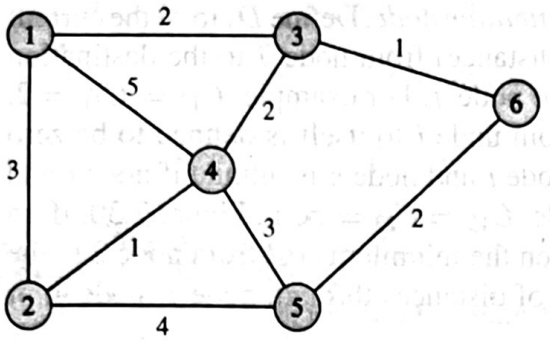

FIGURE 7.30 A network with associated link costs.

packet switch and each link represents a transmission line between two packet switches. Figure 7.30 shows such an example. Associated with each link is a value that represents the *cost* (or *metric*) of using that link. For simplicity, it is assumed that each link is nondirected. If a link is directed, then the cost must be assigned to each direction. If we define the path cost to be the sum of the link costs along the path, then the shortest path between a pair of nodes is the path with the least cost. For example, the shortest path from node 2 to node 6 is 2-4-3-6, and the path cost is 4.

Many metrics can be used to assign a cost to each link, depending on the objective function that is to be optimized. Examples include

1. *Cost* ~ *1/capacity.* The cost is inversely proportional to the link capacity. Here one assigns higher costs to lower-capacity links. The objective is to send a packet through a path with the highest capacity. If each link has equal capacity, then the shortest path is the path with the minimum number of hops.
2. *Cost* ~ *packet delay.* The cost is proportional to an average packet delay, which includes queueing delay in the switch buffer and propagation delay in the link. The shortest path represents the fastest path to reach the destination.
3. *Cost* ~ *congestion.* The cost is proportional to some congestion measure, for example, traffic loading. Thus the shortest path tries to avoid congested links.

7.5.1 The Bellman-Ford Algorithm

The **Bellman-Ford algorithm** (also called the Ford-Fulkerson algorithm) is based on a principle that is intuitively easy to understand: If each neighbor of node A knows the shortest path to node Z, then node A can determine its shortest path to node Z by calculating the cost/distance to node Z through each of its neighbors and picking the minimum.

As an example, suppose that we want to find the shortest path from node 2 to node 6 (the destination) in Figure 7.30. To reach the destination, a packet from node 2 must first go through node 1, node 4, or node 5. Suppose that someone tells us that the shortest paths from nodes 1, 4, and 5 to the destination (node 6) are 3, 3, and 2, respectively. If the packet first goes through node 1, *the total distance* (also called total cost) is 3 + 3, which is equal to 6. Through node 4, the total distance is 1 + 3, equal to 4. Through node 5, the total distance is 4 + 2, equal to 6. Thus the shortest path from node 2 to the destination node is achieved if the packet first goes through node 4.

To formalize this idea, let us first fix *the destination node*. Define D_j to be the current estimate of the minimum cost (or minimum distance) from node j to the destination node and C_{ij} to be the link cost from node i to node j. For example, $C_{13} = C_{31} = 2$, and $C_{45} = 3$ in Figure 7.30. The link cost from node i to itself is defined to be zero (that is, $C_{ii} = 0$), and the link cost between node i and node k is infinite if node i and node k are not directly connected. For example, $C_{15} = C_{23} = \infty$ in Figure 7.30. If, in Figure 7.30, the destination node is node 6, then the minimum cost from node 2 to the destination node 6 can be calculated in terms of distances through node 1, node 4, or node 5:

$$\begin{aligned} D_2 &= \min\{C_{21} + D_1, C_{24} + D_4, C_{25} + D_5\} \\ &= \min\{3 + 3, 1 + 3, 4 + 2\} \\ &= 4 \end{aligned} \tag{7.5}$$

Thus the minimum cost from node 2 to node 6 is through node 4 and is equal to 4.

