A directed graph or digraph $G = (N, A)$ is a finite nonempty set $N$ of nodes and a collection $A$ of ordered pairs of distinct nodes from $N$; each ordered pair of nodes in $A$ is called a directed arc (or simply an arc). Pictorially, a digraph is represented in the same way as a graph, but an arrow is placed on the representation of the arc, going from the first node of the ordered pair to the second (see Fig. 5.29). Note in Fig. 5.29 that (2,4) and (4,2) are different arcs.

Given any digraph $G = (N, A)$, there is an associated (undirected) graph $G' = (N', A')$, where $N' = N$ and $(i,j) \in A'$ if either $(i,j) \in A$, or $(j,i) \in A$, or both. We say that $(n_1, n_2, \ldots, n_k)$ is a walk, path, or cycle in a digraph if it is a walk, path, or cycle in the associated graph. In addition, $(n_1, n_2, \ldots, n_k)$ is a directed walk in a digraph $G$ if $(n_1, n_2, \ldots, n_k)$ is a directed arc in $G$ for $1 \leq i \leq k$. A directed path is a directed walk with no repeated nodes, and a directed cycle is a directed walk $(n_1, \ldots, n_k)$ for $k > 2$ with $n_1 = n_k$ and no repeated nodes. Note that $(n_1, n_2, n_3)$ is a directed cycle if $(n_1, n_2)$ and $(n_2, n_3)$ are both directed arcs, whereas $(n_1, n_2, n_3)$ cannot be an undirected cycle if $(n_1, n_2)$ is an undirected arc.

A digraph is strongly connected if for each pair of nodes $i$ and $j$ there is a directed path $(i = n_1, n_2, \ldots, n_k = j)$ from $i$ to $j$. A digraph is connected if the associated graph is connected. The first graph in Fig. 5.30 is connected but not strongly connected since there is no directed path from 3 to 2. The second graph is strongly connected.

Consider a directed graph $G = (N, A)$ with number of nodes $N$ and number of arcs $A$, in which each arc $(i,j)$ is assigned some real number $d_{ij}$ as the length or distance of the arc. Given any directed path $p = (i,j,k,\ldots,l,m)$, the length of $p$ is defined as $d_{ij} + d_{jk} + \cdots + d_{lm}$. The length of a directed walk or cycle is defined analogously. Given any two nodes $i$ and $m$ of the graph, the shortest path problem is to find a minimum length (i.e., shortest) directed path from $i$ to $m$.

The shortest path problem appears in a surprisingly large number of contexts. $d_{ij}$ is the cost of using a given link $(i,j)$ in a data network, then the shortest path from $i$ to $m$ is the minimum cost route over which to send data. Thus, if the cost of a link equals the average packet delay to cross the link, the minimum cost route is also a minimum delay route. Unfortunately, in a data network, the average link delay depends on the traffic load carried by the link, which in turn depends on the routes selected by the routing algorithm. Because of this feedback effect, the minimum average delay routing problem is more complex than just solving a shortest path problem; however, the shortest path problem is still an integral part of all the formulations of the routing problem to be considered. As another example, if $p_{ij}$ is the probability that a given arc $(i,j)$ in a network is usable, and each arc is usable independently of all the other arcs, then finding the shortest path between $i$ and $j$ with arc distances $-\ln p_{ij}$ is equivalent to finding the most reliable path from $i$ to $j$.

Another application of shortest paths is in the PERT networks used by organizations to monitor the progress of large projects. The nodes of the network correspond to subtasks, and an arc from subtask $i$ to $j$ indicates that the completion of task $j$ is dependent on the completion of $i$. If $t_{ij}$ is the time required to complete $j$ after $i$ is completed, the distance for $(i,j)$ is taken as $d_{ij} = -t_{ij}$. The shortest path from project start to finish is then the most time-consuming path required for completion of the project, and the shortest path indicates the critical subtasks for which delays would delay the entire project. Yet another example is that of discrete dynamic programming problems, which can be viewed as shortest path problems [Ber87]. Finally, many more complex graph-theoretic problems require the solution of shortest path problems as subproblems.

In the following, we develop three standard algorithms for the shortest path problem: the Bellman-Ford algorithm, the Dijkstra algorithm, and the Floyd-Warshall algorithm. The first two algorithms find the shortest paths from all nodes to a given destination node, and the third algorithm finds the shortest paths from all nodes to all other nodes. Note that the problem of finding shortest paths from a given origin node to all other nodes is equivalent to the problem of finding all shortest paths to a given destination node; one version of the problem can be obtained from the other simply by reversing the direction of each arc while keeping its length unchanged. In this subsection we concentrate on centralized shortest path algorithms. Distributed algorithms are considered in Section 5.2.4.
The Bellman–Ford algorithm. Suppose that node 1 is the "destination" node and consider the problem of finding a shortest path from every node to node 1. We assume that there exists at least one path from every node to the destination. To simplify the presentation, let us denote $d_{ij} = \infty$ if $(i,j)$ is not an arc of the graph. Using this convention we can assume without loss of generality that there is an arc between every pair of nodes, since walks and paths consisting of true network arcs are the only ones with length less than $\infty$.

