# 2

# Routing Algorithms: Shortest Path and Widest Path

*"If everybody minded their own business," the Duchess said in a hoarse growl, "the world would go round a deal faster than it does."*

**Lewis Carroll in *Alice in Wonderland***

***Reading Guideline***

Shortest path algorithms are applicable to IP networks and widest path algorithms are useful for telephone network dynamic call routing and quality-of-service-based routing. If you are primarily interested in learning about routing in IP networks, you may read material on shortest path routing algorithms, and then come back to read about widest path algorithms later. If you are interested in understanding routing in a voice over IP (VoIP) environment or a Multiprotocol Label Switching (MPLS) network, researching widest path routing is also recommended.

In this chapter, we will describe two classes of routing algorithms: shortest path routing and widest path routing. They appear in network routing in many ways and have played critical roles in the development of routing protocols. The primary focus of this chapter is to describe how they work, without discussing how they are used by a specific communication network, or in the context of routing protocols. These aspects will be addressed throughout this book.

## 2.1 Background

In general, a communication network is made up of nodes and links. Depending on the type of the network, nodes have different names. For example, in an IP network, a node is called a *router* while in the telephone network a node is either an *end (central) office* or a *toll switch*. In an optical network, a node is an *optical or electro-optical switch*. A link connects two nodes; a link connecting two routers in an IP network is sometimes called an *IP trunk* or simply an IP link, while the end of a link outgoing from a router is called an *interface*. A link in a telephone network is called a *trunkgroup*, or an *intermachine trunk (IMT)*, and sometimes simply a *trunk*.

We first briefly discuss a few general terms. A communication network carries traffic where traffic flows from a *start* node to an *end* node; typically, we refer to the start node as the *source* node (where traffic originates) and the end node as the *destination* node. Consider now the network shown in Figure 2.1. Suppose that we have traffic that enters node 1 destined for node 6; in this case, node 1 is the source node and node 6 is the destination node. We may also have traffic from node 2 to node 5; for this case, the source node will be node 2 and the destination node will be node 5; and so on.

An important requirement of a communication network is to flow or *route* traffic from a source node to a destination node. To do that we need to determine a route, which is a path from the source node to the destination node. A route can certainly be set up manually; such a route is known as a *static route*. In general, however, it is desirable to use a routing *algorithm* to determine a route. The goal of a routing algorithm is in general dictated by the requirement of the communication network and the service it provides as well as any additional or specific goals a service provider wants to impose on itself (so that it can provide a

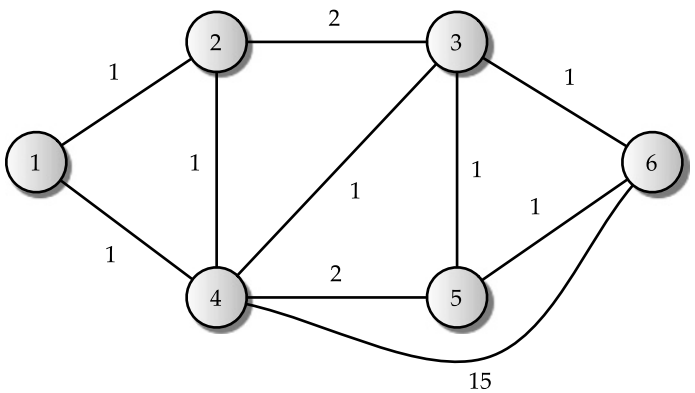



**FIGURE 2.1** A six-node network (the entry shown next to a link is the cost of the link).

better service compared to another provider). While goals can be different for different types of communication networks, they can usually be classified into two broad categories: user-oriented and network-oriented. User-oriented means that a network needs to provide good service to each user so that traffic can move from the source to the destination quickly for this user. However, this should not be for a specific user at the expense of other users between other source–destination nodes in the network. Thus, a network's goal ("network-oriented") generally is to address how to provide an efficient and fair routing so that *most* users receive good and acceptable service, instead of providing the "best" service to a specific user. Such a view is partly required because there are a finite amount of resources in a network, e.g., network capacity.

We next consider two very important algorithms that have profound impact on data networks, in particular on Internet routing. These two algorithms, known as the *Bellman–Ford algorithm* and *Dijkstra's algorithm*, can be classified as falling under user-oriented in terms of the above broad categories. They are both called shortest path routing algorithms, i.e., an algorithm where the goal is to find the shortest path from a source node to a destination node. A simple way to understand a shortest path is from road networks where shortest can be defined in terms of distance, for example, as in what is the shortest distance between two cities, which consists of the link distance between appropriate intermediate places between the end cities. However, it is possible that notions other than the usual distance-based measure may be applicable as well, for instance, time taken to travel between two cities. In other words, an equally interesting question concerns the shortest route between two cities in terms of *time*. This means that the notion of distance need not always be in terms of physical distance; it can be in other measures such as time.

Instead of worrying about the unit of measurement, it is better to have an algorithm that works *independent* of the measuring unit and considers a generic measure for distance for each link in a network. In communication networks, a generic term to refer to a distance measure without assigning any measure units is called *cost*, *link cost*, *distance cost*, or *link metric*. Consider again Figure 2.1. We have assigned a value with each link, e.g., link 4-6 has the value 15; we will say that the link cost, or distance cost, or link metric of link 4-6 is 15. No measuring units are used; for example, in road networks, it could be in miles, kilometers, or minutes. By simple inspection, it is not hard to see that the shortest path between nodes 1 and 6 is the path 1-4-3-6 with a total minimum cost of 3. It may be noted that the shortest path in this case did not include the link 4-6, although from the viewpoint of the number of nodes visited, it would look like the path 1-4-6 is the shortest path between nodes 1 and 6. In fact, this would be the case if the link cost was measured in terms of nodes visited, or *hops*. In other words, if the number of hops is important for measuring distance for a certain network, we can then think about the network in Figure 2.1 by considering the link cost for each link to be 1 instead of the number shown on each link in the figure. Regardless, having an algorithm that works without needing to worry about how cost is assigned to each link is helpful; this is where the Bellman–Ford and Dijkstra's algorithms both fit in.

At this point, it is important to point out that in computing the shortest path, the *additive* property is generally used for constructing the overall distance of a path by adding a cost of a link to the cost of the next link along a path until all links for the path are considered, as we have illustrated above. Thus, we will first start with this property for shortest path routing in describing the Bellman–Ford and Dijkstra's algorithms, and their

variants. You will see later that it is possible to define distance cost between two nodes in terms of *nonadditive* concave properties to determine the widest path (for example, refer to Section 2.7, 2.8, 10.9.2, 17.3, or 19.2.2). To avoid confusion, algorithms that use a non-additive concave property will be generally referred to as *widest* path routing algorithms.

We conclude this section by discussing the relation between a network and a *graph*. A network can be expressed as a graph by mapping each node to a unique vertex in the graph where links between network nodes are represented by edges connecting the corresponding vertices. Each edge can carry one or more weights; such weights may depict cost, delay, bandwidth, and so on. Figure 2.1 depicts a network consisting of a graph of six nodes and ten links where each link is assigned a link cost/weight.

## 2.2 Bellman–Ford Algorithm and the Distance Vector Approach

The Bellman–Ford algorithm uses a simple idea to compute the shortest path between two nodes in a centralized fashion. In a distributed environment, a distance vector approach is taken to compute shortest paths. In this section, we will discuss both the centralized and the distributed approaches.

### 2.2.1 Centralized View: Bellman–Ford Algorithm

To discuss the centralized version of the Bellman–Ford algorithm, we will use two generic nodes, labeled as node $i$ and node $j$, in a network of $N$ nodes. They may be directly connected as a link such as link 4-6 with end nodes 4 and 6 (see Figure 2.1). As can be seen from Figure 2.1, many nodes are not directly connected, for example, nodes 1 and 6; in this case, to find the distance between these two nodes, we need to resort to using other nodes and links. This brings us to an important point; we may have the notion of cost between two nodes, irrespective of whether they are directly connected or not. Thus, we introduce two important notations:

$d_{ij}$ = Link cost between nodes $i$ and $j$

$\overline{D}_{ij}$ = Cost of the computed minimum cost path from node $i$ to node $j$.

Since we are dealing with different algorithms, we will use overbars, underscores, and hats in our notations to help distinguish the computation for different classes of algorithms. For example, overbars are used for all distance computation related to the Bellman–Ford algorithm and its variants. Note that these and other notations used throughout this chapter are summarized later in Table 2.5.

