider a connected network of \(N\) nodes. The conventional routing table used to direct a message from one node to another has \(N\) entries, one for each destination node. The entry is of the type (destination, port number) that identifies the port number to which the packet should be forwarded to reach the specified destination. As the size of the network grows, the size of the routing tables also grows. Can we do something to reduce the growth of the routing tables?

\footnote{1 The clocks are assumed to be approximately synchronized.}
Interval routing is a scheme for reducing the size of routing tables. It was first proposed by Santoro and Khatib [SK85] for tree topologies only. To motivate the discussion on interval routing, consider the network shown in Fig. 10.2. Each node has two ports: port 0 is connected to the node with a higher id, and port 1 is connected with the node of lower id.

![Network Diagram](image)

<table>
<thead>
<tr>
<th>condition</th>
<th>port number</th>
</tr>
</thead>
<tbody>
<tr>
<td>destination &gt; id</td>
<td>0</td>
</tr>
<tr>
<td>destination &lt; id</td>
<td>1</td>
</tr>
<tr>
<td>destination = id</td>
<td>(local delivery)</td>
</tr>
</tbody>
</table>

Fig 10.2. An illustration of compact routing.

The routing policy is described at the bottom.

To take a routing decision, a process simply compares its own id with the id of the destination node. If the destination id is larger than its own id, then the message is routed through port 0. If the destination id is smaller then the message is forwarded through port 1. If the two id’s are equal, then the message is meant for local delivery. Clearly, the number of entries in the routing table does not change with the size of the network. This is an example of a compact routing table.

Interval routing uses this concept. To make a decision about forwarding a message, a node finds out, to which one of a set of predefined intervals the destination id belongs. For a set of \(N\) nodes \(0 \ldots N-1\), the interval \([p,q)\) between \(p\) and \(q\) \((p, q < N)\) is defined as follows:

\[
\text{if } p < q \text{ then } [p,q) = p, p+1, p+2, ..., q-2, q-1 \\
\text{if } p \geq q \text{ then } [p,q) = p, p+1, p+2, ..., N-1, N, 0, 1, ..., q-2, q-1 
\]

As an example, if \(N = 8\), then \([5,5) = 5, 6, 7, 0, 1, 2, 3, 4\). To determine the intervals, the port numbers are read in the anticlockwise order. If port \(q\) is next to port \(p\) in the anticlockwise order, then all messages meant for a process in the interval \([p,q)\) will be sent...
along port \( p \). Fig. 10.3 shows three ports of a node in network of 8 nodes 0,1,2, ...,7 and illustrates how data will be forwarded.

![Diagram showing three ports of a node in a network with destinations 1,2, 3,4, 5,6,7,0](image)

Fig. 10.3. An example of message forwarding in interval routing. There are 8 nodes numbered 0 through 7.

We now consider a tree, and illustrate a scheme for labeling the nodes of the tree, so that messages can be routed from any source node to any destination node using interval routing scheme. Fig. 10.4 shows a labeling scheme. The label of each node is written within the circle representing that node. For each edge that connects a node to a neighbor, there is a port number – these port numbers will be used to define the intervals.

![Diagram showing a labeled tree network with intervals](image)

Fig. 10.4. An interval routing scheme for a tree network
In Fig. 10.4, node 1 will send a message to node 5 through port 3, since the destination 5 is in the interval [3,7). However, if node 1 wants to send the message to node 9, then it has to route it through port 7, since the destination 9 belongs to the interval [7,2).

A labeling scheme that works for many tree networks is as follows:

1. Label the root as node 0.
2. Do a pre-order traversal of the tree, and label the successive nodes in ascending order starting from 1.
3. For each node, label the port towards a child by the node number of the child. Then label the port towards the parent by $L(i) + T(i) + 1 \mod N$, where

- $L(i)$ is the label of the node $i$,
- $T(i)$ is the number of nodes in the subtree under node $i$ (excluding $i$), and
- $N$ is the total number nodes.

Informally, the labeling scheme is justified as follows: if $p$ is the label of the first child of a node with label $i$, then the label $q$ of the next child visited by a post-order traversal will be $p + T(i) + 1$. Then the interval $[p, q)$ will contain the labels of all the nodes in the subtree under $p$. As a consequence of the post-order traversal, the interval $[L(i) + T(i) + 1 \mod N, p)$ includes every destination node that does not belong to the subtree under node $i$.

A simple method of interval routing on non-tree topologies is to construct a spanning tree of the graph, and then apply the routing on the spanning tree. However, this method does not utilize the non-tree edges. Van Leeewen and Tan [LT87] demonstrated how interval routing schemes could be extended to networks of non-tree topologies by making use of the non-tree edges. Fig. 10.5 illustrates an example.

![Intervals Routing on a Ring](image_url)

Fig. 10.5. Interval routing on a ring. The scheme is not optimal. To make it optimal, label the ports of node $i$ with $i+1 \mod 8$ and $i+4 \mod 8$. 


Not all labeling leads to optimal (i.e. the shortest) routes towards the destination. The question does not apply to trees, since there is exactly one path between any pair of nodes.

An interval labeling scheme is called linear, if every destination can be mapped to a single interval of the type \([p,q)\). The example scheme in Fig. 10.5 is linear. There are graphs on which no linear interval routing is possible.

