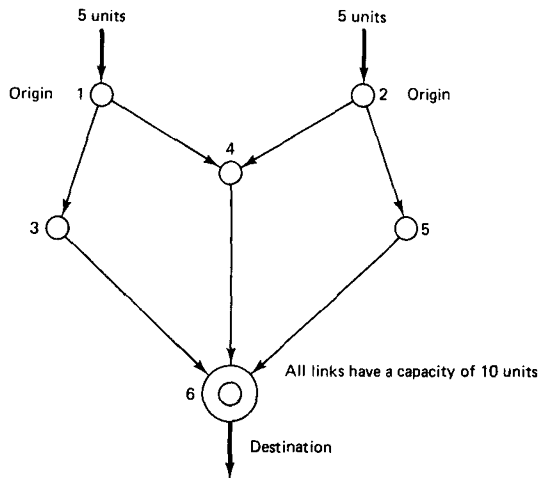
5



Routing in Data Networks

5.1 INTRODUCTION

We have frequently referred to the routing algorithm as the network layer protocol that guides packets through the communication subnet to their correct destination. The times at which routing decisions are made depend on whether the network uses datagrams or virtual circuits. In a datagram network, two successive packets of the same user pair may travel along different routes, and a routing decision is necessary for each individual packet (see Fig. 5.1). In a virtual circuit network, a routing decision is made when each virtual circuit is set up. The routing algorithm is used to choose the communication path for the virtual circuit. All packets of the virtual circuit subsequently use this path up to the time that the virtual circuit is either terminated or rerouted for some reason (see Fig. 5.2).

Routing in a network typically involves a rather complex collection of algorithms that work more or less independently and yet support each other by exchanging services or information. The complexity is due to a number of reasons. First, routing requires coordination between all the nodes of the subnet rather than just a pair of modules as, for example, in data link and transport layer protocols. Second, the routing system must

cope with link and node failures, requiring redirection of traffic and an update of the databases maintained by the system. Third, to achieve high performance, the routing algorithm may need to modify its routes when some areas within the network become congested.

The main emphasis will be on two aspects of the routing problem. The first has to do with selecting routes to achieve high performance. In Sections 5.2.3 to 5.2.5, we discuss algorithms based on shortest paths that are commonly used in practice. In Sections 5.5, 5.6, and 5.7 we describe sophisticated routing algorithms that try to achieve near optimal performance. The second aspect of routing that we will emphasize is broadcasting routing-related information (including link and node failures and repairs) to all network nodes. This issue and the subtleties associated with it are examined in Section 5.3.

The introductory sections set the stage for the main development. The remainder of this section explains in nonmathematical terms the main objectives in the routing problem and provides an overview of current routing practice. Sections 5.2.1 to 5.2.3 present some of the main notions and results of graph theory, principally in connection with shortest paths and minimum weight spanning trees. Section 5.4 uses the material on graph theory to describe methods for topological design of networks. Finally, Section 5.8 reviews the routing system of the Codex network and its relation to the optimal routing algorithm of Section 5.7.

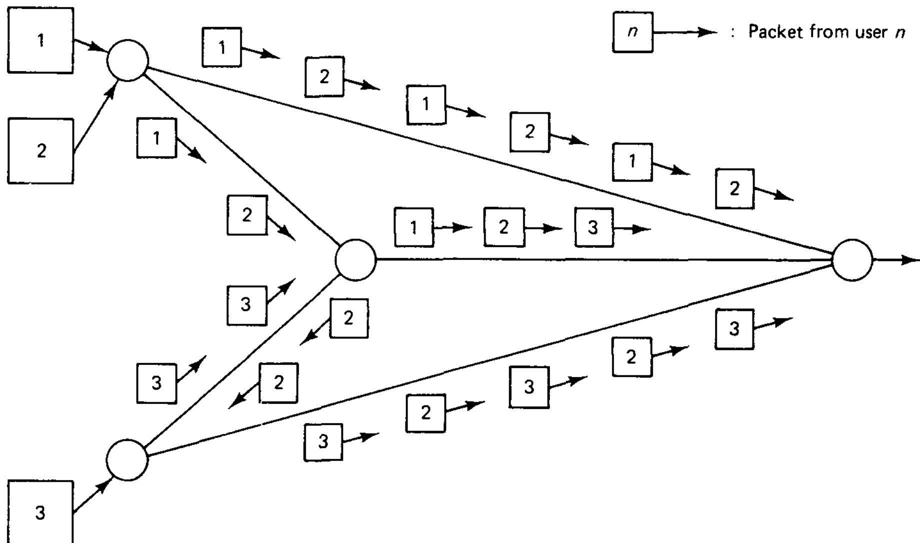


Figure 5.1 Routing in a datagram network. Two packets of the same user pair can travel along different routes. A routing decision is required for each individual packet.

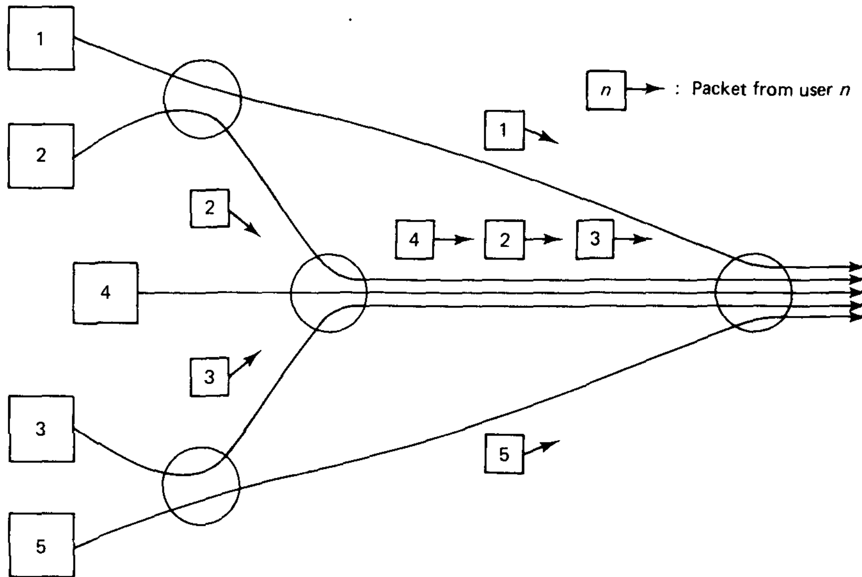


Figure 5.2 Routing in a virtual circuit network. All packets of each virtual circuit use the same path. A routing decision is required only when a virtual circuit is set up.

5.1.1 Main Issues in Routing

The two main functions performed by a routing algorithm are the selection of routes for various origin–destination pairs and the delivery of messages to their correct destination once the routes are selected. The second function is conceptually straightforward using a variety of protocols and data structures (known as routing tables), some of which will be described in the context of practical networks in Section 5.1.2. The focus will be on the first function (selection of routes) and how it affects network performance.

There are two main performance measures that are substantially affected by the routing algorithm—*throughput* (quantity of service) and *average packet delay* (quality of service). Routing interacts with flow control in determining these performance measures

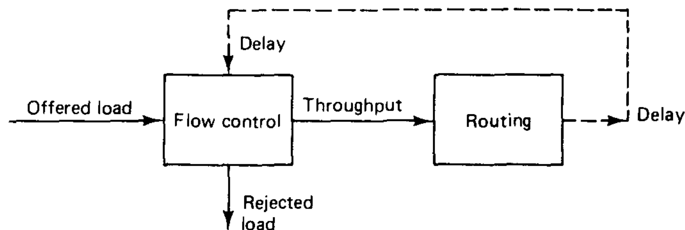


Figure 5.3 Interaction of routing and flow control. As good routing keeps delay low, flow control allows more traffic into the network.

by means of a feedback mechanism shown in Fig. 5.3. When the traffic load offered by the external sites to the subnet is relatively low, it will be fully accepted into the network, that is,

$$\text{throughput} = \text{offered load}$$

When the offered load is excessive, a portion will be rejected by the flow control algorithm and

$$\text{throughput} = \text{offered load} - \text{rejected load}$$

The traffic accepted into the network will experience an average delay per packet that will depend on the routes chosen by the routing algorithm. However, throughput will also be greatly affected (if only indirectly) by the routing algorithm because typical flow control schemes operate on the basis of striking a balance between throughput and delay (*i.e.*, they start rejecting offered load when delay starts getting excessive). Therefore, *as the routing algorithm is more successful in keeping delay low, the flow control algorithm allows more traffic into the network.* While the precise balance between delay and throughput will be determined by flow control, the effect of good routing under high offered load conditions is to realize a more favorable delay-throughput curve along which flow control operates, as shown in Fig. 5.4.

The following examples illustrate the discussion above:

Example 5.1

In the network of Fig. 5.5, all links have capacity 10 units. (The units by which link capacity and traffic load is measured is immaterial and is left unspecified.) There is a single destination (node 6) and two origins (nodes 1 and 2). The offered load from each of nodes 1 and 2 to node 6 is 5 units. Here, the offered load is light and can easily be accommodated with small delay by routing along the leftmost and rightmost paths, $1 \rightarrow 3 \rightarrow 6$ and $2 \rightarrow 5 \rightarrow 6$, respectively. If instead, however, the routes $1 \rightarrow 4 \rightarrow 6$ and $2 \rightarrow 4 \rightarrow 6$ are used, the flow on link (4,6) will equal capacity, resulting in very large delays.

Example 5.2

For the same network, assume that the offered loads at nodes 1 and 2 are 5 and 15 units, respectively (see Fig. 5.6). If routing from node 2 to the destination is done along a single path, then at least 5 units of offered load will have to be rejected since all path capacities

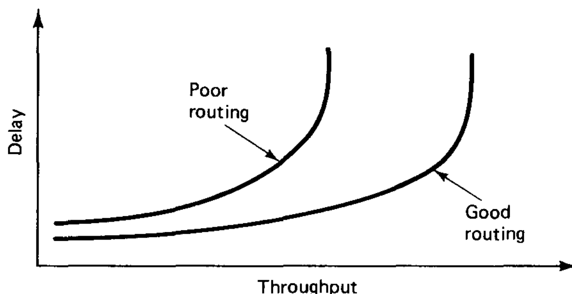


Figure 5.4 Delay–throughput operating curves for good and bad routing.

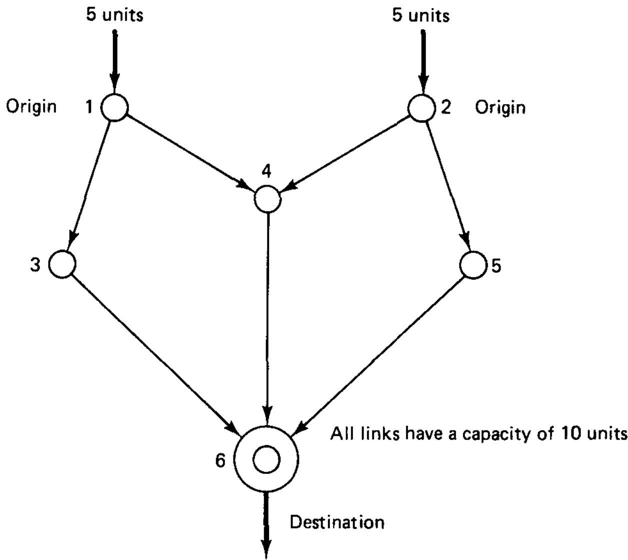


Figure 5.5 Network for Example 5.1. All links have a capacity of 10 units. If all traffic is routed through the middle link (4,6), congestion occurs. If, instead, paths ($1 \rightarrow 3 \rightarrow 6$) and ($2 \rightarrow 5 \rightarrow 6$) are used, the average delay is small.

equal 10. Thus, the total throughput can be no more than 15 units. On the other hand, suppose that the traffic originating at node 2 is evenly split between the two paths $2 \rightarrow 4 \rightarrow 6$ and $2 \rightarrow 5 \rightarrow 6$, while the traffic originating at node 1 is routed along $1 \rightarrow 3 \rightarrow 6$. Then, the traffic arrival rate on each link will not exceed 75% of capacity, the delay per packet will be reasonably small, and (given a good flow control scheme) no portion of the offered load will be rejected. Arguing similarly, it is seen that when the offered loads at nodes 1 and 2 are both large, the maximum total throughput that this network can accommodate is between 10 and 30 units, depending on the routing scheme. This example also illustrates that to achieve high throughput, the traffic of some origin–destination pairs may have to be divided among more than one route.

In conclusion, *the effect of good routing is to increase throughput for the same value of average delay per packet under high offered load conditions and to decrease average delay per packet under low and moderate offered load conditions.* Furthermore, it is evident that the routing algorithm should be operated so as to keep average delay per packet as low as possible for any given level of offered load. While this is easier said than done, it provides a clear-cut objective that can be expressed mathematically and dealt with analytically.

5.1.2 Wide Area Network Routing: An Overview

The purpose of this section is to survey current routing practice in wide area networks, to introduce classifications of different schemes, and to provide a context for the analysis presented later.

There are a number of ways to classify routing algorithms. One way is to divide them into *centralized* and *distributed.* In centralized algorithms, all route choices are made at a central node, while in distributed algorithms, the computation of routes is

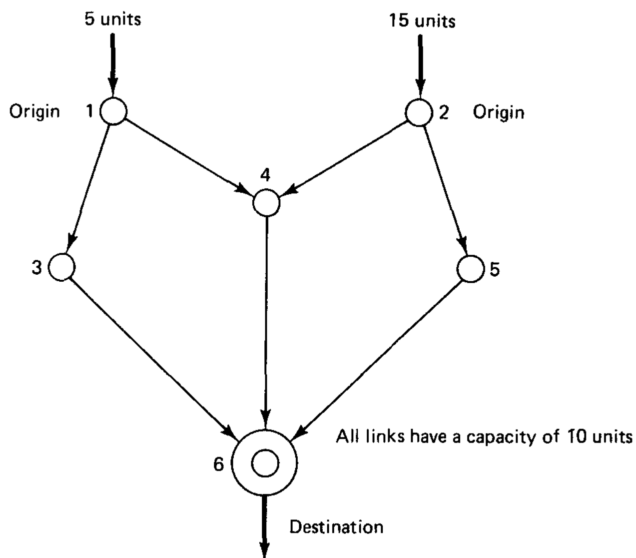


Figure 5.6 Network for Example 5.2. All links have a capacity of 10 units. The input traffic can be accommodated with multiple-path routing, but at least 5 units of traffic must be rejected if a single-path routing is used.

shared among the network nodes with information exchanged between them as necessary. Note that this classification relates mostly to the implementation of an algorithm, and that a centralized and a distributed routing algorithm may be equivalent at some level of mathematical abstraction.

Another classification of routing algorithms relates to whether they change routes in response to the traffic input patterns. In *static* routing algorithms, the path used by the sessions of each origin–destination pair is fixed regardless of traffic conditions. It can only change in response to a link or node failure. This type of algorithm cannot achieve a high throughput under a broad variety of traffic input patterns. It is recommended for either very simple networks or for networks where efficiency is not essential. Most major packet networks use some form of *adaptive* routing where the paths used to route new traffic between origins and destinations change occasionally in response to congestion. The idea here is that congestion can build up in some part of the network due to changes in the statistics of the input traffic load. Then, the routing algorithm should try to change its routes and guide traffic around the point of congestion.

There are many routing algorithms in use with different levels of sophistication and efficiency. This variety is partly due to historical reasons and partly due to the diversity of needs in different networks. In this section we provide a nonmathematical description of some routing techniques that are commonly used in practice. We illustrate these techniques in terms of the routing algorithms of three wide area networks (ARPANET, TYMNET, and SNA). The routing algorithm of another wide area network, the Codex network, will be described in Section 5.8, because this algorithm is better understood after studying optimal routing in Sections 5.5 to 5.7.

Flooding and broadcasting During operation of a data network, it is often necessary to broadcast some information, that is, to send this information from an origin

node to all other nodes. An example is when there are changes in the network topology due to link failures and repairs, and these changes must be transmitted to the entire network. Broadcasting could also be used as a primitive form of routing packets from a single transmitter to a single receiver or, more generally, to a subset of receivers; this use is generally rather inefficient, but may be sensible because it is simple or because the receivers' locations within the network are unknown.

A widely used broadcasting method, known as *flooding*, operates as follows. The origin node sends its information in the form of a packet to its neighbors (the nodes to which it is directly connected with a link). The neighbors relay it to their neighbors, and so on, until the packet reaches all nodes in the network. Two additional rules are also observed, which limit the number of packet transmissions. First, a node will not relay the packet back to the node from which the packet was obtained. Second, a node will transmit the packet to its neighbors at most once; this can be ensured by including on the packet the ID number of the origin node and a sequence number, which is incremented with each new packet issued by the origin node. By storing the highest sequence number received for each origin node, and by not relaying packets with sequence numbers that are less than or equal to the one stored, a node can avoid transmitting the same packet more than once on each of its incident links. Note that with these rules, links need not preserve the order of packet transmissions; the sequence numbers can be used to recognize the correct order. Figure 5.7(a) gives an example of flooding and illustrates that the total number of packet transmissions per packet broadcast lies between L and $2L$, where L is the number of bidirectional links of the network. We note also that one can implement a flooding-like algorithm without using sequence numbers. This possibility is described in Section 5.3, where flooding and the topology broadcast problem are discussed in more detail.

Another broadcasting method, based on the use of a *spanning tree*, is illustrated in Fig. 5.7(b). A spanning tree is a connected subgraph of the network that includes all nodes and has no cycles (see a more detailed discussion in the next section). Broadcasting on a spanning tree is more communication-efficient than flooding. It requires a total of only $N - 1$ packet transmissions per packet broadcast, where N is the number of nodes.

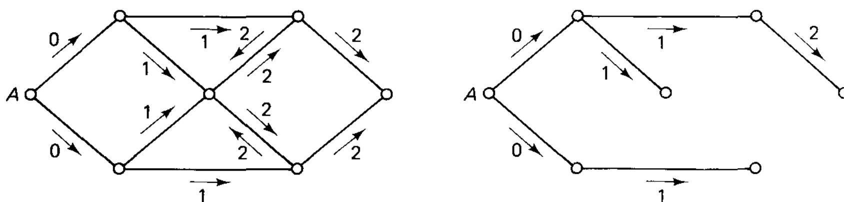


Figure 5.7 Packet broadcasting from node A to all other nodes by using flooding [as in (a)] or a spanning tree [as in (b)]. Arrows indicate packet transmissions at the times shown. Each packet transmission time is assumed to be one unit. Flooding requires at least as many packet transmissions as the spanning tree method and usually many more. In this example, the time required for the broadcast packet to reach all nodes is the same for the two methods. In general, however, depending on the choice of spanning tree, the time required with flooding may be less than with the spanning tree method.

The price for this saving is the need to maintain and update the spanning tree in the face of topological changes.

We note two more uses of spanning trees in broadcasting and routing. Given a spanning tree rooted at the origin node of the broadcast, it is possible to implement flooding as well as routing without the use of sequence numbers. The method is illustrated in Fig. 5.8. Note that flooding can be used to construct a spanning tree rooted at a node, as shown in Fig. 5.8. Also given a spanning tree, one can perform selective broadcasting, that is, packet transmission from a single origin to a limited set of destinations. For this it is necessary that each node knows which of its incident spanning tree links leads to any given destination (see Fig. 5.9). The spanning tree can also be used for routing packets with a single destination and this leads to an important method for bridged local area networks; see the next subsection.

Shortest path routing. Many practical routing algorithms are based on the notion of a *shortest path* between two nodes. Here, each communication link is assigned a positive number called its *length*. A link can have a different length in each direction. Each path (*i.e.*, a sequence of links) between two nodes has a length equal to the sum of the lengths of its links. (See Section 5.2 for more details.) A shortest path routing algorithm routes each packet along a minimum length (or shortest) path between the origin and destination nodes of the packet. The simplest possibility is for each link to have unit length, in which case a shortest path is simply a path with minimum number of links (also called a *min-hop path*). More generally, the length of a link may depend on its transmission capacity and its projected traffic load. The idea here is that a shortest

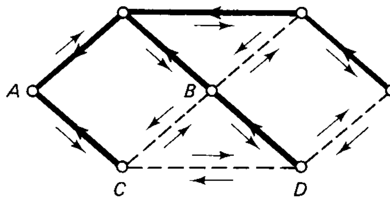


Figure 5.8 Illustration of a method for broadcasting using a tree rooted at the broadcast's origin node A. Each node knows its unique predecessor (or parent) on the tree, but need not know its successors. The tree can be used for routing packets to A from the other nodes using the paths of the tree in the reverse direction. It can also be used for flooding packets from A. The flooding rule for a node other than A is the following: a packet received from the parent is broadcast to all neighbors except the parent; all other packets are ignored. Thus in the figure, node D broadcasts the packet received from its parent B but ignores the packet received from C. Since only the packets transmitted on the spanning tree in the direction pointing away from the root are relayed further, there is no indefinite circulation of copies, and therefore, no need for sequence numbers.

Flooding can also be used to construct the tree rooted at A. Node A starts the process by sending a packet to all its neighbors; the neighbors must send the packet to their neighbors, and so on. All nodes should mark the transmitter of the first packet they receive as their parent on the tree. The nodes should relay the packet to their neighbors only once (after they first receive the packet from their parent); all subsequent packet receptions should be ignored.

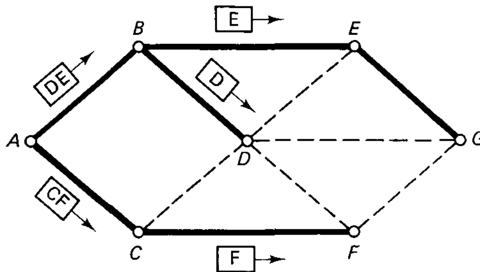


Figure 5.9 Illustration of selective broadcasting using a spanning tree. We assume that each node knows which of the incident spanning tree links lead to any given destination. In this example, node A wishes to broadcast a packet to nodes C, D, E, and F. Node A sends a copy of the packet to each of its incident links that lead to intended destinations, and each copy carries in its header the ID numbers of its intended destinations. Upon reception of a copy, a node strips the ID number from the header (if it is an intended destination) and repeats the process. This eliminates some unnecessary transmissions, for example the transmission on link EG in the figure.

path should contain relatively few and uncongested links, and therefore be desirable for routing.

A more sophisticated alternative is to allow the length of each link to change over time and to depend on the prevailing congestion level of the link. Then a shortest path may adapt to temporary overloads and route packets around points of congestion. This idea is simple but also contains some hidden pitfalls, because by making link lengths dependent on congestion, we introduce a feedback effect between the routing algorithm and the traffic pattern within the network. It will be seen in Section 5.2.5 that this can result in undesirable oscillations. However, we will ignore this possibility for the time being.

An important distributed algorithm for calculating shortest paths to a given destination, known as the *Bellman–Ford method*, has the form

$$D_i := \min_j [d_{ij} + D_j]$$

where D_i is the estimated shortest distance of node i to the destination and d_{ij} is the length of link (i, j). Each node i executes periodically this iteration with the minimum taken over all of its neighbors j. Thus $d_{ij} + D_j$ may be viewed as the estimate of shortest distance from node i to the destination subject to the constraint of going through j, and $\min_j[d_{ij} + D_j]$ may be viewed as the estimate of shortest distance from i to the destination going through the "best" neighbor.

We discuss the Bellman–Ford algorithm in great detail in the next section, where we show that it terminates in a finite number of steps with the correct shortest distances under reasonable assumptions. In practice, the Bellman–Ford iteration can be implemented as an iterative process, that is, as a sequence of communications of the current value of D_j of nodes j to all their neighbors, followed by execution of the shortest distance estimate updates $D_i := \min_j[d_{ij} + D_j]$. A remarkable fact is that this process is very flexible with respect to the choice of initial estimates D_j and the ordering of communications and updates; it works correctly, finding the shortest distances in a finite number of steps, for an essentially arbitrary choice of initial conditions and for an arbitrary order of communications and updates. This allows an asynchronous, real-time distributed implementation of the Bellman-Ford method, which can tolerate changes of the link lengths as the algorithm executes (see Section 5.2.4).

Optimal routing. Shortest path routing has two drawbacks. First, it uses only one path per origin–destination pair, thereby potentially limiting the throughput of the network; see the examples of the preceding subsection. Second, its capability to adapt to changing traffic conditions is limited by its susceptibility to oscillations; this is due to the abrupt traffic shifts resulting when some of the shortest paths change due to changes in link lengths. Optimal routing, based on the optimization of an average delay-like measure of performance, can eliminate both of these disadvantages by splitting any origin–destination pair traffic at strategic points, and by shifting traffic gradually between alternative paths. The corresponding methodology is based on the sophisticated mathematical theory of optimal multicommodity flows, and is discussed in detail in Sections 5.4 to 5.7. Its application to the Codex network is described in Section 5.8.