If two nodes are directly connected, then the link cost $d_{ij}$ takes a finite value. Consider again Figure 2.1. Here, nodes 4 and 6 are directly connected with link cost 15; thus, we can write $d_{46} = 15$. On the other hand, nodes 1 and 6 are not directly connected; thus, $d_{16} = \infty$. What then is the difference between $d_{ij}$ and the minimum cost $\overline{D}_{ij}$? From nodes 4 to 6, we see that the minimum cost is actually 2, which takes path 4-3-6; that is, $\overline{D}_{46} = 2$ while $d_{46} = 15$. For nodes 1 and 6, we find that $\overline{D}_{16} = 3$ while $d_{16} = \infty$. As can be seen, a minimum cost path can be obtained between two nodes in a network regardless of whether they are directly

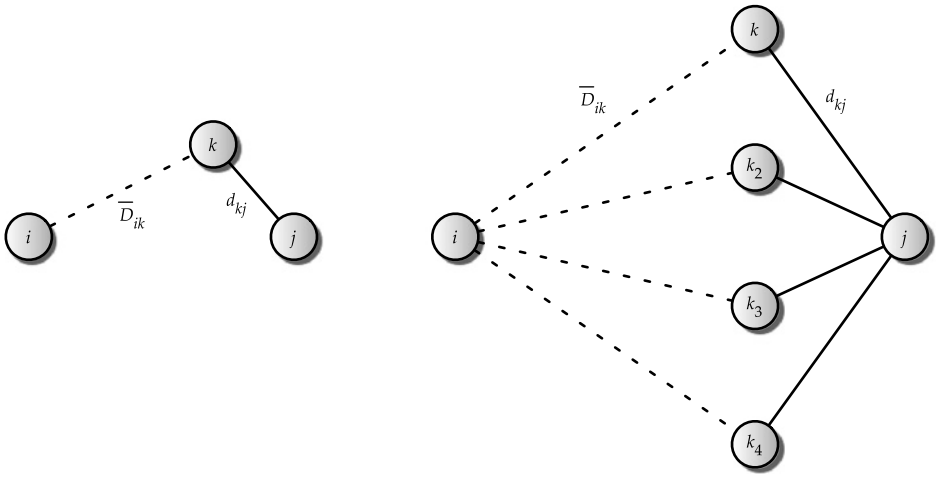



**FIGURE 2.2** Centralized Bellman–Ford Algorithm (solid line denotes a direct link; dashed line denotes distance).

connected or not, as long as one of the end nodes is not completely isolated from the rest of the network.

The question now is how to compute the minimum cost between two nodes in a network. This is where shortest path algorithms come in. To discuss such an algorithm, it is clear from the six-node example that we also need to rely on intermediate nodes. For that, consider a generic node $k$ in the network that is directly connected to either of the end nodes; we assume that $k$ is directly connected to the destination node $j$, meaning $d_{kj}$ has a finite value. The following equations, known as Bellman's equations, must be satisfied by the shortest path from node $i$ to node $j$:

$$\overline{D}_{ii} = 0, \quad \text{for all } i, \tag{2.2.1a}$$

$$\overline{D}_{ij} = \min_{k \neq j} \{\overline{D}_{ik} + d_{kj}\}, \quad \text{for } i \neq j. \tag{2.2.1b}$$

Simply put, Eq. (2.2.1b) states that for a pair of nodes $i$ and $j$, the minimum cost is dependent on knowing the minimum cost from $i$ to $k$ and the direct link cost $d_{kj}$ for link $k$-$j$. A visual is shown in Figure 2.2. Note that there can be multiple nodes $k$ that can be directly connected to the end node $j$ (say they are marked $k$, $k_2$, and so on; note that $k = i$ is not ruled out either); thus, we need to consider all such $k$s to ensure that we can compute the minimum cost. It is important to note that technically, a node $k$ that is not directly connected to $j$ is also considered; since for such $k$, we have $d_{kj} = \infty$, the resulting minimum computation is not impacted. On close examination, note that Eq. (2.2.1b) assumes that we know the minimum cost, $\overline{D}_{ik}$, from node $i$ to $k$ first somehow! Thus, in an actual algorithmic operation, a slight variation of Eq. (2.2.1b) is used where the minimum cost is accounted for by iterating through the number of hops. Specifically, we define the term for the minimum cost in terms of number

of hops $h$ as follows:

$$\overline{D}_{ij}^{(h)} = \text{ cost of the minimum cost path from node } i \text{ to node } j \text{ when up to } h \text{ number of hops are considered.}$$

The Bellman–Ford algorithm that iterates in terms of number of hops is given in Algorithm 2.1. Note the use of $(h)$ in the superscript in Eq. (2.2.2c); while the expression on the right side is up to $h$ hops, with the consideration of one more hop, the expression on the left hand side is now given in $h+1$ hops.

ALGORITHM 2.1 **Bellman–Ford centralized algorithm.**

Initialize for nodes $i$ and $j$ in the network:

$$\overline{D}_{ii}^{(0)} = 0, \quad \text{for all } i; \qquad \overline{D}_{ij}^{(0)} = \infty, \quad \text{for } i \neq j. \tag{2.2.2a}$$

For $h = 0$ to $N-1$ do

$$\overline{D}_{ii}^{(h+1)} = 0, \quad \text{for all } i \tag{2.2.2b}$$

$$\overline{D}_{ij}^{(h+1)} = \min_{k \neq j} \left\{ \overline{D}_{ik}^{(h)} + d_{kj} \right\}, \quad \text{for } i \neq j. \tag{2.2.2c}$$

For the six-node network (Figure 2.1), the Bellman–Ford algorithm is illustrated in Table 2.1. A nice way to understand the hop-iterated Bellman–Ford approach is to visualize through an example. Consider finding the shortest path from node 1 to node 6 as the number of hops increases. When $h = 1$, it means considering a direct link path between 1 and 6; since there is none, $D_{16}^{(1)} = \infty$. With $h = 2$, the path 1-4-6 is the only one possible since this is a two-link path, i.e., it uses two hops, consisting of the links 1-4 and 4-6; in this case, the hop-iterated minimum cost is 16 ($= D_{16}^{(2)}$). At $h = 3$, we can write the Bellman–Ford step as follows (shown only for $k$ for which $d_{k6} < \infty$) since there are three possible paths that need to be considered:

$$\begin{aligned} k = 3: &\quad \overline{D}_{13}^{(2)} + d_{36} = 2 + 1 = 3 \\ k = 5: &\quad \overline{D}_{15}^{(2)} + d_{56} = 3 + 1 = 4 \\ k = 4: &\quad \overline{D}_{14}^{(2)} + d_{46} = 1 + 15 = 16. \end{aligned}$$

In this case, we pick the first one since the minimum cost is 3, i.e., $\overline{D}_{16}^{(3)} = 3$ with the shortest path 1-4-3-6. It is important to note that the Bellman–Ford algorithm computes only the minimum cost; it does not track the actual shortest path. We have included the shortest path in Table 2.1 for ease of understanding how the algorithm works. For many networking environments, it is not necessary to know the entire path; just knowing the next node $k$ for which the cost is minimum is sufficient—this can be easily tracked with the min operation in Eq. (2.2.2c).

**TABLE 2.1** Minimum cost from node 1 to other nodes using Algorithm 2.1.

| $h$ | $\overline{D}_{12}^{(h)}$ | Path | $\overline{D}_{13}^{(h)}$ | Path | $\overline{D}_{14}^{(h)}$ | Path | $\overline{D}_{15}^{(h)}$ | Path | $\overline{D}_{16}^{(h)}$ | Path |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | $\infty$ | – | $\infty$ | – | $\infty$ | – | $\infty$ | – | $\infty$ | – |
| 1 | 1 | 1-2 | $\infty$ | – | 1 | 1-4 | $\infty$ | – | $\infty$ | – |
| 2 | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 16 | 1-4-6 |
| 3 | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 3 | 1-4-3-6 |
| 4 | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 3 | 1-4-3-6 |
| 5 | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 3 | 1-4-3-6 |

### 2.2.2 Distributed View: A Distance Vector Approach

In a computer network, nodes need to work in a distributed fashion in determining the shortest paths to a destination. If we look closely at the centralized version discussed above, we note that a source node needs to know the cost of the shortest path to all nodes immediately prior to the destination, i.e., $\overline{D}_{ik}^{(h)}$, so that the minimum cost to the destination can be computed; this is the essence of Eq. (2.2.2c), communicated through Figure 2.2. This view of the centralized Bellman–Ford algorithm is not directly suitable for a distributed environment. On the other hand, we can consider an important rearrangement in the minimum cost computation to change the view. That is, what if we change the *order* of consideration in Eq. (2.2.1b) and instead use the minimum cost computation step as follows:

$$\overline{D}_{ij} = \min_{k \neq i} \{ d_{ik} + \overline{D}_{kj} \}, \quad \text{for } i \neq j. \tag{2.2.3}$$

Note the subtle, yet distinctive difference between Eq. (2.2.1b) and Eq. (2.2.3); here, we first look at the outgoing link out of node $i$ to a directly connected node $k$ with link cost $d_{ik}$, and then consider the minimum cost $\overline{D}_{kj}$ from $k$ to $j$ without knowing how $k$ determined this value. The list of directly connected nodes of $i$, i.e., neighbors of $i$, will be denoted by $\mathcal{N}_i$. In essence, what we are saying is that if node $i$ finds out from its neighbor the cost of the minimum cost path to a destination, it can then use this information to determine cost to the destination by adding the outgoing link cost $d_{ik}$; this notion is known as the *distance vector approach*, first applied to the original ARPANET routing. With this approach, the computational step Eq. (2.2.3) has a nice advantage in that it helps in building a computational model for a distributed environment.