A shortest walk from a given node $i$ to node 1, subject to the constraint that the walk contains at most $h$ arcs and goes through node 1 only once, is referred to as a shortest ($\leq h$) walk and its length is denoted by $D^h_i$. Note that such a walk may not be a path, that is, it may contain repeated nodes; we will later give conditions under which this is not possible. By convention, we take

$$D^0_i = 0, \quad \text{for all } h$$

We will prove shortly that $D^h_i$ can be generated by the iteration

$$D^{h+1}_i = \min_j \{d_{ij} + D^h_j\}, \quad \text{for all } i \neq 1 \tag{5.1}$$

starting from the initial conditions

$$D^0_i = \infty, \quad \text{for all } i \neq 1 \tag{5.2}$$

This is the Bellman–Ford algorithm, illustrated in Fig. 5.31. Thus, we claim that the Bellman–Ford algorithm first finds the one-arc shortest walk lengths, then finds the two-arc shortest walk lengths, and so forth. Once we show this, we will argue that the shortest walk lengths are equal to the shortest path lengths, under the additional assumption that all cycles not containing node 1 have nonnegative length. We say that the algorithm terminates after $h$ iterations if

$$D^h_i = D^{h-1}_i, \quad \text{for all } i$$

The following proposition provides the main results.

Proposition. Consider the Bellman–Ford algorithm (5.1) with the initial conditions $D^0_i = \infty$ for all $i \neq 1$. Then:

(a) The scalars $D^h_i$ generated by the algorithm are equal to the shortest ($\leq h$) walk lengths from node 1 to node 1.

(b) The algorithm terminates after a finite number of iterations if and only if all cycles not containing node 1 have nonnegative length. Furthermore, if the algorithm terminates, it does so after at most $h \leq N$ iterations, and at termination, $D^h_i$ is the shortest path length from 1 to $i$.

Proof: (a) We argue by induction. From Eqs. (5.1) and (5.2) we have

$$D^1_i = d_{i1}, \quad \text{for all } i \neq 1$$
so \( D_i \) is indeed equal to the shortest (\( \leq 1 \)) walk length from \( i \) to 1. Suppose that \( D_i^j \) is equal to the shortest (\( \leq k \)) walk length from \( i \) to 1 for all \( k \leq h \). We will show that \( D_i^{h+1} \) is the shortest (\( \leq h+1 \)) walk length from \( i \) to 1. Indeed, a shortest (\( \leq h+1 \)) walk from \( i \) to 1 either consists of less than \( h+1 \) arcs, in which case its length is equal to \( D_i^h \), or else it consists of \( h+1 \) arcs with the first arc being (\( i,j \)) for some \( j \neq 1 \), followed by an \( h \)-arc walk from \( j \) to 1 in which node 1 is not repeated. The latter walk must be a shortest (\( \leq h \)) walk from \( j \) to 1. [Otherwise, by concatenating arc \((i,j)\) and a shorter (\( \leq h \)) walk from \( j \) to 1, we would obtain a shorter (\( \leq h+1 \)) walk from \( i \) to 1.] We thus conclude that

\[
\text{Shortest (} \leq h+1 \text{) walk length } = \min \left\{ D_i^h, \min_{j \neq 1} [d_{ij} + D_j^h] \right\} \tag{5.3}
\]

Using the induction hypothesis, we have \( D_i^k \leq D_i^{k-1} \) for all \( k \leq h \) (since the set of (\( \leq k \)) walks from node \( j \) to 1 contains the corresponding set of (\( \leq k-1 \)) walks). Therefore,

\[
D_i^{h+1} = \min_j [d_{ij} + D_j^h] = \min_j [d_{ij} + D_j^{h-1}] = D_i^h \tag{5.4}
\]

Furthermore, we have \( D_i^h \leq D_i^j = d_{ii} = d_{ii} + D_i^h \), so from Eq. (5.3) we obtain

\[
\text{Shortest (} \leq h+1 \text{) walk length } = \min \left\{ D_i^h, \min_{j \neq 1} [d_{ij} + D_j^h] \right\} = \min \{ D_i^h; D_i^{h+1} \}
\]

In view of \( D_i^{h+1} \leq D_i^h \) [cf. Eq. (5.4)], this yields

\[
\text{Shortest (} \leq h+1 \text{) walk length } = D_i^{h+1}
\]

completing the induction proof.