One problem in our calculation of the minimum cost from node 2 to node 6 is that we have assumed that the minimum costs from nodes 1, 4, and 5 to the destination were known. In general, these nodes would not know their minimum costs to the destination without performing similar calculations. So let us apply the same principle to obtain the minimum costs for the other nodes. For example, the cost from node 1 to the destination node 6 is found from

$$D_1 = \min\{C_{12} + D_2, C_{13} + D_3, C_{14} + D_4\} \tag{7.6}$$

and similarly the cost from node 4 is found from

$$D_4 = \min\{C_{41} + D_1, C_{42} + D_2, C_{43} + D_3, C_{45} + D_5\} \tag{7.7}$$

A discerning reader will note immediately that these equations are circular, since D_2 depends on D_1 and D_1 depends on D_2. The magic is that if we keep iterating and updating these equations, the algorithm will eventually converge to the correct result. To see this outcome, assume that initially $D_1 = D_2 = \ldots = D_5 = \infty$. Observe that at each iteration we may discover new shorter paths to the destination, and so the distances from each node to the destination node 6, that is, $D_1, D_2, \ldots, D_5$ are nonincreasing. Because the minimum distances are bounded below, eventually $D_1, D_2, \ldots, D_5$ must converge to the distances corresponding to the shortest path to the given destination.

Now if we define d as the destination node, we can summarize the Bellman-Ford algorithm as follows:

1. Initialization (destination node d is distance 0 from itself)

$$D_i = \infty, \quad \text{for all } i \neq d \tag{7.8}$$

$$D_d = 0 \tag{7.9}$$

2. Updating (find minimum distance to destination through neighbors): For each $i \neq d$,

$$D_i = \min_j \{C_{ij} + D_j\}, \quad \text{for all } j \neq i \tag{7.10}$$

Repeat step 2 until no more changes occur in the iteration.

EXAMPLE Minimum Cost

Using Figure 7.30, apply the Bellman-Ford algorithm to find both the minimum cost from each node to the destination (node 6) and the next node along the shortest path.

Each node i maintains an entry (n, D_i), where n is the next node along the current shortest path and D_i is the current minimum cost from node i to the destination. The next node is given by the value of j in Equation 7.10, which gives the minimum cost. If the next node is not defined, we set n to -1. In the first iteration, the destination node informs its directly-attached neighbors that it is distance zero from itself. This prompts the neighbors to calculate their distance to the destination node. In iteration 2, the directly-attached neighbors inform their neighbors of their current shortest distance to the destination, so all nodes within two hops of the destination have found a path to the destination. At iteration m, all nodes within m hops of the destination node have determined a path to the destination. The algorithm terminates when no more changes in entries are observed. Table 7.2 shows the execution of the Bellman-Ford algorithm for destination node 6.

- Initially all nodes, other than the destination node 6, are at infinite cost (distance) to node 6. Node 6 informs its neighbors it is distance 0 from itself.
- (Iteration 1) Node 3 finds that it is connected to node 6 with cost of 1. Node 5 finds it is connected to node 6 at a cost of 2. Nodes 3 and 5 update their entries and inform their neighbors.
- (Iteration 2) Node 1 finds it can reach node 6, via node 3 with cost 3. Node 2 finds it can reach node 6, via node 5 with cost 6. Node 4 finds it has paths via nodes 3 and 5, with costs 3 and 7 respectively. Node 4 selects the path via node 3. Nodes 1, 2, and 4 update their entries and inform their neighbors.
- (Iteration 3) Node 2 finds that it can reach node 6 via node 4 with distance 4. Node 2 changes its entry to (4, 4) and informs its neighbors.
- (Iteration 4) Nodes 1, 4, and 5 process the new entry from node 2 but do not find any new shortest paths. The algorithm has converged.

TABLE 7.2 Sample processing of Bellman-Ford algorithm. Each entry for node i represents the next node and cost of the current shortest path to destination 6.

Iteration	Node 1	Node 2	Node 3	Node 4	Node 5
Initial	$(-1, \infty)$	$(-1, \infty)$	$(-1, \infty)$	$(-1, \infty)$	$(-1, \infty)$
1	$(-1, \infty)$	$(-1, \infty)$	(6, 1)	$(-1, \infty)$	(6, 2)
2	(3, 3)	(5, 6)	(6, 1)	(3, 3)	(6, 2)
3	(3, 3)	(4, 4)	(6, 1)	(3, 3)	(6, 2)
4	(3, 3)	(4, 4)	(6, 1)	(3, 3)	(6, 2)

EXAMPLE **Shortest-Path Tree**

From the preceding example, draw the shortest path from each node to the destination node. From the last row of Table 7.2, we see the next node of node 1 is node 3, the next node of node 2 is node 4, the next node of node 3 is node 6, and so forth. Figure 7.31 shows the shortest-path tree rooted to node 6.