We illustrate the change of order and its advantage for the distributed environment using Figure 2.3. Suppose that node $i$ periodically receives the minimum cost information $\overline{D}_{kj}$ from its neighboring node $k$ for node $k$'s minimum cost to node $j$; this variation can be addressed by introducing the dependency on the time domain, $t$, using $\overline{D}_{kj}(t)$ for node $k$'s cost to $j$—this will then be available *to* node $i$ (compare this expression to hop-based $\overline{D}_{kj}^{(h)}$). Now, imagine for whatever reason that node $k$ recomputes its cost to $j$ and makes it available to another source node, say $i_2$, but not to node $i$ as shown in Figure 2.3. In other words, from the view of the source node $i$, the best we can say is that the minimum cost value from node $k$ to node $j$ that is available to node $i$ is as node $i$ has been able to receive; that is, it is more appropriate to use the term $\overline{D}_{kj}^{i}(t)$ than $\overline{D}_{kj}(t)$ to indicate that the minimum cost from node $k$ to node $j$, as

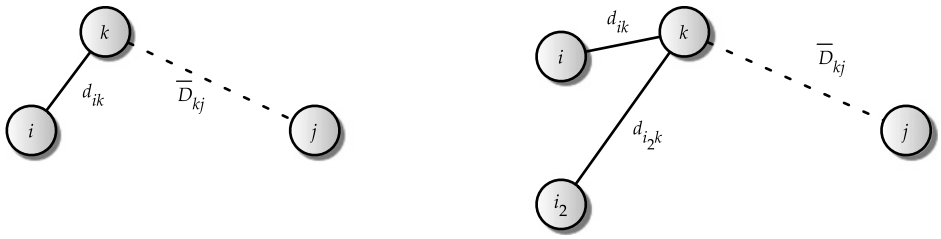



**FIGURE 2.3** Distance vector view for computing the shortest path.

*available* to source node $i$ at time $t$; to distinguish, the cost availability to $i_2$ can be written as $\overline{D}_{kj}^{i_2}(t)$ since $i_2$ may receive at a different time instant $t$. Furthermore, in a dynamic network, the direct link cost $d_{ik}$ from node $i$ to $k$ can also change with time $t$, for example, to reflect change in network load/traffic. Thus, we can generalize the direct link cost to write as $d_{ik}(t)$ to indicate dependency on time $t$. With these changes, we present the distributed distance vector approach in Algorithm 2.2.

ALGORITHM 2.2 **Distance vector algorithm (computed at node $i$).**

Initialize

$$\overline{D}_{ii}(t) = 0; \quad \overline{D}_{ij}(t) = \infty, \quad \text{(for node } j \text{ that node } i \text{ is aware of).} \tag{2.2.4a}$$

For (nodes $j$ that node $i$ is aware of) do

$$\overline{D}_{ij}(t) = \min_{k \text{ directly connected to } i} \left\{ d_{ik}(t) + \overline{D}_{kj}^{i}(t) \right\}, \quad \text{for } j \neq i. \tag{2.2.4b}$$

We will now illustrate the distributed variation. For simplicity, we will assume that node $k$, which is directly connected to node $i$, is sending $\overline{D}_{kj}^{i}(t)$ at the same instant to other directly connected nodes like $i$. Furthermore, we will assume that the direct link cost does not change with time, i.e., $d_{ik}(t)$ does not change with time.

Our illustration will consider the same six-node example by considering computing the shortest path cost from node 1 to node 6 (see Table 2.2). This time, we will interpret the hop-based cost computation in terms of discrete time windows; for example, $t = 0$ means what node 4 sees about cost to node 6 when zero hops away, $t = 1$ means what node 4 sees about cost to node 6 when information from one hop away is received, and so on. Note that node 1 is directly connected to node 2 and node 4. Thus, node 1 will have $\overline{D}_{26}^{1}(t)$, the cost between node 2 and node 6 from node 2, and $\overline{D}_{46}^{1}(t)$, the cost between node 4 and node 6 from node 4.

To summarize, in the distance vector approach, a node relies on its neighboring nodes' known cost to a destination to determine its best path. To do so, it does periodic computation as and when it receives information from its neighbor. For this entire process to work,

**TABLE 2.2** Distance vector based distributed computation at time $t$ from node 1 to node 6.

| Time, $t$ | $\overline{D}^1_{46}(t)$ | $\overline{D}^1_{26}(t)$ | Computation at node 1 $\min\{d_{14}(t)+\overline{D}^1_{46}(t), d_{12}(t)+\overline{D}^1_{26}(t)\}$ | $\overline{D}_{16}(t)$ |
|---|---|---|---|---|
| 0 | $\infty$ | $\infty$ | $\min\{1+\infty, 1+\infty\}$ | $\infty$ |
| 1 | 15 | $\infty$ | $\min\{1+15, 1+\infty\}$ | 16 |
| 2 | 2 | 3 | $\min\{1+2, 1+3\}$ | 3 |

the key idea is that a node $k$ needs to distribute its cost to $j$ given by $\overline{D}_{kj}(t)$ to all its directly connected neighbor $i$—the dependency on $i$ and $t$ means that each node $i$ may get such information potentially at a different time instant $t$. The difference between this idea and the centralized Bellman–Ford algorithm is subtle in that the order of computation along with the link considered in computation leads to different views to computing the shortest path.

## 2.3 Dijkstra's Algorithm

Dijkstra's algorithm is another well-known shortest path routing algorithm. The basic idea behind Dijkstra's algorithm is quite different from the Bellman–Ford algorithm or the distance vector approach. It works on the notion of a candidate neighboring node set as well as the source's own computation to identify the shortest path to a destination. Another interesting property about Dijkstra's algorithm is that it computes shortest paths to all destinations from a source, instead of just for a specific pair of source and destination nodes at a time—which is very useful, especially in a communication network, since a node wants to compute the shortest path to all destinations.

### 2.3.1 Centralized Approach

Consider a generic node $i$ in a network of $N$ nodes from where we want to compute shortest paths to all other nodes in the network. The list of $N$ nodes will be denoted by $\mathcal{N} = \{1, 2, \ldots, N\}$. A generic destination node will be denoted by $j$ ($j \neq i$). We will use the following two terms:

$d_{ij}$ = link cost between node $i$ and node $j$

$\underline{D}_{ij}$ = cost of the minimum cost path between node $i$ and node $j$.

Note that to avoid confusing this with the computation related to the Bellman–Ford algorithm or the distance vector approach, we will be using *underscores* with uppercase $D$, as in $\underline{D}_{ij}$, for the cost of the path between nodes $i$ and $j$ in Dijkstra's algorithm.

Dijkstra's algorithm divides the list $\mathcal{N}$ of nodes into two lists: it starts with permanent list $\mathcal{S}$, which represents nodes already considered, and tentative list $\mathcal{S}'$, for nodes not considered yet. As the algorithm progresses, list $S$ expands with new nodes included while list $\mathcal{S}'$ shrinks when nodes newly included in $\mathcal{S}$ are deleted from this list; the algorithm stops when

ALGORITHM 2.3 **Dijkstra's shortest path first algorithm (centralized approach).**

1. Start with source node $i$ in the permanent list of nodes considered, i.e., $\mathcal{S} = \{i\}$; all the rest of the nodes are put in the tentative list labeled as $\mathcal{S}'$. Initialize

$$\underline{D}_{ij} = d_{ij}, \quad \text{for all } j \in \mathcal{S}'.$$

2. Identify a neighboring node (intermediary) $k$ not in the current list $\mathcal{S}$ with the minimum cost path from node $i$, i.e., find $k \in \mathcal{S}'$ such that $\underline{D}_{ik} = \min_{m \in \mathcal{S}'} \underline{D}_{im}$.

   Add $k$ to the permanent list $\mathcal{S}$, i.e., $\mathcal{S} = \mathcal{S} \cup \{k\}$,

   Drop $k$ from the tentative list $\mathcal{S}'$, i.e., $\mathcal{S}' = \mathcal{S}' \backslash \{k\}$.

   If $\mathcal{S}'$ is empty, stop.

3. Consider the list of neighboring nodes, $\mathcal{N}_k$, of the intermediary $k$ (but do not consider nodes already in $\mathcal{S}$) to check for improvement in the minimum cost path, i.e., for $j \in \mathcal{N}_k \cap \mathcal{S}'$

$$\underline{D}_{ij} = \min\{\underline{D}_{ij}, \underline{D}_{ik} + d_{kj}\}. \tag{2.3.1}$$

   Go to Step 2.

list $\mathcal{S}'$ becomes empty. Initially, we have $\mathcal{S} = \{i\}$ and $\mathcal{S}' = \mathcal{N} \backslash \{i\}$ (i.e., all nodes in $\mathcal{N}$ except node $i$).

The core of the algorithm has two parts: (1) how to expand the list $\mathcal{S}$, and (2) how to compute the shortest path to nodes that are neighbors of nodes of list $\mathcal{S}$ (but nodes not in this list yet). List $\mathcal{S}$ is expanded at each iteration by considering a neighboring node $k$ of node $i$ with the least cost path from node $i$. At each iteration, the algorithm then considers the neighboring nodes of $k$, which are not already in $S$, to see if the minimum cost changes from the last iteration.