(b) If the Bellman-Ford algorithm terminates after \( h \) iterations, we must have

\[
D_i^h = D_i^k, \quad \text{for all } i \text{ and } k \geq h \tag{5.5}
\]

so we cannot reduce the lengths of the shortest walks by allowing more and more arcs in these walks. It follows that there cannot exist a negative-length cycle not containing node 1, since such a cycle could be repeated an arbitrarily large number of times in walks from some nodes to node 1, thereby making their length arbitrarily small and contradicting Eq. (5.5). Conversely, suppose that all cycles not containing node 1 have nonnegative length. Then by deleting all such cycles from shortest (\( \leq h \)) walks, we obtain paths of less or equal length. Therefore, for every \( i \) and \( h \), there exists a path that is a shortest (\( \leq h \)) walk from \( i \) to 1, and the corresponding shortest path length is equal to \( D_i^h \). Since paths have no cycles, they can contain at most \( N-1 \) arcs. It follows that

\[
D_i^N = D_i^{N-1}, \quad \text{for all } i
\]

implying that the algorithm terminates after at most \( N \) iterations. Q.E.D.
Dijkstra's algorithm. This algorithm requires that all arc lengths are nonnegative (fortunately, the case for most data network applications). Its worst-case computational requirements are considerably less than those of the Bellman–Ford algorithm. The general idea is to find the shortest paths in order of increasing path length. The shortest of the shortest paths to node 1 must be the single-arc path from the closest neighbor of node 1, since any multiple-arc path cannot be shorter than the first arc length because of the nonnegative-length assumption. The next shortest of the shortest paths must either be the single-arc path from the next closest neighbor of 1 or the shortest two-arc path through the previously chosen node, and so on. To formalize this procedure into an algorithm, we view each node i as being labeled with an estimate \( \hat{D}_i \) of the shortest path length to node 1. When the estimate becomes certain, we regard the node as being permanently labeled and keep track of this with a set \( P \) of permanently labeled nodes.

The node added to \( P \) at each step will be the closest to node 1 out of those that are not yet in \( P \). Figure 5.34 illustrates the main idea. The detailed algorithm is as follows:

1. **Step 1:** (Find the next closest node.) Find \( i \notin P \) such that
   \[ D_i = \min_{j \notin P} D_j. \]

   Set \( P := P \cup \{ i \} \). If \( P \) contains all nodes, then stop; the algorithm is complete.

   ![Diagram of Dijkstra's algorithm](image)

   **Figure 5.34** Basic idea of Dijkstra's algorithm. At the \( k \)th step we have the set \( P \) of the \( k \) closest nodes to node 1 as well as the shortest distance \( \hat{D}_i \) from each node \( i \) in \( P \) to node 1. Of all paths connecting some node \( i \) not in \( P \) with node 1, there is a shortest one that passes exclusively through nodes in \( P \) (since \( d_{ij} \geq 0 \)). Therefore, the \( (k+1) \)th closest node and the corresponding shortest distance are obtained by minimizing over \( j \notin P \) the quantity \( \min_{i \in P} (d_{ij} + \hat{D}_j) \). This calculation can be organized efficiently as discussed in the text, resulting in an \( O(N^2) \) computational complexity.
Step 2: (Updating of labels.) For all \( j \notin P \) set
\[
D_j := \min \{ d_{ij} + D_i \}
\]
Go to step 1.

To see why the algorithm works, we must interpret the estimates \( D_j \). We claim that at the beginning of each step 1:
(a) \( D_i \leq D_j \) for all \( i \in P \) and \( j \notin P \).
(b) \( D_j \) is, for each node \( j \), the shortest distance from \( j \) to 1 using paths with all nodes except possibly \( j \) belonging to the set \( P \).

Indeed, condition (a) is satisfied initially, and since \( d_{ij} \geq 0 \) and \( D_i = \min_{j \notin P} D_j \), it is preserved by the formula \( D_j := \min \{ d_{ij} + D_i \} \) for all \( j \notin P \), in step 2. We show condition (b) by induction. It holds initially. Suppose that it holds at the beginning of some step 1, let \( i \) be the node added to \( P \) at that step, and let \( D_i \) be the label of each node \( k \) at the beginning of that step. Then condition (b) holds for \( j = i \) by the induction hypothesis. It is also seen to hold for all \( j \in P \), in view of condition (a) and the induction hypothesis. Finally, for a node \( j \notin P \cup \{i\} \), consider a path from \( j \) to 1 which is shortest among those with all nodes except \( j \) belonging to the set \( P \cup \{i\} \), and let \( D'_j \) be the corresponding shortest distance. Such a path must consist of an arc \((j,k)\) for some \( k \in P \cup \{i\} \), followed by a shortest path from \( k \) to 1 with nodes in \( P \cup \{i\} \). Since we just argued that the length of this \( k \) to 1 path is \( D_k \), we have
\[
D'_j = \min_{k \in P \cup \{i\}} \{ d_{jk} + D_k \} = \min_{k \in P \cup \{i\}} \{ \min_{k \in P \cup \{i\}} \{ d_{jk} + D_k \} + d_{ji} + D_i \}.
\]

Similarly, the induction hypothesis implies that \( D_j = \min_{k \in P \cup \{i\}} \{ d_{jk} + D_k \} \), so we obtain \( D'_j = \min \{ d_{ij} + D_i \} \). Thus in step 2, \( D_j \) is set to the shortest distance \( D'_j \) from \( j \) to 1 using paths with all nodes except \( j \) belonging to \( P \cup \{i\} \). The induction proof of condition (b) is thus complete.