Hot potato (deflection) routing schemes. In networks where storage space at each node is limited, it may be important to modify the routing algorithm so as to minimize buffer overflow and the attendant loss of packets. The idea here is for nodes to get rid of their stored packets as quickly as possible, transmitting them on whatever link happens to be idle—not necessarily one that brings them closer to their destination.

To provide an example of such a scheme, let us assume that all links of the communication network can be used simultaneously and in both directions, and that each packet carries a destination address and requires unit transmission time on every link. Assume also that packets are transmitted in slots of unit time duration, and that slots are synchronized so that their start and end are simultaneous at all links. In a typical routing scheme, each node, upon reception of a packet destined for a different node, uses table lookup to determine the next link on which the packet should be transmitted; we refer to this link as the *assigned link of the packet*. It is possible that more than one packet with the same assigned link is received by a node during a slot; then at most one of these packets can be transmitted by the node in the subsequent slot, and the remaining packets must be stored by the node in a queue. The storage requirement can be eliminated by modifying the routing scheme so that all these packets are transmitted in the next slot; one of them is transmitted on its assigned link, and the others are transmitted on some other links chosen, possibly at random, from the set of links that are not assigned to any packet received in the previous slot. It can be seen that with this modification, at any node with d incident links, there can be at most d packets received in any one slot, and out of these packets, the ones that are transient (are not destined for the node) will be transmitted along some link (not necessarily their assigned one) in the next slot. Therefore, assuming that at most $d - k$ new packets are generated at a node in a slot where k transient packets are to be transmitted, there will be no queueing, and the storage space at the node need not exceed $2d$ packets. A scheme of this type is used in the Connection Machine, a massively parallel computing system [Hil85]. A variation is obtained when storage space for more than $2d$ packets is provided, and the transmission of packets on links other than the ones assigned to them is allowed only when the available storage space falls below a certain threshold.

The type of method just described was suggested in [Bar64] under the name *hot potato routing*. It has also been known more recently as *deflection routing*. Its drawback is that successive packets of the same origin–destination pair may be received out of

order, and some packets may travel on long routes to their destinations; indeed, one should take precautions to ensure that a packet cannot travel on a cycle indefinitely. For some recent analyses of deflection routing, see [BrC91a], [BrC91b], [HaC90], [Haj91], [GrH89], [Max87], [Syz90], [Var90].

Cut-through routing. We have been implicitly assuming thus far that a packet must be fully received at a node before that node starts relaying it to another node. There is, however, an incentive to split a long message into several smaller packets in order to reduce the message's delay on multiple link paths, taking advantage of pipelining (see the discussion in Section 2.5.5). The idea of message splitting, when carried to its extreme, leads to a transmission method called *cut-through routing*, whereby a node can start relaying any portion of a packet to another node without waiting to receive the packet in its entirety. In the absence of additional traffic on the path, the delay of the packet is equal to the transmission time on the slowest link of the path plus the propagation delay, regardless of the number of links on the path. Thus, delay can be reduced by as much as a factor of n on a path with n links.

Note that the nature of cut-through routing is such that error detection and retransmission cannot be done on a link-by-link basis; it must be done on an end-to-end basis. Another issue is that pieces of the same packet may simultaneously be traveling on different links while other pieces are stored at different nodes. To keep the pieces of the packet together, one imposes the requirement that once a packet starts getting transmitted on a link, the link is reserved until all bits of the packet are transmitted. The reservation is made when the link transmits the packet's first bit, and it is released when the link transmits the packet's last bit. Thus, at any time a packet is reserving a portion of its path consisting of several links as shown in Fig. 5.10. Portions of the packet may be stored simultaneously at several nodes either because two or more links on the path have unequal transmission capacities, or because the packet's header had to wait in queue for a link to become available. To avoid the need for complex error recovery procedures, it is generally advisable to ensure that when the packet's first bit is transmitted by a link,

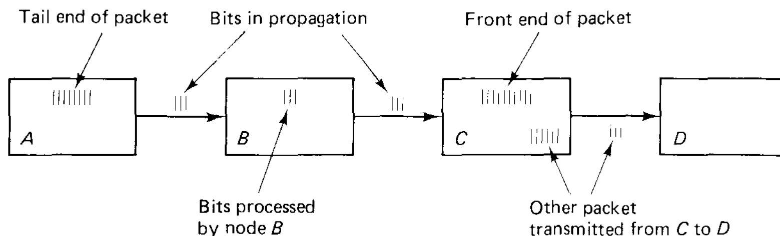


Figure 5.10 Illustration of cut-through routing. Here a packet originates at node A and is destined for node D, after going through nodes B and C. The header of the packet traveled to node B, found the link BC free, and continued to node C without waiting for the rest of the packet to arrive at B. Upon arrival to C, the header found that the link CD was in use, so it was forced to wait at node C. The header reserves links as it travels through them. A link reservation is relinquished only after the last bit of the packet goes through the link, so a packet stays together even though portions of it may be residing in several different nodes.

there is enough buffer space to store the entire packet at the other end of the link. Note that cut-through routing encounters a serious problem if a slow link is followed by a fast link along a packet's path. Then the fast link must effectively be slowed down to the speed of the slow link, and the associated idle fill issues may be hard to resolve.

A variation of cut-through routing, motivated by storage concerns, drops a packet once its first bit encounters a busy link along its path. Thus a packet is either successful in crossing its path without any queueing delay, or else it is discarded and must be retransmitted in its entirety, similar to circuit switching. Indeed, with some thought, it can be seen that this version of cut-through routing is equivalent to a form of circuit switching; the circuit's path is set up by the first bit of the packet, and once the circuit is established, its duration is equal to the packet's transmission time from origin to destination.

ARPANET: An example of datagram routing. The routing algorithm of the ARPANET, first implemented in 1969, has played an important historical role in the development of routing techniques. This was an ambitious, distributed, adaptive algorithm that stimulated considerable research on routing and distributed computation in general. On the other hand, the algorithm had some fundamental flaws that were finally corrected in 1979 when it was replaced by a new version. By that time, however, the original algorithm had been adopted by several other networks.

Shortest path routing is used in both ARPANET algorithms. The length of each link is defined by some measure of traffic congestion on the link, and is updated periodically. Thus, the ARPANET algorithms are adaptive and, since the ARPANET uses datagrams, two successive packets of the same session may follow different routes. This has two undesirable effects. First, packets can arrive at their destination out of order and must, therefore, be put back in order upon arrival. (This can also happen because of the data link protocol of the ARPANET, which uses eight logical channels; see the discussion in Chapter 2.) Second, the ARPANET algorithms are prone to oscillations. This phenomenon is explained in some detail in Section 5.2.5, but basically the idea is that selecting routes through one area of the network increases the lengths of the corresponding links. As a result, at the next routing update the algorithm tends to select routes through different areas. This makes the first area desirable at the subsequent routing update with an oscillation resulting. The feedback effect between link lengths and routing updates was a primary flaw of the original ARPANET algorithm that caused difficulties over several years and eventually led to its replacement. The latest algorithm is also prone to oscillations, but not nearly as much as the first (see Section 5.2.5).

In the original ARPANET algorithm, neighboring nodes exchanged their estimated shortest distances to each destination every 625 msec. The algorithm for updating the shortest distance estimate D_i of node i to a given destination is based on the Bellman–Ford method

$$d_i := \min_j [d_{ij} + D_j]$$

outlined previously and discussed further in Sections 5.2.3 and 5.2.4. Each link length d_{ij} was made dependent on the number of packets waiting in the queue of link (i, j) at the time of the update. Thus, link lengths were changing very rapidly, reflecting statistical traffic fluctuations as well as the effect of routing updates. To stabilize oscillations, a

large positive constant was added to the link lengths. Unfortunately, this reduced the sensitivity of the algorithm to traffic congestion.

In the second version of the ARPANET algorithm [MRR80], known as *Shortest Path First* (or SPF for short), the length of each link is calculated by keeping track of the delay of each packet in crossing the link. Each link length is updated periodically every 10 sec, and is taken to be the average packet delay on the link during the preceding 10-sec period. The delay of a packet on a link is defined as the time between the packet's arrival at the link's start node and the time the packet is correctly delivered to the link's end node (propagation delay is also included). Each node monitors the lengths of its outgoing links and broadcasts these lengths throughout the network at least once every 60 sec by using a flooding algorithm (see Section 5.3.1). Upon reception of a new link length, each node recalculates a shortest path from itself to each destination, using an incremental form of Dijkstra's algorithm (see Section 5.2.3). The algorithm is asynchronous in nature in that link length update messages are neither transmitted nor received simultaneously at all nodes. It turns out that this tends to improve the stability properties of the algorithm (see Section 5.2.5). Despite this fact, with increased traffic load, oscillations became serious enough to warrant a revision of the method for calculating link lengths. In the current method, implemented in 1987, the range of possible link lengths and the rate at which these lengths can change have been drastically reduced, with a substantial improvement of the algorithm's dynamic behavior ([KhZ89] and [ZVK89]).

Note that a node is not concerned with calculating routing paths that originate at other nodes even though the information available at each node (network topology plus lengths of all links) is sufficient to compute a shortest path from every origin to every destination. In fact, the routing tables of an ARPANET node contain only the first outgoing link of the shortest path from the node to each destination (see Fig. 5.11). When the node receives a new packet, it checks the packet's destination, consults its routing table, and places the packet in the corresponding outgoing link queue. If all nodes visited by a packet have calculated their routing tables on the basis of the same link length information, it is easily seen that the packet will travel along a shortest path. It is actually possible that a packet will occasionally travel in a loop because of changes in routing tables as the packet is in transit. However, looping of this type is rare, and when it happens, it is unlikely to persist.

The ARPANET routing tables can also be used conveniently for broadcasting a packet to a selected set of destinations. Multiple copies of the packet are transmitted along shortest paths to the destinations, with each copy guided by a header that includes a subset of the destinations. However, care is taken to avoid unnecessary proliferation of the copies. In particular, let D be the set of destinations and for every link l incident to the packet's origin, let D_l be the subset of destinations for which l is the first link on the shortest path from the origin (the origin knows D_l by looking at its routing tables). The origin starts the broadcast by transmitting a copy of the packet on each link l for which D_l is nonempty and includes the destination set D_l in the copy's header. Each node s that subsequently receives a copy of the packet repeats this process with D replaced by the set of remaining destinations (the set of destinations in the copy's header other than s; if s is the only destination on the copy, the node does not relay it further, as shown

Routing table of 2	
Destination	Out link
1	(2, 4)
3	(2, 4)
4	(2, 4)

Routing table of 1	
Destination	Out link
2	(1, 4)
3	(1, 4)
4	(1, 4)

Routing table of 3	
Destination	Out link
1	(3, 4)
2	(3, 4)
4	(3, 4)

Routing table of 4	
Destination	Out link
1	(4, 1)
2	(4, 2)
3	(4, 3)

Figure 5.11 Routing tables maintained by the nodes in the ARPANET algorithm. Each node calculates a shortest path from itself to each destination, and enters the first link on that path in its routing table.

in Fig. 5.12). It can be seen with a little thought that if the routing tables of the nodes are consistent in the sense that they are based on the same link length information, the packet will be broadcast along the shortest path tree to all its destinations and with a minimal number of transmissions.

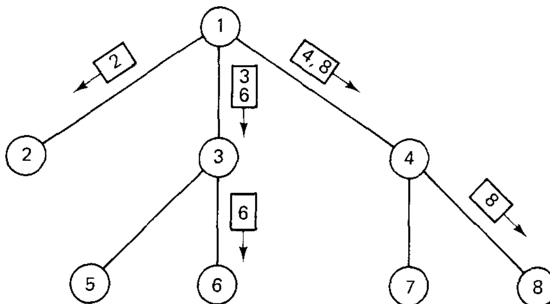


Figure 5.12 Broadcasting a packet to a selected set of destinations using the ARPANET routing tables. In this example, the broadcast is from node 1 to nodes 2, 3, 4, 6, and 8. The figure shows the links on which a copy of the packet is transmitted, the header of each copy, and the shortest path tree from node 1 to all destinations (each node actually maintains only its outgoing links on the tree). The number of transmissions is five. If a separate copy were to be transmitted to each destination, the number of transmissions would be seven.

TYMNET: An example of virtual circuit routing. The TYMNET routing algorithm, originally implemented in 1971, is based on the shortest path method, and is adaptive like the ARPANET algorithms. However, the implementation of the adaptive shortest path idea is very different in the two networks.

In the TYMNET, the routing algorithm is centralized and is operated at a special node called the supervisor. TYMNET uses virtual circuits, so a routing decision is needed only at the time a virtual circuit is set up. A virtual circuit request received by the supervisor is assigned to a shortest path connecting the origin and destination nodes of the virtual circuit. The supervisor maintains a list of current link lengths and does all the shortest path calculations. The length of each link depends on the load carried by the link as well as other factors, such as the type of the link and the type of virtual circuit being routed [Tym81]. While the algorithm can be classified as adaptive, it does not have the potential for oscillations of the ARPANET algorithms. This is due primarily to the use of virtual circuits rather than datagrams, as explained in Section 5.2.5.

Once the supervisor has decided upon the path to be used by a virtual circuit, it informs the nodes on the path and the necessary information is entered in each node's routing tables. These tables are structured as shown in Fig. 5.13. Basically, a virtual circuit is assigned a channel (or port) number on each link it crosses. The routing table at each node matches each virtual circuit's channel number on the incoming link with a channel number on the outgoing link. In the original version of TYMNET (now called TYMNET I), the supervisor maintains an image of the routing tables of all nodes and explicitly reads and writes the tables in the nodes. In the current version (TYMNET II), the nodes maintain their own tables. The supervisor establishes a new virtual circuit by sending a "needle" packet to the origin node. The needle contains routing information and threads its way through the network, building the circuit as it goes, with the user data following behind it.

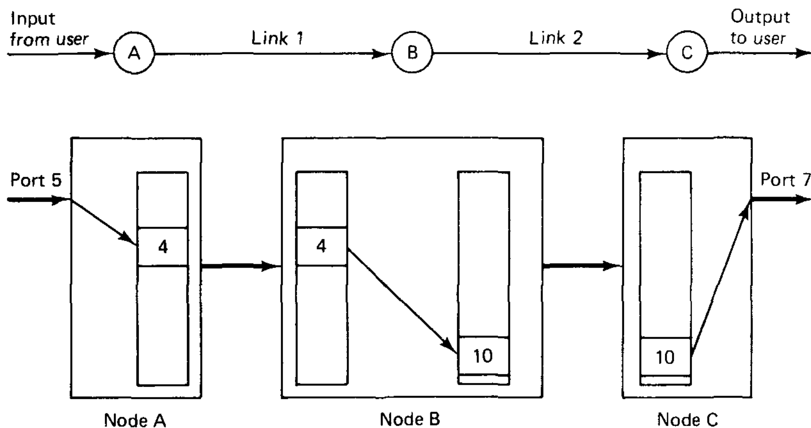


Figure 5.13 Structure of TYMNET routing tables. For the virtual circuit on the path ABC, the routing table at node A maps input port 5 onto channel 4 on link 1. At node B, incoming channel 4 is mapped into outgoing channel 10 on link 2. At node C, incoming channel 10 is mapped into output port 7.

The routing information includes the node sequence of the routing path, and some flags indicating circuit class. When a needle enters a TYMNET II node, its contents are checked. If the circuit terminates at this node, it is assigned to the port connected with the appropriate external site. Otherwise, the next node on the route is checked. If it is unknown (because of a recent link failure or some other error), the circuit is zapped back to its origin. Otherwise, the routing tables are updated and the needle is passed to the next node.

The TYMNET algorithm is well thought out and has performed well over the years. Several of its ideas were implemented in the Codex network, which is a similarly structured network (see Section 5.8). The Codex algorithm, however, is distributed and more sophisticated. A potential concern with the TYMNET algorithm is its vulnerability to failure of the supervisor node. In TYMNET, this is handled by using backup supervisor nodes. A distributed alternative, used in the Codex network, is to broadcast all link lengths to all nodes (as in the ARPANET), and to let each node choose the routing path for each circuit originating at that node. This path may be established by updating the routing tables of the nodes on the path, just as in TYMNET. Another possibility is to let the originating node include the routing path on each packet. This makes the packets longer but may save substantially in processing time at the nodes, when special switching hardware are used. For high-speed networks where the packet volume handled by the nodes is very large, it is essential to use such special hardware and to trade communication efficiency for reduced computation.

Routing in SNA. Routing in IBM's SNA is somewhat unusual in that the method for choosing routes is left partially to the user. SNA provides instead a framework within which a number of routing algorithm choices can be made. Specifically, for every origin–destination pair, SNA provides several paths along which virtual circuits can be built. The rule for assignment of virtual circuits to paths is subject to certain restrictions but is otherwise left unspecified. The preceding oversimplified description is couched in the general terminology of paths and virtual circuits used in this book. SNA uses a different terminology, which we now review.

The architecture of SNA does not fully conform to the OSI seven-layer architecture used as our primary model. The closest counterpart in SNA of the OSI network layer, called the *path control layer*, provides virtual circuit service to the immediately higher layer, called the *transmission control layer*. The path control layer has three functions: *transmission group control*, *explicit route control*, and *virtual route control*. The first fits most appropriately in the data link control layer of the OSI model, whereas the second and third correspond to the routing and flow control functions, respectively, of the network layer in the OSI model.

A transmission group in SNA terminology is a set of one or more physical communication lines between two neighboring nodes of the network. The transmission group control function makes the transmission group appear to higher layers as a single physical link. There may be more than one transmission group connecting a pair of nodes. The protocol places incoming packets on a queue, sends them out on the next available physical link, and sequences them if they arrive out of order. Resequencing of packets is done at every node, unlike the ARPANET, which resequences only at the destination node.

An explicit route in SNA terminology is what we have been referring to as a path within a subnet. Thus, an explicit route is simply a sequence of transmission groups providing a physical path between an origin and a destination node. There are several explicit routes provided (up to eight in SNA 4.2) for each origin–destination pair, which can be modified only under supervisory control. The method for choosing these explicit routes is left to the network manager. Each node stores the next node for each explicit route in its routing tables, and each packet carries an explicit route number. Therefore, when a packet arrives at a node, the next node on the packet's path (if any) is determined by a simple table lookup.

A virtual route in SNA terminology is essentially what we have been calling a virtual circuit. A user pair conversation (or session) uses only one virtual route. However, a virtual route can carry multiple sessions. Each virtual route belongs to a priority (or service) class. Each transmission group can serve up to three service classes, each with up to eight virtual routes, for a maximum of 24 virtual routes. When a user requests a session through a subnet entry node, a service class is specified on the basis of which the entry node will attempt to establish the session over one of its virtual routes. If all the communication lines of a transmission group fail, every session using that transmission group must be rerouted. If no alternative virtual route can be found, the session is aborted. When new transmission groups are established, all nodes are notified via special control packets, so each node has a copy of the network topology and knows which explicit routes are available and which are not.

Routing in circuit switching networks. Routing in circuit switching networks is conceptually not much different than routing in virtual circuit networks. In both cases a path is established at the time the circuit is set up, and all subsequent communication follows that path. In both cases the circuit will be blocked when the number of existing circuits on some link of the path reaches a certain threshold, although the choice of threshold may be based on different considerations in each case.

Despite these similarities, in practice, telephone networks have used quite different routing methods than virtual circuit networks. A common routing method in circuit switching networks, known as *hierarchical routing*, has been to use a fixed first-choice route, and to provide for one or more alternative routes, to be tried in a hierarchical order, for the case where the first-choice route is blocked. However, the differences in routing philosophy between circuit switching and virtual circuit networks have narrowed recently. In particular, there have been implementations of adaptive routing in telephone networks, whereby the path used by a call is chosen from several alternative paths on the basis of a measure of congestion (*e.g.*, the number of circuits these paths carry). We refer to [Ash90], [KeC90], [Kri90], [ReC90], and [WaO90] for detailed descriptions and discussion of such implementations.

5.1.3 Interconnected Network Routing: An Overview

As networks started to proliferate, it became necessary to interconnect them using various interface devices. In the case of wide-area networks, the interfaces are called *gateways*,

and usually perform fairly sophisticated network layer tasks, including routing. Gateways typically operate at the internet sublayer of the network layer. A number of engineering and economic factors, to be discussed shortly, motivated the interconnection of local area networks (LANs) at the MAC sublayer. The devices used for interconnection, known as *bridges*, perform a primitive form of routing. LANs can also be connected with each other or with wide-area networks using more sophisticated devices called *routers*. These provide fairly advanced routing functions at the network layer, possibly including adaptive and multiple-path routing. In this section we survey various routing schemes for interconnected networks, with a special emphasis on bridged LANs.

Conceptually, one may view an interconnected network of subnetworks as a single large network, where the interfaces (gateways, bridges, or routers) are additional nodes, each connected to two or more of the constituent subnetworks (see Fig. 5.14). Routing based on this conceptual model is called *nonhierarchical* and is similar to routing in a wide area network.

It is also possible to adopt a coarser-grain view of an interconnected network of subnetworks, whereby each subnetwork is considered to be a single node connected to interface nodes. Routing based on this model is called *hierarchical*, and consists of two levels. At the lower level, there is a separate routing algorithm for each subnetwork that handles local subnetwork traffic and also directs internetwork traffic to the subnetwork's interfaces with other subnetworks. At the higher level, there is a routing algorithm that determines the sequence of subnetworks and interface nodes that each internetwork packet must follow (see Fig. 5.15). Basically, the low-level algorithms handle "local" packets, and the high-level algorithm delivers "long-distance" packets to the appropriate low-level algorithm. An example of a hierarchical network is the Internet discussed in Section 2.8.4. Hierarchical routing is generally recommended for large internetworks; for example, telephone networks are typically organized hierarchically. We contrast the two approaches to internetworking by means of an example.

Example 5.3 Hierarchical versus Nonhierarchical Routing

Consider a country with several cities each having its own data network, and suppose that the city networks are connected into a national network using gateways. Each packet must have an internetwork destination address, consisting of a city address (or city ID number) and a local address (an ID number for the destination node within the destination city). Assume for concreteness that we want to use an ARPANET-like algorithm, where each

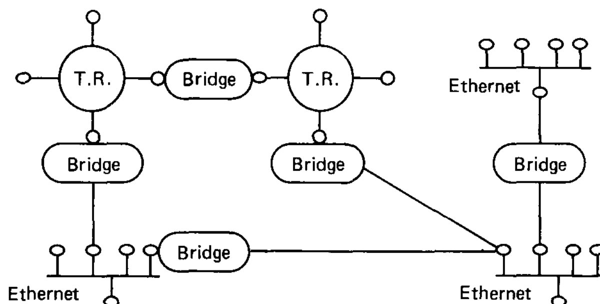


Figure 5.14 Conceptual view of an interconnected network of subnetworks. In a nonhierarchical approach to routing the internetwork is viewed as a single large network.

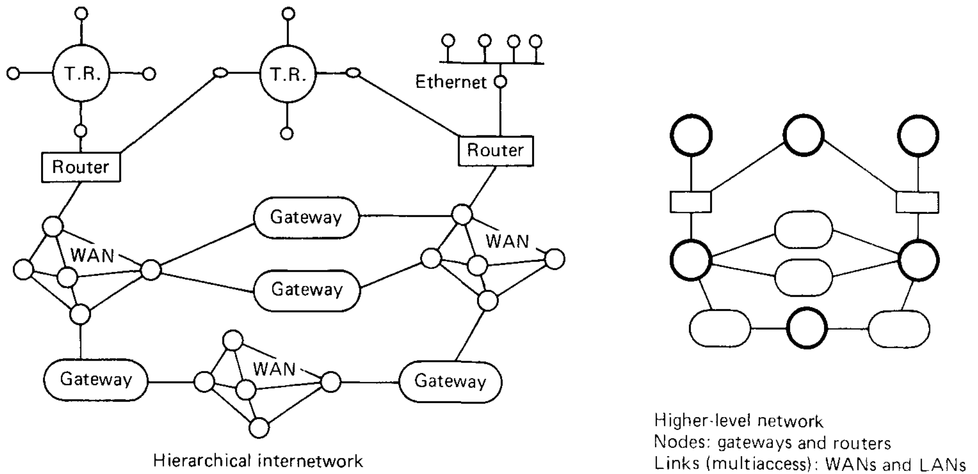


Figure 5.15 Conceptual view of hierarchical routing in an interconnected network of subnetworks. There is a lower-level algorithm that handles routing within each subnetwork and a higher-level algorithm that handles routing between subnetworks.

node has a unique entry in its routing tables for every destination (the next outgoing link on its shortest path to the destination).

In a nonhierachical routing approach, there is no distinction between the city networks; each node of each city must have a separate entry in its routing tables for every node in every other city. Thus the size of the routing tables is equal to the total number of nodes in the national network. Furthermore, a shortest path calculation to update the routing tables must be performed for the entire national network. Thus we see the main drawbacks of the nonhierarchical approach: the size of the routing tables and the calculations to update them may become excessive.