We will illustrate Dijkstra's algorithm using the network given in Figure 2.1. Suppose that node 1 wants to find shortest paths to all other nodes in the network. Then, initially, $\mathcal{S} = \{1\}$, and $\mathcal{S}' = \{2, 3, 4, 5, 6\}$, and the shortest paths to all nodes that are direct neighbors of node 1 can be readily found while for the rest, the cost remains at $\infty$, i.e.,

$$\underline{D}_{12} = 1, \underline{D}_{14} = 1, \quad \underline{D}_{13} = \underline{D}_{15} = \underline{D}_{16} = \infty.$$

For the next iteration, we note that node 1 has two directly connected neighbors: node 2 and node 4 with $d_{12} = 1$ and $d_{14} = 1$, respectively; all the other nodes are not directly connected to node 1, and thus, the "direct" cost to these nodes remains at $\infty$. Since both nodes 2 and 4 are neighbors with the same minimum cost, we can pick either of them to break the tie. For our illustration, we pick node 2, and this node becomes intermediary, $k$. Thus, we now have $\mathcal{S} = \{1, 2\}$, and $\mathcal{S}'$ becomes the list $\{3, 4, 5, 6\}$. Then, we ask node 2 for cost to its direct

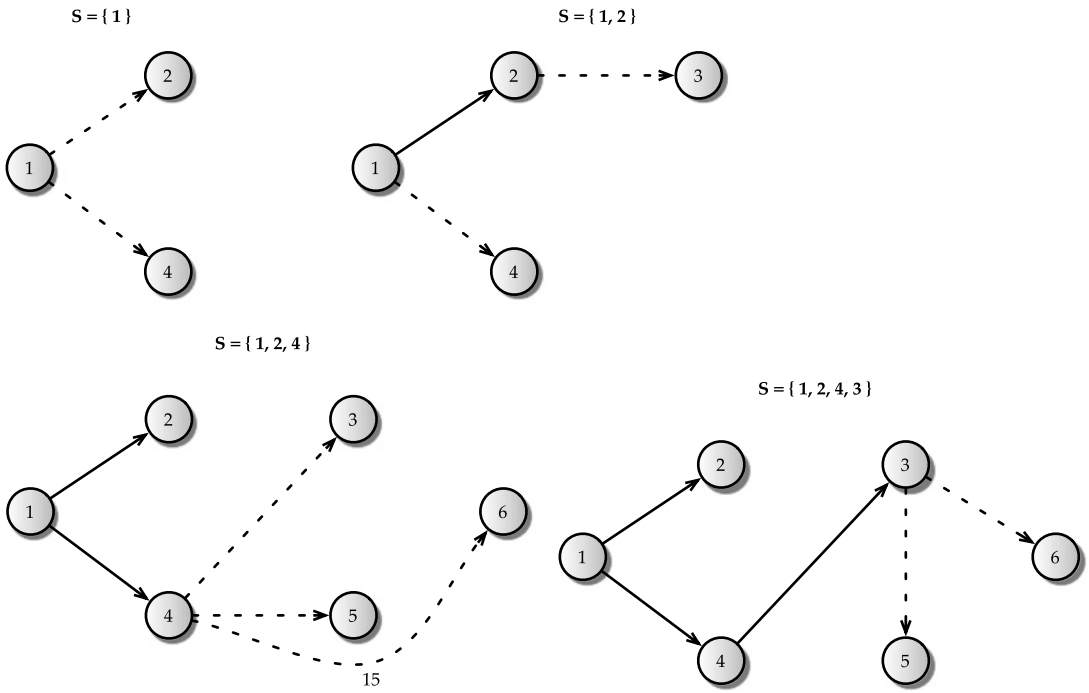



**FIGURE 2.4** Iterative view of Dijkstra's algorithm.

neighbors not already in set $\mathcal{S}$. We can see from Figure 2.1 that node 2's neighbors are node 3 and node 4. Thus, we compare and compute cost from node 1 for these two nodes, and see if there is any improvement:

$$\underline{D}_{13} = \min\{\underline{D}_{13}, \underline{D}_{12} + d_{23}\} = \min\{\infty, 1+2\} = 3$$
$$\underline{D}_{14} = \min\{\underline{D}_{14}, \underline{D}_{12} + d_{24}\} = \min\{1, 1+1\} = 1.$$

Note that there is no improvement in cost to node 4; thus, we keep the original shortest path. For node 3, we now have a shortest path, 1-2-3. For the rest of the nodes, the cost remains at $\infty$. This completes this iteration. We then move to the next iteration and find that the next intermediary is $k = 4$, and the process is continued as before. In Table 2.3, we summarize all the steps until all nodes are considered in list $\mathcal{S}$, and in Figure 2.4, we give a visual illustration to how the algorithm adds a new intermediary $k$ to list $\mathcal{S}$. The centralized version of Dijkstra's algorithm is formally presented in Algorithm 2.3.

### 2.3.2 Distributed Approach

The distributed variant of Dijkstra's algorithm is very similar to the centralized version. The main difference is that the link cost of a link received by one node could be different from another node since this information is disseminated in an asynchronous manner. Thus, we

**TABLE 2.3** Iterative steps in Dijkstra's algorithm.

| Iteration | List, $\mathcal{S}$ | $\underline{D}_{12}$ | Path | $\underline{D}_{13}$ | Path | $\underline{D}_{14}$ | Path | $\underline{D}_{15}$ | Path | $\underline{D}_{16}$ | Path |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | {1} | 1 | 1-2 | $\infty$ | – | 1 | 1-4 | $\infty$ | – | $\infty$ | – |
| 2 | {1, 2} | 1 | 1-2 | 3 | 1-2-3 | 1 | 1-4 | $\infty$ | – | $\infty$ | – |
| 3 | {1, 2, 4} | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 16 | 1-4-6 |
| 4 | {1, 2, 4, 3} | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 3 | 1-4-3-6 |
| 5 | {1, 2, 4, 3, 5} | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 3 | 1-4-3-6 |
| 6 | {1, 2, 4, 3, 5, 6} | 1 | 1-2 | 2 | 1-4-3 | 1 | 1-4 | 3 | 1-4-5 | 3 | 1-4-3-6 |

denote the cost of link *k*-*m* as received at node *i* at time *t* by $d^i_{km}(t)$. Similarly, the minimum distance from *i* to *j* is time-dependent and is denoted by $\underline{D}_{ij}(t)$.

Dijkstra's algorithm for the distributed environment is presented in Algorithm 2.4. The steps are similar to the centralized version. Thus, in the distributed version, it is really the communication of the link cost information in a distributed manner that is taken into account by the algorithm. The steps are the same as in Table 2.3—this time we can think of the iterative

ALGORITHM 2.4 **Dijkstra's shortest path first algorithm (a distributed approach).**

1. Discover nodes in the network, $\mathcal{N}$, and cost of link *k*-*m*, $d^i_{km}(t)$, as known to node *i* at the time of computation, *t*.
2. Start with source node *i* in the permanent list of nodes considered, i.e., $\mathcal{S} = \{i\}$; all the rest of the nodes are put in the tentative list labeled as $\mathcal{S}'$. Initialize

$$\underline{D}_{ij}(t) = d^i_{ij}(t), \quad \text{for all } j \in \mathcal{S}'.$$

3. Identify a neighboring node (intermediary) $k$ not in the current list $\mathcal{S}$ with the minimum cost path from node *i*, i.e., find $k \in \mathcal{S}'$ such that $\underline{D}_{ik}(t) = \min_{m \in \mathcal{S}'} \underline{D}_{im}(t)$.

   Add $k$ to the permanent list $\mathcal{S}$, i.e., $\mathcal{S} = \mathcal{S} \cup \{k\}$,

   Drop $k$ from the tentative list $\mathcal{S}'$, i.e., $\mathcal{S}' = \mathcal{S}' \backslash \{k\}$.

   If $\mathcal{S}'$ is empty, stop.

4. Consider neighboring nodes $\mathcal{N}_k$ of the intermediary $k$ (but do not consider nodes already in $\mathcal{S}$) to check for improvement in the minimum cost path, i.e.,
   for $j \in \mathcal{N}_k \cap \mathcal{S}'$

$$\underline{D}_{ij}(t) = \min\{\underline{D}_{ij}(t), \underline{D}_{ik}(t) + d^i_{kj}(t)\}. \tag{2.3.2}$$

   Go to Step 3.

ALGORITHM 2.5 **Dijkstra's shortest path first algorithm (with tracking of next hop).**

```
0   // Computation at time t
1   S = {i}    // permanent list; start with source node i
2   S' = N \ {i}    // tentative list (of the rest of the nodes)
3   for (j in S') do
4     if (d_ij^i(t) < ∞) then    // if i is directly connected to j
5       D_ij(t) = d_ij^i(t)
6       H_ij = j   // set i's next hop to be j
7     else
8       D_ij(t) = ∞
9       H_ij = −1       // next hop not set
10    endif
11  endfor
12  while (S' is not empty) do    // while tentative list is not empty
13    Dtemp = ∞    // find minimum cost neighbor k
14    for (m in S') do
15      if (D_im(t) < Dtemp) then
16        Dtemp = D_im(t)
17        k = m
18      endif
19    endfor
20    S = S ∪ {k}   // add to permanent list
21    S' = S'\{k}   // delete from tentative list
22    for (j in N_k ∩ S') do
23      if (D_ij(t) > D_ik(t) + d_kj^i(t)) then    // if cost improvement via k
24        D_ij(t) = D_ik(t) + d_kj^i(t)
25        H_ij = H_ik    // next hop for destination j; inherit from k
26      endif
27    endfor
28  endwhile
```

steps as the increment in time in terms of learning about different links and link costs in the network.