While compactness of the routing table is the motivation behind interval routing, a major obstacle is its poor ability of adaptation to changes in the topology. Every time a new node is added to a network, or an existing node is removed from the network, in general all node labels and port labels have to be recomputed. The change is not an incremental change that is required for conventional routing tables.

Prefix routing is an alternative technique for routing using compact routing tables. It overcomes the poor adaptivity of the classical interval routing scheme to changes in the network topology. Fig. 10.6 illustrates the idea behind prefix routing.

In prefix routing, each label is a string from an alphabet \(\emptyset\). The labeling rule is as follows:

1. Label the root by \(\emptyset\) that represents the empty string. By definition, \(\emptyset\) is the prefix of every string consisting of symbols from \(\emptyset\).
2. If the parent has a label $L$, then label each child by $L.a$, where $a$ is an element of $S$ and $L.a$ is not the label of any other child of that parent.

3. Label every port from a parent to its child by the label of the child, and every port from a child to its parent by the empty string $\square$.

Let $X$ be the label of a destination node towards which a message is to be routed. When a node with label $Y$ receives this message, it makes the routing decision as follows:

$$\begin{align*}
\text{if } X &= Y \quad \text{deliver the message locally} \\
X &\neq Y \quad \text{find the port with the longest prefix of } X \text{ as its label;} \\
&\text{forward the message towards that port} \\
fi
\end{align*}$$

When a new node is added to a tree, new labels are assigned to that node and the edges without modifying the labels of any of the existing nodes and ports. The scheme can be easily generalized to non-tree topologies.

### 10.3 Graph Traversal

Given an undirected connected graph $G=(V, E)$ where $V$ is a set of nodes and $E$ is a set of edges, a traversal algorithm of $G$ enables a visitor sent out by any one of the nodes to visit every node of the graph before returning to the initiator. The visitor here can be a token or a message or a query. Since no node is assumed to possess global knowledge about the topology of $G$, the routing decision at each node is completely local. Traversal algorithms have numerous applications, starting from simple broadcast / multicast, to information retrieval in database, web crawling, global state collection, network routing and AI related problems.

Graph traversal on specific topologies like a ring or a clique is fairly straightforward. We will focus on traversal on general graphs only. In addition to the correctness of the method, the intellectually challenging part is to accomplish the task in shortest time or using fewest messages.

Traversal of a general graph is done via a spanning tree, and is classified by the order in which the nodes are visited. The two important traversal orders are the depth-first-search (DFS) order and the breadth-first-search (BFS) order. Chany-Misra’s shortest path algorithm generates a BFS tree when the edge weights are 1. Some of these algorithms have been discovered and rediscovered with minor variations. We will study here a couple of algorithms for the construction of spanning trees.
10.3.1 Spanning tree construction

Let $G = (V, E)$ represent the topology is represented by an undirected connected graph. A spanning tree of $G$ is a maximal connected subgraph $T = (V, E')$, $E' \subseteq E$, such that if one more edge from $(E \setminus E')$ is added to $T$, then the subgraph ceases to be a tree.

A rooted spanning tree is one in which a specific node is designated as the root. The root acts as a coordinator or initiator of activities. The following asynchronous message-passing algorithm for constructing a rooted spanning tree was proposed by Chang [C82]. It belongs to the class of probe-echo algorithms. The computation is initiated by the root, and is similar to the termination-detection algorithm proposed in [DS80] (which also generates a spanning tree as an aside) the difference being that there is no notion of an underlying computation here. Processes send out probes (empty messages) to their neighbors and wait to receive the corresponding echoes (acknowledgements). As in [DS80], for each process, we represent the number of unacknowledged probes on the parent link by $C$, and that on the non-parent links by $D$. The steps are as follows:

```
program probe-echo
define N: integer \{number of neighbors\}
    C, D: integer
initially parent = i, C=0, D=0

\{for the initiator\}
send probes to each neighbor; D := number of neighbors;
do D \neq 0 \triangleright echo \triangleright D := D-1 od

\{for a non-initiator process i\}
do
    probe \triangleright parent = i \triangleright C=0 \triangleright C=1; parent := sender;
    if i is a leaf \triangleright send echo to parent;
        i is not a leaf \triangleright send probes to non-parent neighbors;
        D:= no of non-parent neighbors
    fi
    echo \triangleright D:= D-1
    probe \triangleright parent \neq i \triangleright send echo to sender;
    C=1 \triangleright D=0 \triangleright send echo to parent; C:=0
od
```
The computation terminates, when a message has been sent and received through every edge exactly once, and for all nodes, \( C = 0 \) and \( D = 0 \). The set of all nodes along with the directed edges connecting the nodes with their parents define the spanning tree. The proof follows from that of Dijkstra-Scholten’s algorithm [DS80] in Chapter 8 and is omitted.

![Graph of spanning tree](image)

**Fig 10.7. A spanning tree generated by Chang’s algorithm.**

An example is shown in Fig. 10.7. Note that the structure of spanning tree is determined by the message propagation delays through the different edges. Since these delays are arbitrary, different runs of the above algorithm may lead to different spanning trees.

The message complexity of the above algorithm is \( 2|E| \) since through each edge, a probe and an echo travel exactly once. If the root of the spanning tree is not designated, then to use the above algorithm, a root has to be identified first. This requires a leader election step. Leader election will be addressed in a subsequent chapter.