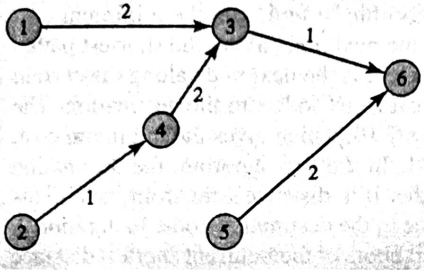

FIGURE 7.31 Shortest-path tree to node 6.

One nice feature of the Bellman-Ford algorithm is that it lends itself readily to a distributed implementation. The process involves having each node independently compute its minimum cost to each destination and periodically broadcast the vector of minimum costs to its neighbors. Changes in the routing table should also trigger a node to broadcast the minimum costs to its neighbors to speed up convergence. This mechanism is called *triggered updates*. It turns out that the distributed algorithm would also converge to the correct minimum costs under mild assumptions. Upon convergence, each node would know the minimum cost to each destination and the corresponding next node along the shortest path. Because only cost vectors (or distance vectors) are exchanged among neighbors, the protocol implementing the distributed Bellman-Ford algorithm is often referred to as a *distance vector protocol*. Each node i participating in the distance vector protocol computes the following equation:

$$D_{ii} = 0 \tag{7.11}$$

$$D_{ij} = \min_k \{C_{ik} + D_{kj}\}, \quad \text{for all } k \neq i \tag{7.12}$$

where D_{ij} is the minimum cost from node i to the destination node j. Upon updating, node i broadcasts the vector $\{D_{i1}, D_{i2}, D_{i3}, \ldots\}$ to its neighbors. The distributed version can adapt to changes in link costs or topology as the next example shows.

EXAMPLE **Recomputing Minimum Cost**

Suppose that after the distributed algorithm stabilizes for the network shown in Figure 7.30, the link connecting node 3 and node 6 breaks. Compute the minimum cost from each node to the destination node (node 6), assuming that each node immediately recomputes its cost after detecting changes and broadcasts its routing updates to its neighbors. The new network topology is shown in Figure 7.32.

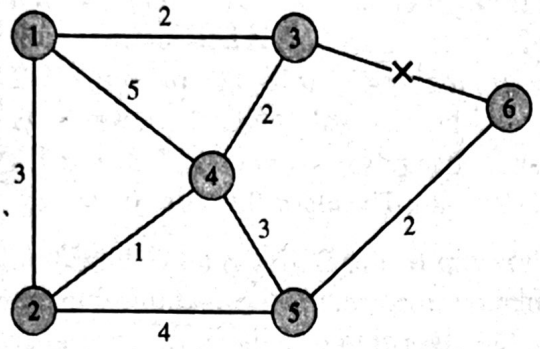

FIGURE 7.32 New network topology following break from node 3 to 6.

Upon receiving routing updates, we assume that nodes recompute their shortest paths in parallel simultaneously. When the computations are completed, we also assume that routing updates are transmitted simultaneously. The results of the computations are shown in Table 7.3. The results are obtained as follows (again the reader is encouraged to perform the algorithm before reading the discussion).

- (Update 1) As soon as node 3 detects that link (3,6) breaks, node 3 recomputes the minimum cost to node 6. Node 3 looks for paths to node 6 through its neighbors, node 1 and node 4, and its calculations indicate that the new shortest path is through node 4 at a cost of 5. (At this point, node 3 does not realize that the shortest path that node 4 is advertising happens to go through node 3, so a loop has been created!) Node 3 then sends the new routing update to its neighbors, which are nodes 1 and 4.
- (Update 2) Nodes 1 and 4 then recompute their minimum costs: node 1 finds its shortest path is still through node 3 but the cost has increased to 7; node 4 finds its shortest path is through either node 2 or node 5 with a cost of 5. We suppose that node 4 chooses node 2 (which creates a new loop between 4 and 2). Node 1 transmits its routing update to nodes 2, 3, and 4, and node 4 transmits its routing update to nodes 1, 2, 3, and 5.
- (Update 3) Node 1 finds its shortest path is still through node 3. Node 2 finds its shortest path is still through node 4 but the cost has increased to 6. Node 3 finds that its shortest path is still through node 4 but the cost has increased to 7. Nodes 4 and 5 find their shortest paths have not changed. Node 2 transmits its update to nodes 1, 4, and 5, and node 3 transmits its update to nodes 1 and 4.