Determination of the next hop is important in many networking environments; next hop refers to the next directly connected node that the source node $i$ should go to for reaching a destination $j$; ideally, the next hop should be on the optimal path. In Algorithm 2.5, we present a somewhat formal version of Dijkstra's algorithm—the purpose is to highlight the logic conditions for the benefit of the interested reader. In this algorithm, we have also included another identifier $H_{ij}$ to track the next hop from $i$ for destination $j$. Finally, in many situations, the shortest path to a specific destination $j$, instead of being to all destinations, is sufficient to compute. This can be easily incorporated in Algorithm 2.5 by inserting the following operation between line 19 and line 20: "if ($k$ is same as destination $j$), then *exit* the while loop;" this means that we have found the shortest path to destination $j$.

## 2.4 Comparison of the Bellman–Ford Algorithm and Dijkstra's Algorithm

This is a good time to do a quick comparison between Dijksta's algorithm (Algorithm 2.3) and the Bellman–Ford algorithm (Algorithm 2.1). First, the Bellman–Ford algorithm com-

putes the shortest path to one destination at a time while Dijksta's algorithm computes the shortest paths to all destinations (sometimes called the shortest path tree). When we compare the minimum cost computation for each algorithm, i.e., between Eq. (2.2.1b) and Eq. (2.3.1), it may seem that they are similar. However, there are actually very important subtle differences. In both of them, there is an intermediary node $k$; in the case of the Bellman–Ford algorithm, node $k$ is over *all* nodes to find the best next hop to node $j$, while in the case of Dijkstra's algorithm, node $k$ is an intermediary first determined and fixed, and then the shortest path computation is done to all $j$, not already covered. Table 2.1 and Table 2.3 are helpful in understanding this difference.

Since there are various operations in an algorithm, it is helpful to know the computational complexity (see Appendix B.3) so that a comparison of two or more algorithms can be done in terms of computational complexity using the "big-$O$" notation. Given $N$ as the total number of nodes and $L$ as the total number of links, the computational complexity of the Bellman–Ford algorithm is $O(LN)$. The complexity of Dijkstra's algorithm is $O(N^2)$ but can be improved to $O(L + N \log N)$ using a good data structure. Note that if a network is fully connected, the number of bidirectional links is $N(N-1)/2$; thus, for a fully connected network, the complexity of the Bellman–Ford algorithm is $O(N^3)$ while for Dijkstra's algorithm, it is $O(N^2)$.

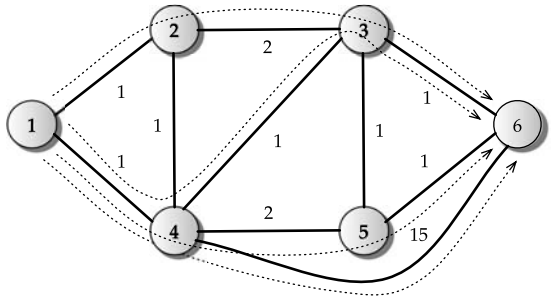
Two key routing protocol concepts, the distance vector protocol concept and the link-state protocol concept, have fairly direct relation to the Bellman–Ford algorithm (or the distance vector-based shortest path computation approach) and Dijkstra's algorithm, respectively. These two key routing protocol concepts will be discussed later in Sections 3.3 and 3.4.

## 2.5 Shortest Path Computation with Candidate Path Caching

We will next deviate somewhat from the Bellman–Ford algorithm and Dijkstra's algorithm. There are certain networking environments where a list of possible paths is known or determined ahead of time; such a path list will be referred to as the *candidate* path list. *Path caching* refers to storing of a candidate path list at a node ahead of time. If through a distributed protocol mechanism the link cost is periodically updated, then the shortest path computation at a node becomes very simple when the candidate path list is already known.

Consider again the six-node network shown in Figure 2.1. Suppose that node 1 somehow *knows* that there are four paths available to node 6 as follows: 1-2-3-6, 1-4-3-6, 1-4-5-6, and 1-4-6; they are marked in Figure 2.5.

Using the link cost, we can then compute path cost for each path as shown in the table in Figure 2.5. Now, if we look for the least cost path, we will find that path 1-4-3-6 is the most preferred path due to its lowest end-to-end cost. Suppose now that in the next time period, the link cost for link 4-3 changes from 1 to 5. If we know the list of candidate paths, we can then recompute the path cost and find that path 1-4-3-6 is no longer the least cost; instead, both 1-2-3-6 and 1-4-5-6 are now the shortest paths—either can be chosen based on a tie-breaker rule.




| Path | Cost |
|---|---|
| 1-2-3-6 | $d_{12}+d_{23}+d_{36}=4$ |
| 1-4-3-6 | $d_{14}+d_{43}+d_{36}=3$ |
| 1-4-5-6 | $d_{14}+d_{45}+d_{56}=4$ |
| 1-4-6 | $d_{14}+d_{46}=16$ |

**FIGURE 2.5** Paths identified from node 1 to node 6, along with associated path cost.

We will now write the shortest path calculation in the presence of path caching in a generic way, considering that this calculation is done at time $t$. We consider a candidate path $p$ between nodes $i$ and node $j$, and its cost at time $t$ as

$$\widehat{D}_{ij/p}(t) = \sum_{\text{link } l\text{-}m \text{ in path } p} d^i_{lm}(t), \tag{2.5.1}$$

where $d^i_{lm}(t)$ is the cost of link $l$-$m$ at time $t$ as known to node $i$, and the summation is over all such links that are part of path $p$. The list of candidate paths for the pair $i$ and $j$ will be denoted by $\mathcal{P}_{ij}$; the best path will be identified by $\widehat{p}$. The procedure to compute the shortest path is given in Algorithm 2.6.

ALGORITHM 2.6 **Shortest path computation when candidate paths are known.**

*At source node i, a list of candidate paths $\mathcal{P}_{ij}$ to destination node j is available, and link cost, $d^i_{lm}(t)$, of link l-m at time t is known:*
// Initialize the least cost:
$\widehat{D}_{ij}(t) = \infty$
// Consider each candidate path in the list
for ($p$ in $\mathcal{P}_{ij}$) do
  $\widehat{D}_{ij/p}(t) = 0$
  for (link $l$-$m$ in path $p$) do  // add up cost of links for this path
  $\widehat{D}_{ij/p}(t) = \widehat{D}_{ij/p}(t) + d^i_{lm}(t)$ (2.5.2)
  end for
  if ($\widehat{D}_{ij/p}(t) < \widehat{D}_{ij}(t)$) then  // if this is cheaper, note it
    $\widehat{D}_{ij}(t) = \widehat{D}_{ij/p}(t)$
    $\widehat{p} = p$
  end if
end do

It is important to note that the candidate path list is not required to include all possible paths between node $i$ and $j$, only a sublist of paths that are, for some reason, preferable to consider for a particular networking environment. The way to think about this is to think of a

road network in a city where to go from your home to school/office, you are likely to use only a selected set of paths. In a communication network, this approach of computing the shortest path involves a trade-off between storage and time complexity. That is, by storing multiple candidate paths ahead of time, the actual computation is simple when new link costs are received. The set of candidate paths can be determined using, for example, the $K$-shortest path algorithm (see Algorithm 2.10); since the interest in the case of path caching is obtain a good working set, any reasonable link cost can be assumed; for example, we can set all link costs to 1 (known also as hop count) and use Algorithm 2.10 to obtain a set of $K$ candidate paths.

It is worth noting that such a candidate path-based approach can potentially miss a good path. For example, if a node is configured to keep only three candidate paths, it can potentially miss including 1-4-3-6; thus, in the first cycle of computation before the link cost $d_{43}$ for link 4-3 was updated, this path would not be chosen at all.

## 2.6 Widest Path Computation with Candidate Path Caching

So far, we have assumed that the shortest path is determined based on the additive cost property. There are many networking environments in which the additive cost property is not applicable; for example, dynamic call routing in the voice telephone network (refer to Chapter 10) and quality of service based routing (refer to Chapter 17). Thus, determining paths when the cost is nonadditive is also an important problem in network routing; an important class among the nonadditive cost properties is *concave* cost property that leads to widest path routing. We will first start with the case in which path caching is used, so that it is easy to transition and compare where and how the nonadditive concave case is different from the additive case described in the previous section.