In a hierarchical approach, there is a lower-level (local) ARPANET-like routing algorithm within each city, and a higher-level ARPANET-like routing algorithm between cities. The local routing algorithm of a city operates only at the nodes of that city. Every node of a city has a routing table entry for every node of the same city and a routing table entry for every other city. The higher-level algorithm provides for routing between cities and operates only at the gateways. Each gateway has a routing table entry for every destination city. The latter entry is the next city network to which an internetwork packet must be routed on its way to its destination.

To see how this algorithm works, consider a packet generated at a node of some city A and destined for a node of a different city. The origin node checks the packet's address, consults its routing table entry for the packet's destination city, and sends the packet on the corresponding outgoing link. Each node of city A subsequently visited by the packet does the same, until the packet arrives at an appropriate gateway. At this point, the higher-level routing algorithm takes over and determines (based on the gateway's routing table) the next city network within which the packet will travel. The packet is then guided by the latter network's local routing algorithm, and the process is repeated until the packet arrives at the destination city's network. Then the packet is routed to its destination node by the local routing algorithm of the destination city.

The routing tables of a city network algorithm may be generated or updated using an adaptive shortest-path algorithm such as the ARPANET SPF algorithm. To do this, the nodes of the city network must have a destination gateway for every other city (the gateway connected to the city network to which packets destined for the other city will be guided by the local routing algorithm). This information must be provided by the higher-level algorithm that operates at the gateways. The higher-level algorithm may itself be static or adaptive, adjusting its intercity routes to temporary congestion patterns.

It can be seen from the preceding example that in the hierarchical approach, the size of the routing tables and the calculations to update them are considerably reduced over the nonhierarchical approach. In fact, the hierarchical approach may be recommended for a single large network (no internetworking) to reduce the routing algorithm's overhead and storage. The disadvantage of the hierarchical approach is that it may not result in routing that is optimal for the entire internetwork with respect to a shortest path or some other criterion.

Note that hierarchical networks may be organized in more than two levels. For example, consider an international network interconnecting several national networks. In a three-level hierarchy, there could be a routing algorithm between countries at the highest level, a routing algorithm between cities of each country at the middle level, and a routing algorithm within each city at the lowest level.

Bridged local area networks. As we discussed in Chapter 4, the performance of a local area network tends to degrade as its number of users increases. In the case of Ethernet-based LANs the performance also degrades as the ratio of propagation delay to transmission capacity increases. A remedy is to connect several LANs so that a large geographical area with many users can be covered without a severe performance penalty.

Routing for interconnected LANs has a special character because LANs operate in a "promiscuous mode," whereby all nodes, including the interface nodes connected to a LAN, hear all the traffic broadcast within the LAN. Furthermore, in a LAN internetwork, interface nodes must be fast and inexpensive, consistent with the characteristics of the LANs connected. This has motivated the use of simple, fast, and relatively inexpensive bridges. A bridge listens to all the packets transmitted within two or more LANs through connections known as *ports*. It also selectively relays some packets from one LAN to another. In this way, packets can travel from one user to another, going through a sequence of LANs and bridges.

An important characteristic of bridges is that they connect LANs at the MAC sublayer. The resulting internetwork is, by necessity, nonhierarchical; it may be viewed as a single network, referred to as the *extended LAN*. To alleviate the disadvantages of nonhierarchical routing mentioned earlier, bridges use some special routing techniques, the two most prominent of which are known as *spanning tree routing* and *source routing*. Both of these schemes assume that each station in the extended LAN has a unique ID number known through a directory that can be accessed by all stations. A station's ID number need not provide information regarding the LAN to which the station is connected. Also, the location of a station is not assumed known, thus allowing stations to be turned on and off and to be moved from one LAN to another at will. To appreciate

the difficulties of this, think of a telephone company trying to operate a network where customers can change their residence without notifying the company, while maintaining their phone numbers. We now describe the two major routing appoaches.

Spanning tree routing in bridged local area networks. In the spanning tree method, the key data structure is a spanning tree, which is used by a bridge to determine whether and to which LAN it should relay a packet (see Fig. 5.16). The spanning tree consists of a subset of ports that connect all LANs into a network with no cycles. An important property of a spanning tree is that each of its links separates the tree in two parts; that is, each node of the tree lies on exactly one "side" of the link. From this it is seen that there is a unique path from every LAN to every other LAN on the spanning tree. This path is used for communication of all packets between the LANs. The ports lying on the spanning tree are called *designated* or *active* ports. The spanning tree may need to be modified due to bridge failures, but we postpone a discussion of this issue for later, assuming for the time being that the spanning tree is fixed.

To understand how routing works, note that each bridge can communicate with any given station through a unique active port (see Fig. 5.16). Therefore, to implement spanning tree routing, it is sufficient to maintain for each active port a list of the stations with which the port communicates; this is known as the *forwarding database* (FDB) of the port. Assuming that the FDBs at each bridge are complete (they contain all stations), routing would be implemented as follows: If a packet is heard on the LAN corresponding to an active port A of a bridge, then if the destination ID of the packet is included in the FDB of an active bridge port B different from A, the packet is transmitted on the LAN connected to B; otherwise, the packet is discarded by the bridge (see Fig. 5.17).

Unfortunately, the routing method just described does not quite work because the FDBs of the active ports are not complete at all times. The reason is that stations may be turned on and off or change location, and also the spanning tree may change.

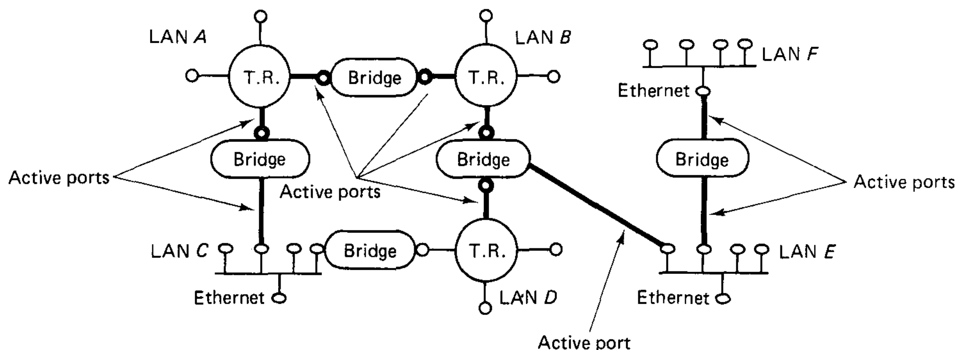


Figure 5.16 Illustration of spanning tree in a network of LANs interconnected with bridges. The spanning tree consists of a set of bridge ports (called active), which connect all LANs into a network without cycles. Only active ports are used for routing packets and thus there is a unique route connecting a pair of LANs. For example, a packet of a station connected to LAN C will go to a station in LAN E through LANs A and B.

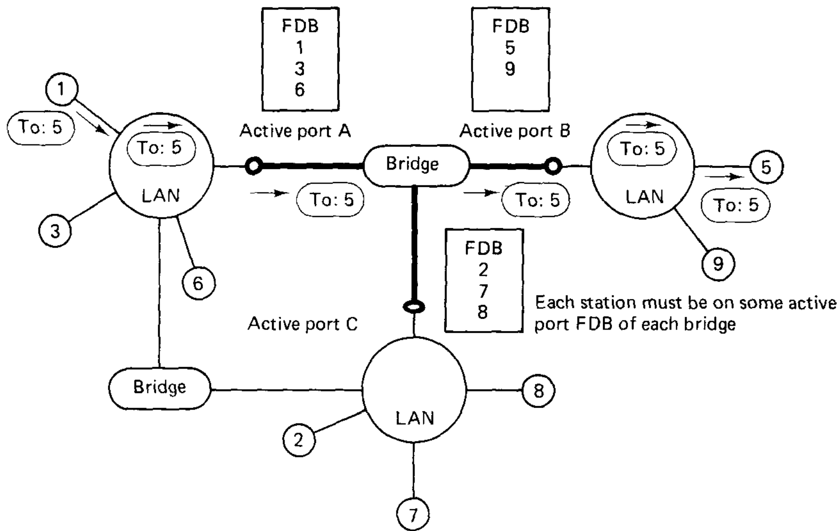


Figure 5.17 Using forwarding databases (FDBs) to effect spanning tree routing. Each active port maintains an FDB, that is, a list of the stations with which it communicates. An active port, say A, checks the destination of each packet transmitted within its associated LAN. If the destination appears in the FDB of another active port of the same bridge, say B, the bridge broadcasts the packet on the LAN connected with port B.

For this reason, the FDBs are constantly being updated through a process known as *bridge learning*. Each active port adds to its FDB the source ID numbers of the packets transmitted on the corresponding LAN. Also each active port deletes from its FDB the ID numbers of stations whose packets have not been transmitted on the corresponding LAN for a certain period of time; this helps to deal with situations where a station is disconnected with one LAN and is connected to another, and also cleans up automatically the FDBs when the spanning tree changes. When a port first becomes active its FDB is initialized to empty.

Since not all station IDs will become known through bridge learning, the routing method includes a search feature. If the destination ID of an incoming packet at some active port is unknown at each of the bridge's FDBs, the bridge relays the packet on all the other active ports. An illustration of the algorithm's operation in this case is shown in Fig. 5.18. Here station 1 sends a packet P to station 2, but because 2 has not yet transmitted any packet, its ID number does not appear on any FDB. Then P is broadcast on the entire spanning tree, eventually reaching the destination 2. When 2 answers P with a packet R of its own, this packet will follow the unique spanning tree route to 1 (because 1's ID number was entered in all of the appropriate FDBs of active ports of the spanning tree when P was broadcast through the spanning tree). As a result, 2's ID number will be entered in the FDBs of the active ports along the route that connects 2 with 1, and a subsequent packet from 1 to 2 will follow this route.

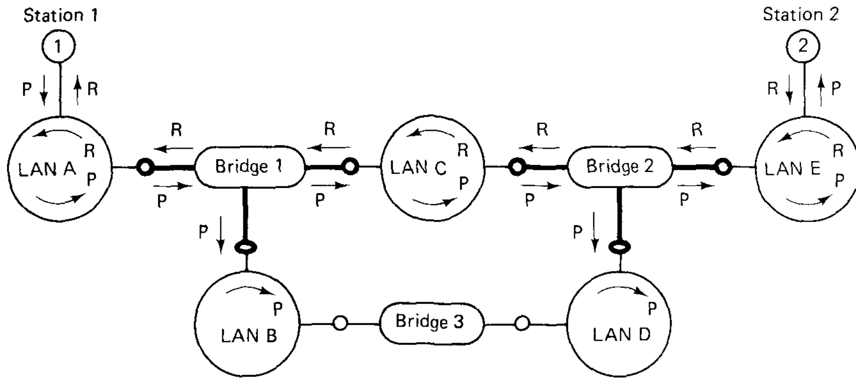


Figure 5.18 Illustration of bridge learning and FDB updating. Station 1 sends a packet P to station 2, but the ID of station 2 does not appear in the FDB of any active port of bridge 1 (because, for example, station 2 has not transmitted any packet for a long time). Packet P is broadcast into LANs B and C by bridge 1, and then it is broadcast into LANs D and E by bridge 2. In the process the ID of node 1 is entered in the left-side FDBs of bridges 1 and 2 (if not already there). Station 2 replies with a packet R, which goes directly to its destination station 1 through bridges 2 and 1, because the ID number of station 1 appears in the left-side FDBs of these bridges.

The method by which the spanning tree is updated in the event of a bridge or LAN failure or repair remains to be discussed. There are several algorithms for constructing spanning trees, some of which are discussed in Section 5.2. What is needed here, however, is a distributed algorithm that can work in the face of multiple and unpredictable failures and repairs. A relatively simple possibility is to use a spanning tree of shortest paths from every LAN and bridge to a particular destination (or root). We will see in Section 5.2 that the Bellman–Ford algorithm is particularly well suited for this purpose. There is still, however, an extra difficulty, namely how to get all bridges to agree on the root and what to do when the root itself fails. Such issues involve subtle difficulties, which are unfortunately generic in distributed algorithms dealing with component failures. The difficulty can be resolved by numbering the bridges and by designating as root of the tree the bridge with the smallest ID number. Each bridge keeps track of the smallest bridge ID number that it knows of. A bridge must periodically initiate a spanning tree construction algorithm if it views itself as the root based on its own ID number and the ID numbers of other bridges it is aware of. Also, each bridge ignores messages from bridges other than its current root. To cope with the situation where the root fails, bridges may broadcast periodically special "hello" messages to other bridges. These messages may also carry additional information relating to the spanning tree construction and updating algorithm. (See [Bac88] for a description of the corresponding IEEE 802.1 MAC Bridge Draft Standard.)

Source routing in bridged local area networks. Source routing is an alternative to the spanning tree method for bridged LANs. The idea is to discover a route between two stations A and B wishing to communicate and then to include the route on the header of each packet exchanged between A and B. There are two phases here:

1. Locating B through the extended LAN. This can be done by broadcasting an exploratory packet from A through the extended LAN (either flooding or a spanning tree can be used for this).
2. Selecting a route out of the many possible that connect A and B. One way of doing this is to send a packet from B to A along each possible loop-free route. The sequence of LANs and bridges is recorded on the packet as it travels along its route. Thus A gets to know all possible routes to B and can select one of these routes using, for example, a shortest path criterion.

Note the similarity of source routing with virtual circuit routing; in both cases, all packets of a user pair follow the same path.

The main advantage of the spanning tree method is that it does not require any cooperation of the user/stations in the routing algorithm. By contrast, in source routing, the origin station must select a route along which to establish a virtual circuit. Thus the software of the stations must provide for this capability. This is unnecessary in spanning tree routing, where the routing process is transparent to the stations.

The main advantage of source routing is that with each packet having its route stamped on it, there is no need for routing table lookup at the bridges. This simplifies packet processing and may result in bridges with higher packet throughput. Another advantage of source routing is that it may use the available network transmission capacity more efficiently. While spanning tree routing uses a single possibly nonshortest path for each origin–destination pair, with source routing, shortest path and multiple-path routing can be used. Figure 5.19 shows an example where spanning tree–based routing is inefficient.

We finally mention that both spanning tree and source routing suffer from a common disadvantage. They must often broadcast packets throughout the network either to update FDBs or to establish new connections between user pairs. These broadcasts are needed

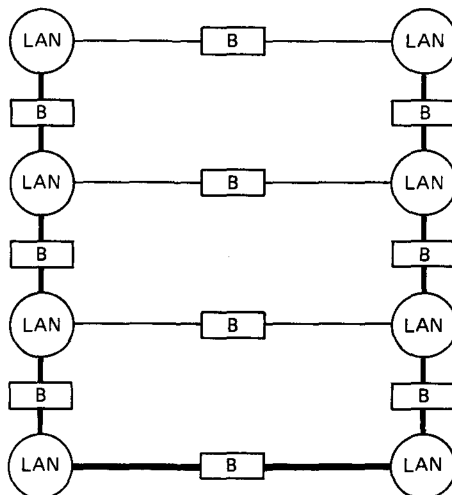


Figure 5.19 Example of inefficient spanning tree routing. The routing paths are good for origin LANs near the bottom of the figure but poor for origin and destination LANs near the top of the figure.

to allow dynamic relocation of the stations while keeping the bridge hardware simple. They may, however, become a source of congestion.

5.2 NETWORK ALGORITHMS AND SHORTEST PATH ROUTING

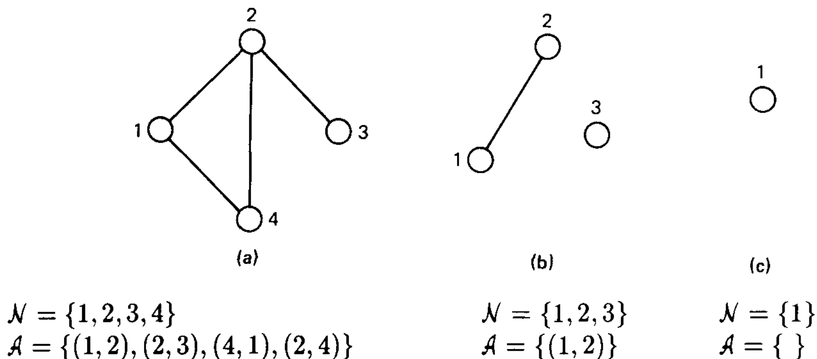
Routing methods involve the use of a number of simple graph-theoretic problems such as the shortest path problem, discussed in the preceding section. In this section we consider shortest path routing in detail together with related problems of interest. We start by introducing some basic graph-theoretic notions.

5.2.1 Undirected Graphs

We define a *graph*, $G = (\mathcal{N}, \mathcal{A})$, to be a finite nonempty set $\mathcal{N}$ of *nodes* and a collection $\mathcal{A}$ of pairs of distinct nodes from $\mathcal{N}$. Each pair of nodes in $\mathcal{A}$ is called an *arc*. Several examples of graphs are given in Fig. 5.20. The major purpose of the formal definition $G = (\mathcal{N}, \mathcal{A})$ is to stress that the location of the nodes and the shapes of the arcs in a pictorial representation are totally irrelevant. If n_1 and n_2 are nodes in a graph and (n_1, n_2) is an arc, this arc is said to be *incident* on n_1 and n_2. Some authors define graphs so as to allow arcs to have both ends incident on the same node, but we have intentionally disallowed such loops. We have also disallowed multiple arcs between the same pair of nodes.

A *walk* in a graph G is a sequence of nodes $(n_1, n_2, \ldots, n_\ell)$ of nodes such that each of the pairs (n_1, n_2), (n_2, n_3), ..., $(n_{\ell-1}, n_\ell)$ are arcs of G. A walk with no repeated nodes is a *path*. A walk $(n_1, ..., n_\ell)$ with $n_1 = n_\ell$, $\ell > 3$, and no repeated nodes other than $n_1 = n_\ell$ is called a *cycle*. These definitions are illustrated in Fig. 5.21.

We say that a graph is *connected* if for each node i there is a path $(i = n_1, n_2, \ldots, n_\ell = j)$ to each other node j. Note that the graphs in Fig. 5.20(a) and (c) are connected, whereas the graph in Fig. 5.20(b) is not connected. We spot an absence of connectivity in a graph by seeing two sets of nodes with no arcs between them. The following lemma captures this insight. The proof is almost immediate and is left as an exercise.



$\mathcal{N} = \{1,2,3,4\}$
$\mathcal{A} = \{(1,2),(2,3),(4,1),(2,4)\}$

$\mathcal{N} = \{1,2,3\}$
$\mathcal{A} = \{(1,2)\}$

$\mathcal{N} = \{1\}$
$\mathcal{A} = \{\ \}$

Figure 5.20 Examples of graphs with a set of nodes $\mathcal{N}$ and a set of arcs $\mathcal{A}$.

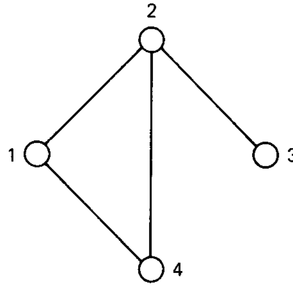


Figure 5.21 Illustration of walks, paths, and cycles. The sequences (1,4,2,3), (1,4,2,1), (1,4,2,1,4,1), (2,3,2), and (2) are all walks. The sequences (1,4,2,3) and (2) are each paths; and (1,4,2,1) is a cycle. Note that (2) and (2,3,2) are not considered cycles.

Lemma. Let G be a connected graph and let S be any nonempty strict subset of the set of nodes N. Then at least one arc (i, j) exists with $i \in S$ and $j \notin S$.

We say that $G' = (\mathcal{N}', \mathcal{A}')$ is a *subgraph* of $G = (\mathcal{N}, \mathcal{A})$ if G' is a graph, $\mathcal{N}' \subset \mathcal{N}$, and $\mathcal{A}' \subset \mathcal{A}$. For example, the last three graphs in Fig. 5.22 are subgraphs of the first.

A *tree* is a connected graph that contains no cycles. A *spanning tree* of a graph G is a subgraph of G that is a tree and that includes all the nodes of G. For example, the subgraphs in Fig. 5.22(b) and (c) are spanning trees of the graph in Fig. 5.22(a). The following simple algorithm constructs a spanning tree of an arbitrary connected graph $G = (\mathcal{N}, \mathcal{A})$:

1. Let n be an arbitrary node in $\mathcal{N}$. Let $\mathcal{N}' = \{n\}$, and let $\mathcal{A}'$ be the empty set $\emptyset$.
2. If $\mathcal{N}' = \mathcal{N}$, then stop [$G' = (\mathcal{N}', \mathcal{A}')$ is a spanning tree]; else go to step 3.
3. Let $(i, j) \in \mathcal{A}$ be an arc with $i \in \mathcal{N}'$, $j \in \mathcal{N} - \mathcal{N}'$. Update $\mathcal{N}'$ and $\mathcal{A}'$ by

$$\mathcal{N}' := \mathcal{N}' \cup \{j\}$$
$$\mathcal{A}' := \mathcal{A}' \cup \{(i, j)\}$$

Go to step 2.

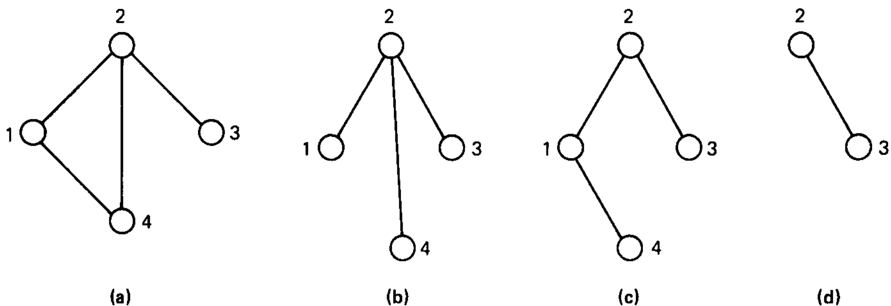


Figure 5.22 A graph (a) and three subgraphs (b), (c), and (d). Subgraphs (b) and (c) are spanning trees.

To see why the algorithm works, note that step 3 is entered only when $\mathcal{N}'$ is a proper subset of $\mathcal{N}$, so that the earlier lemma guarantees the existence of the arc (i, j). We use induction on successive executions of step 3 to see that $G' = (\mathcal{N}', \mathcal{A}')$ is always a tree. Initially, $G' = (\{n\}, \emptyset)$ is trivially a tree, so assume that $G' = (\mathcal{N}', \mathcal{A}')$ is a tree before the update of step 3. This ensures that there is a path between each pair of nodes in $\mathcal{N}'$ using arcs of $\mathcal{A}'$. After adding node j and arc (i, j), each node has a path to j simply by adding j to the path to i, and similarly j has a path to each other node. Finally, node j cannot be in any cycles since (i, j) is the only arc of G' incident to j, and there are no cycles not including j by the inductive hypothesis. Figure 5.23 shows G' after each execution of step 3 for one possible choice of arcs.

Observe that the algorithm starts with a subgraph of one node and zero arcs and adds one node and one arc on each execution of step 3. This means that the spanning tree, G', resulting from the algorithm always has N nodes, where N is the number of nodes in G, and $N - 1$ arcs. Since G' is a subgraph of G, the number of arcs, A, in G must satisfy $A \geq N - 1$; this is true for every connected graph G. Next, assume that $A = N - 1$. This means that the algorithm uses all arcs of G in the spanning tree G', so that $G = G'$, and G must be a tree itself. Finally, if $A \geq N$, then G contains at least one arc (i, j) not in the spanning tree G' generated by the algorithm. Letting $(j, \ldots, i)$ be the path from j to i in G', it is seen that $(i, j, \ldots, i)$ is a cycle in G and G cannot be a tree. The following proposition summarizes these results:

Proposition. Let G be a connected graph with N nodes and A arcs. Then:

1. G contains a spanning tree.
2. $A \geq N - 1$.
3. G is a tree if and only if $A = N - 1$.

Figure 5.24 shows why connectedness is a necessary assumption for the last part of the proposition.

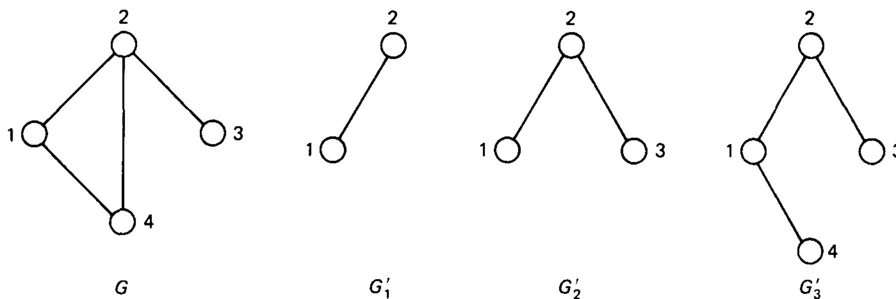


Figure 5.23 Algorithm for constructing a spanning tree of a given graph G.