### 10.3.2. Tarry’s graph traversal algorithm

Graph traversal is a problem in which an initiator node sends out a token, each non-initiator receives the token and forwards to a neighbor, until the token returns to the initiator. A correct traversal algorithm must guarantee that (1) the token visits every node in the graph and (2) eventually returns to the initiator, which signals termination. Books on data structures extensively cover depth-first-search (DFS) and breadth-first-search (BFS) traversals as two basic methods of traversing a graph. In fact, any spanning tree rooted at the initiator can be used for traversal.
In 1895, Tarry proposed an algorithm [Ta1895] that solves the traversal problem, and generates a spanning tree of the graph in that process. It is the oldest known traversal algorithm, and hence an interesting candidate for study. Define the parent of a node as one from which the token is received for the first time. All other neighboring nodes will simply be called neighbors. The root is the initiator, and it does not have a parent. The following two rules define the algorithm:

**Rule 1.** Send the token towards any neighbor exactly once.

**Rule 2.** If Rule 1 cannot be used to send the token, then send the token to its parent.

When the token returns to the root, the entire graph has been traversed.

![Graph](image)

**Fig. 10.8.** A possible traversal route 0 1 2 5 3 1 4 6 2 6 4 1 3 5 2 1 0.

In the graph of Fig. 10.8, a possible traversal route for the token is 0 1 2 5 3 1 4 6 2 6 4 1 3 5 2 1 0. Each edge is traversed twice, once in each direction, and that the directed edges connecting each node with its parent form a spanning tree. Tarry’s algorithm does not guarantee that the traversal will follow the DFS order.

To prove that Tarry’s algorithm is a traversal algorithm, we need to show that (i) a least one of the rules is applicable until the token returns to the root, and (ii) eventually every node is visited.

**Lemma 10.3.** The token has a valid move until it returns to the root.

**Proof (by induction)**

**(Basis)** When the token is at the root, Rule 1 is applicable.
(Inductive case) Assume that the token is at a node \( i \) that is not the root. If Rule 1 does not apply, then Rule 2 must be applicable since the path from \( i \) to its parent remains to be traversed. It is not feasible for the token to stay at \( i \) if that path is already traversed. So, the token has a valid move.

Lemma 10.4. Using Tarry’s algorithm, eventually every node is visited by the token.

Proof (by contradiction). Consider a process \( j \), such that \( j \) is visited, but one of its neighbors \( k \) is not, and the token has returned to the root. Since the token finally leaves \( j \) via the edge towards its parent (Rule 2), \( j \) must have forwarded the token every neighbor prior to this. This includes \( k \), and contradicts the statement of the lemma.

These two lemmas establish that Tarry’s algorithm is a correct traversal algorithm. Since the token traverses each edge exactly twice, once in each direction, the message complexity of the algorithm is \( 2|E| \).

10.3.3. Minimum Spanning Tree

A given graph, in general, can have many different spanning trees. If each edge has a weight, then the spanning tree for which the sum of weights of all the edges is the smallest, is called the minimum spanning tree (MST). The minimum spanning tree has many applications. As an example, when the nodes represent cities and the edges represent air routes between the cities, the MST can be used to find the least expensive routes connecting all the cities. Two well-known sequential algorithms for computing the MST are Prim’s algorithm (also called Prim-Dijkstra algorithm) and Kruskal’s algorithm.

Prim’s algorithm builds the MST one node at a time. It starts at any node \( i \) in the graph, and connects it with a neighbor \( j \) via the edge with least weight. Next, it finds a node \( k \) that is connected with either \( i \) or \( j \) by an edge of least weight, and augments the tree with that edge and the node \( k \). Each augmentation step adds a new node and an edge, so no cycle is created. Eventually, all the vertices are connected, and an MST is generated. In case a choice has to be made between two or more edges with the same cost, anyone can be chosen.

Kruskal’s algorithm is a greedy algorithm and works as follows: Take the \( N \) nodes of the graph, and in each step, keep adding the edge of least weight, while avoiding the creation of cycles. When all \( N-1 \) edges have been added, the MST is formed. As in Prim’s
algorithm, when a choice has to be made between two or more edges with the same weight, anyone of them can be chosen.

Before we present a distributed algorithm for MST, consider the following lemma.

**Lemma 10. 5.** If the weight of every edge is distinct, then the MST is unique.

**Proof (by contradiction).** Suppose the MST is not unique. Then there must be at least two MST’s: MST1 and MST2 of the given graph. Let \( e_1 \) be the edge of smallest weight that belongs to MST1 but not MST2. Add \( e_1 \) to MST2 - this will form a cycle. Now, break the cycle by deleting an edge \( e_2 \) that does not belong to MST1 (clearly \( e_2 \) must exist). This process yields a tree whose total weight is lower than that of MST1. So MST1 cannot be a minimum spanning tree.