Suppose a network link has a certain bandwidth available, sometimes referred to as *residual capacity*; to avoid any confusion, we will denote the available bandwidth by $b_{lm}$ for link *l-m*, as opposed to $d_{lm}$ for the additive case. Note that $b_{lm} = 0$ then means that the link is not feasible since there is no bandwidth; we can also set $b_{lm} = 0$ if there is no link between nodes $l$ and $m$ (compare this with $d_{lm} = \infty$ for the additive case). We start with a simple illustration. Consider a path between node 1 and node 2 consisting of three links: the first link has 10 units of bandwidth available, the second link has 5 units of bandwidth available, and the third link has 7 units of bandwidth available. Now, if we say the cost of this path is additive, i.e., $22(= 10 + 5 + 7)$, it is unlikely to make any sense. There is another way to think about it. Suppose that we have new requests coming in, each requiring a unit of *dedicated* bandwidth for a certain duration. What is the maximum number of requests this path can handle? It is easy to see that this path would be able to handle a maximum of five additional requests simultaneously since if it were more than five, the link in the middle in this case would not be able to handle more than five requests. That is, we arrive at the availability of the path by doing $\min\{10, 5, 7\} = 5$. Thus the path "cost" is 5; certainly, this is a strange definition of a path cost; it is easier to see this as the *width* of a path (see Figure 2.6). Formally, similar to Eq. (2.5.1), for all links *l-m* that make up a path $p$, we can write the width of the path as

$$\widehat{B}_{ij/p}(t) = \min_{\text{link } l\text{-}m \text{ in path } p} \left\{ b^i_{lm}(t) \right\}. \tag{2.6.1}$$


**FIGURE 2.6** Width of a path—a visual depiction.

Regardless, the important point to note is that this path cost is computed using a non-additive cost property, in this case the minimum function. It may be noted that the minimum function is not the only nonadditive cost property possible for defining cost of a path; there are certainly other possible measures, such as the nonadditive *multiplicative* property given by Eq. (B.8.1) discussed in Appendix B.8.

Now consider a list of candidate paths; how do we define the most preferable path? One way to define it is to find the path with the largest amount of available bandwidth. This is actually easy to do once the path "cost" for each path is determined since we can then take the maximum of all such paths. Consider the topology shown in Figure 2.7 with available bandwidth on each link as marked. Now consider three possible paths between node 1 and node 5:

| Path | Cost |
|---|---|
| 1-2-3-5 | $\min\{b_{12}, b_{23}, b_{35}\} = 10$ |
| 1-4-3-5 | $\min\{b_{14}, b_{43}, b_{35}\} = 15$ |
| 1-4-5 | $\min\{b_{14}, b_{45}\} = 20$ |

ALGORITHM 2.7 **Widest path computation (non-additive, concave) when candidate paths are known.**

*At source node i, a list of candidate paths $\mathcal{P}_{ij}$ to destination node j is available, and link bandwidth, $b^i_{lm}(t)$, of link l-m at time t is known:*
// Initialize the least bandwidth:
$\widehat{B}_{ij}(t) = 0$
for $p$ in $\mathcal{P}_{ij}$ do
  $\widehat{B}_{ij/p}(t) = \infty$
  for (link *l-m* in path $p$) do  // find bandwidth of the bottleneck link
    $\widehat{B}_{ij/p}(t) = \min\left\{\widehat{B}_{ij/p}(t), b^i_{lm}(t)\right\}$ (2.6.2)
  end for
  if $(\widehat{B}_{ij/p}(t) > \widehat{B}_{ij}(t))$ then  // if this has more bandwidth, note it
    $\widehat{B}_{ij}(t) = \widehat{B}_{ij/p}(t)$
    $\widehat{p} = p$
  end if
end do

It is easy to see that the third path, 1-4-5, has the most bandwidth and is thus the preferred path. This means that we need to do a maximum over all paths in the case of the nonadditive property to find the widest path as opposed to the minimum over all paths when additive cost property is used. A widest path so selected is sometimes referred to as the *maximal residual capacity* path. The procedure is presented in detail in Algorithm 2.7. It is helpful to contrast this algorithm with its counterpart, Algorithm 2.6, where the additive cost property was used; for example, you can compare Eq. (2.6.2) with Eq. (2.5.2), especially the logical "if" condition statements.

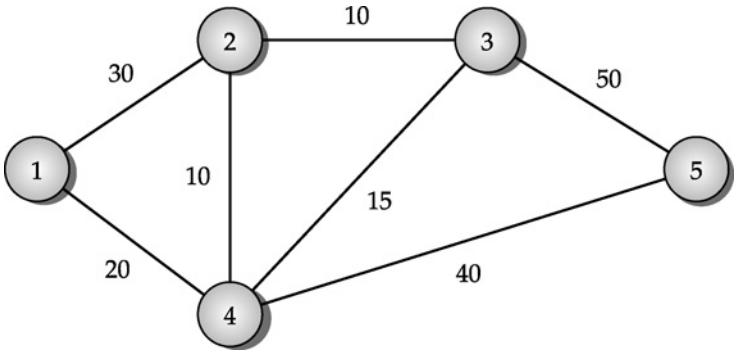



**FIGURE 2.7** Network example for widest path routing.

**Remark 2.1.** *Relation between shortest path routing and widest path routing.*

While the cost of a path is determined differently, by using the additive property in Eq. (2.5.1), and by the nonadditive property in Eq. (2.6.1), there is a direct relation between shortest and widest. If we imagine the path cost to be the *negative* of the quantity given in Eq. (2.6.1), then widest translates to being the minimum of this negative cost. Thus, the widest path is the cheapest path in the sense of this negative representation. In other words, the widest path can be thought of as the *nonadditive (concave) shortest path*. ♦

## 2.7 Widest Path Algorithm

We are coming back full circle to *no* path caching for widest path routing algorithms. We present two approaches: first we show an extension of Dijkstra's shortest path first algorithm; next, we extend the Bellman–Ford algorithm.

### 2.7.1 Dijkstra-Based Approach

When there is no path caching, the algorithm is very similar to Dijkstra's algorithm that is adapted from [731], and is listed in Algorithm 2.8.

Consider the network topology shown in Figure 2.7 where each link is marked with an available bandwidth. The algorithmic steps with Algorithm 2.8 are detailed in Table 2.4 for

ALGORITHM 2.8 **Widest path algorithm, computed at node $i$ (Dijkstra-based).**

1. Discover list of nodes in the network, $\mathcal{N}$ and available bandwidth of link $k$-$m$, $b^i_{km}(t)$, as known to node $i$ at the time of computation, $t$.
2. Initially, consider only source node $i$ in the set of nodes considered, i.e., $\mathcal{S} = \{i\}$; mark the set with all the rest of the nodes as $\mathcal{S}'$. Initialize

$$\underline{B}_{ij}(t) = b^i_{ij}(t).$$

3. Identify a neighboring node (intermediary) $k$ not in the current list $\mathcal{S}$ with the maximum bandwidth from node $i$, i.e., find $k \in \mathcal{S}'$ such that $\underline{B}_{ik}(t) = \max_{m\in\mathcal{S}'} \underline{B}_{im}(t)$

   Add $k$ to the list $\mathcal{S}$, i.e., $\mathcal{S} = \mathcal{S} \cup \{k\}$

   Drop $k$ from $\mathcal{S}'$, i.e., $\mathcal{S}' = \mathcal{S}'\backslash\{k\}$.

   If $\mathcal{S}'$ is empty, stop.

4. Consider nodes in $\mathcal{S}'$ to update maximum bandwidth path, i.e., for $j \in \mathcal{S}'$

$$\underline{B}_{ij}(t) = \max\{\underline{B}_{ij}(t), \min\{\underline{B}_{ik}, b^i_{kj}(t)\}\}. \tag{2.7.1}$$

   Go to Step 3.

**TABLE 2.4** Iterative steps based on Algorithm 2.8.

| Iteration | List, $\mathcal{S}$ | $\underline{B}_{12}$ | Path | $\underline{B}_{13}$ | Path | $\underline{B}_{14}$ | Path | $\underline{B}_{15}$ | Path |
|---|---|---|---|---|---|---|---|---|---|
| 1 | {1} | 30 | 1-2 | 0 | – | 20 | 1-4 | 0 | – |
| 2 | {1, 2} | 30 | 1-2 | 10 | 1-2-3 | 20 | 1-4 | 0 | – |
| 3 | {1, 2, 4} | 30 | 1-2 | 15 | 1-4-3 | 20 | 1-4 | 20 | 1-4-5 |
| 4 | {1, 2, 4, 3} | 30 | 1-2 | 15 | 1-4-3 | 20 | 1-4 | 20 | 1-4-5 |
| 5 | {1, 2, 4, 3, 5} | 30 | 1-2 | 15 | 1-4-3 | 20 | 1-4 | 20 | 1-4-5 |

the distributed time-dependent case from the point of view of node 1, i.e., suppose that node 1 wants to find the path with most available bandwidth to all other nodes in the network. Then, initially, $\mathcal{S} = \{1\}$ and $\mathcal{S}' = \{2, 3, 4, 5\}$, and the widest paths to all nodes that are direct neighbors of node 1 can be readily found while for the rest, the "cost" remains at 0, i.e.,

$$\underline{B}_{12} = 30, \quad \underline{B}_{14} = 20, \quad \underline{B}_{13} = \underline{B}_{15} = 0.$$

Since $\max_{j\in\mathcal{S}'} \overline{B}_{1j} = 30$ is attained for $j = 2$, we add node 2 to list $\mathcal{S}$. Thus, we have the updated lists: $\mathcal{S} = \{1, 2\}$ and $\mathcal{S}' = \{3, 4, 5\}$.