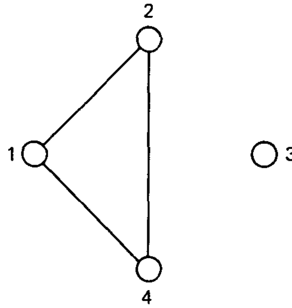


Figure 5.24 Graph with $A = N - 1$, which is both disconnected and contains a cycle.

5.2.2 Minimum Weight Spanning Trees

It is possible to construct a spanning tree rooted at some node by sending a packet from the node to all its neighbors; the neighbors must send the packet to their neighbors, and so on, with all nodes marking the transmitter of the first packet they receive as their "father" on the tree, as described in Fig. 5.8. However, such a tree has no special properties. If the communication cost or delay of different links should be taken into account in constructing a spanning tree, we may consider using a minimum weight spanning tree, which we now define.

Consider a connected graph $G = (\mathcal{N}, \mathcal{A})$ with N nodes and A arcs, and a weight w_{ij} for each arc $(i, j) \in \mathcal{A}$. A *minimum weight spanning tree* (MST for short) is a spanning tree with minimum sum of arc weights. An arc weight represents the communication cost of a message along the arc in either direction, and the total spanning tree weight represents the cost of broadcasting a message to all nodes along the spanning tree.

Any subtree (*i.e.*, a subgraph that is a tree) of an MST will be called a *fragment*. Note that a node by itself is considered a fragment. An arc having one node in a fragment and the other node not in this fragment is called an *outgoing arc* from the fragment. The following proposition is of central importance for MST algorithms.

Proposition 1. Given a fragment F, let $\alpha = (i, j)$ be a minimum weight outgoing arc from F, where the node j is not in F. Then F, extended by arc α and node j, is a fragment.

Proof: Denote by M the MST of which F is a subtree. If arc α belongs to M, we are done, so assume otherwise. Then there is a cycle formed by α and the arcs of M. Since node j does not belong to F, there must be some arc $\beta \neq \alpha$ that belongs to the cycle and to M, and which is outgoing from F (see Fig. 5.25). Deleting β from M and adding α results in a subgraph M' with $(N - 1)$ arcs and no cycles which, therefore, must be a spanning tree. Since α has less or equal weight to β, M' must be an MST, so F extended by α and j forms a fragment. **Q.E.D.**

Proposition 1 can be used as the basis for MST construction algorithms. The idea is to start with one or more disjoint fragments and enlarge or combine them by adding minimum weight outgoing arcs. One method (the Prim–Dijkstra algorithm) starts with an

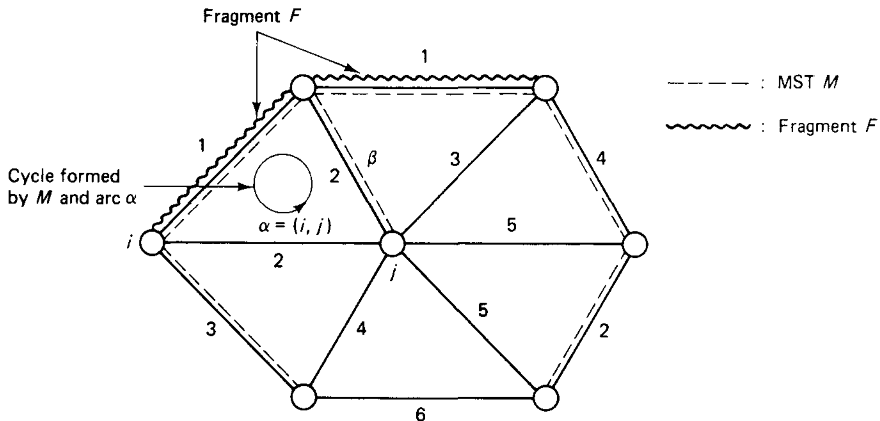


Figure 5.25 Proof of Proposition 1. The numbers next to the arcs are the arc weights. F is a fragment which is a subtree of an MST M. Let α be a minimum weight outgoing arc from F not belonging to M. Let $\beta \neq \alpha$ be an arc that is outgoing from F and simultaneously belongs to M and to the cycle formed by α and M. Deleting β from M and adding α in its place forms another MST M'. When F is extended by α, it forms a fragment.

arbitrarily selected single node as a fragment and enlarges the fragment by successively adding a minimum weight outgoing arc. Another method (Kruskal's algorithm) starts with each node being a single node fragment; it then successively combines two of the fragments by using the arc that has minimum weight over all arcs that when added to the current set of fragments do not form a cycle (see Fig. 5.26). Both of these algorithms terminate in $N - 1$ iterations.

Kruskal's algorithm proceeds by building up simultaneously several fragments that eventually join into an MST; however, only one arc at a time is added to the current set of fragments. Proposition 1 suggests the possibility of adding a minimum weight outgoing arc simultaneously to each fragment in a distributed algorithmic manner. This is possible if there is a unique MST, as we now discuss.

A distributed algorithm for constructing the MST in a graph with a unique MST is as follows. Start with a set of fragments (these can be the nodes by themselves, for example). Each fragment determines its minimum weight outgoing arc, adds it to itself, and informs the fragment that lies at the other end of this arc. It can be seen that as long as the arc along which two fragments join is indeed a minimum weight outgoing arc for some fragment, the algorithm maintains a set of fragments of the MST at all times, and no cycle is ever formed. Furthermore, new arcs will be added until there are no further outgoing arcs and there is only one fragment (by necessity the MST). Therefore, the algorithm cannot stop short of finding the MST. Indeed, for the algorithm to work correctly, it is not necessary that the procedure for arc addition be synchronized for all fragments. What is needed, however, is a scheme for the nodes and arcs of a fragment to determine the minimum weight outgoing arc. There are a number of possibilities along these lines, but it is of interest to construct schemes that accomplish this with low

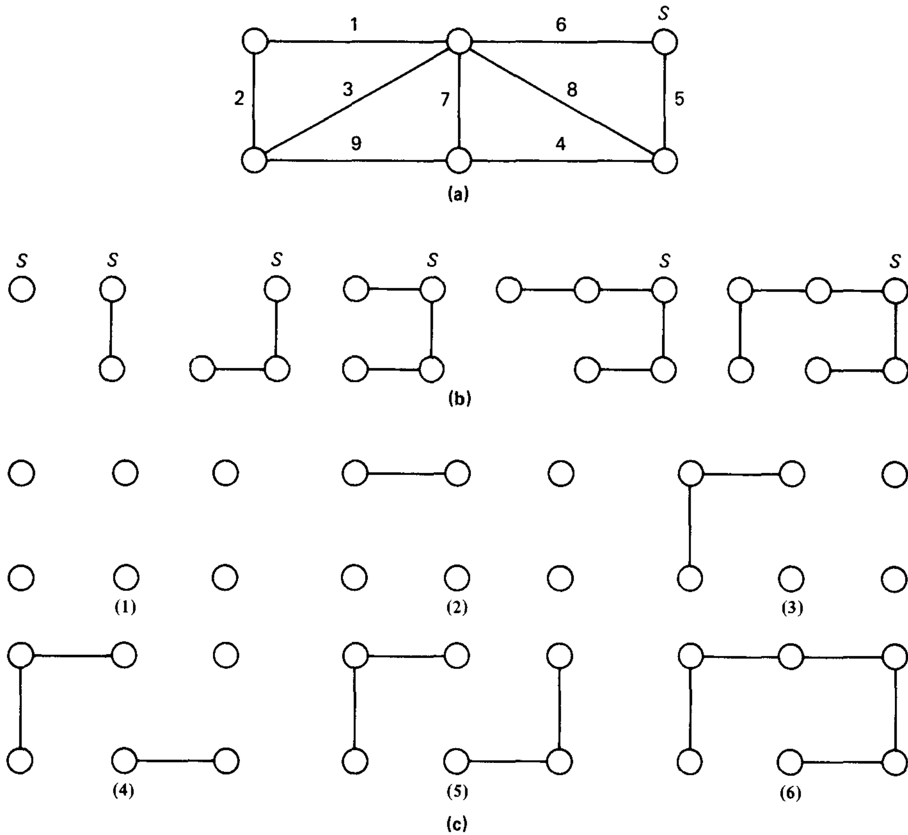


Figure 5.26 Minimum weight spanning tree construction. (a) Graph with arc weights as indicated. (b) Successive iterations of the Prim–Dijkstra algorithm. The starting fragment consists of the single node marked S. The fragment is successively extended by adding a minimum weight outgoing arc. (c) Successive iterations of the Kruskal algorithm. The algorithm starts with all nodes as fragments. At each iteration, we add the arc that has minimum weight out of all arcs that are outgoing from one of the fragments.

communication overhead. This subject is addressed in [GHS83], to which we refer for further details. Reference [Hum83] considers the case where the arc weights are different in each direction. A distributed MST algorithm of the type just described has been used in the PARIS network [CGK90] (see Section 6.4 for a description of this network).

To see what can go wrong in the case of nonunique MSTs, consider the triangular network of Fig. 5.27 where all arc lengths are unity. If we start with the three nodes as fragments and allow each fragment to add to itself an arbitrary, minimum weight outgoing arc, there is the possibility that the arcs (1,2), (2,3), and (3,1) will be added simultaneously by nodes 1, 2, and 3, respectively, with a cycle resulting.

The following proposition points the way on how to handle the case of nonunique MSTs.

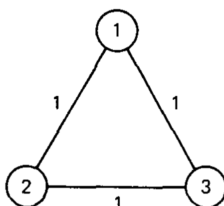


Figure 5.27 Counterexample for distributed MST construction with nondistinct arc weights. Nodes 1, 2, and 3 can add simultaneously to their fragments arcs (1,2), (2,3), and (3,1), respectively, with a cycle resulting.

Proposition 2. If all arc weights are distinct, there exists a unique MST.

Proof: Suppose that there exist two distinct MSTs denoted M and M'. Let α be the minimum weight arc that belongs to either M or M' but not to both. Assume for concreteness that α belongs to M. Consider the graph formed by the union of M' and α (see Fig. 5.28). It must contain a cycle (since $\alpha \notin M'$), and at least one arc of this cycle, call it β, does not belong to M (otherwise M would contain a cycle). Since the weight of α is strictly smaller than that of β, it follows that deleting β from M' and adding α in its place results in a spanning tree of strictly smaller weight than M'. This is a contradiction since M' is optimal. **Q.E.D.**

Consider now an MST problem where the arc weights are not distinct. The ties between arcs with the same weight can be broken by using the identities of their nodes; that is, if arcs (i, j) and (k, l) with $i < j$ and $k < l$ have equal weight, prefer arc (i, j) if $i < k$ or if $i = k$ and $j < l$, and prefer arc (k, l) otherwise. Thus, by implementing, if necessary, the scheme just described, one can assume without loss of generality that arc weights are distinct and that the MST is unique, thereby allowing the use of distributed MST algorithms that join fragments along minimum weight outgoing arcs.

5.2.3 Shortest Path Algorithms

In what follows we shall be interested in traffic flowing over the arcs of a network. This makes it necessary to distinguish the direction of the flow, and thus to give a reference direction to the arcs themselves.

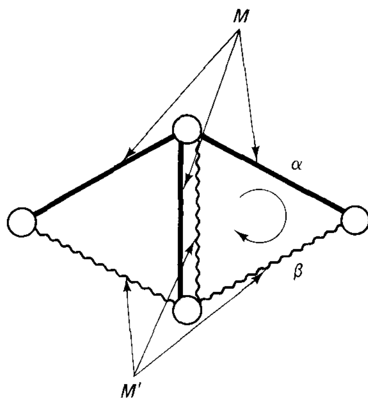


Figure 5.28 Proof of MST uniqueness when all arc weights are distinct. Let M and M' be two MSTs and assume that α is the minimum weight arc that belongs to M and M' but not both. Suppose that $\alpha \in M$ and let β be an arc of the cycle of $M' \cup \{\alpha\}$ which does not belong to M. By deleting β from M' and adding α in its place, we obtain a spanning tree because the corresponding subgraph has $N - 1$ arcs and is seen to be connected. The weight of this spanning tree is smaller than the one of M', because the weight of α is smaller than the weight of β—a contradiction.

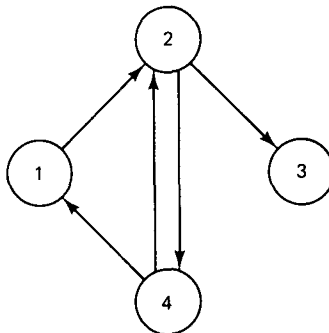
A *directed graph* or *digraph* $G = (\mathcal{N}, \mathcal{A})$ is a finite nonempty set $\mathcal{N}$ of nodes and a collection $\mathcal{A}$ of *ordered* pairs of distinct nodes from $\mathcal{N}$; each ordered pair of nodes in $\mathcal{A}$ is called a *directed arc* (or simply arc). Pictorially, 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 $G = (\mathcal{N}, \mathcal{A})$, there is an associated (undirected) graph $G' = (\mathcal{N}', \mathcal{A}')$, where $\mathcal{N}' = \mathcal{N}$ and $(i,j) \in \mathcal{A}'$ if either $(i,j) \in \mathcal{A}$, or $(j,i) \in \mathcal{A}$, or both. We say that $(n_1, n_2, ..., n_\ell)$ is a walk, path, or cycle in a digraph if it is a walk, path, or cycle in the associated graph. In addition, $(n_1, n_2, ..., n_\ell)$ is a *directed walk* in a digraph G if (n_i, n_{i+1}) is a directed arc in G for $1 \leq i \leq \ell - 1$. A *directed path* is a directed walk with no repeated nodes, and a *directed cycle* is a directed walk $(n_1, \ldots, n_\ell)$, for $\ell > 2$ with $n_1 = n_\ell$ and no repeated nodes. Note that (n_1, n_2, n_1) is a directed cycle if (n_1, n_2) and (n_2, n_1) are both directed arcs, whereas (n_1, n_2, n_1) 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 $(i = n_1, n_2, \ldots, n_\ell = j)$ 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 $G = (\mathcal{N}, \mathcal{A})$ 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 $p = (i, j, k, \ldots, \ell, m)$, the *length* of p is defined as $d_{ij} + d_{jk} + \cdots + d_{\ell m}$. The length of a directed walk or cycle is defined analogously. Given any two nodes i, 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. If 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



$\mathcal{N} = \{1, 2, 3, 4\}$

$\mathcal{A} = \{(1,2), (2,3), (2,4), (4,2), (4,1)\}$

Figure 5.29 Representation of a directed graph.

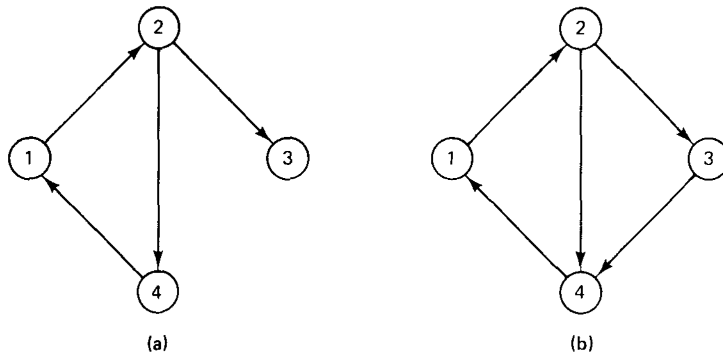


Figure 5.30 Both digraphs (a) and (b) are connected; (b) is strongly connected, but (a) is not strongly connected since there is no directed path from 3 to 2.

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 m with arc distances $(-\ln p_{ij})$ is equivalent to finding the most reliable path from i to m.

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 ∞.

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_i^h. 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

$$D_1^h = 0, \qquad \text{for all } h$$

We will prove shortly that D_i^h can be generated by the iteration

$$D_i^{h+1} = \min_j \left[d_{ij} + D_j^h\right], \qquad \text{for all } i \neq 1 \tag{5.1}$$

starting from the initial conditions

$$D_i^0 = \infty, \qquad \text{for all } i \neq 1 \tag{5.2}$$

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

$$D_i^h = D_i^{h-1}, \qquad \text{for all } i$$

The following proposition provides the main results.

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

(a) The scalars D_i^h generated by the algorithm are equal to the shortest ($\leq h$) walk lengths from node i 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_i^h is the shortest path length from i to 1.

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

$$D_i^1 = d_{i1}, \qquad \text{for all } i \neq 1$$

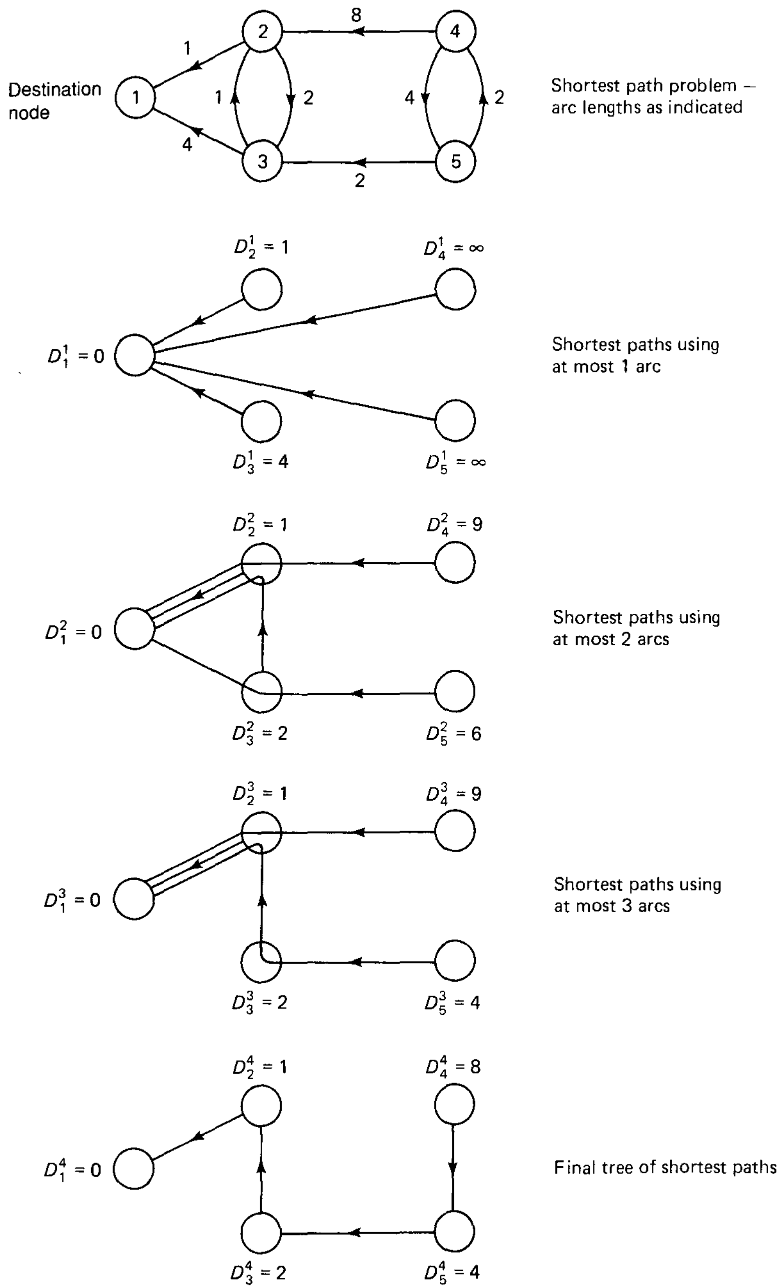


Figure 5.31 Successive iterations of the Bellman–Ford method. In this example, the shortest ($\leq h$) walks are paths because all arc lengths are positive and therefore all cycles have positive length. The shortest paths are found after $N - 1$ iterations, which is equal to 4 in this example.

so D_i^1 is indeed equal to the shortest (≤ 1) walk length from i to 1. Suppose that D_i^k 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

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

Using the induction hypothesis, we have $D_j^k \leq D_j^{k-1}$ 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,

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

Furthermore, we have $D_i^h \leq D_i^1 = d_{i1} = d_{i1} + D_1^h$, so from Eq. (5.3) we obtain

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

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

$$D_i^k = D_i^h, \qquad \text{for all } i \text{ and } k \geq h \tag{5.5}$$

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

$$D_i^N = D_i^{N-1}, \qquad \text{for all } i$$

implying that the algorithm terminates after at most N iterations. **Q.E.D.**

Note that the preceding proposition is valid even if there is no path from some nodes i to node 1 in the original network. Upon termination, we will simply have for those nodes $D_i^h = \infty$.

To estimate the computation required to find the shortest path lengths, we note that in the worst case, the algorithm must be iterated N times, each iteration must be done for $N - 1$ nodes, and for each node the minimization must be taken over no more than $N - 1$ alternatives. Thus, the amount of computation grows at worst like N^3, which is written as $O(N^3)$. Generally, the notation $O\big(p(N)\big)$, where $p(N)$ is a polynomial in N, is used to indicate a number depending on N that is smaller than $cp(N)$ for all N, where c is some constant independent of N. Actually, a more careful accounting shows that the amount of computation is $O(mA)$, where A is the number of arcs and m is the number of iterations required for termination (m is also the maximum number of arcs contained in a shortest path).

The example in Fig. 5.32 shows the effect of negative-length cycles not involving node 1, and illustrates that one can test for existence of such cycles simply by comparing D_i^N with D_i^{N-1} for each i. As implied by part (b) of the preceding proposition, there exists such a negative-length cycle if and only if $D_i^N < D_i^{N-1}$ for some i.

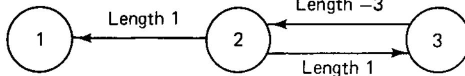


Figure 5.32 Graph with a negative cycle. The shortest path length from 2 to 1 is 1. The Bellman–Ford algorithm gives $D_2^2 = 1$ and $D_2^3 = -1$, indicating the existence of a negative-length cycle.

Bellman's equation and shortest path construction. Assume that all cycles not containing node 1 have nonnegative length and denote by D_i the shortest path length from node i to 1. Then upon termination of the Bellman–Ford algorithm, we obtain

$$D_i = \min_j \left[d_{ij} + D_j\right], \qquad \text{for all } i \neq 1 \tag{5.6a}$$

$$D_1 = 0 \tag{5.6b}$$

This is called *Bellman's equation* and expresses that the shortest path length from node i to 1 is the sum of the length of the arc to the node following i on the shortest path plus the shortest path length from that node to node 1. From this equation it is easy to find the shortest paths (as opposed to the shortest path lengths) if all cycles not including node 1 have a positive length (as opposed to zero length). To do this, select, for each $i \neq 1$, one arc (i, j) that attains the minimum in the equation $D_i = \min_j[d_{ij} + D_j]$, and consider the subgraph consisting of these $N - 1$ arcs (see Fig. 5.33). To find the shortest path from any node i, start at i and follow the corresponding arcs of the subgraph until node 1 is reached. Note that the same node cannot be reached twice before reaching node 1, since a cycle would be formed that (on the basis of the equation $D_i = \min_j[d_{ij} + D_j]$) would have zero length [let $(i_1, i_2, \ldots, i_k, i_1)$ be the cycle and add the equations

$$D_{i_1} = d_{i_1 i_2} + D_{i_2}$$

$$\cdots$$

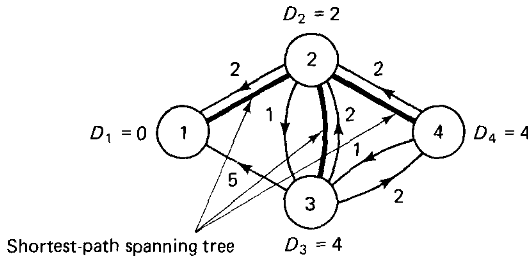


Figure 5.33 Construction of a shortest path spanning tree. For every $i \neq 1$, a neighbor node j_i minimizing $d_{ij} + D_j$ is selected. The tree of the $N - 1$ arcs (i, j_i) defines a set of shortest paths.

$$D_{i_{k-1}} = d_{i_{k-1}i_k} + D_{i_k}$$
$$D_{i_k} = d_{i_k i_1} + D_{i_1}$$

obtaining $d_{i_1 i_2} + \cdots + d_{i_k i_1} = 0$]. Since the subgraph connects every node to node 1 and has $N - 1$ arcs, it must be a spanning tree. We call this subgraph the *shortest path spanning tree* and note that it has the special structure of having a root (node 1), with every arc of the tree directed toward the root. Problem 5.7 shows how such a spanning tree can be obtained when there are cycles of zero length. Problem 5.4 explores the difference between a shortest path spanning tree and a minimum weight spanning tree.