In [GHS83], Gallager, Humblet and Spira proposed a distributed algorithm for constructing the MST of a connected graph in which the edge weights are unique. Their algorithm works on a message-passing model, and can be viewed as a distributed implementation of Kruskal’s algorithm. It uses a bottom-up approach (see Fig. 10.9). The main idea is as follows:

**Strategy for MST construction.** Let \( T_1 \) and \( T_2 \) be two trees (called *fragments* in GHS83) covering disjoint subsets of nodes \( V_1 \) and \( V_2 \) respectively, such that \( T_1 \) is the tree of minimum weight covering \( V_1 \), and \( T_2 \) is the tree of minimum weight covering \( V_2 \). Let \( e \) be an edge with the minimum weight connecting \( T_1 \) and \( T_2 \). Then \( T_1 [ e ] T_2 \) is the tree of minimum weight covering the nodes in \( V_1 [ e ] V_2 \).

In [GHS83], initially each node is a fragment. The repeated application of the above procedure forms the MST of the given graph. However, in a distributed environment, there are some challenges to be met in implementing the above idea:

**Challenge 1.** How will the nodes in a given fragment identify the edge to be used for connecting to a different fragment?

In each fragment, a coordinator called the *root* will initiate the search, and choose one among the different outgoing edges connecting to a different fragment.
Challenge 2. How will a node in $T_1$ determine if a given edge connects to a node of a different tree $T_2$ or the same tree $T_1$? In Fig. 10.9, why will node 0 choose the edge $e$ with weight 8, and not the edge with weight 4?

The answer is that all nodes in the same fragment must acquire the same name before the augmentation takes place. The augmenting edge must connect to a node belonging to a fragment with a different name.

In [GHS83], each fragment belongs to a level. Initially each individual node is a fragment at level 0. Fragments join with one another in the following two ways:

(Merge) A fragment at level $L$ connects to another fragment at the same level. In this case, the level of the resulting fragment becomes $L+1$, and the resulting fragment is named after the edge joining the two fragments. In Fig. 10.9, the combined fragment will be named 8, which is the weight of the edge $e$.

(Absorb) A fragment at level $L$ joins a fragment at level $L' > L$. In this case, the level of the combined fragment becomes $L'$.

As a consequence of the above two operations, each fragment at level $L$ has at least $2^L$ nodes in it. The grand plan is to generate the MST in at most $\log N$ levels, where $N=|V|$.

Each fragment is a rooted tree. The root of a fragment is the node on which the lowest weight outgoing edge is incident. In Fig. 10.9, 0 is the root of $T_1$ and 2 is the root of $T_2$. Every node in a fragment has a parent, and any node can reach the root following the edge towards its parent. Initially every node is a fragment, so every node is the root of its own fragment. Before a fragment joins with another, it has to identify the lowest weight

Fig 10.9. An example showing two fragments $T_1$ and $T_2$ being joined by a minimum cost edge $e$ into a larger fragment
outgoing edge that connects to a different fragment, and the node (i.e. the root) on which it is incident.

**Detection of least weight outgoing edge.** When the root sends an *initiate* message, the nodes of that fragment search for the least weight outgoing edge (lwoe). Each node reports its finding through a *report* message to its parent. When the root receives the report from every process in its fragment, it determines the least weight outgoing edge from that fragment. The total number of messages required to detect the lwoe is $O(|V_i|)$, where $V_i$ is the set of nodes in the given fragment.

To test if a given edge is *outgoing*, a node sends a *test* message through that edge. The node at the other end may respond with a *reject* message (when it is the same fragment as the sender) or an *accept* message (when it is certain that it belongs to a different fragment). While rejection is straightforward, acceptance in some cases may be tricky. For example, it may be the case that the responding node belongs to a different fragment name when it receives the *test* message, but it is in the process of merging with the fragment of the sending node. To deal with this dilemma, when $i$ sends a *test* message (containing its *name* and *level*) to $j$, the responses by $j$ will be as follows:

**Case 1.** If $\text{name}\ (i) = \text{name}\ (j)$ then send *reject*

**Case 2.** If $\text{name}\ (i) \neq \text{name}\ (j) \land \text{level}\ (i) \leq \text{level}\ (j)$ then send *accept*

**Case 3.** If $\text{name}\ (i) \neq \text{name}\ (j) \land \text{level}\ (i) > \text{level}\ (j)$ then wait until $\text{level}\ (i) = \text{level}\ (j)$.

Note that the level numbers never decrease, and by allowing a node to send an accept message only when its level is at least as large as that of the sending node (and the fragment names are different), the dilemma is resolved.

To guarantee progress, we need to establish that the wait in Case 3 is finite. Suppose this is not true. Then there must exist a finite chain of fragments $T_0, T_1, T_2, \ldots$ of progressively decreasing levels, such that $T_i$ has sent a *test* message to $T_{i+1}$. But then the last fragment in the chain must also have sent a *test* message to another fragment of the same or higher level, and it is guaranteed to receive a response, enabling it to combine with another fragment and raise its level. Thus the wait is only finite.

For the sake of bookkeeping, each edge is classified into one of the three categories: *basic*, *branch*, and *rejected*. Initially every edge is a *basic* edge. When a *reject* message is sent through an edge, it is classified as *rejected*. Finally when a basic edge becomes a tree edge, its status changes to *branch*. 
**Property 2.** The attributes *branch* and *rejected* are stable.

As a consequence of Property 2, while searching for the least weight output edge, *test* messages are sent through the basic edges only.