Now for $j$ not in $\mathcal{S}$, we update the available bandwidth to see if it is better than going via node 2, as follows:

$$\underline{B}_{13} = \max\{\underline{B}_{13}, \min\{B_{12}, b_{23}\}\} = \max\{0, \min\{30, 10\}\} = 10 \quad // \text{ use 1-2-3}$$

$$\underline{B}_{14} = \max\{\underline{B}_{14}, \min\{B_{12}, b_{24}\}\} = \max\{20, \min\{30, 10\}\} = 20 \quad // \text{ stay on 1-2}$$

$$\underline{B}_{15} = \max\{\underline{B}_{15}, \min\{B_{12}, b_{25}\}\} = \max\{0, \min\{30, 0\}\} = 0 \quad // \text{ no change}$$

Now we are in the second pass of the algorithm. This time, $\max_{j\in\mathcal{S}'} \overline{B}_{1j} = \max\{\underline{B}_{13}, \underline{B}_{14}, \underline{B}_{15}\} = 20$. This is attained for $j = 4$. Thus, $\mathcal{S}$ becomes $\{1, 2, 4\}$. Updating the available bandwidth to check via node 4 will be as follows:

$$\underline{B}_{13} = \max\{\underline{B}_{13}, \min\{B_{14}, b_{43}\}\} = \max\{10, \min\{20, 15\}\} = 15 \quad // \text{ use 1-4-3}$$

$$\underline{B}_{15} = \max\{\underline{B}_{15}, \min\{B_{14}, b_{45}\}\} = \max\{0, \min\{20, 40\}\} = 20 \quad // \text{ use 1-4-5}$$

This time, $j = 3$ will be included in $\mathcal{S}$. Thus, $\mathcal{S}$ becomes $\{1, 2, 4, 3\}$. There is no further improvement in the final pass of the algorithm.

### 2.7.2 Bellman–Ford-Based Approach

The widest path algorithm that uses the Bellman–Ford-based approach is strikingly similar to the Bellman–Ford shortest path routing algorithm given in Algorithm 2.2. For completeness, this is listed in Algorithm 2.9.

ALGORITHM 2.9 **Widest path algorithm, computed at node $i$ (Bellman–Ford-based).**

Initialize

$$\overline{B}_{ii}(t) = 0; \quad \overline{B}_{ij}(t) = 0, \quad \text{(for node } j \text{ that node } i \text{ is aware of).} \tag{2.7.2a}$$

For (nodes $j$ that node $i$ is aware of) do

$$\overline{B}_{ij}(t) = \max_{k \text{ directly connected to } i} \min \left\{ b_{ik}(t), \overline{B}^{i}_{kj}(t) \right\}, \quad \text{for } j \neq i. \tag{2.7.2b}$$

## 2.8 *k*-Shortest Paths Algorithm

We now go back to the class of shortest path algorithms to consider an additional case. In many networking situations, it is desirable to determine the second shortest path, the third shortest path, and so on, up to the $k$-th shortest path between a source and a destination. Algorithms used for determining paths beyond just the shortest paths are generally referred to as $k$-shortest paths algorithms.

A simple way to generate additional paths would be to start with, say Dijkstra's shortest path first algorithm, to determine the shortest path; then, by temporarily deleting each link on the shortest path one at a time, we can consider the reduced graph where we can apply again

ALGORITHM 2.10 ***k*-shortest paths algorithm.**

1. Initialize $k := 1$.
2. Find the shortest path $\mathcal{P}$ between source ($i$) and destination ($j$) in graph $\mathcal{G}$, using Dijkstra's Algorithm.

   Add $\mathcal{P}$ to permanent list $\mathcal{K}$, i.e., $\mathcal{K} := \{\mathcal{P}\}$.

   If $K = 1$, stop.

   Add $\mathcal{P}$ to set $\mathcal{X}$ and pair $(\mathcal{P}, i)$ to set $\mathcal{S}$, i.e., $\mathcal{X} := \{\mathcal{P}\}$ and $\mathcal{S} := \{(\mathcal{P}, i)\}$.
3. Remove $\mathcal{P}$ from $\mathcal{X}$, i.e., $\mathcal{X} := \mathcal{X} \backslash \{\mathcal{P}\}$.
4. Find the unique pair $(\mathcal{P}, w) \in \mathcal{S}$, and corresponding deviation node $w$ associated with $\mathcal{P}$.
5. For each node $v$, except $j$, on subpath of $\mathcal{P}$ from $w$ to $j$ ($sub_{\mathcal{P}}(w, j)$):

   Construct graph $\mathcal{G}'$ by removing the following from graph $\mathcal{G}$:

   (a) All the vertices on subpath of $\mathcal{P}$ from $i$ to $v$, except $v$.

   (b) All the links incident on these deleted vertices.

   (c) Links outgoing from $v$ toward $j$ for each $\mathcal{P}' \in \mathcal{K} \cup \{\mathcal{P}\}$, such that $sub_{\mathcal{P}}(i, v) = sub_{\mathcal{P}'}(i, v)$.

   Find the shortest path $\mathcal{Q}'$ from $v$ to $j$ in graph $\mathcal{G}'$ using Dijkstra's Algorithm.

   Concatenate subpath of $\mathcal{P}$ from $i$ to $v$ and path $\mathcal{Q}'$, i.e., $\mathcal{Q} = sub_{\mathcal{P}}(i, v) \oplus \mathcal{Q}'$.

   Add $\mathcal{Q}$ to $\mathcal{X}$ and pair $(\mathcal{Q}, v)$ to $\mathcal{S}$, i.e., $\mathcal{X} := \mathcal{X} \cup \{\mathcal{Q}\}$ and $\mathcal{S} := \mathcal{S} \cup \{(\mathcal{Q}, v)\}$.
6. Find the shortest path $\mathcal{P}$ among the paths in $\mathcal{X}$ and add $\mathcal{P}$ to $\mathcal{K}$, i.e., $\mathcal{K} := \mathcal{K} \cup \mathcal{P}$.
7. Increment $k$ by 1.
8. If $k < K$ and $\mathcal{X}$ is not empty, go to Step 4, else stop.

Dijkstra's shortest path first algorithm. This will then give us paths that are longer than the shortest path. By identifying the cost of each of these paths, we can sort them in order of successively longer paths. For example, consider finding $k$-shortest paths from node 1 to node 6 in Figure 2.1. Here, the shortest path is 1-4-3-6 with path cost 3. Through this procedure, we can find longer paths such as 1-2-3-6 (path cost 4), 1-4-5-6 (path cost 4), and 1-4-3-5-6 (path cost 4). It is easy to see that paths so determined may have one or more links in common.

Suppose that we want to find $k$-shortest link disjoint paths. In this case, we need to temporarily delete all the links on the shortest path and run Dijkstra's algorithm again on the reduced graph—this will then give the next shortest link disjoint path; we can continue this process until we find $k$-shortest link disjoint paths. Sometimes it might not be possible to find two or more link disjoint paths, if the reduced graph is isolated into more than one network. Consider again Figure 2.1. Here, the shortest path from node 1 to node 6 is 1-4-3-6 with path cost 3. If we temporarily delete the links in this path, we find the next link-disjoint shortest

path to be 1-2-4-5-6 of path cost 5. If we now delete links in this path, node 1 becomes isolated in the newly obtained reduced graph.

In Algorithm 2.10, we present a $k$-shortest path algorithm that is based on an idea, originally outlined in [756]; see also [454], [549] for additional references for this method. In this algorithm, a fairly complicated process is applied beyond finding the shortest path. For example, it uses an auxiliary list $\mathcal{S}$ in order to track/determine longer paths. This is meant for die-hard readers, though. A description with each step is included in Algorithm 2.10 to convey the basic idea behind this algorithm.

Finally, recall that we discussed widest path computations with candidate path caching; such candidate paths to be cached can also be determined using a $k$-shortest paths algorithm. Typically, in such situations, the link cost for all links can be set to 1 because usually hop-length-based $k$-shortest paths are sufficient to determine candidate paths to cache.

## 2.9 Summary

We first start with notations. In discussing different shortest path algorithms, we have used a set of notations. While the notations might look confusing at first, there is some structure to the notations used here.

First, for a link $i$-$k$ connecting node $i$ and node $k$, the link cost for the additive case has been denoted by $d_{ik}$, while the link cost for the nonadditive case was denoted by $b_{ik}$. From a computational results point of view, we needed to track the minimum path cost between node $i$ and $j$ for various algorithms that can be distinctly identifiable—they can be classified as follows:

| Algorithm | Indicator | Additive | Nonadditive (Widest) |
|---|---|---|---|
| Bellman–Ford | "overbar" | $\overline{D}_{ij}$ | $\overline{B}_{ij}$ |
| Dijkstra | "underscore" | $\underline{D}_{ij}$ | $\underline{B}_{ij}$ |
| Path caching | "hat" | $\widehat{D}_{ij}$ | $\widehat{B}_{ij}$ |

A superscript is used when we discuss information as known to a node, especially node $i$ where the algorithmic computation is viewed from in the distributed environment. Thus, we have used $\overline{D}^i_{kj}$ to denote the minimum additive path cost from node $k$ to node $j$ as known to node $i$. Finally, the temporal aspect is incorporated by making an expression a function of time $t$. Thus, we use $\overline{D}^i_{kj}(t)$ to indicate dependency on time $t$. While there are a few more notations, such as path list, these notations basically capture the essence and distinction of various algorithms. In any case, all notations are summarized in Table 2.5.