We now note that a new node is added to \( P \) with each iteration, so the algorithm terminates after \( N - 1 \) iterations, with \( P \) containing all nodes. By condition (b), \( D_j \) is then equal to the shortest distance from \( j \) to 1.

To estimate the computation required by Dijkstra's algorithm, we note that there are \( N - 1 \) iterations and the number of operations per iteration is proportional to \( N \). Therefore, in the worst case, the computation is \( O(N^2) \), comparing favorably with the worst-case estimate \( O(N^3) \) of the Bellman–Ford algorithm; in fact, with proper implementation, the worst-case computational requirements for Dijkstra's algorithm can be reduced considerably (see [Ber91] for a variety of implementations of Dijkstra's algorithm). However, there are many problems where \( A \ll N^2 \), and the Bellman–Ford algorithm terminates in very few iterations (say \( m \ll N \), in which case its required computation \( O(mA) \) can be less than the \( O(N^2) \) requirement of the straightforward implementation of Dijkstra's algorithm. Generally, for nondistributed applications, efficiently implemented variants of the Bellman-Ford and Dijkstra algorithms appear to be competitive (see [Ber91] for a more detailed discussion and associated codes).

The Floyd–Warshall algorithm. This algorithm, unlike the previous two, finds the shortest paths between all pairs of nodes together. Like the Bellman–Ford algorithm, the arc distances can be positive or negative, but again there can be no negative-length cycles. All three algorithms iterate to find the final solution, but each iterates on something different. The Bellman–Ford algorithm iterates on the number of arcs in a path, the Dijkstra algorithm iterates on the length of the path, and finally the Floyd–Warshall algorithm iterates on the set of nodes that are allowed as intermediate nodes on the paths. The Floyd–Warshall algorithm starts like both other algorithms with single arc distances.

![Example using the Bellman–Ford and Dijkstra algorithms.](image)
algorithm are allowed to be essentially arbitrary. These assumptions provide an adequate model for a situation where the link lengths stay fixed after some time \( t_0 \) following a number of changes that occurred before \( t_0 \).

We focus on the shortest distance \( D_i \) from each node \( i \) to a generic destination node taken for concreteness to be node 1. (In reality, a separate version of the algorithm must be executed for each destination node.) Under our assumptions, these distances are the unique solution of Bellman’s equation,

\[
D_i = \min_{j \in N(i)} \{ d_{ij} + D_j \}, \quad i \neq 1
\]

\[
D_1 = 0
\]

where \( N(i) \) denotes the set of current neighbors of node \( i \) (i.e., the nodes connected with \( i \) via an up link). This was shown in Section 5.2.3; see the discussion following Eq. (5.6).

The Bellman–Ford algorithm is given by

\[
D_{i+1}^{h+1} = \min_{j \in N(i)} \{ d_{ij} + D_j^h \}, \quad i \neq 1
\]

\[
D_1^{h+1} = 0
\]

In Section 5.2.3, we showed convergence to the correct shortest distances for the initial conditions

\[
D_i^0 = \infty, \quad i \neq 1
\]

\[
D_1^0 = 0
\]

The algorithm is well suited for distributed computation since the Bellman–Ford iteration (5.8) can be executed at each node \( i \) in parallel with every other node. One possibility is for all nodes \( i \) to execute the iteration simultaneously, exchange the results of the computation with their neighbors, and execute the iteration again with the index \( h \) increased by one. When the infinite initial conditions of Eqs. (5.10) and (5.11) are used, the algorithm will terminate after at most \( N \) iterations (where \( N \) is the number of nodes), with each node \( i \) knowing both the shortest distance, \( D_i \), and the outgoing link on the shortest path to node 1.

Unfortunately, implementing the algorithm in such a synchronous manner is not as easy as it appears. There is a twofold difficulty here. First, a mechanism is needed for getting all nodes to agree to start the algorithm. Second, a mechanism is needed to abort the algorithm and start a new version if a link status or length changes as the algorithm is running. Although it is possible to cope successfully with these difficulties [Seg81], the resulting algorithm is far more complex than the pure Bellman–Ford method given by Eqs. (5.8) and (5.9).

A simpler alternative is to use an asynchronous version of the Bellman–Ford algorithm that does not insist on maintaining synchronism between nodes, and on starting with the infinite initial conditions of Eqs. (5.10) and (5.11). This eliminates the need for either an algorithm initiation or an algorithm restart protocol. The algorithm simply operates indefinitely by executing from time to time at each node \( i \neq 1 \) the iteration

\[
D_i := \min_{j \in N(i)} \{ d_{ij} + D_j \}
\]
using the last estimates \( D_j \) received from the neighbors \( j \in N(i) \), and the latest status and lengths of the outgoing links from node \( i \). The algorithm also requires that each node \( i \) transmit from time to time its latest estimate \( D_i \) to all its neighbors. However, there is no need for either the iterations or the message transmissions to be synchronized at all nodes. Furthermore, no assumptions are made on the initial values \( D_j, j \in N(i) \) available at each node \( i \). The only requirement is that a node \( i \) will eventually execute the Bellman–Ford iteration (5.12) and will eventually transmit the result to the neighbors.