Using the preceding construction, it can be shown that *if there are no zero (or negative)-length cycles, then Bellman's equation (5.6) (viewed as a system of N equations with N unknowns) has a unique solution.* This is an interesting fact which will be useful in Section 5.2.4 when we consider the Bellman–Ford algorithm starting from initial conditions other than ∞ [cf. Eq. (5.2)]. For a proof, we suppose that $\tilde{D}_i, i = 1, \ldots, N$, are another solution of Bellman's equation (5.6) with $\tilde{D}_1 = 0$, and we show that $\tilde{D}_i$ are equal to the shortest path lengths D_i. Let us repeat the path construction of the preceding paragraph with $\tilde{D}_i$ replacing D_i. Then $\tilde{D}_i$ is the length of the corresponding path from node i to node 1, showing that $\tilde{D}_i \geq D_i$. To show the reverse inequality, consider the Bellman–Ford algorithm with two different initial conditions. The first initial condition is $D_i^0 = \infty$, for $i \neq 1$, and $D_1^0 = 0$, in which case the true shortest path lengths D_i are obtained after at most $N - 1$ iterations, as shown earlier. The second initial condition is $D_i^0 = \tilde{D}_i$, for all i, in which case $\tilde{D}_i$ is obtained after every iteration (since the $\tilde{D}_i$ solve Bellman's equation). Since the second initial condition is, for every i, less than or equal to the first, it is seen from the Bellman–Ford iteration $D_i^{h+1} = \min_j[d_{ij} + D_j^h]$ that $\tilde{D}_i \leq D_i$, for all i. Therefore, $\tilde{D}_i = D_i$, and the only solution of Bellman's equation is the set of the true shortest path lengths D_i. It is also possible to show that if there are zero-length cycles not involving node 1, then Bellman's equation has a nonunique solution [although the Bellman-Ford algorithm still terminates with the correct shortest path lengths from the initial conditions $D_i^0 = \infty$ for all $i \neq 1$; see Problem 5.8].

It turns out that the Bellman–Ford algorithm works correctly even if the initial conditions D_i^0 for $i \neq 1$ are arbitrary numbers, and the iterations are done in parallel for different nodes in virtually any order. This fact will be shown in the next subsection and accounts in part for the popularity of the Bellman–Ford algorithm for distributed applications.

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 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: Initially, $P = \{1\}$, $D_1 = 0$, and $D_j = d_{j1}$ for $j \neq 1$.

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.

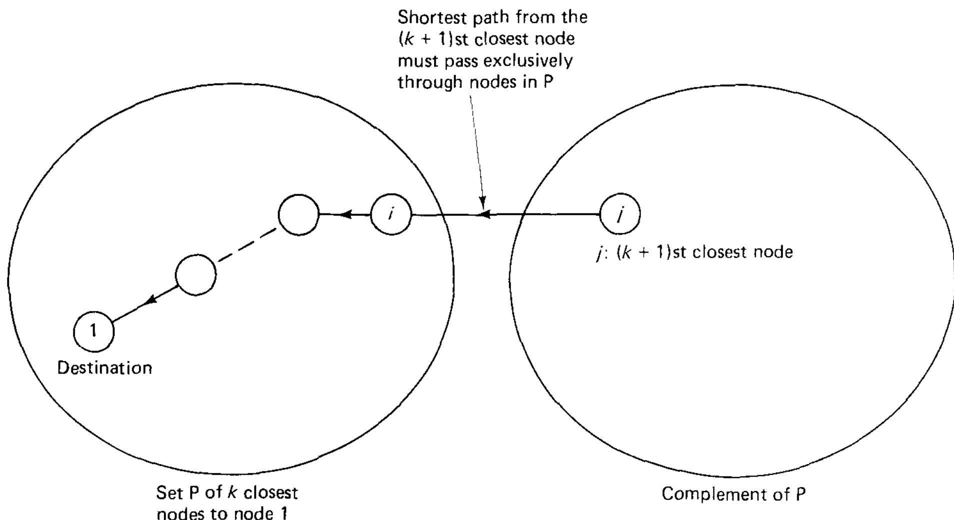


Figure 5.34 Basic idea of Dijkstra's algorithm. At the kth step we have the set P of the k closest nodes to node 1 as well as the shortest distance D_i from each node i in P to node 1. Of all paths connecting some node not in P with node 1, there is a shortest one that passes exclusively through nodes in P (since $d_{ij} \geq 0$). Therefore, the $(k+1)$st closest node and the corresponding shortest distance are obtained by minimizing over $j \notin P$ the quantity $\min_{i \in P}\{d_{ji} + D_i\}$. 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

$$D_j := \min\,[D_j, d_{ji} + D_i]$$

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 $d_{ji} \geq 0$ and $D_i = \min_{j \notin P} D_j$, it is preserved by the formula $D_j := \min[D_j, d_{ji} + D_i]$ 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_k 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

$$D'_j = \min_{k \in P \cup \{i\}} \left[d_{jk} + D_k\right] = \min\left[\min_{k \in P}\left[d_{jk} + D_k\right], d_{ji} + D_i\right]$$

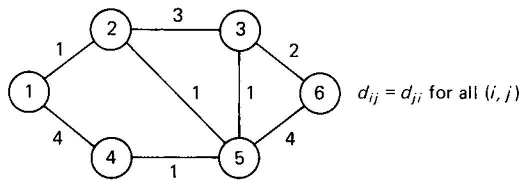
Similarly, the induction hypothesis implies that $D_j = \min_{k \in P}[d_{jk} + D_k]$, so we obtain $D'_j = \min[D_j, d_{ji} + D_i]$. Thus in step 2, D_j is set to the shortest distance D'_j from j to 1 using paths with all nodes except j belonging to $P \cup \{i\}$. The induction proof of condition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

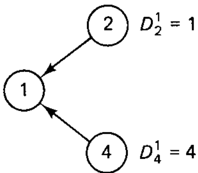
To estimate the computation required by Dijkstra's algorithm, we note that there are $N - 1$ iterations and the number of operations per iteration is proportional to N. Therefore, in the worst case the computation is $O(N^2)$, comparing favorably with the worst-case estimate $O(N^3)$ of the Bellman–Ford 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).

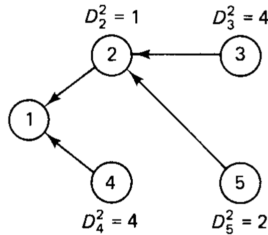
The Dijkstra and Bellman–Ford algorithms are contrasted on an example problem in Fig. 5.35.

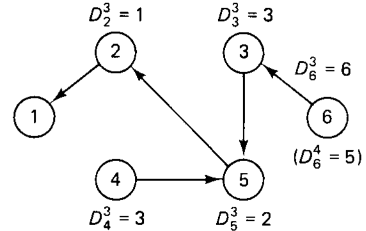
The Floyd–Warshall algorithm. This algorithm, unlike the previous two, finds the shortest paths between all pairs of nodes together. Like the Bellman–Ford algorithm, the arc distances can be positive or negative, but again there can be no negative-length cycles. All three algorithms iterate to find the 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



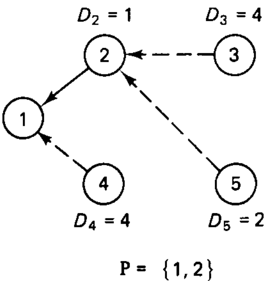
Bellman-Ford

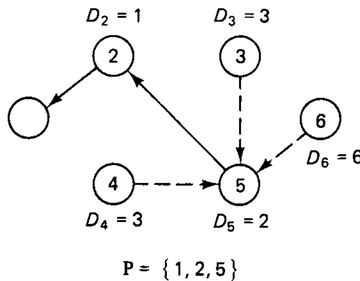






Dijkstra





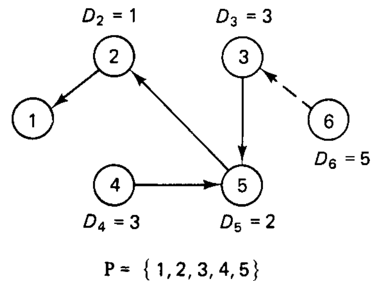


Figure 5.35 Example using the Bellman–Ford and Dijkstra algorithms.

(*i.e.*, no intermediate nodes) as starting estimates of shortest path lengths. It then calculates shortest paths under the constraint that only node 1 can be used as an intermediate node, and then with the constraint that only nodes 1 and 2 can be used, and so forth.

To state the algorithm more precisely, let D_{ij}^n be the shortest path length from node i to j with the constraint that only nodes $1, 2, \ldots, n$ can be used as intermediate nodes on paths. The algorithm then is as follows: Initially,

$$D_{ij}^0 = d_{ij}, \qquad \text{for all } i,\ j, \qquad i \neq j$$

For $n = 0, 1, \ldots, N-1$,

$$D_{ij}^{n+1} = \min\left[D_{ij}^n, D_{i(n+1)}^n + D_{(n+1)j}^n\right], \qquad \text{for all } i \neq j \tag{5.7}$$

To see why this works, we use induction. For $n = 0$, the initialization clearly gives the shortest path lengths subject to the constraint of no intermediate nodes on paths. Now, suppose that for a given n, D_{ij}^n in the algorithm above gives the shortest path lengths using nodes 1 to n as intermediate nodes. Then the shortest path length from i to j, allowing nodes 1 to $n+1$ as intermediate nodes, either contains node $n+1$ on the shortest path, or does not contain node $n+1$. For the first case, the constrained shortest path from i to j goes first from i to $n+1$ and then from $n+1$ to j, giving the length in the final term of Eq. (5.7). For the second case, the constrained shortest path is the same as the one using nodes 1 to n as intermediate nodes, yielding the length of the first term in the minimization of Eq. (5.7).

Since each of the N steps above must be executed for each pair of nodes, the computation involved in the Floyd–Warshall algorithm is $O(N^3)$, the same as if the Dijkstra algorithm were repeated for each possible choice of source node.

5.2.4 Distributed Asynchronous Bellman–Ford Algorithm

Consider a routing algorithm that routes each packet along a shortest path from the packet's origin to its destination, and suppose that the link lengths may change either due to link failures and repairs, or due to changing traffic conditions in the network. It is therefore necessary to update shortest paths in response to these changes. We described several such algorithms in Section 5.1.2 in connection with the ARPANET and the TYMNET.

In this subsection we consider an algorithm similar to the one originally implemented in the ARPANET in 1969. It is closely related to the one used in DNA (DEC's Digital Network Architecture) [Wec80]. The idea is to compute the shortest distances from every node to every destination by means of a distributed version of the Bellman–Ford algorithm. An interesting aspect of this algorithm is that it requires very little information to be stored at the network nodes. Indeed, a node need not know the detailed network topology. It suffices for a node to know the length of its outgoing links and the identity of every destination.

We assume that each cycle has positive length. We also assume that the network always stays strongly connected, and that if (i, j) is a link, then (j, i) is also a link. We envision a practical situation where the lengths d_{ij} can change with time. In the analysis, however, it is assumed that the lengths d_{ij} are fixed while the initial conditions for the

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,

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

$$D_1 = 0$$

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

$$D_i^{h+1} = \min_{j \in N(i)} \left[d_{ij} + D_j^h\right], \qquad i \neq 1 \tag{5.8}$$

$$D_1^{h+1} = 0 \tag{5.9}$$

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

$$D_i^0 = \infty, \qquad i \neq 1 \tag{5.10}$$

$$D_1^0 = 0 \tag{5.11}$$

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 incremented 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

$$D_i := \min_{j \in N(i)} [d_{ij} + D_j] \tag{5.12}$$

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 major 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:

$D_j^i(t) =$ Estimate of the shortest distance of each neighbor node $j \in N(i)$ which was last communicated to node i

$D_i(t) =$ Estimate of the shortest distance of node i which was last computed at node i according to the Bellman–Ford iteration

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

$$D_1(t) = 0, \qquad \text{for all } t \geq t_0$$

$$D_1^i(t) = 0, \qquad \text{for all } t \geq t_0, \text{ and } i \text{ with } 1 \in N(i)$$

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

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

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

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_j^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^i 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^i 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 $\bar{t} \geq t_0$, there exists a time $\tilde{t} > \bar{t}$ such that estimates D_j computed at a node j prior to time $\bar{t}$ are not received at any neighbor node i after time $\tilde{t}$].

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 [BeT89].)

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

$$D_i(t) = D_i, \qquad \text{for all } t \geq t_m, \; i = 1, \ldots, N$$

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 $\{\underline{D}_i^k\}$ and $\{\overline{D}_i^k\}$ with

$$\underline{D}_i^k \leq \underline{D}_i^{k+1} \leq D_i \leq \overline{D}_i^{k+1} \leq \overline{D}_i^k \tag{5.13}$$

and

$$\underline{D}_i^k = D_i = \overline{D}_i^k, \qquad \text{for all } k \text{ sufficiently large} \tag{5.14}$$

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

$$\underline{D}_i^k \leq D_i(t) \leq \overline{D}_i^k, \qquad \text{for all } t \text{ sufficiently large} \tag{5.15}$$

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 $\overline{D}_j$ and $\tilde{D}_j$,

$$\overline{D}_j \geq \tilde{D}_j, \qquad \text{for all } j \in N(i)$$

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

$$\min_{j \in N(i)} [d_{ij} + \overline{D}_j] \geq \min_{j \in N(i)} [d_{ij} + \tilde{D}_j]$$

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 $D_i^1 \geq D_i^0$ for each i, then $D_i^{k+1} \geq D_i^k$, for each i and k. Similarly, if $D_i^1 \leq D_i^0$, for each i, then $D_i^{k+1} \leq D_i^k$, for each i and k.

Consider the Bellman–Ford algorithm given by Eqs. (5.8) and (5.9). Let $\overline{D}_i^k, i = 1, \ldots, N$ be the kth iterate of this algorithm when the initial condition is

$$\overline{D}_i^0 = \infty, \qquad i \neq 1 \tag{5.16a}$$

$$\overline{D}_1^0 = 0 \tag{5.16b}$$

and let $\underline{D}_i^k, i = 1, \ldots, N$ be the kth iterate when the initial condition is

$$\underline{D}_i^0 = D_i - \delta, \qquad i \neq 1 \tag{5.17a}$$

$$\underline{D}_1^0 = 0 \tag{5.17b}$$

where δ is a positive scalar large enough so that $\underline{D}_i^0$ is smaller than all initial node estimates $D_i(t_0)$, $D_j^i(t_0)$, $j \in N(i)$ 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 $\{\underline{D}_i^k\}$ and $\{\overline{D}_i^k\}$ 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,

$$\overline{D}_i^k = D_i, \qquad k \geq N - 1 \tag{5.18}$$

so only $\underline{D}_i^k = D_i$, for sufficiently large k, remains to be established. To this end, we first note that by an induction argument it follows that $\underline{D}_i^k$ is, for every k, the length of a walk from i to some other node j involving no more than k links, plus $\underline{D}_j^0$. 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 $D_i \geq \underline{D}_i^k = L(i,k) + \underline{D}_j^0$ 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 $\underline{D}_i^k$, for $i = 1, \ldots, N$ and $k = 0, 1, \ldots$, is finite. Since $\underline{D}_i^k$ is monotonically nondecreasing in k for all i, it follows that for some h

$$\underline{D}_i^{h+1} = \underline{D}_i^h, \qquad \text{for all } i \tag{5.19}$$

Therefore, the scalars $\underline{D}_i^h$ satisfy Bellman's equation, which as shown in Section 5.2.3,

has as its unique solution the shortest distances D_i. It follows that

$$\underline{D}_i^k = D_i, \qquad \text{for all } i, \quad k \geq h \tag{5.20}$$

Equations (5.18) and (5.20) show Eq. (5.14). **Q.E.D.**

We now complete the proof of the proposition by showing by induction that for every k there exists a time $t(k)$ such that for all $t \geq t(k)$,

$$\underline{D}_i^k \leq D_i(t) \leq \overline{D}_i^k, \qquad i = 1, 2, \ldots, N \tag{5.21}$$

$$\underline{D}_j^k \leq D_j^i(t) \leq \overline{D}_j^k, \qquad j \in N(i), \quad i = 1, 2, \ldots, N \tag{5.22}$$

and for all $t \in T_j^i$, $t \geq t(k)$,

$$\underline{D}_j^k \leq D_j\left[\tau_j^i(t)\right] \leq \overline{D}_j^k, \qquad j \in N(i), \quad i = 1, 2, \ldots, N \tag{5.23}$$

where $\tau_j^i(t)$ is the largest time at which the estimate $D_j^i(t)$ available at node i at time t was computed using the iteration $D_j := \min_{k \in N(j)}[d_{jk} + D_k]$ at node j. [Formally, $\tau_j^i(t)$ is defined as the largest time in T^j that is less than t, and is such that $D_j[\tau_j^i(t)] = D_j^i(t)$.]

Indeed, the induction hypothesis is true for $k = 0$ [for $t(0) = t_0$] because δ was chosen large enough in the initial conditions (5.17). Assuming that the induction hypothesis is true for a given k, it will be shown that there exists a time $t(k+1)$ with the required properties. Indeed, from relation (5.22) and the monotonicity of the Bellman–Ford iteration, we have that for every $t \in T^i, t \geq t(k)$ [*i.e.*, a time t for which $D_i(t)$ is updated via the Bellman–Ford iteration],

$$\underline{D}_i^{k+1} \leq D_i(t) \leq \overline{D}_i^{k+1}$$

Therefore, if $t'(k)$ is the smallest time $t \in T^i$, with $t \geq t(k)$, then

$$\underline{D}_i^{k+1} \leq D_i(t) \leq \overline{D}_i^{k+1}, \qquad \text{for all } t \geq t'(k), \quad i = 1, 2, \ldots, N \tag{5.24}$$

Since $\tau_j^i(t) \to \infty$ as $t \to \infty$ (by Assumptions 1 and 2) we can choose a time $t(k+1) \geq t'(k)$ so that $\tau_j^i(t) \geq t'(k)$, for all i, $j \in N(i)$, and $t \geq t(k+1)$. Then, in view of Eq. (5.24), for all $t \geq t(k+1)$,

$$\underline{D}_j^{k+1} \leq D_j^i(t) \leq \overline{D}_j^{k+1}, \qquad j \in N(i), \quad i = 1, 2, \ldots, N \tag{5.25}$$

and for all $t \in T_j^i$, $t \geq t(k+1)$,

$$\underline{D}_j^{k+1} \leq D_j\left[\tau_j^i(t)\right] \leq \overline{D}_j^{k+1}, \qquad j \in N(i), \quad i = 1, 2, \ldots, N \tag{5.26}$$

This completes the induction proof. **Q.E.D.**

The preceding analysis assumes that there is a path from every node to node 1. If not, the analysis applies to the portion of the network that is connected with node 1. For a node i that belongs to a different network component than node 1 and has at least one neighbor (so that it can execute the algorithm), it can be seen that $D_i(t) \to \infty$ as $t \to \infty$. This property can be used by a node to identify the destinations to which it is not connected.

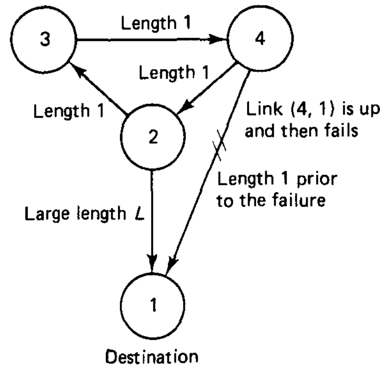


Figure 5.36 Example where the number of iterations of the (synchronous) Bellman–Ford algorithm is excessive. Suppose that the initial conditions are the shortest distances from nodes 2, 3, and 4 to node 1 before link (4,1) fails ($D_2 = 3$, $D_3 = 2$, $D_4 = 1$). Then, after link (4,1) fails, nearly L iterations will be required before node 2 realizes that its shortest path to node 1 is the direct link (2,1). This is an example of the so-called "bad news phenonenon," whereby the algorithm reacts slowly to a sudden increase in one or more link lengths.

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

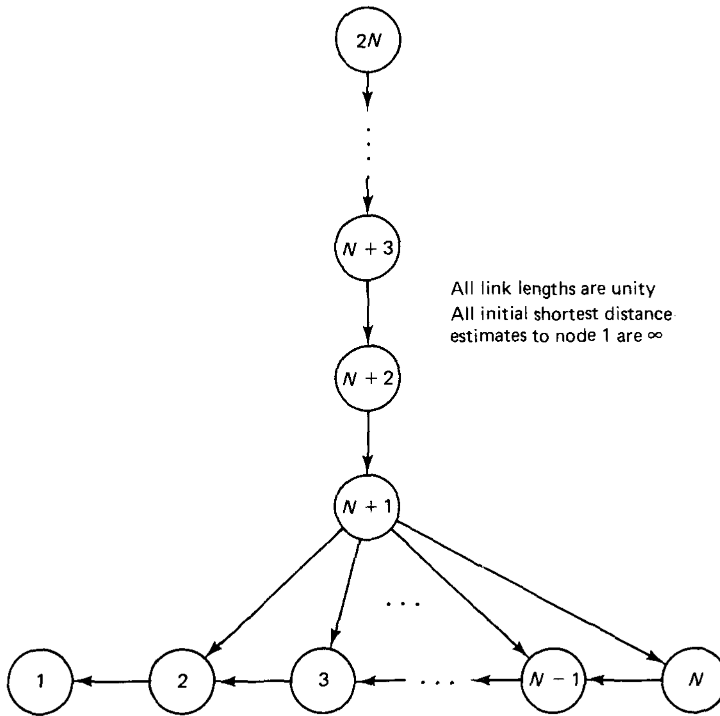


Figure 5.37 Example where the asynchronous version of the Bellman–Ford algorithm requires many more message transmissions than the synchronous version. The initial estimates of shortest distance of all nodes is ∞. Consider the following sequence of events, where all messages are received with zero delay:

1. Node 2 updates its shortest distance and communicates the result to 3. Node 3 updates and communicates the result to 4.... Node $N-1$ updates and communicates the result to N. Node N updates and communicates the result to $N+1$. (These are $N-1$ messages.)
2. Node $N+1$ updates and communicates the result to $N+2$.... Node $2N-1$ updates and communicates the result to $2N$. Node $2N$ updates. (These are $N-1$ messages.)
3. For $i = N-1, N-2, \ldots, 2$, in that order, node i communicates its update to $N+1$ and sequence 2 above is repeated. [These are $N(N-2)$ messages.]

The total number of messages is $N^2 - 2$. If the algorithm were executed synchronously, with a message sent by a node to its neighbors only when its shortest distance estimate changes, the number of messages needed would be $3(N-1)-1$. The difficulty in the asynchronous version is that a lot of unimportant information arrives at node $N+1$ early and triggers a lot of unnecessary messages starting from $N+1$ and proceeding all the way to $2N$.

let the traffic input of node 8 be $\epsilon > 0$, where ϵ 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

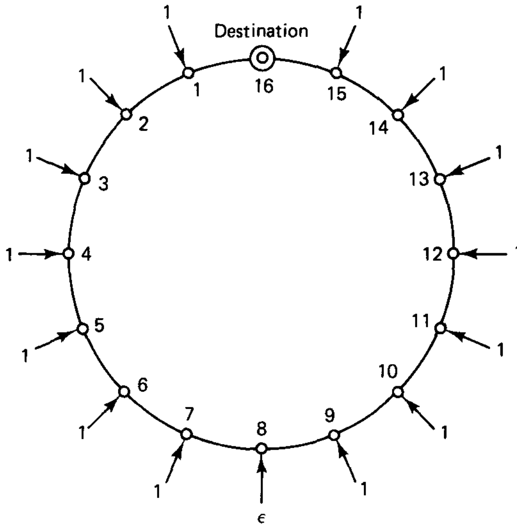
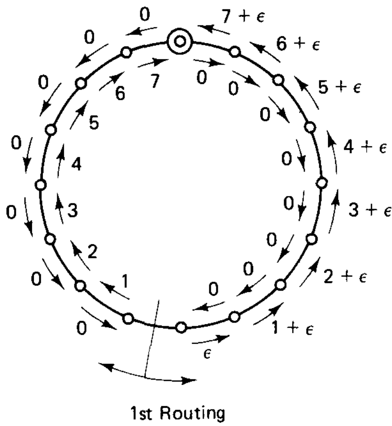


Figure 5.38 Sixteen-node ring network of Example 5.4. Node 16 is the only destination.

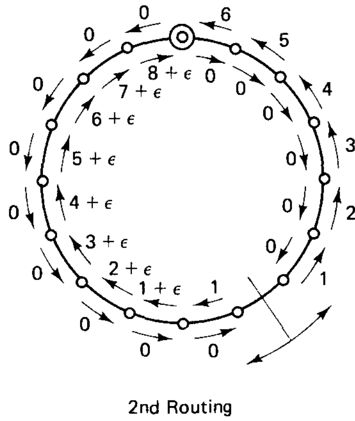
for the next T seconds. Assume that we start with nodes 1 through 7 routing clockwise and nodes 8 through 15 routing counterclockwise. This is a rather good routing, balancing the traffic input between the two directions. Figure 5.39 shows the link rates corresponding to the initial and subsequent shortest 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 generically 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 α (link length at zero load) is known as a *bias* factor.