TABLE 7.3 Next node and cost of current shortest path to node 6 using the distributed version.

Update	Node 1	Node 2	Node 3	Node 4	Node 5
Before break	(3, 3)	(4, 4)	(6, 1)	(3, 3)	(6, 2)
1	(3, 3)	(4, 4)	(4, 5)	(3, 3)	(6, 2)
2	(3, 7)	(4, 4)	(4, 5)	(2, 5)	(6, 2)
3	(3, 7)	(4, 6)	(4, 7)	(2, 5)	(6, 2)
4	(2, 9)	(4, 6)	(4, 7)	(5, 5)	(6, 2)
5	(2, 9)	(4, 6)	(4, 7)	(5, 5)	(6, 2)

- (Update 4) Node 1 finds its shortest path is through either node 2 or node 3 with a cost of 9. Suppose that node 1 chooses node 2. Node 4 now finds a new shortest path through node 5 with a cost of 5 (which removes the loop), since the cost through 2 has increased to 7. Node 5 does not change its shortest path. Node 1 transmits its update to nodes 2, 3, and 4, and node 4 transmits its update to nodes 1, 2, 3, and 5.
- (Update 5) None of the nodes finds a new shorter path. The algorithm has converged.

This example shows that the distributed version of the Bellman-Ford algorithm requires a series of exchanges of update information to correct inaccurate information until convergence is achieved. Because of this, the algorithm usually converges rather slowly. Note also that during the calculation in the transient state, packets already in transit may loop among nodes. After convergence in the steady state, packets eventually find the destination.

EXAMPLE **Reaction to Link Failure**

This example shows that the distributed Bellman-Ford algorithm may react very slowly to a link failure. To see this, consider the topology shown in Figure 7.33a with node 4 as the destination. Suppose that after the algorithm stabilizes, link (3,4) breaks, as shown in Figure 7.33b. Recompute the minimum cost from each node to the destination node (node 4).

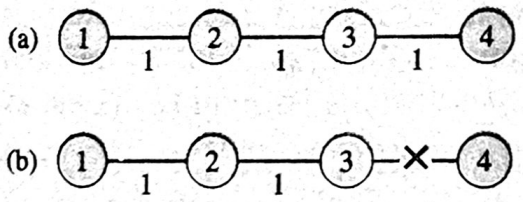

FIGURE 7.33 Topology before and after link failure.

TABLE 7.4 Next node and cost of current shortest path to node 4.

Update	Node 1	Node 2	Node 3
Before break	(2, 3)	(3, 2)	(4, 1)
After break	(2, 3)	(3, 2)	(2, 3)
1	(2, 3)	(3, 4)	(2, 3)
2	(2, 5)	(3, 4)	(2, 5)
3	(2, 5)	(3, 6)	(2, 5)
4	(2, 7)	(3, 6)	(2, 7)
5	(2, 7)	(3, 8)	(2, 7)
...	...	...	...

Note: Dots in the last row indicate that the table continues to infinity

The computation of minimum costs is shown in Table 7.4. As the table shows, each node keeps updating its cost (in increments of 2 units). At each update, node 2 thinks that the shortest path to the destination is through node 3. Likewise, node 3 thinks the best path is through node 2. As a result, a packet in either of these two nodes bounces back and forth until the algorithm stops updating. Unfortunately, in this case the algorithm keeps iterating until the minimum cost is infinite (or very large, in practice), at which point, the algorithm realizes that the destination node is unreachable. This problem is often called **counting to infinity**. It is easy to see that if link (3,4) is restored, the algorithm will converge very quickly. Therefore: Good news travels quickly, bad news travels slowly.