Thus, a totally asynchronous mode of operation is envisioned.

It turns out that the algorithm is still valid when executed asynchronously as described above. It will be shown that if a number of link length changes occur up to some time \( t_0 \), and no other changes occur subsequently, then within finite time from \( t_0 \), the asynchronous algorithm finds the correct shortest distance of every node \( i \). The shortest distance estimates available at time \( t_0 \) can be arbitrary numbers, so it is not necessary to reinitialize the algorithm after each link status or link length change.

The original 1969 ARPANET algorithm was based on the Bellman–Ford iteration (5.12), and was implemented asynchronously much like the scheme described above. Neighboring nodes exchanged their current shortest distance estimates \( D_j \) needed in the iteration every 625 msec, but this exchange was not synchronized across the network. Furthermore, the algorithm was not restarted following a link length change or a link failure. The main difference between the ARPANET algorithm and the one analyzed in the present subsection is that the link lengths \( d_{ij} \) were changing very frequently in the ARPANET algorithm, and the eventual steady state assumed in our analysis was seldom reached.

We now state formally the distributed, asynchronous Bellman–Ford algorithm and proceed to establish its validity. At each time \( t \), a node \( i \neq 1 \) has available:

\[
D_j(t) = \text{Estimate of the shortest distance of each neighbor node } j \in N(i) \text{ which was last communicated to node } i
\]

\[
D_i(t) = \text{Estimate of the shortest distance of node } i \text{ which was last computed at node } i \text{ according to the Bellman–Ford iteration}
\]

The distance estimates for the destination node 1 are defined to be zero, so

\[
D_1(t) = 0, \quad \text{for all } t \geq t_0
\]

\[
D_i(t) = 0, \quad \text{for all } t \geq t_0, \text{ and } i \text{ with } i \in N(i)
\]

Each node \( i \) also has available the link lengths \( d_{ij} \), for all \( j \in N(i) \), which are assumed constant after the initial time \( t_0 \). We assume that the distance estimates do not change except at some times \( t_0, t_1, t_2, \ldots, \) with \( t_{m+1} > t_m \), for all \( m \), and \( t_m \to \infty \) as \( m \to \infty \), when at each processor \( i \neq 1 \), one of three events happens:

1. Node \( i \) updates \( D_i(t) \) according to

\[
D_i(t) := \min_{j \in N(i)} [d_{ij} + D_j(t)]
\]

and leaves the estimates \( D_j(t), j \in N(i) \), unchanged.

Sec: 5.2 Network Algorithms and Shortest Path Routing

2. Node \( i \) receives from one or more neighbors \( j \in N(i) \) the value of \( D_j \) which was computed at node \( j \) at some earlier time, updates the estimate \( D_i \), and leaves all other estimates unchanged.

3. Node \( i \) is idle, in which case all estimates available at \( i \) are left unchanged.

Let \( T_i \) be the set of times for which an update by node \( i \) as in case 1 occurs, and \( T_j \) the set of times when a message is received at \( i \) from node \( j \), as in case 2. We assume the following:

**Assumption 1.** Nodes never stop updating their own estimates and receiving messages from all their neighbors [i.e., \( T_i \) and \( T_j \) have an infinite number of elements for all \( i \neq 1 \) and \( j \in N(i) \)].

**Assumption 2.** Old distance information is eventually purged from the system [i.e., given any time \( t \geq t_0 \), there exists a time \( t' \geq t \) such that \( D_j \) computed at a node \( j \) prior to time \( t' \) is not received at any neighbor node \( i \) after time \( t' \)].

The following proposition shows that the estimates \( D_i(t) \) converge to the correct shortest distances within finite time. The proof (which can be skipped without loss of continuity) is interesting in that it serves as a model for proofs of validity of several other asynchronous distributed algorithms. (See the convergence proof of the algorithm of Section 5.3.3 and references [Ber82a], [Ber83], and [BeTr91].)

**Proposition.** Suppose that each cycle has positive length and that the initial conditions \( D_i(t_0), D_j(t_0) \) be arbitrary numbers for \( i = 2, \ldots, N \) and \( j = 2, \ldots, N \). Then there is a time \( t_m \) such that

\[
D_i(t) = D_i, \quad \text{for all } t \geq t_m, \quad i = 1, \ldots, N
\]

where \( D_i \) is the correct shortest distance from node \( i \) to the destination node 1.