Once the lwoe has been found, the root node sends out a *changeroot* message. After receiving this message, the node on which the lwoe is incident sends out a *join* message to the process at the other end of the lwoe, indicating its willingness to join. The *join* message initiates a *merge* or an *absorb* operation. Some possible scenarios are summarized below:

**Scenario 1: Merge.** A fragment at level $L$ sends a (*join, level = L, name= T*) message to another fragment at level $L'$ ($L' > L$) across its lwoe, and receives an (*initiate, level = L', name = T'*') message in return. The *initiate* message indicates that the fragment at level $L$ has been absorbed by the other fragment at level $L'$. The fragment at level $L$ changes its level to $L'$, and acquires the name $T'$ of the other fragment. Then they collectively search for the lwoe. The edge through which the *join* message is sent changes its status to *branch*, and becomes a tree edge.

![Diagram](Fig. 10.10. (a) merge operation (b) absorb operation)
**Scenario 2: Absorb.** A fragment at level $L$ sends out a $(\text{join}, L, \text{name})$ message to another fragment at the same level $L' = L$, and receives a $(\text{join}, L', \text{name})$ message in return. Thereafter, the edge through which the join message is sent, changes its status to branch, and becomes a tree edge. Each root broadcasts an $(\text{initiate}, L+1, \text{name})$ message to the nodes in its own fragment, where name corresponds to the tree edge that joined the fragments. All nodes change the level $L+1$.

Following a merge operation the node with larger id across the lwoe becomes the new root. All nodes in the combined fragment now orient their parent pointers appropriately. The algorithm terminates and the MST is formed when no new outgoing edge is found in a fragment. A complete example of MST formation is illustrated in Fig 10.11.

An apparent concern here is that of deadlock. What if every fragment sends a join message to a different fragment, but no fragment receives a reciprocating join message to complete the handshake? Can such a situation arise, leading to a circular waiting? The next lemma shows that this is impossible.

**Lemma 10.6.** Until the MST is formed, there must always be a pair of fragments, such that their roots that will send a join message to each other.

**Proof.** Consider the fragments across the edge of least weight. They must send join messages to each other. This will trigger either a merge or an absorb operation.

As a consequence of Lemma 10.6, no node indefinitely waits to unite with another fragment, and progress is guaranteed.

**Message complexity.** Since a fragment at level $k$ has at most $2^k$ nodes, the maximum level cannot exceed $\log N$. To coordinate the joining of two fragment, in each level, $O(N)$ messages are sent along tree edges. This requires $O(N \log N)$ messages.

Furthermore, to search for the lwoe, the edges of the graph have to be tested. If each edge was tested at every level, then the number of messages needed to compute the lwoe would have been $O(|E| \log N)$. Fortunately, each edge is rejected at most once. Once an edge becomes a tree edge, it is no more tested. However, an edge across which an accept message has been sent, can be tested repeatedly – in each level, such an edge can be tested once until the edge becomes a tree edge, or the algorithm terminates. In each level, such an edge $(i, j)$ is tested by the node $i$ or $j$ at most once, leading to a message complexity of
O(N \cdot \log N). Adding to it the overhead of sending at most one reject message through each edge, the message complexity for \textit{lwoe} determination becomes $O(N \log N + |E|)$.

Therefore, the overall message complexity of the MST generation algorithm is $O(|E| + N \log N)$.

![Diagram](image)

Fig 10.11. An Example of MST formation using GHS83. In (a), node 3 sends a join request to 5, but 5 does not respond until it has formed a fragment (part b) by joining with node 2. Note how the root changes in each step.
10.4 Graph Coloring

The problem of node coloring in graphs can be stated as follows: Given a graph \( G = (V, E) \), assign color to the nodes in \( V \) from a given set of colors, so that no two neighboring nodes have the same color. Graph coloring has been extensively investigated as a classic problem in graph theory. The algorithms become particularly challenging when the color palette is small, or its size approaches the lower bound. In a distributed environment, no node knows anything about the graph beyond its immediate neighbors. To realize the difficulties in the construction of such algorithms, consider the graph in Fig. 10.12.

![Graph](image)

Fig. 10.12. A graph that can be colored with two colors 0 and 1

The nodes of the graph (Fig. 10.13) can be colored using two colors \( \{0, 1\} \). Let \( c[i] \) denote the color of node \( i \). Assume that initially \( c[i] = 0 \). We assume a shared memory model, and try a naïve algorithm for every process \( i \):

\[
\text{do } \forall j \in \text{neighbor}(i) : c[i] = c[j] \quad \forall j \quad c[i] := 1 - c[i] \quad \text{od}
\]

Assume that there is a central demon, so only one process executes a step at any time. We encourage the readers to verify that the naive algorithm does not terminate.

10.4.1. A simple coloring algorithm

As a first illustration of a distributed algorithm for graph coloring, consider a graph, in which the maximum degree of any node is \( D \). We demonstrate an algorithm for coloring such a graph with \( D+1 \) colors. We will designate the set of all colors by \( \mathbb{C} \).

The algorithm runs on a shared memory model under a serial demon. No fairness is assumed. The atomicity is coarse-grained, so that a process can read the states of all its neighbors and execute an action in a single step. Let \( \text{neighbor}(i) \) denote the set of all
neighbors of node \( i \). Also define \( nc(i) \) to be the set of colors of the nodes of \( \text{neighbor}(i) \).