To avoid the counting-to-infinity problem, several changes to the algorithm have been proposed, but unfortunately, none of them work satisfactorily in all situations. One particular method that is widely implemented is called the **split horizon**, whereby the minimum cost to a given destination is not sent to a neighbor if the neighbor is the next node along the shortest path. For example, if node X thinks that the best route to node Y is via node Z, then node X should not send the corresponding minimum cost to node Z. Another variation called **split horizon with poisoned reverse** allows a node to send the minimum costs to all its neighbors; however, the minimum cost to a given destination is set to infinity if the neighbor is the next node along the shortest path. Here, if node X thinks that the best route to node Y is via node Z, then node X should set the corresponding minimum cost to infinity before sending it to node Z.

EXAMPLE **Split Horizon with Poisoned Reverse**

Consider again the topology shown in Figure 7.33a. Suppose that after the algorithm stabilizes, link (3,4) breaks. Recompute the minimum cost from each node to the destination node (node 4), using the split horizon with poisoned reverse.

The computation of minimum costs is shown in Table 7.5. After the link breaks, node 3 sets the cost to the destination equal to infinity, since the minimum cost node 3 has received from node 2 is also infinity. When node 2 receives the update message, it also sets the cost to infinity. Next node 1 also learns that the destination is unreachable. Thus split horizon with poisoned reverse speeds up convergence in this case.

TABLE 7.5 Minimum costs by using split horizon with poisoned reverse.

Update	Node 1	Node 2	Node 3
Before break	(2, 3)	(3, 2)	(4, 1)
After break	(2, 3)	(3, 2)	$(-1, \infty)$
1	(2, 3)	$(-1, \infty)$	$(-1, \infty)$
2	$(-1, \infty)$	$(-1, \infty)$	$(-1, \infty)$

7.5.2 Dijkstra's Algorithm

Dijkstra's algorithm is an alternative algorithm for finding *the shortest paths from a source node to all other nodes in a network.* It is generally more efficient than the Bellman-Ford algorithm but requires each link cost to be positive, which is fortunately the case in communication networks. The main idea of Dijkstra's algorithm is to progressively identify the closest nodes from the source node in order of increasing path cost. The algorithm is iterative. At the first iteration the algorithm finds the closest node from the source node, which must be the neighbor of the source node if link costs are positive. At the second iteration the algorithm finds the second-closest node from the source node. This node must be the neighbor of either the source node or the closest node to the source node; otherwise, there is a closer node. At the third iteration the third-closest node must be the neighbor of the source node or the first two closest nodes, and so on. Thus at the kth iteration, the algorithm will have determined the k closest nodes from the source node.

The algorithm can be implemented by maintaining a set N of *permanently labeled nodes,* which consists of those nodes whose shortest paths have been determined. At each iteration the next-closest node is added to the set N and the distance to the remaining nodes via the new node is evaluated. To formalize the algorithm, let us define D_i to be the current minimum cost from the source node (labeled s) to node i. Dijkstra's algorithm can be described as follows:

1. Initialization:

$$N = \{s\} \tag{7.13}$$
$$D_j = C_{sj}, \quad \text{for all } j \neq s \tag{7.14}$$
$$D_s = 0 \tag{7.15}$$

2. Finding the next closest node: Find node $i \notin N$ such that

$$D_i = \min_{j \notin N} D_j \tag{7.16}$$

Add i to N.
If N contains all the nodes, stop.

3. Updating minimum costs after node i added to N: For each node $j \notin N$

$$D_j = \min\{D_j, D_i + C_{ij}\} \tag{7.17}$$

Go to step 2.

EXAMPLE **Finding the Shortest Path**

Using Figure 7.30, apply Dijkstra's algorithm to find the shortest paths from the source node (assumed to be node 1) to all the other nodes.

Table 7.6 shows the execution of Dijkstra's algorithm at the end of the initialization and each iteration. At each iteration the value of the minimum cost of the next closest node is underlined. In case of a tie, the closest node can be chosen randomly. The minimum cost for each node not permanently labeled is then updated sequentially. The last row records the minimum cost to each node.

TABLE 7.6 Execution of Dijkstra's algorithm.