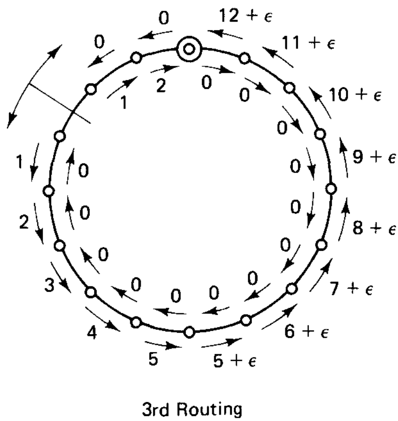
Figure 5.39 Oscillations in a ring network for link lengths d_{ij} equal to the link arrival rates F_{ij}. Each node sends one unit of input traffic to the destination except for the middle node 8, which sends $\epsilon > 0$, where ϵ is very small. The numbers next to the links are the link rates in 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 $(= 0 + 1 + 2 + \cdots + 7)$, and the length of the counterclockwise path as $28 + 8\epsilon$ $(= \epsilon + 1 + \epsilon + 2 + \epsilon + \cdots + 7 + \epsilon)$, and switches its traffic to the shortest (clockwise) path at the second routing. The corresponding numbers for node 9 are 28 and $28 + 7\epsilon$, 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.



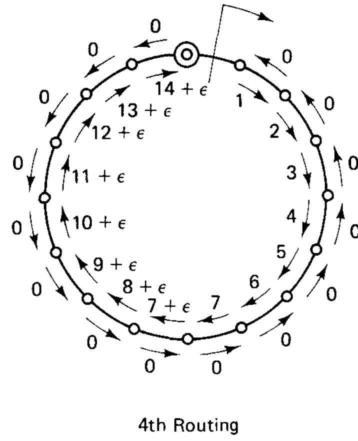
1st Routing



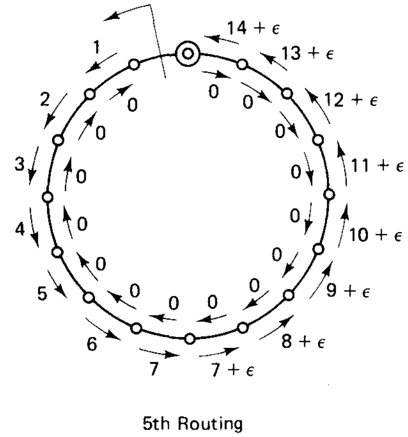
2nd Routing



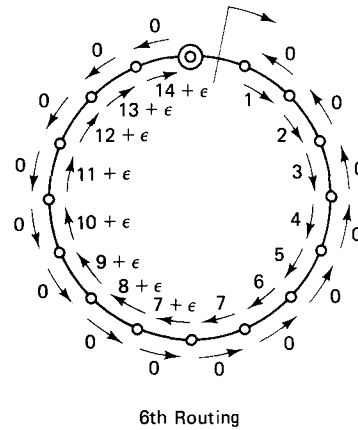
3rd Routing



4th Routing



5th Routing



6th Routing

Oscillations in the original ARPANET algorithm discussed in Section 5.1.2 were damped by using a substantial bias factor. Indeed, if α 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 α 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 ([KhZ89] and [ZVK89]), 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], [Ber79b], 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 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 paths almost instantaneously.

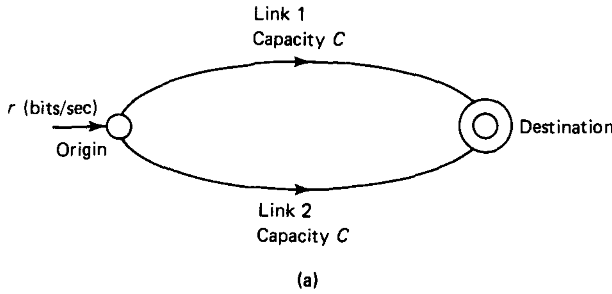
The situation is quite different in a virtual circuit network, where every session is assigned a fixed communication path at the time it is first established. There the average duration of a virtual circuit is often large relative to the shortest path updating period. As a result, the network reaction 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 Fig. 5.40(a). 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.



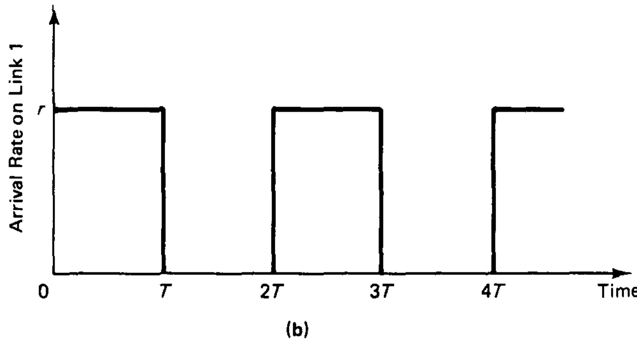


Figure 5.40 (a) Two-link network of Example 5.5. (b) Arrival rate on link 1 in Example 5.5 using the shortest path rule in the datagram case. Essentially, only one path is used for routing at any one time if the shortest path update period is much larger than the time required to empty the queue of waiting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

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 λ 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 $M/M/\infty$ queueing results (cf. Section 3.4.2), the number of active virtual circuits is Poisson distributed with mean λ/μ. If γ is the average communication rate (bits/sec) of a virtual circuit, we must have $r = (\lambda/\mu)\gamma$, or

$$\gamma = \frac{r\mu}{\lambda} \tag{5.27}$$

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 μT 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 λT 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 $\gamma\lambda T$ bits/sec or, using the fact that $\gamma = r\mu/\lambda$ [cf. Eq. (5.27)], $r\mu T$ bits/sec. Therefore, the average rates x_1^k and x_2^k (bits/sec) on the two links at the kth interval will evolve approximately according to

$$x_i^{k+1} = \begin{cases} (1-\mu T)x_i^k + r\mu T, & \text{if } i \text{ is shortest } (i.e., x_i^k = \min\{x_1^k, x_2^k\}) \\ (1-\mu T)x_i^k, & \text{otherwise} \end{cases} \tag{5.28}$$

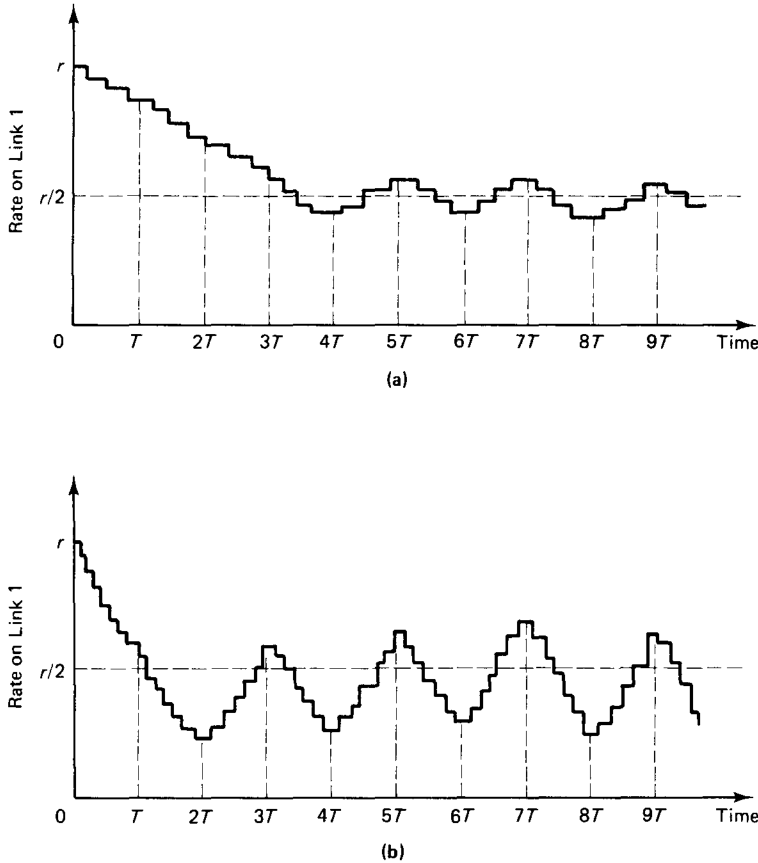


Figure 5.41 Arrival rate on link 1 in Example 5.5 when virtual circuits are used. (a) Virtual circuits last a long time, and the shortest paths are updated frequently (μT is small). (b) Virtual circuits last a short time, and the shortest paths are updated frequently (μT is moderate).

Figures 5.41 and 5.42 provide examples of the evolution of the average rate on link 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 μT is large, the analysis above indicates considerable oscillation of the link rates. Figures 5.41 and 5.42 demonstrate the relation between μ, 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

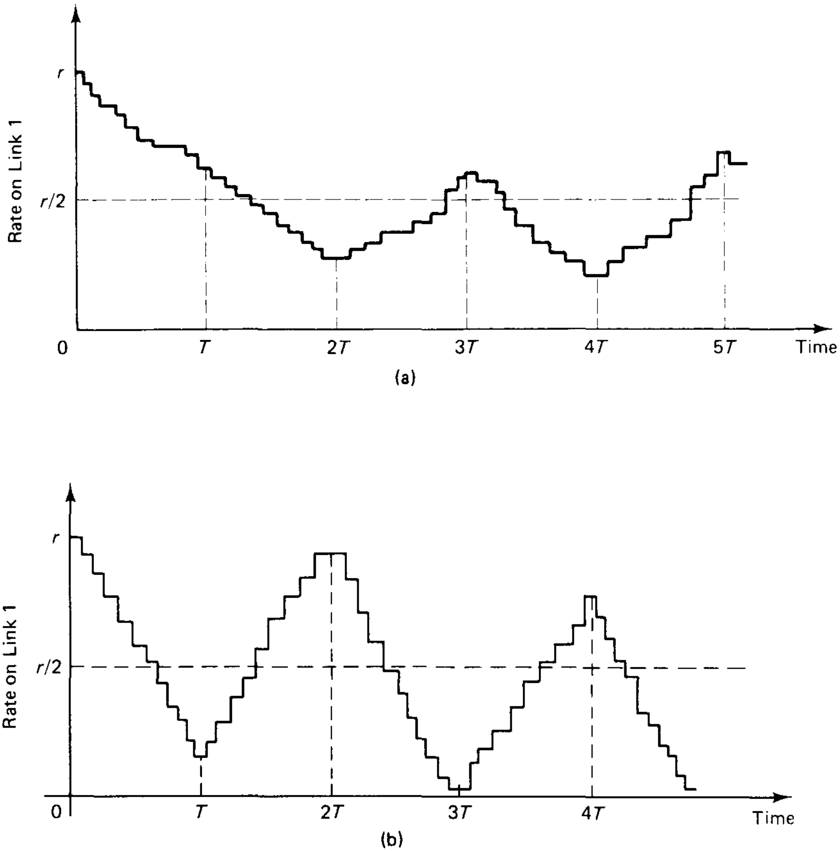


Figure 5.42 Arrival rate on link 1 in Example 5.5 when virtual circuits are used. (a) Virtual circuits last a long time, and the shortest paths are updated infrequently (μT is moderate). (b) Virtual circuits last a short time, and shortest paths are updated infrequently (μT is large). In the limit as $\mu \to \infty$, the datagram case is obtained.

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.

5.3 BROADCASTING ROUTING INFORMATION: COPING WITH LINK FAILURES

A problem that often arises in routing is the transfer of control information from points in the network where it is collected to other points where it is needed. This problem is surprisingly challenging when links are subject to failure. In this section we explain the nature of the difficulties and we consider ways to address them. We also take the opportunity to look at several instructive examples of distributed algorithms.

One practical example of broadcasting routing information was discussed in Section 5.1.2 in connection with the current ARPANET algorithm. Here all link lengths are periodically broadcast to all nodes, which then proceed to update their routing tables through a shortest path computation. Other cases include situations where it is desired to alert higher layers about topological changes, or to update data structures that might be affected by changes in the network topology, such as a spanning tree used for broadcasting messages to all nodes. [The term *network topology* will be used extensively in this section. It is synonymous with the list of nodes and links of the network together with the status (up or down) of each link.]

Getting routing information reliably to the places where it is needed in the presence of potential link failures involves subtleties that are generally not fully appreciated. Here are some of the difficulties:

1. Topological update information must be communicated over links that are themselves subject to failure. Indeed, in some networks, one must provide for the event where portions of the network get disconnected from each other. As an example consider a *centralized* algorithm where all information regarding topological changes is communicated to a special node called the network control center (NCC). The NCC maintains in its memory the routing tables of the entire system, and updates them upon receipt of new topological information by using some algorithm that is of no concern here. It then proceeds to communicate to the nodes affected all information needed for local routing operations. Aside from the generic issue of collecting and disseminating information over failure-prone links, a special problem here is that the NCC may fail or may become disconnected from a portion of the network. In some cases it is possible to provide redundancy and ensure that such difficulties will occur very rarely. In other cases, however, resolution of these difficulties may be neither simple nor foolproof, so that it may be advisable to abandon the centralized approach altogether and adopt a distributed algorithm.
2. One has to deal with multiple topological changes (such as a link going down and up again within a short time), so there is a problem of distinguishing between old and new update information. As an example, consider a flooding method for broadcasting topological change information (cf. Section 5.1.2). Here a node monitors the status of all its outgoing links, and upon detecting a change, sends a packet to all its neighbors reporting that change. The neighbors send this packet to their neighbors, and so on. Some measures are needed to prevent update packets from circulating indefinitely in the network, but this question is discussed

later. Flooding works when there is a single topological change, but can fail in its pure form when there are multiple changes, as shown in the example of Fig. 5.43. Methods for overcoming this difficulty are discussed later in this section, where it is shown that apparently simple solutions sometimes involve subtle failure modes.

3. Topological update information will be processed by some algorithm that computes new routing paths (*e.g.*, a shortest path algorithm). It is possible, however, that new topological update information arrives as this algorithm is running. Then, either the algorithm must be capable of coping with changes in the problem data as it executes, or else it must be aborted and a new version must be started upon receipt of new data. Achieving either of these may be nontrivial, particularly when the algorithm is distributed.
4. The repair of a single link can cause two parts of the network which were disconnected to reconnect. Each part may have out-of-date topology information about the other part. The algorithm must ensure that eventually the two parts agree and adopt the correct network topology.

The difficulties discussed above arise also in the context of broadcasting information related to the congestion status of each link. An important difference, however, is that incorrect congestion information typically has less serious consequences than does incorrect topological information; the former can result at worst in the choice of inferior routes, while the latter can result in the choice of nonexistent routes. Thus, one can afford to allow for a small margin of error in an algorithm that broadcasts routing information other than topological updates, if that leads to substantial algorithmic simplification or reduction in communication overhead. Note, however, that in some schemes, such as the ARPANET flooding algorithm to be discussed shortly, topological update information is embedded within congestion information, and the same algorithm is used to broadcast both.

As we approach the subject of broadcasting topological update information, we must first recognize that it is impossible for every node to know the correct network topology at all times. Therefore, the best that we can expect from an algorithm is that it

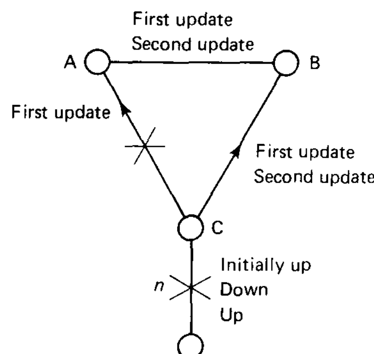


Figure 5.43 Example where the flooding algorithm in its pure form fails. Link n is initially up, then it goes down, then up again. Suppose that the two updates travel on the path CBA faster than the first update travels on the link CA. Suppose also that link CA fails after the first update, but before the second update travels on it. Then, the last message received by A asserts that link n is down, while the link is actually up. The difficulty here is that outdated information is mistakenly taken to be new.

can cope successfully with any finite number of topological changes within finite time. By this we mean that if a finite number of changes occur up to some time and no other changes occur subsequently, all nodes within each connected portion of the network should know the correct status of each link in that portion within finite time.

In the discussion of validity of various schemes in the sense described above, we will assume the following:

1. Network links preserve the order and correctness of transmissions. Furthermore, nodes maintain the integrity of messages that are stored in their memory.
2. A link failure is detected by both end nodes of the link, although not necessarily simultaneously. By this we mean that any link declared down by one end node will also eventually be declared down by the other, before the first end node declares it up again.
3. There is a data link protocol for recognizing, at the ends of a failed link, when the link is again operational. If one of the end nodes declares the link to be up, then within finite time, either the opposite end node also declares the link up or the first declares it down again.
4. A node can crash, in which case each of its incident links is, within finite time, declared down by the node at the opposite end of the link.

The preceding assumptions provide a basis for discussion and analysis of specific schemes, but are not always satisfied in practice. For example, on rare occasions, Assumption 1 is violated because damaged data frames may pass the error detection test of Data Link Control, or because a packet may be altered inside a node's memory due to hardware malfunction. Thus, in addition to analysis of the normal case where these assumptions are satisfied, one must weigh the consequences of the exceptional case where they are not. Keep in mind here that topology updating is a low-level algorithm on which other algorithms rely for correct operation.

Note that it is not assumed that the network will always remain connected. In fact, the topology update algorithm together with some protocol for bringing up links should be capable of starting up the network from a state where all links are down.

5.3.1 Flooding—The ARPANET Algorithm

Flooding was described previously as an algorithm whereby a node broadcasts a topological update message to all nodes by sending the message to its neighbors, which in turn send the message to their neighbors, and so on. Actually, the update messages may include other routing-related information in addition to link status. For example, in the ARPANET, each message originating at a node includes a time average of packet delay on each outgoing link from the node. The time average is taken over 10-sec intervals. The time interval between update broadcasts by the node varies in the ARPANET from 10 to 60 sec, depending on whether there has been a substantial change in any single-link average delay or not. In addition, a message is broadcast once a status change in one of the node's outgoing links is detected.

A serious difficulty with certain forms of flooding is that they may require a large number of message transmissions. The example of Fig. 5.44 demonstrates the difficulty and shows that it is necessary to store enough information in update messages and network nodes to ensure that each message is transmitted by each node only a finite number of times (and preferably only once). In the ARPANET, update messages from each node are marked by a *sequence number*. When a node j receives a message that originated at some node i, it checks to see if its sequence number is greater than the sequence number of the message last received from i. If so, the message together with its sequence number is stored in memory, and its contents are transmitted to all neighbors of j except the one from which the message was received. Otherwise, the message is discarded. Assuming that the sequence number field is large enough, one can be sure that wraparound of the sequence numbers will never occur under normal circumstances. (With a 48-bit field and one update per millisecond, it would take more than 5000 years for wraparound to occur.)

The use of sequence numbers guarantees that each topological update message will be transmitted at most once by each node to its neighbors. Also, it resolves the difficulty of distinguishing between new and old information, which was discussed earlier and illustrated in the example of Fig. 5.43. Nonetheless, there are subtle difficulties with sequence numbers relating to exceptional situations, such as when portions of the network become disconnected or when equipment malfunctions. For example, when a complete or partial network reinitialization takes place (perhaps following a crash of one or more nodes), it may be necessary to reset some sequence numbers to zero, since a node may not remember the sequence number it was using in the past. Consider a time period when two portions of the network are disconnected. During this period, each portion cannot learn about topological changes occurring in the other portion. If sequence numbers are

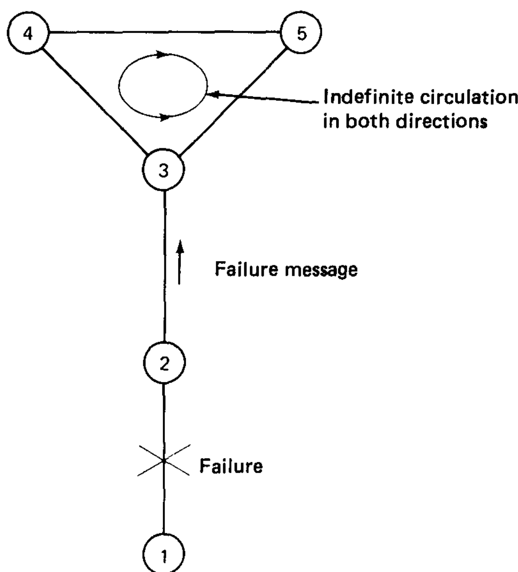


Figure 5.44 Example of a form of flooding where the transmission of messages never terminates. The rule is that a node that receives a message relays it to all of its neighbor nodes except the one from which the message was received. A failure of link (1, 2) is communicated to node 3, which triggers an indefinite circulation of the failure message along the loop (3,4,5) in both directions. This form of flooding works only if the network has no cycles.

reset in one of the two portions while they are disconnected, there may be no way for the two portions to figure out the correct topology after they connect again by relying on the sequence number scheme alone. Another concern is that a sequence number could be altered either inside a node's memory due to a malfunction, or due to an undetected error in transmission. Suppose, for example, that an update message originating at node i reaches another node j, has its sequence number changed accidentally to a high number, and is flooded throughout the network. Then the erroneous sequence number will take over, and in the absence of a correcting mechanism, node i will not be listened to until its own (correct) sequence number catches up with the wrong one. Similarly, the sequence number of a node that is accidentally set to a high number inside the node itself can wrap around, at which time all subsequent update messages from the node will be ignored by the network.

The ARPANET resolves the problems associated with sequence numbers by using two devices:

1. Each update message includes an *age field*, which indicates the amount of time that the message has been circulating inside the network. Each node visited by a message keeps track of the message's arrival time and increments its age field (taking into account the transmission and propagation time) before transmitting it to its neighbors. (The age field is incremented slightly differently in the ARPANET; see [Per83].) Thus, a node can calculate the age of all messages in its memory at any time. A message whose age exceeds a prespecified limit is not transmitted further. Otherwise, the message is transmitted to all neighbor nodes except the one from which the message was received.
2. Each node is required to transmit update messages *periodically* in addition to the messages transmitted when a link status change is detected. (There is at least one of these from every node every 60 sec.)

The rule used regarding the age field is that an "aged" message is superseded by a message that has not "aged" yet, regardless of sequence numbers. (A message that has not "aged" yet is superseded only if the new message has a higher sequence number.) This rule guarantees that damaged or incorrect information with a high sequence number will not be relied upon for too long. The use of periodic update messages guarantees that up-to-date information will become available within some fixed time following reconnection of two portions of the network. Of course, the use of periodic updates implies a substantial communication overhead penalty and is a major drawback of the ARPANET scheme. The effect of this, however, is lessened by the fact that the update packets include additional routing-related information, namely the average packet delay on the node's outgoing links since the preceding update.

5.3.2 Flooding without Periodic Updates

It is possible to operate the flooding scheme with sequence numbers correctly without using an age field or periodic updates. The following simple scheme (suggested by P. Humblet) improves on the reliability of the basic sequence number idea by providing

a mechanism for coping with node crashes and some transmission errors. It requires, however, that the sequence number field be so large that wraparound never occurs. The idea is to modify the flooding rules so that the difficulties with reconnecting disconnected network components are dealt with adequately. We describe briefly the necessary modifications, leaving some of the details for the reader to work out. We concentrate on the case of broadcasting topological information, but a similar scheme can be constructed to broadcast other routing information.

As before, we assume that when the status of an outgoing link from a node changes, that node broadcasts an update message containing the status of all its outgoing links to all its current neighbors. The message is stamped by a sequence number which is either zero or is larger by one than the last sequence number used by the node in the past. There is a restriction here, namely that a zero sequence number is allowed only when the node is recovering from a crash (defined as a situation where all of the node's incident links are down and the node is in the process of bringing one or more of these links up). As before, there is a separate sequence number associated with each origin node.

The first flooding rule modification has to do with bringing up links that have been down. When this happens, the end nodes of the link, in addition to broadcasting a regular update message on all of their outgoing links, should exchange their current views of the network topology. By this we mean that they should send to each other all the update messages stored in their memory that originated at other nodes together with the corresponding sequence numbers. These messages are then propagated through the network using the modified flooding rules described below. This modification guarantees that, upon reconnection, the latest update information will reach nodes in disconnected network portions. It also copes with a situation where, after a crash, a node does not remember the sequence number it used in the past. Such a node must use a zero sequence number following the crash, and then, through the topology exchange with its neighbors that takes place when its incident links are brought up, become aware of the highest sequence number present in the other nodes' memories. The node can then increment that highest sequence number and flood the network with a new update message.

For the scheme to work correctly we modify the flooding rule regarding the circumstances under which an update message is discarded. To this end we order, for each node i, the topological update messages originated at i. For two such messages A and B, we say that $A > B$ if A has a greater sequence number than B, or if A and B have identical sequence numbers and the content of A is greater than the content of B according to some lexicographic rule (*e.g.*, if the content of A interpreted as a binary number is greater than the similarly interpreted content of B). Any two messages A and B originating at the same node can thus be compared in the sense that $A > B$, or $B > A$, or $A = B$, the last case occurring only if the sequence numbers and contents of A and B are identical.

Suppose that node j receives an update message A that has originated at node i, and let B be the message originated at node i and currently stored in node j's memory. The message is discarded if $A < B$ or $A = B$. If $A > B$, the flooding rule is now as follows:

1. If $j \neq i$, node j stores A in its memory in place of B and sends a copy of A on all its incident links except the one on which A was received.