Then the coloring algorithm is as follows:

```
program simple coloring;

{program for node i}

do
    \( \forall j \in \text{neighbor}(i) : c[i] = c[j] \)
    \( c[i] := b : b \in \{\} \setminus nc(i) \)

od
```

**Theorem 10.7.** Program simple coloring produces a correct coloring of the nodes.

**Proof.** Each action by a node resolves its color with respect to those of its neighbors. Once a node resolves its color with respect to its neighbors, its guard is never enabled by an action of a neighboring node. So regardless of the initial colors of the nodes, each node executes its action at most once, and the algorithm requires at most \((N-1)\) steps to terminate.

```
In many cases, the size of the color palette used in the simple algorithm may be far from optimum. For example, consider a star graph where \( N-1 \) nodes are connected to a single node that acts as the hub. The simple algorithm will require \( N \) distinct colors, whereas the graph can be colored using two colors only!

Converting the graph into a dag (directed acyclic graph) sometimes helps reduce the size of the color palette. In a dag, let \( \text{succ}(i) \) denote the successors of node \( i \) defined as the set of all nodes \( j \) such that a directed edge \((i,j)\) exists. Also, let \( sc(i) \) represent the set of colors of the nodes in \( \text{succ}(i) \). Then the following is an adaptation of the simple algorithm when the topology is a dag:

```
program dag coloring;

{program for node i}

do
    \( \forall j \in \text{succ}(i) : c[i] = c[j] \)
    \( c[i] := b : b \in \{\} \setminus sc(i) \)

od
For the star graph mentioned earlier, if all edges are directed towards the hub, then each node at the periphery has only the hub as its successor, and the above algorithm can trivially produce coloring with two colors only.

We present here an inductive proof of the dag-coloring algorithm. Any dag has at least one node with no outgoing edges – call it a leaf node. According to the program, the leaf nodes do not execute actions since they have no successors. So their colors are stable, (i.e. their colors do not change from now onwards). This is the base case.

After the successors of a node \( i \) attain a stable color, it requires one more step for \( c[i] \) to become stable, and such a color can always be found since the set \( \{i \in \text{sc}(i) \} \) is non-empty. Thus, the nodes at distance one from a leaf node acquire a stable color in at most one step, those at a maximum distance 2 attains a stable color in at most \((1+2)\) steps, and so on. Eventually all nodes are colored, and the maximum number of required steps is \( 1+2+3+ \ldots +L = L(L+1)/2 \), where \( L \) is the length of the longest path in the graph.

The proposed method works for directed graphs only. What remains is to devise a method for converting undirected graphs into directed ones. A straightforward approach is to construct a BFS spanning tree, and direct each edge towards a node of higher level, but the outdegree of some nodes (and consequently the color palette) may still be large. In some cases, we can do even better. The next section addresses this issue with an example.

### 10.4.2. Planar graph coloring

In this section, we demonstrate an algorithm of coloring the nodes of any planar graph with at most six colors. The color palette \( S = \{0, 1, 2, 3, 4, 5\} \). The basic principle is to transform any given planar graph into a directed acyclic graph where the degree of every node is < 6, and execute the coloring algorithm on this dag. We begin with the assumption of coarse-grain atomicity – in a single atomic step each node examines the states of all its neighbors and executes an action. A central demon arbitrarily serializes the actions of the nodes.

For any planar graph \( G = (V, E) \), if \( e = |E| \) and \( p = |V| \), then the following results can be found in most books on graph theory (for example, see [Ha72]).

**Theorem 10.8 (Euler’s polyhedron formula)** If \( p > 2 \), then \( e \geq 3p-6 \)

**Corollary 10.8.1.** For any planar graph, there is at least one node with degree \( \leq 5 \).
Call a node whose degree is $\leq 5$ a *core* node. The algorithm to assign edge directions works as follows. Initially, all edges are undirected. Let each node execute the following algorithm:

\begin{verbatim}
do
    number of undirected edges $\leq 5$ make all undirected edges outgoing
od
\end{verbatim}

At the beginning, the core nodes of $G$ will mark all edges incident on them as outgoing. The remainder graph obtained by deleting the core nodes and the directed edges from $G$ is also a planar graph, so the core nodes of the remainder graph now mark the undirected edges incident on them as outgoing. This continues, until the remainder graph is empty, and all edges are directed. That the resulting graph is a dag, and every node has an outdegree $\leq 5$ trivially follow.

Interestingly, the coloring algorithm need not wait for the dag-generation algorithm to terminate – both of them can run concurrently. However, the first algorithm runs only when the outdegree $D$ of each node is $\leq 5$, so the composite algorithm will be as follows:

\begin{verbatim}
program planar graph coloring;

{program for node i}
do
    outdegree(i) $\leq 5$ $\forall$ $j \in$ succ(i) : $c[i] = c[j]$
    $c[i] := b : b \in \{\} \setminus$ sc(i)
    number of undirected edges $\leq 5$
    make all undirected edges outgoing
od
\end{verbatim}

An example of dag generation is shown in Fig. 10.13.