**Proof:** The idea of the proof is to define for every node \( i \) two sequences \( \{D_i^k\} \) and \( \{\overline{D}_i^k\} \) with

\[
D_i^k \leq \overline{D}_i^{k+1} \leq D_i \leq \overline{D}_i^{k+1} \leq D_i^k
\]

and

\[
D_i^k = D_1, \quad \text{for all } k \text{ sufficiently large}
\]

These sequences are obtained from the Bellman–Ford algorithm by starting at two different initial conditions. It is then shown that for every \( k \) and \( i \), the estimates \( D_i(t) \) satisfy

\[
D_i^k \leq D_i(t) \leq \overline{D}_i^k, \quad \text{for all } t \text{ sufficiently large}
\]

A key role in the proof is played by the monotonicity property of the Bellman–Ford iteration. This property states that if for some scalars \( D_j \) and \( D_j \),

\[
D_j \geq D_j, \quad \text{for all } j \in N(i)
\]

then the direction of the inequality is preserved by the iteration, that is,

\[
\min_{j \in N(i)} [d_{ij} + D_j] \geq \min_{j \in N(i)} [d_{ij} + D_j]
\]
A consequence of this property is that if $D^i_k$ are sequences generated by the Bellman-Ford iteration (5.8) and (5.9) starting from some initial condition $D^i_0$, $i = 1, \ldots, N$, and we have $D^i_k \geq D^j_k$ for each $i$, then $D^{i+k}_k \geq D^{j+k}_k$, for each $i$ and $k$. Similarly, if $D^i_k \leq D^j_k$, for each $i$, then $D^{i+k}_k \leq D^{j+k}_k$, for each $i$ and $k$.

Consider the Bellman-Ford algorithm given by Eqs. (5.8) and (5.9). Let $D^i_k$, $i = 1, \ldots, N$ be the $k$th iterate of this algorithm when the initial condition is

\begin{equation}
D^i_0 = \infty, \quad i \neq 1
\end{equation}

and let $D^i_0, i = 1, \ldots, N$ be the $k$th iterate when the initial condition is

\begin{equation}
D^i_0 = D_i - \delta, \quad i \neq 1
\end{equation}

where $\delta$ is a positive scalar large enough so that $D^i_0$ is smaller than all initial node estimates $D_i(t_0)$, $D^j_k(t_0)$, $j \in N(i)$ and all estimates $D_i$ that were communicated by node $i$ before $t_0$ and will be received by neighbors of $i$ after $t_0$.

**Lemma.** The sequences $\{D^i_k\}$ and $\{D^k_i\}$ defined above satisfy Eqs. (5.13) and (5.14).

**Proof:** Relation (5.13) is shown by induction using the monotonicity property of the Bellman-Ford iteration, and the choice of initial conditions above. To show Eq. (5.14), first note that from the convergence analysis of the Bellman-Ford algorithm of the preceding section, for all $i$,

\begin{equation} D^i_0 = D_i, \quad k \geq N - 1 \end{equation}

so only $D^i_k = D_i$, for sufficiently large $k$, remains to be established. To this end, we first note that by an induction argument it follows that $D^i_0$ is, for every $k$, the length of a walk from $i$ to some other node $j$ involving no more than $k$ links, plus $D^j_0$. Let $L(i, k)$ and $n(i, k)$ be the length and the number of links of this walk, respectively. We can decompose this walk into a path from $i$ to $j$ involving no more than $N - 1$ links plus a number of cycles each of which must have positive length by our assumptions. Therefore, if $\lim_{k \to \infty} n(i, k) = \infty$ for any $i$, we would have $\lim_{k \to \infty} L(i, k) = \infty$, which is impossible since $D^i_0 \geq D^{i+k}_k = L(i, k) + D^j_0$ and $D_i$ is finite (the network is assumed strongly connected). Therefore, $n(i, k)$ is bounded with respect to $k$ and it follows that the number of all possible values of $D^i_k$, for $i = 1, \ldots, N$ and $k = 0, 1, \ldots$, is finite. Since $D^i_k$ is monotonically nondecreasing in $k$ for all $i$, it follows that for some $h$

\begin{equation} D^{i+k}_k = D^i_k, \quad \text{for all } i \end{equation}

Therefore, the scalars $D^i_k$ satisfy Bellman's equation, which as shown in Section 5.2.3,
We close this section with a discussion of two weaknesses of the asynchronous Bellman-Ford method. The first is that in the worst case, the algorithm may require an excessive number of iterations to terminate (see the example of Fig. 5.36). This is not due to the asynchronous nature of the algorithm, but rather to the arbitrary choice of initial conditions [an indication is provided by the argument preceding Eq. (5.19)]. A heuristic remedy is outlined in Problem 5.6: for other possibilities, see [JAM82], [Gar87], and [Hum91]. The second weakness, demonstrated in the example of Fig. 5.37, is that in the worst case, the algorithm requires an excessive number of message transmissions. The example of Fig. 5.37 requires an unlikely sequence of events. An analysis given in [TsS90] shows that under fairly reasonable assumptions, the average number of messages required by the algorithm is bounded by a polynomial in $N$.

5.2.5 Stability of Adaptive Shortest Path Routing Algorithms

We discussed in earlier sections the possibility of using link lengths that reflect the traffic conditions on the links in the recent past. The idea is to assign a large length to a congested link so that the shortest path algorithm will tend to exclude it from a routing path. This sounds attractive at first, but on second thought one gets alerted to the possibility of oscillations. We will see that this possibility is particularly dangerous in datagram networks.