2. If $j = i$ (*i.e.*, node i receives a message it issued some time in the past), node i sends to all its neighbors a new update message carrying the current status of all its outgoing links together with a sequence number that is 1 plus the sequence number of A.

Figure 5.45 shows by example why it is necessary to compare the contents of A and B in the case of equal sequence numbers.

The algorithm of this section suffers from a difficulty that seems generic to all event-driven algorithms that do not employ periodic updates: it is vulnerable to memory and transmission errors. Suppose, for example, that a message's sequence number is altered to a very high number either during a transmission or while residing inside a node's memory. Then the message's originating node may not be listened to until its sequence number catches up with the high numbers that are unintentionally stored in some nodes' memories. It turns out that this particular difficulty can be corrected by modifying the flooding rule discussed earlier. Specifically, if node j receives A and has B with $A < B$ in its memory, then message A is discarded as in the earlier rule, but in addition message B is sent back to the neighbor from which A was received. (The neighbor will then propagate B further.) Suppose now that a node issues an update message carrying a sequence number which is less than the sequence number stored in some other node's memory. Then, because of the modification just described, if the two nodes are connected by a path of up links, the higher sequence number (call it k) will be propagated back to the originator node, which can then issue a new update message with sequence number $k + 1$ according to rule 2 above. Unfortunately, there is still a problem associated with a possible sequence number wraparound (*e.g.*, when k above is the highest number possible within the field of possible sequence numbers). There is also still the problem of update messages themselves being corrupted by memory or

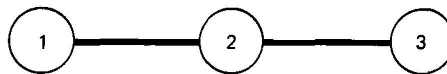


Figure 5.45 Example showing why it is necessary to compare the contents of A and B in the case of equal sequence numbers in the flooding algorithm without periodic updates of Section 5.3.2. Here, there are three nodes, 1, 2, and 3, connected in tandem with two (undirected) links (1,2) and (2,3). Suppose, initially, that both links are up, and the messages stored in all the nodes' memories carry the correct link status information and have a sequence number of zero. Consider a scenario whereby link (2,3) goes down, then link (1,2) goes down, and then link (2,3) comes up while node 2 resets its sequence number to zero. Then, nodes 2 and 3 exchange their (conflicting) view of the status of the directed links (1,2) and (2,1), but if the earlier flooding rules are used, both nodes discard each other's update message since it carries a sequence number zero which is equal to the one stored in their respective memories. By contrast, if the steps of the modified flooding algorithm described above are traced, it is seen that (depending on the lexicographic rule used) either the (correct) view of node 2 regarding link (2,1) will prevail right away, or else node 2 will issue a new update message with sequence number 1 and its view will again prevail. Note that node 3 may end up with an incorrect view of the status of link (1,2). This happens because nodes 1 and 3 are not connected with a path of up links. For the same reason, however, the incorrect information stored by node 3 regarding the outgoing link of node 1 is immaterial.

transmission errors (rather than just their sequence numbers being corrupted). It seems that there is no clean way to address these difficulties other than ad hoc error recovery schemes using an age field and periodic updates (see Problem 5.13).

5.3.3 Broadcast without Sequence Numbers

In this subsection we discuss a flooding-like algorithm called the *Shortest Path Topology Algorithm* (SPTA), which, unlike the previous schemes, does not use sequence numbers and avoids the attendant reset and wraparound problems.

To understand the main idea of the algorithm, consider a generic problem where a special node, called the *center*, has some information that it wishes to broadcast to all other nodes. We model this information as a variable V taking values in some set. As a special case, V could represent the status of the incident links of the center, as in the topology broadcast problem. We assume that after some initial time, several changes of V and several changes of the network topology occur, but there is some time after which there are no changes of either V or the network topology. We wish to have an algorithm by which all nodes get to know the correct value of V after a finite time from the last change.

Consider a scheme whereby each node i stores in memory a value V_i, which is an estimate of the correct value of V. Each time V_i changes, its value is transmitted to all neighbors of i along each of the currently operating outgoing links of i. Also, when a link (i, j) becomes operational after being down, the end nodes i and j exchange their current values V_i and V_j. Each node i stores the last value received by each of its neighbors j, denoted as V_j^i.

We now give a general rule for changing the estimate V_i of each node i. Suppose that we have an algorithm running in the network that maintains a tree of directed paths from all nodes to the center (*i.e.*, a tree rooted at the center; see Fig. 5.46). What we mean here is that this algorithm constructs such a tree within a finite time following the last change in the network topology. Every node i except for the center has a unique successor $s(i)$ in such a tree and we assume that this successor eventually becomes known to i. The rule for changing V_i then is for node i to make it equal to the value $V_{s(i)}^i$ last transmitted by the successor $s(i)$ within a finite time after either the current successor transmits a new value or the successor itself changes. It is evident under these assumptions that each node i will have the correct value V_i within a finite time following the last change in V or in the network topology.

It should be clear that there are many candidate algorithms that maintain a tree rooted at the center in the face of topological changes. One possibility is the asynchronous Bellman–Ford algorithm described in the preceding section, in which case the tree rooted at the center is a shortest path tree. The following SPTA algorithm constructs a separate shortest path tree for each node/center more economically than the asynchronous Bellman–Ford algorithm. The algorithm is given for the special case where the broadcast information is the network topology; in this case the tree construction algorithm and the topological change broadcast procedure are integrated into a single algorithm. The SPTA algorithm can be generalized, however, to broadcast any kind of information.

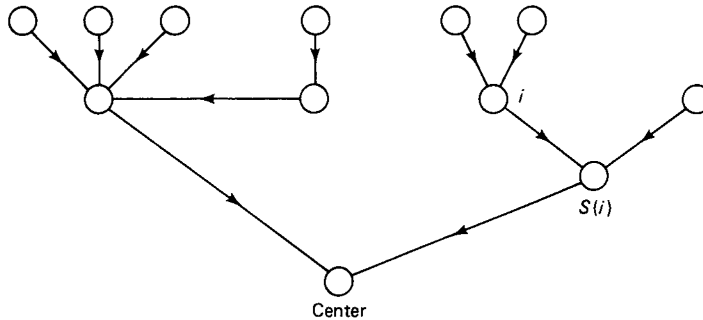


Figure 5.46 Information broadcast over a tree rooted at the center that initially holds the value V. Here there is a unique directed path from every processor to the center and a unique successor $s(i)$ of every node i on the tree. The idea of the broadcast algorithm is to maintain such a tree in the presence of topological changes and to require each node i to adopt (eventually) the latest received message from its successor $s(i)$ (i.e., $V_i = V^i_{s(i)}$). This guarantees that the correct information will eventually be propagated and adopted along the tree even after a finite number of topological changes that may require the restructuring of the tree several times.

Throughout this subsection we will make the same assumptions as with flooding about links preserving the order and correctness of transmissions, and nodes being able to detect incident link status changes.

The data structures that each node i maintains in the SPTA are (see Fig. 5.47):

1. The *main topology table* T_i where node i stores the believed status of every link. This is the "official" table used by the routing algorithm at the node. The main topology tables of different nodes may have different entries at different times. The goal of the SPTA is to get these tables to agree within finite time from the last topological change.
2. The *port topology tables* T^i_j. There is one such table for each neighbor node j. Node i stores the status of every link in the network, as communicated latest by neighbor node j, in T^i_j.

The algorithm consists of five simple rules:

Communication Rules

1. When a link status entry of a node's main topology table changes, the new entry is transmitted on each of the node's operating incident links.
2. When a failed link comes up, each of its end nodes transmits its entire main topology table to the opposite end node over that link. Upon reception of this table the opposite end node enters the new link status in its main and port topology tables.

Topology Table Update Rules

3. When an incident link of a node fails, the failed status is entered into the node's main and port topology tables.

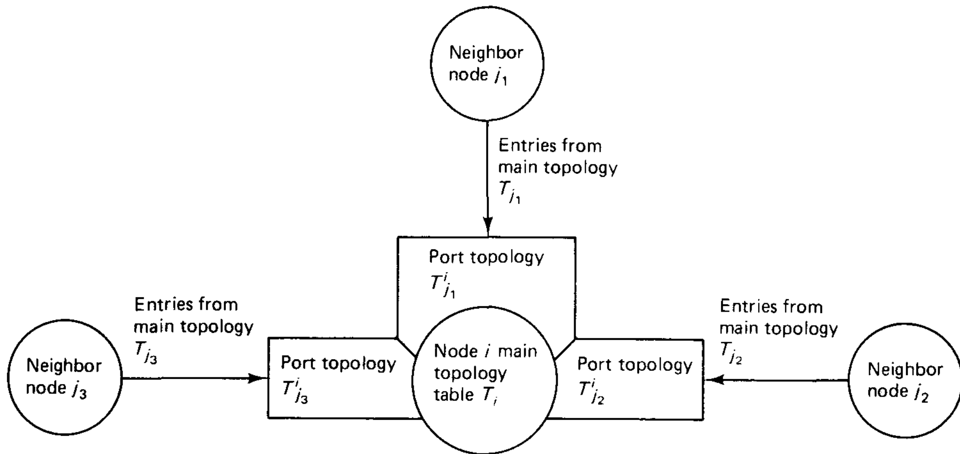


Figure 5.47 Data structures of SPTA at node i are the main topology table T_i and the port topology tables $T^i_{j_1}$, $T^i_{j_2}$, and $T^i_{j_3}$. Changes in the main topology tables of the neighbor nodes j_1, j_2, and j_3 are transferred with some delay to the corresponding port topology tables of node i, except if the change corresponds to a link that is incident to i. Such a change is entered directly in its main and port topology tables. Also, when an incident failed link comes up, node i transmits its entire main topology table over that link. Finally, when an entry of a port topology changes, the entire main topology table is recalculated using the main topology update algorithm.

4. When a node receives a link status message from a neighbor, it enters the message into the port topology table associated with that neighbor.
5. When an entry of a main topology T_i changes due to a status change of an incident link, or an entry of a port topology T^i_j changes due to a communication from neighbor node j, node i updates its main topology table by using the following algorithm. (It is assumed that the algorithm is restarted if an incident link changes status or a new message is received from a neighbor node while the algorithm executes.) The idea in the algorithm is for node i to believe, for each link ℓ, the status communicated by a neighbor that lies on a shortest path from node i to link ℓ. A shortest path is calculated on the basis of a length of unity for each up link and a length of infinity for each down link.

Main Topology Update Algorithm at Node i. The algorithm proceeds in iterations that are similar to those of Dijkstra's shortest path algorithm when all links have a unit length (Section 5.2.3). At the start of the kth iteration, there is a set of nodes P_k. Each node in P_k carries a label which is just the ID number of some neighbor node of i. The nodes in P_k are those that can be reached from i via a path of k links or fewer that are up according to node i's main topology table at the start of the kth iteration. The label of a node $m \in P_k$ is the ID number of the neighbor of i that lies on a path from i to m that has a minimum number of (up) links. In particular, P_1 consists of the nodes connected with i via an up link, and each node in P_1 is labeled with its own ID number. During the kth iteration, P_k is either augmented by some new nodes to form P_{k+1} or

else the algorithm terminates. Simultaneously, the status of the links with at least one end node in P_k but no end node in P_{k-1}, that is, the set of links

$$L_k = \{(m,n) \mid m \notin P_{k-1}, n \notin P_{k-1}, \ m \text{ or } n \text{ (or both) belongs to } P_k\}$$

is entered in the main topology table T_i. (For notational completeness, we define $P_0 = \{i\}$ in the equation for L_1.) The kth iteration is as follows:

Step 1: For each link $(m,n) \in L_k$, do the following: Let (without loss of generality) m be an end node belonging to P_k, and let "j" be the label of m. Copy the status of (m,n) from the port topology T_j^i into the main topology table T_i. If this status is up and $n \notin P_k$, give to node n the label "j".

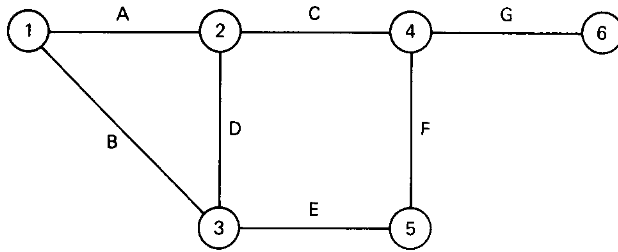
Step 2: Let M_k be the set of nodes that were labeled in step 1. If M_k is empty, terminate. Otherwise, set $P_{k+1} = P_k \cup M_k$, and go to the $(k+1)^{\text{st}}$ iteration.

The algorithm is illustrated in Fig. 5.48. Since each link is processed only once in step 1 of the algorithm, it is clear that the computational requirements are proportional to the number of links. It is straightforward to verify the following properties of the main topology update algorithm:

1. The set P_k, for $k \geq 1$, is the set of nodes m with the property that in the topology T_i finally obtained, there is a path of k or fewer up links connecting m with i.
2. The set L_k, $k \geq 1$, is the set of links ℓ with the property that in the topology T_i finally obtained, all paths of up links connecting one of the end nodes of ℓ with node i has no fewer than k links, and there is exactly one such path with k links.
3. The final entry of T_i for a link $\ell \in L_k$ is the entry from the port topology of a neighbor that is on a path of k up links from i to one of the end nodes of ℓ. If there are two such neighbors with conflicting entries, the algorithm arbitrarily chooses the entry of one of the two.
4. When the main topology update algorithm executes at node i in response to failure of an incident link (i,j), the information in the port topology T_j^i is in effect disregarded. This is due to the fact that node j is not included in the set P_1, and therefore no node is subsequently given the label "j" during execution of the algorithm.

The operation of the SPTA for the case of a single-link status change is shown in Fig. 5.49. The algorithm works roughly like flooding in this case, but there is no need to number the update messages to bound the number of message transmissions. Figure 5.50 illustrates the operation of the SPTA for the case of multiple-link status changes, and illustrates how the SPTA handles the problem of distinguishing old from new information (cf. Fig. 5.43). A key idea here is that when a link (i,j) fails, all information that came over that link is effectively disregarded, as discussed above.

We now provide a proof of correctness of the SPTA. At any given time the correct topology table (as seen by an omniscient observer) is denoted by T^*. We say that at that time *link ℓ is connected with node i* if there is a path connecting i with one of the end nodes of ℓ, and consisting of links that are up according to T^*. We say that at that



(a)

$$P_1 = \{2,3\}, \quad P_2 = \{2,3,5\}, \quad P_3 = \{2,3,5,4\}, \quad P_4 = \{2,3,5,4,6\}$$
$$L_1 = \{D,C,E\}, \quad L_2 = \{F\}, \quad L_3 = \{G\}$$

(b)

Link	Port Topology T_2^1	Port Topology T_3^1	Distance	Believes Neighbor	Final Main Topology of Node 1
A	—	—	0	—	U
B	—	—	0	—	U
C	D	U	1	2	D
D	U	U	1	2,3	U
E	U	U	1	3	U
F	U	U	2	3	U
G	D	U	3	3	U

(c)

Figure 5.48 Illustration of the main topology update algorithm for node 1 in the network (a). The node sets P_k and the link sets L_k are given in (b). The contents of the port topologies T_2^1 and T_3^1 at node 1 are given in the table in (c) (where D means down, U means up). There is conflicting information on the status of links C and G. Node 1 believes node 2 on the status of link C since node 2 is closer to this link. Node 1 believes node 3 on the status of link G since node 3 is closer to this link (given that link C was declared down by node 1).

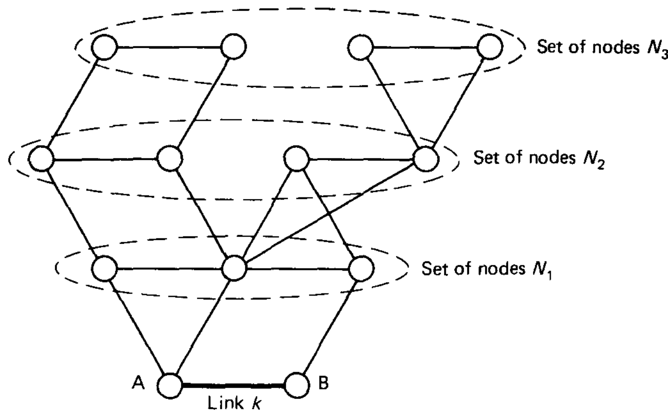
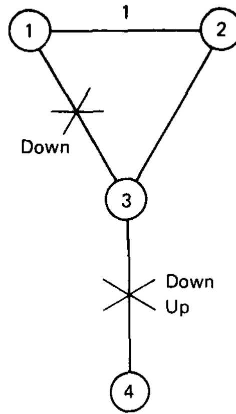


Figure 5.49 Operation of the SPTA for a single-link status change. The status change information propagates to nodes in the set N_1 (one link away from A or B or both), then to nodes in N_2 (two links away from A or B or both), then to N_3, and so on. Specifically, when link k changes status, the change is entered in the topology tables of A and B and is broadcast to the nodes in N_1. It is entered in the port topology tables T_A^i and/or T_B^i of nodes i in N_1. Since the distance to link k through nodes $j \neq A, B$ is greater than zero, the new status of link k will eventually be entered in the main topology tables of each node in N_1. It will then be broadcast to nodes in N_2 and entered in the corresponding port topology tables, and so on. The number of required message transmissions does not exceed $2L$, where L is the number of undirected network links.

time, node j is an *active neighbor* of node i if link (i, j) is up according to T*. We say that at that time, *node i knows the correct topology* if the main topology table T_i agrees with T^* on the status of all links connected with i. We assume that at some initial time t_s, each node knows the correct topology (*e.g.*, at network reset time when all links are down), and all active port topologies T_j^i agree on all entries with the corresponding main topologies T_j. After that time, several links change status. However, there is a time t_0 after which no link changes status and the end nodes of each link are aware of the correct link status. We will show the following:

Proposition. The SPTA works correctly in the sense that under the preceding assumptions, there is a time $t_f \geq t_0$ such that each node knows the correct topology for all $t \geq t_f$.

Figure 5.50 Operation of the SPTA for the topology change scenario of Fig. 5.43. The example demonstrates how the SPTA copes with a situation where pure flooding fails. (a) Example network with four links and four nodes. Initially, all links are up. Three changes occur: (1) link (3,4) goes down, (2) link (3,4) goes up, and (3) link (1,3) goes down. (b) Assumed sequence of events, message transmissions, and receptions of the SPTA. All topology table updating is assumed to take place instantaneously following an event. (c) Main and port topology table contents at each node following each event. U and D mean up and down, respectively. For example, a table entry UUUD means that links (1,2), (2,3), and (1,3) are up, and link (3,4) is down. Tables that change after an event are shown in bold. Note that the last event changes not only the entry for link (1,3) in T_1, but also the entry for link (3,4). Also, after the last event, the port topology table T_1^3 is incorrect, but this is inconsequential since the connecting link (1,3) is down.



(a)

Event Number	1	2	3
Event	Link (3,4) goes down. Message is transmitted from 3 to 1 and 2. Message is received at 2.	Message is transmitted from 2 to 1. Message is received at 1.	Link (3,4) goes up. Message is transmitted from 3 to 1 and 2. Message is received at 2.
Event Number	**4**	**5**	**6**
Event	Message is transmitted from 2 to 1. Message is received at 1.	Message from 3 to 1 sent during Event 1 is received.	Link (1,3) goes down. Messages from 1 and 3 are sent to 2 and 4. Messages received at 2 and 4.

(b)

Event Number		1	2	3	4	5	6
	T_1	UUUU	UUUU	UUUU	UUUU	**UUUD**	**UUDU**
Tables at Node 1	T_1^2	UUUU	**UUUD**	UUUD	**UUUU**	UUUU	**UUDU**
	T_1^3	UUUU	UUUU	UUUU	UUUU	**UUUD**	**UUDD**
	T_2	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
Tables at Node 2	T_2^1	UUUU	UUUU	UUUU	UUUU	UUUU	**UUDU**
	T_2^3	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
	T_3	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
Tables at Node 3	T_3^1	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
	T_1^2	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
	T_1^4	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
Tables at Node 4	T_4	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**
	T_4^3	**UUUD**	UUUD	**UUUU**	UUUU	UUUU	**UUDU**

(c)

Proof: In what follows we say that a link ℓ is at distance n away from i if in the graph defined by T^*, the shortest path from i to the closest end node of ℓ is n links long. We will show by induction that for each integer $n \geq 0$, there is a time $t_n \geq t_0$ after which, for each node i, T_i agrees with T^* for all links that are at a distance n or less from i. The induction hypothesis is clearly true for $n = 0$, since each node i knows the correct status of its incident links and records them in its main topology table T_i. We first establish the following lemma.

Lemma. Assume that the induction hypothesis is true for time t_n. Then, there is a time $t'_{n+1} \geq t_n$ after which the port topology table T_j^i for each active neighbor j of node i agrees with T^* for each link at a distance n or less from j.

Proof: Consider waiting a sufficient amount of time after t_n for all messages which were sent from j to i before t_n to arrive. By rules 1, 2, and 4 of the algorithm, T_j^i agrees with T_j for all links which are not incident to i. Therefore, by the induction hypothesis, T_j^i agrees with T^* for each link at a distance of n or less from j. This proves the lemma.

To complete the proof of the proposition, one must show that there is a time $t_{n+1} \geq t'_{n+1}$ such that for all $t \geq t_{n+1}$ and nodes i, T_i agrees with T^* for each link ℓ which is at a distance $n+1$ from i. Consider the first time that link ℓ is processed by the topology update algorithm after all port topologies T_j^i and T^* agree on all links at distance n or less from j as per the preceding lemma. Then link ℓ will belong to the set L_{n+1}. Also, the closest end node(s) of ℓ to node i will belong to the set P_{n+1} and will have a label which is one of the active neighbors of i (say j) that is at distance n from ℓ. By the lemma, the port topology T_j^i agrees with T^* on all links at distance n from j. Therefore, the entry of T_j^i for link ℓ, which will be copied into T_i when link ℓ is processed, will agree with the corresponding entry of T^*. Because the conclusion of the lemma holds for all t after t'_{n+1}, the entry for link ℓ in T_i will also be correct in all subsequent times at which link ℓ will be processed by the main topology update algorithm. **Q.E.D.**

Note that the lemma above establishes that the active port topology tables also eventually agree with the corresponding main topology tables. This shows that the initial conditions for which validity of the algorithm was shown reestablish themselves following a sufficiently long period for which no topological changes occur.

The main advantage of the SPTA over the ARPANET flooding algorithm is that it does not require the use of sequence numbers with the attendant reset difficulties, and the use of an age field, the proper size of which is network dependent and changes as the network expands. Furthermore, the SPTA is entirely event driven and does not require the substantial overhead associated with the regular periodic updates of the ARPANET algorithm.

Consider now the behavior of SPTA when we cannot rely on the assumption that errors in transmission or in nodes' memories are always detected and corrected. Like all event-driven algorithms, SPTA has no ability to correct such errors. Without periodic

retransmission, an undetected error can persist for an arbitrary length of time. Thus, in situations where extraordinary reliability is needed, a node should periodically retransmit its main table to each of its neighbors. This would destroy the event-driven property of SPTA, but would maintain its other desirable characteristics and allow recovery from undetected database errors.

As mentioned earlier, the SPTA can be generalized to broadcast information other than link status throughout the network. For example, in the context of an adaptive routing algorithm, one might be interested in broadcasting average delay or traffic arrival rate information for each link. What is needed for the SPTA is to allow for additional, perhaps nonbinary, information entry in the main and port topology tables in addition to the up–down status of links. When this additional information depends on the link direction, it is necessary to make a minor modification to the SPTA so that there are separate entries in the topology tables for each link direction. This matter is considered in Problem 5.14.

5.4 FLOW MODELS, OPTIMAL ROUTING, AND TOPOLOGICAL DESIGN

To evaluate the performance of a routing algorithm, we need to quantify the notion of traffic congestion. In this section we formulate performance models based on the traffic arrival rates at the network links. These models, called *flow models* because of their relation to network flow optimization models, are used to formulate problems of optimal routing that will be the subject of Sections 5.5 to 5.7. Flow models are also used in our discussion of the design of various parts of the network topology in Sections 5.4.1 to 5.4.3.

Traffic congestion in a data network can be quantified in terms of the statistics of the arrival processes of the network queues. These statistics determine the distributions of queue length and packet waiting time at each link. It is evident that desirable routing is associated with a small mean and variance of packet delay at each queue. Unfortunately, it is generally difficult to express this objective in a single figure of merit suitable for optimization. A principal reason is that, as seen in Chapter 3, there is usually no accurate analytical expression for the means or variances of the queue lengths in a data network.