Iteration	N	D_2	D_3	D_4	D_5	D_6
Initial	{1}	3	2	5	∞	∞
1	{1,3}	3	2	4	∞	3
2	{1,2,3}	3	2	4	7	3
3	{1,2,3,6}	3	2	4	5	3
4	{1,2,3,4,6}	3	2	4	5	3
5	{1,2,3,4,5,6}	3	2	4	5	3

If we also keep track of the predecessor node of the next-closest node at each iteration, we can obtain a shortest-path tree rooted at node 1, such as shown in Figure 7.34. When the algorithm stops, it knows the minimum cost to each node and the next node along the shortest path. For a datagram network, the routing table at node 1 looks like Table 7.7.

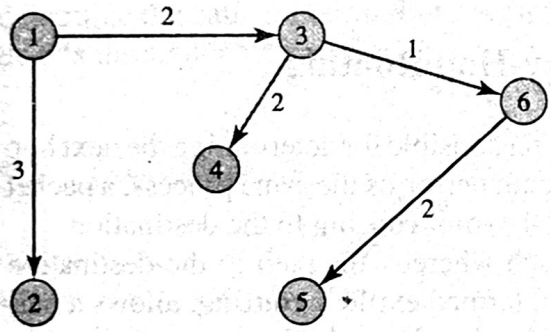

FIGURE 7.34 Shortest-path tree from node 1 to other nodes.

The data in the table is obtained as follows (again we encourage the reader to develop the results in Table 7.6 and Table 7.7, before reading on):

- (Iteration 1) Node 1 compares its costs to its directly attached nodes and finds that node 3 is the closest with a cost of 2. Node 3 is added to the set N. The tree diagram leading to Figure 7.34 is started by linking node 1 to node 3. The new minimum costs from node 1 to other nodes via node 3 are determined. It is found that a new shortest path to node 4 is through node 3 with cost of 4 so the third entry in iteration 1 is changed. It is also determined that node 6 can be reached through node 3 at a cost of 3.
- (Iteration 2) Node 2 and node 6 are tied for the second closest from node 1. Suppose that node 2 is selected so it is added to the set N. The tree diagram is updated by

connecting node 1 to node 2. The new minimum costs from node 1 are now calculated for the paths through node 2. It is found that node 5 has a cost of 7 through node 2.

- (Iteration 3) Node 6 is next added to N and connected in the tree diagram to node 1 via node 3. A new shortest path to node 5 through node 6 is found with a cost of 5.
- (Iteration 4) Node 4 is found to be the fourth closest node from node 1. It is connected to node 1 via node 3. No new shortest path is found through node 4.
- (Iteration 5) Node 5 if finally added to N with a cost of 5. Node 5 is connected to node 1 via nodes 3 and 6. This completes the algorithm.

TABLE 7.7 Routing table at node 1 for Figure 7.34.

Destination	Next node	Cost
2	2	3
3	3	2
4	3	4
5	3	5
6	3	3

To calculate the shortest paths, Dijkstra's algorithm requires the costs of all links to be available to the algorithm. Thus these link values must be communicated to the processor that is carrying out the computation. The *link-state protocol* uses this approach to calculate shortest paths.

7.5.3 Source Routing versus Hop-by-Hop Routing

In the datagram network, typically each node is responsible for determining the next hop along the shortest path. If each node along the path performs the same process, a packet traveling from the source is said to follow **hop-by-hop routing** to the destination.

Source routing is another routing approach whereby the path to the destination is determined by the source. Another variation, termed **explicit routing**, allows a particular node (not necessarily the source) to determine the path. Source routing works in either datagram or virtual-circuit packet switching. Before the source can send a packet, the source has to know the path to the destination in order to include the path information in the packet header. The path information contains the sequence of nodes to traverse and should give the intermediate node sufficient information to forward the packet to the next node until the packet reaches the destination. Figure 7.35 shows how source routing works in a datagram network.

Each node examines the header, strips off the address identifying the node, and forwards the packet to the next node. The source (host A) initially includes the entire path (1, 3, 6, B) in the packet to be destined to host B. Node 1 strips off its address and forwards the packet to the next node, which is node 3. The path specified in the header now contains 3, 6, B. Nodes 3 and 6 perform the same function until the packet reaches host B, which finally verifies that it is the intended destination