**Stability issues in datagram networks.** For a simple example of oscillation, consider a datagram network, and suppose that there are two paths along which an origin can send traffic to a destination. Routing along a path during some time period will increase its length, so the other path will tend to be chosen for routing in the next time period, resulting in an oscillation between the two paths. It turns out that a far worse type of oscillation is possible, as illustrated in the following example.

**Example 5.4**

Consider the 16-node network shown in Fig. 5.38, where node 16 is the only destination. Let the traffic input (in data units/sec) at each node $i = 1, \ldots, 7, 9, \ldots, 15$ be one unit and let the traffic input of node 8 be $\epsilon > 0$, where $\epsilon$ is very small. Assume that the length of link $(i, j)$ is

$$d_{ij} = F_{ij}$$

where $F_{ij}$ is the arrival rate at the link counting input and relayed traffic. Suppose that all nodes compute their shortest path to the destination every $T$ seconds using as link lengths the arrival rates $F_{ij}$ during the preceding $T$ seconds, and route all their traffic along that path.
for the next 7 seconds. Assume that we start with nodes 1 through 7 routing clockwise and
nodes 8 through 15 routing counterclockwise. This is a rather good routing, balancing the
traffic input between the two directions. Figure 5.39 shows the link rates corresponding to
the initial and subsequent shortest path routings. Thus, after three shortest path updates, the
algorithm is locked into an oscillatory pattern whereby all traffic swings from the clockwise
to the counterclockwise direction and back at alternate updates. This is certainly the worst
type of routing performance that could occur.

The difficulty in Example 5.4 is due to the fact that link arrival rates depend on
routing, which in turn depends on arrival rates via the shortest path calculation,
with a feedback effect resulting. This is similar to the stability issue in feedback
control theory, and can be analyzed using a related methodology [Ber82b]. Actually,
it can be shown that the type of instability illustrated above will occur gener-
ically if the length \(d_{ij}\) of link \((i,j)\) increases continuously and monotonically with the
link arrival rate \(F_{ij}\), and \(d_{ij} = 0\) when \(F_{ij} = 0\). It is possible to damp the
oscillations by adding a positive constant to the link length so that \(d_{ij} = \alpha > 0\)
when \(F_{ij} = 0\). The scalar \(\alpha\) (link length at zero load) is known as a bias factor.

Figure 5.39 Oscillations in a ring network for link lengths \(d_{ij}\) equal to the link arrival rates \(F_{ij}\).
Each node sends one unit of input traffic to the destination except for the middle node 8, which
sends \(e > 0\), where \(e\) is very small. The numbers next to the links are the link rates in each of the
two directions. As an example of the shortest path calculations, at the first iteration the middle node
8 computes the length of the clockwise path as 28 = 0 + 1 + 2 + \ldots + 7, and the length of the
clockwise path as 28 + \(8e\) = 1 + \(e\) + 2 + \(e\) + \ldots + 7 + \(e\), and switches its traffic to the
shortest (clockwise) path at the second routing. The corresponding numbers for node 9 are 28 and
28 + 7\(e\), so node 9 also switches its traffic to the clockwise path. All other nodes find that the path
used at the first routing is shortest, and therefore they do not switch their traffic to the other path.
Oscillations in the original ARPANET algorithm discussed in Section 5.1.2 were damped by using a substantial bias factor. Indeed, if \( \alpha \) is large enough relative to the range of link lengths, it can be seen that the corresponding shortest paths will have the minimum possible number of links to the destination. This is static routing, which cannot exhibit any oscillation of the type seen earlier, but is also totally insensitive to traffic congestion. In the current ARPANET algorithm described in Section 5.1.2, the bias \( \alpha \) in effect equals the sum of the average packet transmission time, the processing delay, and the propagation delay along a link. With the 1987 algorithm modifications ([KZh69], [BZ97], and [ZV97]), the bias became fairly large relative to the range of allowable link lengths. As a result, the algorithm became quite stable but also less sensitive to congestion. The second possibility to damp oscillations is to introduce a mechanism for averaging the lengths of links over a time period spanning more than one shortest path update. This tends to improve the stability of the algorithm, albeit at the expense of reducing its speed of response to congestion. It turns out that asynchronous shortest path updating by the network nodes results in a form of length averaging that is beneficial for stability purposes. (See [MRR78], [Ber79], and [Ber82b] for further discussion.)

**Stability issues in virtual circuit networks.** The oscillatory behavior exhibited above is associated principally with datagram networks. It will be shown that oscillations are less severe in virtual circuit networks. A key feature of a datagram network in this regard is that each of a packet of a user pair is not required to travel on the same path as the preceding packet. Therefore, the time that an origin-destination pair will continue to use a shortest path after it is changed due to a routing update is very small. As a result, a datagram network reacts very fast to a shortest path update, with all traffic switching to the new shortest path almost instantaneously.