The second action is executed at most $|V|$ times, after which all edges are directed. In case the first action is executed before all edges have been assigned directions, some nodes may acquire interim colors – however, these will be overwritten by the first action after the second guard is permanently disabled for all nodes.
Fig. 10.13. An example of generating a dag from a planar graph. The core nodes are shaded. In (a), the core nodes execute their actions in the order 0, 1, 3, 5, 2, 7, 6, 9, 10. In part (b), after node 4 executes its move, all edges are appropriately directed, and the node 8 becomes a leaf node.
The termination of the composite program reflects the principle of *convergence stairs* proposed in \([AG]\). Let \(A\) and \(B\) be two algorithms, such that from a given initial condition \(\text{init}\), the following holds:

- \{\text{init}\} \ A \ {P}\)
- \{P\} \ B \ {Q}\), and
- The actions of algorithm \(B\) do not negate the guards of algorithm \(A\)

Then \{\text{init}\} \ \text{AllB} \ \{Q\} \ holds, where \text{AllB} denotes the concurrent execution of the two programs \(A\) and \(B\).

The composition rule can be justified as follows: Since the actions of \(B\) do not negate the guards of \(A\), algorithm \(A\) is guaranteed to converge to a configuration that satisfies the postcondition \(P\). Thereafter, the suffix of the computation guarantees convergence to \(Q\) since \{P\} \ B \ \{Q\}. For the composition illustrated in the graph coloring algorithm, \(P = \{V: \text{outdegree}(i) \leq 5\}\), and \(Q = \{(i,j): \text{c}(i) \neq \text{c}(j)\}\).

To implement the graph-coloring algorithm, each process has to know the direction of the edges incident on it. This can be done as follows. Let each node maintain a local variable \(\text{succ}\) which is a set containing the names of the successors of that node. Initially, \(\square i: \text{succ}(i) = \emptyset\). Node \(i\) has an outgoing edge to a neighbor \(j\), when \(j \in \text{succ}(i)\). An edge \((i,j)\) is undirected, when \(j \in \text{succ}(i) \land i \in \text{succ}(j)\). Thus, when a node \(i\) makes the edge \((i,j)\) outgoing, it appends \(j\) to \(\text{succ}(i)\), a condition that can be tested by both \(i\) and \(j\).

**Complexity analysis.** To determine the edge directions, each process executes at most one step, so the complexity for this part in only \(O(N)\) steps. Once the edge directions have been determined, the complexity of the coloring part is \(O(L^2)\) steps, where \(L\) is the length of the longest directed path in the transformed dag. Note that if a node \(i\) colors itself using the coloring algorithm before the edge directions have stabilized, then that color \(\text{c}[i]\) remains stable, since the actions of any other node \(j \neq i\) can change some of its undirected edges incident on \(I\) into incoming ones, and does not increase the size of \(\text{succ}(i)\). Since the upper bound of \(L\) is \(N\), the time complexity is \(O(N^2)\) steps.
10.5. Concluding Remarks

Many applications in distributed computing center around a few common graph problems -- this chapter addresses a few of them. An algorithm is considered robust, when it works on dynamic graphs, i.e. it survives changes in topology. Mobile ad-hoc networks add a new dimension to the fragility of the network topology, because the transmission range of each node is limited. With such networks, power consumption is a major issue – a low consumption of power adds to the life of the system. Therefore, a useful performance metric for mobile ad-hoc networks is the amount of power used by the nodes, before the system reaches quiescence.

Numerous applications require broadcasting or multicasting, for which spanning trees are useful. In a network of \( N \) nodes, a spanning tree has \( N-1 \) edges, so the message complexity of broadcasting any message is \( N \). This is smaller than the number of messages required by flooding algorithms, for which the message complexity is at least \( O(|E|) \), regardless of the topology.

The GHS algorithm for MST construction has been extensively studied in the published literature. Interestingly, the construction of an arbitrary spanning tree also has the same message complexity of \( O(N \cdot \log N + |E|) \) as that of the GHS algorithm.

The link-state algorithm has the nice property of self-stabilization: the algorithm correctly and efficiently responds to changes in the topology without generating much traffic. This makes it attractive for implementation in real life. The downside is that, the space complexity per process is large, so scalability is a major issue. There is yet another routing algorithm, called distance-vector routing, that is used in networking. Unlike the link-state algorithm where a node communicates with every other node in the network to deliver the information about its immediate neighbors, in distance-vector routing, every node sends the indirect information about whatever it has learned so far about the topology to its immediate neighbors. The algorithm is not self-stabilizing: it is possible that certain failures of nodes or links may cause individual nodes to accumulate incorrect information about the topology, and make the routing table incorrect.

Interval routing is a topic that has generated sufficient interest among theoreticians. It addresses the scalability problem typical of many commercial algorithms. However, as of now, its inability to adapt to changes in topology limits its applicability. So far, it has been used only for communication in some transputer-based\(^2\) distributed systems. Some attempts of using it in sensor networks have recently been reported.

---

\(^2\) Transputers were introduced by INMOS as building blocks of distributed systems.
Bibliographic Notes

Chandy-Misra’s shortest path algorithm [CM82] is an adaptation of the Bellman-Ford algorithm used with ARPANET during 1969-1979. The adaptation included a mechanism for termination detection, and a mechanism to deal with negative edge weights. The link-state routing algorithm also originated from ARPANET and was first proposed by McQuillan et al [MQ78]. After several modifications, it was adopted by the ISO as an OSPF protocol. Santoro and Khatib [SK85] introduced interval routing. This paper demonstrated the feasibility for tree topologies only. Van Leewuen and Tan [LT87] extended the idea to non-tree topologies. The probe algorithm for computing the spanning tree is due to Chang [Ch82] -- Segall [Se83] presented a slightly different version of it. In [GHS83] Gallager, Humblet, and Spira presented their minimum spanning tree algorithm – it has been extensively studied in the field of distributed algorithms, and many different correctness proofs have been proposed. Tarry’s traversal algorithm [T1995] proposed for exploring an unknown maze, is one of the oldest known distributed algorithms. The distributed algorithm for coloring planar graphs is due to Ghosh and Karaata [GK93] – the original paper proposed a self-stabilizing algorithm for the problem. The version presented in this chapter is a simplification of that algorithm.