We have presented several shortest path and widest path routing algorithms that are useful in communication network routing. We started with the centralized version of the Bellman–Ford algorithm and then presented the distance vector approach, first used in the ARPANET distributed environment. Similarly, we presented Dikstra's algorithm, both the centralized and its distributed variant. We then considered routing algorithms when a non additive cost property (based on minimum function) is applicable; such algorithms can be classified as widest path routing when nonadditive cost property is concave. It may be noted that there may be several widest paths between two nodes, each with a different number of

**TABLE 2.5** Summary of notations used in this chapter

| Notation | Remark |
|---|---|
| $i$ | Source node |
| $j$ | Destination node |
| $k$ | Intermediate node |
| $\mathcal{N}$ | List of nodes in a network |
| $\mathcal{S}$ | Permanent list of nodes in the Dijkstra's algorithm (considered so far in the calculation) |
| $\mathcal{S}'$ | Tentative list of nodes in the Dijkstra's algorithm (yet to be considered in calculation) |
| $\mathcal{N}_k$ | List of neighboring nodes of node $k$ |
| $d_{ij}$ | Link cost between nodes $i$ and $j$ |
| $d_{ij}(t)$ | Link cost between nodes $i$ and $j$ at time $t$ |
| $\overline{D}_{ij}$ | Cost of the minimum cost path from node $i$ to node $j$ (Bellman–Ford) |
| $\overline{D}_{ij}^{(h)}$ | Cost of the minimum cost path from node $i$ to node $j$ when $h$ hops have been considered |
| $\overline{D}_{ij}(t)$ | Cost of the minimum cost path from node $i$ to node $j$ at time $t$ |
| $d_{kj}^{i}(t)$ | Link cost between nodes $k$ and $j$ at time $t$ as known to node $i$ |
| $\overline{D}_{kj}^{i}(t)$ | Cost of the minimum cost path from node $k$ to node $j$ at time $t$ as known to node $i$ |
| $\underline{D}_{ij}$ | Cost of the minimum cost path from node $i$ to node $j$ (Dijkstra) |
| $\widehat{D}_{ij/p}(t)$ | Cost of path $p$ from node $i$ to node $j$ (path caching) |
| $\underline{B}_{ij}(t)$ | Nonadditive cost (width) of the best path from node $i$ to node $j$ at time $t$ (Dijkstra) |
| $\widehat{B}_{ij}(t)$ | Nonadditive cost (width) of the best path from node $i$ to node $j$ at time $t$ (path caching) |
| $\mathcal{P}_{ij}$ | The list of cached path at node $i$ for destination $j$ |
| $H_{ij}$ | Next hop for source $i$ for destination $j$ |

hops. It is sometimes useful to identify the *shortest-widest* path; if "shortest" is meant in terms of the number of hops, then it can be more appropriately referred to as the *least-hop-widest* path, i.e., the widest path that uses the least number of hops between two nodes. Another consideration is the determination of the *widest-shortest path*, i.e., a feasible path with the minimum cost, for example, in terms of hop count; if there are several such paths, one with the maximum bandwidth is used. These will be discussed later in Chapter 17.

Note that in this chapter, we have presented our discussion using origin and destination nodes. When a network structure is somewhat complicated, that is, when we have a backbone network that is the carrier of traffic between access networks, we also use the term *ingress* node to refer to a entry point in the core network and the term *egress* node to refer to an exit point in the core network. It is important to keep this in mind.

Finally, it is worth noting that the Bellman–Ford algorithm can operate with negative link cost while Dijkstra's algorithm requires the link costs to be nonnegative; on the other hand, Dijkstra's algorithm can be modified to work with negative link cost as well. Communication network routing protocols such as Open Shortest Path First (OSPF) and Intermediate System-to-Intermediate System (IS-IS) (refer to Chapter 6) that are based on the link state protocol concept do not allow negative weights. Thus, for all practical purposes, negative link cost rarely plays a role in communication networking protocols. Thus, in this book, we primarily consider the case when link costs are nonnegative. Certainly, from a graph theory point of view, it is important to know whether a particular algorithm works with negative link cost; interested readers may consult books such as [624].

## Further Lookup

The Bellman–Ford shortest path algorithm for computing the shortest path in a centralized manner was proposed by Ford [231] in 1956. Bellman [68] described a version independently in 1958, by using a system of equations that has become known as Bellman's equations. Moore also independently presented an algorithm in 1957 that was published in 1959 [499]. Thus, what is often known as the Belllman–Ford algorithm, especially in communications networking, is also known as the *Bellman–Ford–Moore* algorithm in many circles.

The distance vector approach for the shortest path computation in a distributed environment is subtly as well as uniquely different from the centralized Bellman–Ford approach. The distance vector approach is also known as the original or "old" ARPANET routing algorithm, yet is sometimes attributed as the "distributed Bellman–Ford" algorithm. For a discussion on how these different naming and attributions came to be known, refer to [725]. For a comprehensive summary of ARPANET design decisions in the early years, see [464].

In 1959, Dijkstra presented his shortest path first algorithm for a centralized environment [178]. The "new" ARPANET routing took the distributed view of Dijkstra's algorithm along with considerations for numerous practical issues in a distributed environment; for example, see [368], [462], [463], [599].

Widest path routing with at most two links for a path has been known since the advent of dynamic call routing in telephone networks in the early 1980s. In this context, it is often known as *maximum residual capacity routing* or *maximum available trunk routing*; for example, see [680]. The widest path algorithm based on Dijkstra's framework given in Algorithm 2.8 and the distance vector framework given in Algorithm 2.9 are adapted from [731]. Widest path routing and its variations are applicable in a quality-of-service routing framework.

The *k*-shortest paths algorithm and its many variants have been studied by numerous researchers; see [202] for an extensive bibliography.

## Exercises

2.1. Review questions:

(a) In what ways, are the Bellman–Ford algorithm (Algorithm 2.1) and the distance vector algorithm (Algorithm 2.2) different?

(b) What are the main differences between shortest path routing and widest path routing?

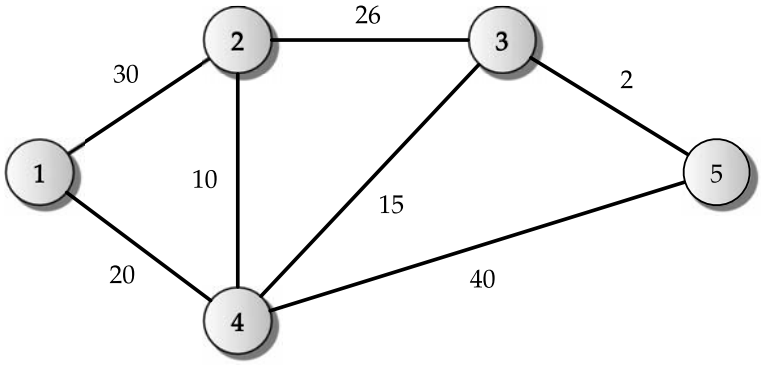



**FIGURE 2.8** A 5-node example.

(c) What is the difference between minimum hop routing and shortest path routing?

2.2. For the network example presented in Figure 2.1, compute the shortest paths from node 2 to all other nodes using the centralized Bellman–Ford algorithm (Algorithm 2.1).

2.3. For the network example presented in Figure 2.1, use Dijkstra's algorithm (Algorithm 2.3) to compute the shortest paths from node 6 to the other nodes. Next, consider that link 3-6 fails; recompute the shortest paths from node 6 to the other nodes.

2.4. Consider the network topology in Figure 2.1. Assume now that the links have the following bandwidth: 1-2: 1, 1-4: 1, 2-3: 2, 2-4: 2, 3-4: 1, 3-5: 1, 3-6: 4-5: 2; 4-6: 3; 5-6: 2. Determine the widest paths from node 6 to all the other nodes.

2.5. Consider the network topology in Figure 2.8. The number listed next to the links are link costs.

(a) Determine the shortest path from node 1 to node 5 using Algorithm 2.2 and also using Algorithm 2.3.

(b) Now suppose that the link cost for link 1-4 is changed to 45. Determine again the shortest path from node 1 to node 5 using Algorithm 2.2 and also using Algorithm 2.3. Also, generate an iterative view similar to Figure 2.4.

2.6. Consider the network topology in Figure 2.8. The number listed next to the links are assumed to be bandwidth. Determine the widest path from node 2 to node 5 using Algorithm 2.8.

2.7. Identify networking environments where path caching might be helpful that require either the shortest path or the widest path computation.

2.8. Develop a specialized *k*-shortest paths algorithm, given that a path cannot consist of more than two links.

2.9. Implement the *k*-shortest paths algorithm described in Algorithm 2.10.

This page intentionally left blank