The situation is quite different in a virtual circuit network, where every session is assigned a fixed communication path at the time it is first established. The average duration of a virtual circuit is often large relative to the shortest path updating period. As a result, the network reaction to a shortest path update is much more gradual since old sessions continue to use their established communication paths and only new sessions are assigned to the most recently calculated shortest paths.

As the following example demonstrates, the critical parameters for stability are the "speed" of the virtual circuit arrival and departure processes, and the frequency with which shortest paths are updated.

**Example 5.5**

Consider the simple two-link network with one origin and one destination shown in Figure 5.40. Suppose that the arrival rate is \( r \) bits/sec. Assuming that the two links have equal capacity \( C \), an "optimal" routing algorithm should somehow divide the input \( r \) equally between the two links, thereby allowing a throughput up to \( 2C \).

Consider a typical adaptive algorithm based on shortest paths for this example network. The algorithm divides the time axis into \( T \)-second intervals. It measures the average arrival rate (bits/second) on both links during each \( T \)-second interval, and directs all traffic (datagrams or virtual circuits) generated during every \( T \)-second interval along the link that had the smallest rate during the preceding \( T \)-second interval.

![Figure 5.40 (a) Two-link network of Example 5.5. (b) Arrival rate on link 1 in Example 5.5 using the shortest path rule in the datagram case. Essentially, only one path is used for routing at any one time if the shortest path update period is much larger than the time required to empty the queue of waiting packets at the time of an update.](image)

If the network uses datagrams, then, assuming that \( T \) is much larger than the time required to empty the queue of waiting packets at the time of an update, each link will essentially carry either no traffic or all the input traffic \( r \) at alternate time intervals, as shown in Fig. 5.40(b).

Next consider the case where the network uses virtual circuits, which are generated according to a Poisson process at a rate \( \lambda \) per second. Each virtual circuit uses the link on which it was assigned by the routing algorithm for its entire duration, assumed exponentially distributed with mean \( 1/\mu \) seconds. Therefore, according to the \( M|M|\infty \) queuing results (cf. Section 3.4.2), the number of active virtual circuits is Poisson distributed with mean \( \lambda/\mu \). If \( \gamma \) is the average communication rate (bits/sec) of a virtual circuit, we must have

\[
\gamma = \frac{r(\lambda/\mu)}{\lambda} \quad \text{(5.27)}
\]

Suppose that the shortest path updating interval \( T \) is small relative to the average duration of the virtual circuits \( 1/\mu \). Then, approximately a fraction \( \mu T \) of the virtual circuits that were carried on each link at the beginning of a \( T \)-second interval will be terminated at the end of the interval. There will be \( N \mu T \) virtual circuits on the average added on the link that carried the least traffic during the preceding interval. This amounts to an added arrival rate of \( \gamma N \mu T \) bits/sec or, using the fact that \( \gamma = r(\lambda/\mu) \) (cf. Eq. (5.27)), \( rN \mu T \) bits/sec.

Therefore, the average rates \( x_1^i \) and \( x_2^i \) (bits/sec) on the two links at the \( i \)-th interval will evolve approximately according to

\[
x_{i+1}^{k+1} = \begin{cases} (1 - \mu T)x_i^k + r_i T, & \text{if } i \text{ is shortest (i.e., } x_i^k = \min\{x_1^k, x_2^k\}) \\ (1 - \mu T)x_i^k, & \text{otherwise} \end{cases} \quad \text{(5.28)}
\]
Figures 5.41 and 5.42 provide examples of the evolution of the average rate on line 1. From Eq. (5.28) it can be seen that as $k \to \infty$, the average rates $x_1$ and $x_2$ will tend to oscillate around $r/2$ with a magnitude of oscillation roughly equal to $r/(\mu T)$. Therefore, if the average duration of a virtual circuit is large relative to the shortest path updating interval ($\mu T \ll 1$), the routing algorithm performs almost optimally, keeping traffic divided in nearly equal proportions among the two links. Conversely, if the product $\mu T$ is large, the analysis above indicates considerable oscillation of the link rates. Figures 5.41 and 5.42 demonstrate the relation between $\mu$, $T$, and the magnitude of oscillation.

Example 5.5 illustrates behavior that has been demonstrated analytically for general virtual circuit networks [GaB83]. It is also shown in [GaB83] that the average duration of a virtual circuit is a critical parameter for the performance of adaptive shortest path routing. If it is very large, the rate of convergence of the algorithm will be slow, essentially because virtual circuits that are misplaced on congested paths persist for a long time [cf. Figs. 5.41(a) and 5.42(a)]. If it is very small, the shortest path update interval must be accordingly small for the oscillation around optimality to be small [cf. Fig. 5.41(b)]. Unfortunately, there is a practical limit on how small the update interval can be, because frequent updates require more overhead and because sufficient time between updates is needed to measure accurately the current link lengths. The problem with a large duration of virtual circuits would not arise if virtual circuits could be rerouted during their lifetime. In the Codex network described in Section 5.8, such rerouting is allowed, thereby resulting in an algorithm that is more efficient than the one described in this section.