Exercises

1. Let $G= (V,E)$ be a directed graph. A maximal strongly connected component of $G$ is a subgraph $G'$ such that (i) for every pair of vertices $u, v$ in the $G'$, there is a directed path from $u$ to $v$ and a directed path from $v$ to $u$, and (ii) no other subgraph of $G$ has $G'$ as its subgraph. Propose a distributed algorithm to compute the maximal strongly connected component of a graph.

2. Let $G = (V, E)$ be an undirected graph and let $V' \subseteq V$. The elements of $V'$ are interpreted as the members of the group $V'$. Each edge has a unit weight. The nodes in $V'$ want to communicate with one another via a multicast tree which is a tree of least cost that includes all members of $V'$. Design a distributed algorithm to construct a multicast tree.

3. Devise an interval labeling scheme for optimal routing on a (i) $4 \times 4$ grid of 16 processes, and (ii) a 4-cube of 16 processes.
4. Is it true that in the following tree, no linear interval-labeling scheme exists?

5. **The background:**

The routing table is an array of tuples (destination, nexthop, distance). To send a packet to a given destination, it is forwarded to the process in the corresponding nexthop field of the tuple. In *distance vector routing* each node updates its knowledge about the topology by computing its distance from other nodes, and shares it with other neighbors. In a graph with \( N \) nodes, the distance vector \( D \) for each node \( i \) contains \( N \) elements, and \( D[i,j] \) defines the its distance from node \( i \) to node \( j \). Initially

\[
D[i,j] = 0 \quad \text{when } i = j, \\
= 1 \quad \text{when } j \text{ is a neighbor of } i, \text{ and} \\
= \bullet \quad \text{when } i \neq j \text{ and } j \text{ is not a neighbor of } i
\]

Each node broadcasts its distance vector to its immediate neighbors. The recipient node executes the following program:

\[
\text{do } \quad \text{vector from node } j \text{ is received} \\
\quad \text{[k : } D[i,j] + D[j,k] < D[i,k] \text{]} \\
\quad D[i,k] := D[i,j] + D[j,k] \\
\quad \text{nexthop for destination } k \text{ is } j \\
\text{od}
\]

When a failure hits a node or a link, its neighbors detect the failure, and set the corresponding distance to \( \bullet \), and the algorithm re-computes the routing table.

Unfortunately, depending on when a failure is detected, and when the advertisements are sent out, the routing table may not stabilize. Consider the following network.
Initially, $d(1,3)=2$ and $d(2,3)=1$. As the link $(2,3)$ fails, $d(2,3)$ becomes infinity. But node 1 may still pass on the stale information $d(1,3) = 2$ to nodes 0 and 2. As a result, node 2 updates $d(2,3)$ to 3.

**The question:**
(a) Work out the next few steps to show that there is a valid computation that prevents the distance vectors from stabilizing.
(b) Suggest a method to fix this problem.

6. Modify Chang’s spanning tree algorithm so that the generated spanning tree is always a DFS tree.

7. Modify Tarry’s traversal algorithm so that the traversal path follows a DFS tree.

8. In the two-dimensional grid shown below, the nodes can be colored using at most three colors. Suggest a distributed algorithm for coloring the nodes with no more than three colors. Provide a correctness proof of your algorithm.
9. The classical algorithm for generating a spanning tree requires $O(|E|)$ messages, and is completed in $O(|E|)$ time. Devise an algorithm that generates the spanning tree in $O(N)$ time, where $N = |V|$. [Hint: See [Aw85]. When the token visits a node $i$ for the first time, it lets every neighbor $j$ know that it has been visited. The token is not forwarded until it has received an acknowledgement from node $j$. Since $j$ knows that $i$ has been visited, it will not forward the token to $i$.] Analyze the time and the message complexities.

10. Given an undirected graph $G = (V, E)$, the maximal matching problem computes a maximal subset $E'$ of $E$ such that each vertex is incident with exactly one edge in $E'$. Suggest a distributed algorithm for computing $E'$. When the algorithm terminates, each node must know its matching neighbor, if such a match exists.

11. Devise a distributed algorithm for computing a spanning tree of a graph in which no root is designated.

12. In a spanning tree of a graph, there is exactly one path between any pair of nodes. If a spanning tree is used for broadcasting a message, and a process crashes, some nodes will not be able to receive the broadcast. Our goal is to improve the connectivity of the subgraph used for broadcast, so that it can tolerate the crash of one process.

   What kind of minimal subgraph would you use for broadcasting, so that messages will reach every process even if one process fails? Suggest a distributed algorithm for constructing such a subgraph. How many messages will you need to complete a broadcast?

12. Design a distributed algorithm to find the center of a tree. (Include the definition of center first)

13. Extend the prefix routing scheme to non-tree topologies.