A convenient but somewhat imperfect alternative is to measure congestion at a link in terms of the *average* traffic carried by the link. More precisely, we assume that the statistics of the arrival process at each link (i, j) change due only to routing updates, and that we measure congestion on (i, j) via the traffic arrival rate F_{ij}. We call F_{ij} the *flow* of link (i, j), and we express it in data units/sec where the data units can be bits, packets, messages, and so on. Sometimes, it is meaningful to express flow in units that are assumed directly proportional to data units/sec, such as virtual circuits traversing the link.

Implicit in flow models is the assumption that the statistics of the traffic entering the network do not change over time. This is a reasonable hypothesis when these statistics change very slowly relative to the average time required to empty the queues in the network. A typical network where such conditions hold is one accommodating a large number of users for each origin–destination pair, with the traffic rate of each of these users being small relative to the total rate (see Section 3.4.2).

An expression of the form

$$\sum_{(i,j)} D_{ij}(F_{ij}) \tag{5.29}$$

where each function D_{ij} is monotonically increasing, is often appropriate as a cost function for optimization. A frequently used formula is

$$D_{ij}(F_{ij}) = \frac{F_{ij}}{C_{ij} - F_{ij}} + d_{ij}F_{ij} \tag{5.30}$$

where C_{ij} is the transmission capacity of link (i,j) measured in the same units as F_{ij}, and d_{ij} is the processing and propagation delay. With this formula, the cost function (5.29) becomes the average number of packets in the system based on the hypothesis that each queue behaves as an $M/M/1$ queue of packets—a consequence of the Kleinrock independence approximation and Jackson's Theorem discussed in Sections 3.6 and 3.8. Although this hypothesis is typically violated in real networks, the cost function of Eqs. (5.29) and (5.30) represents a useful measure of performance in practice, principally because it expresses qualitatively that congestion sets in when a flow F_{ij} approaches the corresponding link capacity C_{ij}. Another cost function with similar qualitative properties is given by

$$\max_{(i,j)} \left\{ \frac{F_{ij}}{C_{ij}} \right\} \tag{5.31}$$

(*i.e.*, maximum link utilization). A computational study [Vas79] has shown that it typically makes little difference whether the cost function of Eqs. (5.29) and (5.30) or that of Eq. (5.31) is used for routing optimization. This indicates that one should employ the cost function that is easiest to optimize. In what follows we concentrate on cost functions of the form $\sum_{(i,j)} D_{ij}(F_{ij})$. The analysis and computational methods of Sections 5.5 to 5.7 extend to more general cost functions (see Problem 5.27).

We now formulate a problem of optimal routing. For each pair $w = (i,j)$ of distinct nodes i and j [also referred to as an origin–destination (or OD) pair], the input traffic arrival process is assumed stationary with rate r_w.* Thus r_w (measured in data units/sec) is the arrival rate of traffic entering the network at node i and destined for node j. The routing objective is to divide each r_w among the many paths from origin to destination in a way that the resulting total link flow pattern minimizes the cost function (5.29). More precisely, denote

W = Set of all OD pairs

P_w = Set of all directed paths connecting the origin and destination nodes of OD pair w (in a variation of the problem, P_w is a given subset of the set of all directed paths connecting origin and destination of w; the optimality conditions and algorithms to be given later apply nearly verbatim in this case)

*Sometimes it is useful to adopt a broader view of an OD pair and consider it simply as a class of users that shares the same set of paths. This allows modeling of multiple priority classes of users. In this context we would allow several OD pairs to have the same origin and destination nodes. The subsequent analysis and algorithms extend to this broader context almost verbatim. To simplify the following exposition, however, we assume that there is at most one OD pair associated with each pair of nodes.

x_p = Flow (data units/sec) of path p

Then the collection of all path flows $\{x_p \mid w \in W, p \in P_w\}$ must satisfy the constraints

$$\sum_{p \in P_w} x_p = r_w, \qquad \text{for all } w \in W,$$

$$x_p \geq 0, \qquad \text{for all } p \in P_w, w \in W$$

as shown in Fig. 5.51. The total flow F_{ij} of link (i, j) is the sum of all path flows traversing the link

$$F_{ij} = \sum_{\substack{\text{all paths } p \\ \text{containing } (i,j)}} x_p$$

Consider a cost function of the form

$$\sum_{(i,j)} D_{ij}(F_{ij}) \tag{5.32}$$

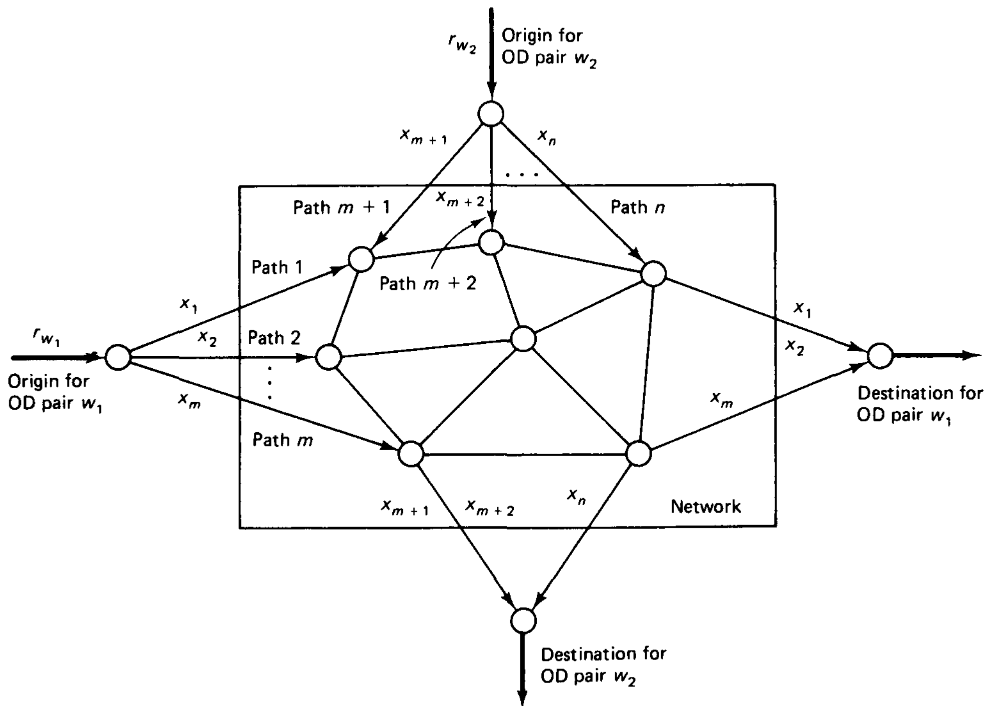


Figure 5.51 Schematic representation of a network with two OD pairs w_1 and w_2. The paths of the OD pairs are $P_{w_1} = \{1, 2, \ldots, m\}$ and $P_{w_2} = \{m+1, m+2, \ldots, n\}$. The traffic inputs r_{w_1} and r_{w_2} are to be divided into the path flows $x_1, \ldots, x_m$ and $x_{m+1}, \ldots, x_n$, respectively.

and the problem of finding the set of path flows $\{x_p\}$ that minimize this cost function subject to the constraints above.

By expressing the total flows F_{ij} in terms of the path flows in the cost function (5.32), the problem can be written as

$$\text{minimize} \sum_{(i,j)} D_{ij}\left[\sum_{\substack{\text{all paths } p \\ \text{containing } (i,j)}} x_p\right]$$

$$\text{subject to} \sum_{p\in P_w} x_p = r_w, \qquad \text{for all } w \in W \tag{5.33}$$

$$x_p \geq 0, \quad \text{for all } p \in P_w, w \in W$$

Thus, the problem is formulated in terms of the unknown path flows $\{x_p \mid p \in P_w, w \in W\}$. This is the main optimal routing problem that will be considered. A characterization of its optimal solution is given in Section 5.5, and interestingly, it is expressed in terms of shortest paths. Algorithms for its solution are given in Sections 5.6 and 5.7.

The optimal routing problem just formulated is amenable to analytical investigation and distributed computational solution. However, it has some limitations that are worth explaining. The main limitation has to do with the choice of the cost function $\sum_{(i,j)} D_{ij}(F_{ij})$ as a figure of merit. The underlying hypothesis here is that one achieves reasonably good routing by optimizing the average levels of link traffic without paying attention to other aspects of the traffic statistics. Thus, the cost function $\sum_{(i,j)} D_{ij}(F_{ij})$ is insensitive to undesirable behavior associated with high variance and with correlations of packet interarrival times and transmission times. To illustrate this fact, consider the example given in Section 3.6, where we had a node A sending Poisson traffic to a node B along two equal-capacity links. We compared splitting of the traffic between the two links by using randomization and by using metering. It was concluded that metering is preferable to randomization in terms of average packet delay. Yet randomization and metering are rated equally by the cost function $\sum_{(i,j)} D_{ij}(F_{ij})$, since they result in the same flow on each link. The problem here is that delay on each link depends on second and higher moments of the arrival process, while the cost function reflects a dependence on just the first moment.

Since metering is based on keeping track of current queue lengths, the preceding example shows that routing can be improved by using queue length information. Here is another example that illustrates the same point.

Example 5.6

Consider the network shown in Fig. 5.52. Here there are three origin–destination pairs, each with an arrival rate of 1 unit. The OD pairs (2,4) and (3,4) route their traffic exclusively along links (2,4) and (3,4) respectively. Since all links have a capacity of 2 units, the two paths (1,2,4) and (1,3,4) are equally attractive for the traffic of OD pair (1,4). An optimal routing algorithm based on time average link rates would alternate sending packets of OD pair (1,4) along the two paths, thereby resulting in a total rate of 1.5 units on each of the bottleneck links (2,4) and (3,4). This would work reasonably well if the traffic of the OD

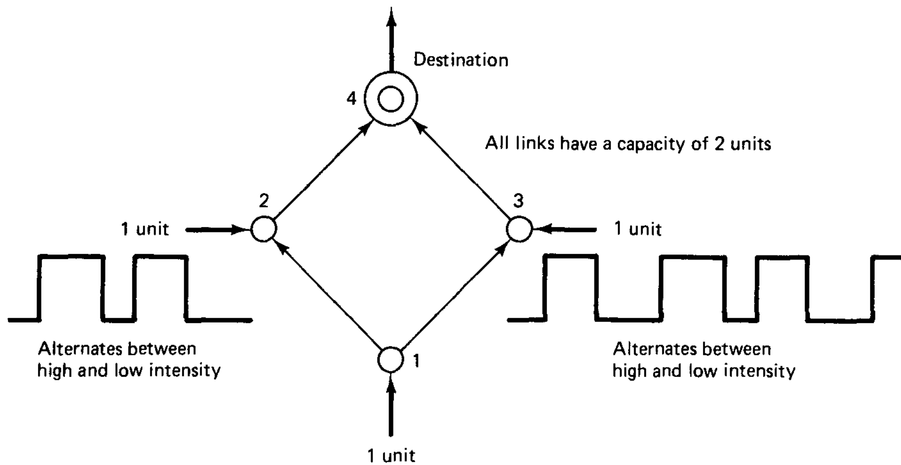


Figure 5.52 Network of Example 5.6 where routing based on queue length information is beneficial. The traffic originating at nodes 2 and 3 alternates in intensity between 2 units and 0 units over large time intervals. A good form of routing gets information on the queue sizes at links (2,4) and (3,4) and routes the traffic originating at node 1 on the path of least queue size.

pairs (2,4) and (3,4) had a Poisson character. Consider, however, the situation where instead, this traffic alternates in intensity between 2 units and 0 units over large time intervals. Then the queues in link (2,4) and (3,4) would build up (not necessarily simultaneously) over some large time intervals and empty out during others. Suppose now that a more dynamic form of routing is adopted, whereby node 1 is kept informed about queue sizes at links (2,4) and (3,4), and routes packets on the path of least queue size. Queueing delay would then be greatly reduced.

Unfortunately, it is impractical to keep nodes informed of all queue lengths in a large network. Even if the overhead for doing so were not prohibitive, the delays involved in transferring the queue length information to the nodes could make this information largely obsolete. It is not known at present how to implement effective and practical routing based on queue length information, so we will not consider the subject further. For an interesting but untested alternative, see Problem 5.38.

The remainder of this section considers the use of network algorithms, flow models, and the optimal routing problem in the context of topological design of a network. The reader may skip this material without loss of continuity.

5.4.1 An Overview of Topological Design Problems

In this subsection we discuss algorithms for designing the topology of a data network. Basically, we are given a set of traffic demands, and we want to put together a network that will service these demands at minimum cost while meeting some performance requirements. A common, broadly stated formulation of the problem is as follows:

We assume that we are given:

1. The geographical location of a collection of devices that need to communicate with each other. For simplicity, these devices are called *terminals.*
2. A traffic matrix giving the input traffic flow from each terminal to every other terminal.

We want to design (cf. Fig. 5.53):

1. The *topology of a communication subnet* to service the traffic demands of the terminals. This includes the location of the nodes, the choice of links, and the capacity of each link.
2. The *local access network* (*i.e.*, the collection of communication lines that will connect the terminals to entry nodes of the subnet).

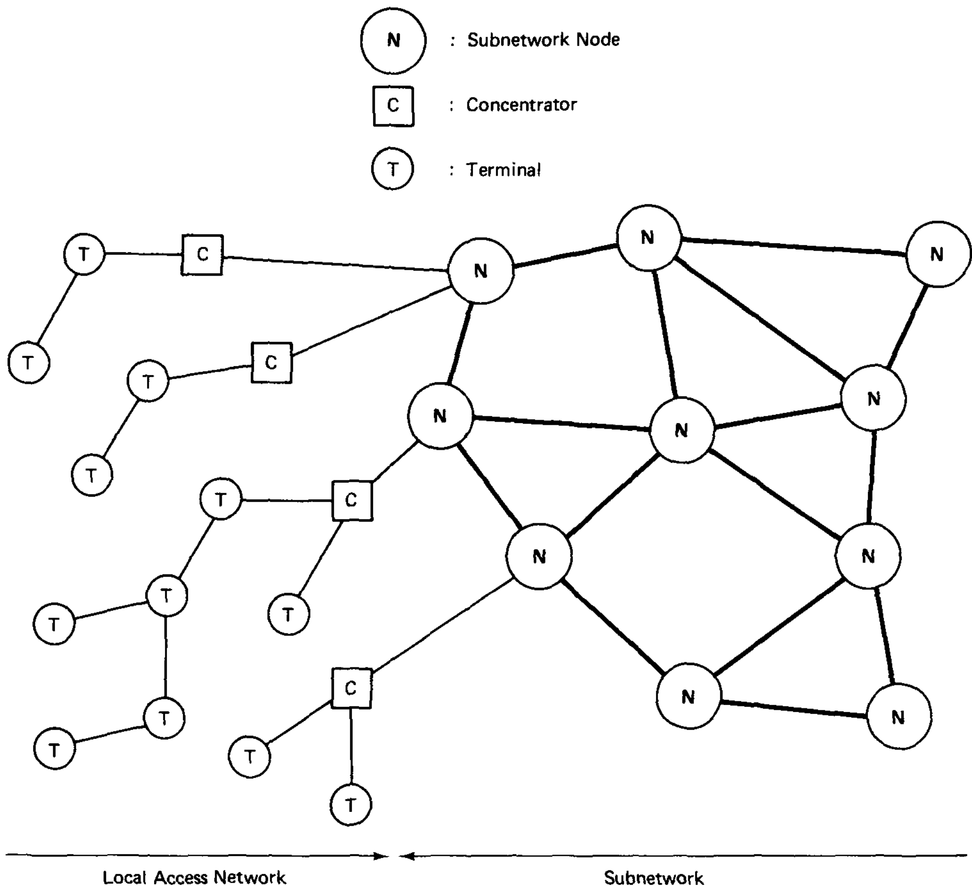


Figure 5.53 The main aspects of the topology design problem are the design of the local access network and the communication subnet.

The design objectives are:

1. Keep the average delay per packet or message below a given level (for the given nominal traffic demands and assuming some type of routing algorithm)
2. Satisfy some reliability constraint to guarantee the integrity of network service in the face of a number of link and node failures
3. Minimize a combination of capital investment and operating costs while meeting objectives 1 and 2

It is evident that this is, in general, a very broad problem that may be difficult to formulate precisely, let alone solve. However, in many cases, the situation simplifies somewhat. For example, part of the problem may already be solved because portions of the local access network and/or the subnet might already be in place. There may be also a natural decomposition of the problem—for example, the subnet topology design problem may be naturally decoupled from the local access network design problem. In fact, this will be assumed in the presentation. Still, one seldom ends up with a clean problem that can be solved exactly in reasonable time. One typically has to be satisfied with an approximate solution obtained through heuristic methods that combine theory, trial and error, and common sense.

In Section 5.4.2 we consider the subnet design problem, assuming that the local access network has been designed and therefore, the matrix of input traffic flow for every pair of subnet nodes is known. Subsequently, in Section 5.4.3, the local access network design problem is considered.

5.4.2 Subnet Design Problem

Given the location of the subnet nodes and an input traffic flow for each pair of these nodes, we want to select the capacity and flow of each link so as to meet some delay and reliability constraints while minimizing costs. Except in very simple cases, this turns out to be a difficult combinatorial problem. To illustrate the difficulties, we consider a simplified version of the problem, where we want to choose the link capacities so as to minimize a linear cost. We impose a delay constraint but neglect any reliability constraints. Note that by assigning zero capacity to a link we effectively eliminate the link. Therefore, the capacity assignment problem includes as a special case the problem of choosing the pairs of subnet nodes that will be directly connected with a communication line.

Capacity assignment problem. The problem is to choose the capacity C_{ij} of each link (i, j) so as to minimize the linear cost

$$\sum_{(i,j)} p_{ij} C_{ij} \tag{5.34}$$

where p_{ij} is a known positive price per unit capacity, subject to the constraint that the average delay per packet should not exceed a given constant T.

The flow on each link (i, j) is denoted F_{ij} and is expressed in the same units as capacity. We adopt the $M/M/1$ model based on the Kleinrock independence approximation, so we can express the average delay constraint as

$$\frac{1}{\gamma} \sum_{(i,j)} \frac{F_{ij}}{C_{ij} - F_{ij}} \leq T \tag{5.35}$$

where γ is the total arrival rate into the network. We assume that there is a given input flow for each origin–destination pair and γ is the sum of these. The link flows F_{ij} depend on the known input flows and the scheme used for routing. We will first assume that routing and therefore also the flows F_{ij} are known, and later see what happens when this assumption is relaxed.

When the flows F_{ij} are known, the problem is to minimize the linear cost $\sum_{(i,j)} p_{ij} C_{ij}$ over the capacities C_{ij} subject to the constraint (5.35), and it is intuitively clear that the constraint will be satisfied as an equality at the optimum. We introduce a Lagrange multiplier β and form the Lagrangian function

$$L = \sum_{(i,j)} \left(p_{ij} C_{ij} + \frac{\beta}{\gamma} \frac{F_{ij}}{C_{ij} - F_{ij}} \right)$$

In accordance with the Lagrange multiplier technique, we set the partial derivatives $\partial L / \partial C_{ij}$ to zero:

$$\frac{\partial L}{\partial C_{ij}} = p_{ij} - \frac{\beta F_{ij}}{\gamma (C_{ij} - F_{ij})^2} = 0$$

Solving for C_{ij} gives

$$C_{ij} = F_{ij} + \sqrt{\frac{\beta F_{ij}}{\gamma p_{ij}}} \tag{5.36}$$

and substituting in the constraint equation, we obtain

$$T = \frac{1}{\gamma} \sum_{(i,j)} \frac{F_{ij}}{C_{ij} - F_{ij}} = \sum_{(i,j)} \sqrt{\frac{p_{ij} F_{ij}}{\beta \gamma}}$$

From the last equation,

$$\sqrt{\beta} = \frac{1}{T} \sum_{(i,j)} \sqrt{\frac{p_{ij} F_{ij}}{\gamma}} \tag{5.37}$$

which when substituted in Eq. (5.36) yields the optimal solution:*

$$C_{ij} = F_{ij} + \frac{1}{T}\sqrt{\frac{F_{ij}}{\gamma p_{ij}}}\sum_{(m,n)}\sqrt{\frac{p_{mn}F_{mn}}{\gamma}}$$

The solution, after rearranging, can also be written as

$$C_{ij} = F_{ij}\left(1 + \frac{1}{\gamma T}\frac{\sum_{(m,n)}\sqrt{p_{mn}F_{mn}}}{\sqrt{p_{ij}F_{ij}}}\right) \tag{5.38}$$

Finally, by substitution in the cost function $\sum_{(i,j)} p_{ij}C_{ij}$, the optimal cost is expressed as

$$\text{optimal cost} = \sum_{(i,j)} p_{ij}F_{ij} + \frac{1}{\gamma T}\left(\sum_{(i,j)}\sqrt{p_{ij}F_{ij}}\right)^2 \tag{5.39}$$

Consider now the problem of optimizing the network cost with respect to both the capacities C_{ij} and the flows F_{ij} (equivalently, the routing). This problem can be solved by minimizing the cost (5.39) with respect to F_{ij} and then obtaining the optimal capacities from the closed-form expression (5.38). Unfortunately, it turns out that the cost (5.39) has many local minima and is very difficult to minimize. Furthermore, in these local minima, there is a tendency for many of the flows F_{ij} and corresponding capacities C_{ij} to be zero. The resulting networks tend to have low connectivity (few links with large capacity) and may violate reliability constraints. The nature of this phenomenon can be understood by considering the simple network of Fig. 5.54 involving two nodes and n links. Suppose that we want to choose the capacities $C_1, \ldots, C_n$ so as to minimize the sum $C_1 + \cdots + C_n$ while dividing the input rate λ among the n links so as to meet an average delay constraint. From our comparison of statistical and time-division multiplexing (Sections 3.1 and 3.3), we know that delay increases if a transmission line is divided into smaller lines, each serving a fraction of the total traffic. It is therefore evident that the optimal solution is the minimal connectivity topology whereby all links are eliminated, except one which has just enough capacity to meet the delay constraint.

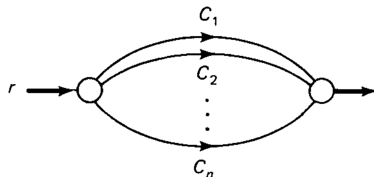


Figure 5.54 Minimizing the total capacity $C_1 + \cdots + C_n$ while meeting an average delay constraint results in a minimum connectivity network (all links are eliminated except one).

*It is rigorous to argue at this point that the capacities C_{ij} of Eq. (5.38) are optimal. The advanced reader can verify this by first showing that the average delay expression of Eq. (5.35) is convex and differentiable over the set of all C_{ij} with $C_{ij} > F_{ij}$, and by arguing that the equation $\partial L/\partial C_{ij} = 0$ is a sufficient condition for optimality that is satisfied by the positive Lagrange multiplier β of Eq. (5.37) and the capacities C_{ij} of Eq. (5.38) (see [Las70], p. 84, [Roc70], p. 283, or [BeT89], p. 662).

It makes no difference which lines are eliminated, so this problem has n local minima. Each of these happens to be also a global minimum, but this is a coincidence due to the extreme simplicity of the example.

The conclusion from the preceding discussion is that the simultaneous optimization of link flows and capacities is a hard combinatorial problem. Furthermore, even if this problem is solved, the network topology obtained will tend to concentrate capacity in a few large links, and possibly violate reliability constraints. The preceding formulation is also unrealistic because, in practice, capacity costs are not linear. Furthermore, the capacity of a leased communication line usually must be selected from a finite number of choices. In addition, the capacity of a line is typically equal in both directions, whereas this is not necessarily true for the optimal capacities of Eq. (5.38). These practical constraints enhance further the combinatorial character of the problem and make an exact solution virtually impossible. As a result, the only real option for addressing the problem subject to these constraints is to use the heuristic schemes that are described next.

Heuristic methods for capacity assignment. Typically, these methods start with a network topology and then successively perturb this topology by changing one or more link capacities at a time. Thus, these methods search "locally" around an existing topology for another topology that satisfies the constraints and has lower cost. Note that the term "topology" here means the set of capacities of all potential links of the network. In particular, a link with zero capacity represents a link that in effect does not exist. Usually, the following are assumed:

1. The nodes of the network and the input traffic flow for each pair of nodes are known.
2. A routing model has been adopted that determines the flows F_{ij} of all links (i, j) given all link capacities C_{ij}. The most common possibility here is to assume that link flows minimize a cost function $\sum_{(i,j)} D_{ij}(F_{ij})$ as in Eq. (5.33). In particular, F_{ij} can be determined by minimizing the average packet delay,

$$D = \frac{1}{\gamma} \sum_{(i,j)} \left(\frac{F_{ij}}{C_{ij} - F_{ij}} + d_{ij} F_{ij} \right) \tag{5.40}$$

   based on $M/M/1$ approximations [cf. Eq. (5.30)], where γ is the known total input flow into the network, and C_{ij} and d_{ij} are the known capacity and the processing and propagation delay, respectively, of link (i, j). Several algorithms described in Sections 5.6 and 5.7 can be used for this purpose.
3. There is a delay constraint that must be met. Typically, it is required that the chosen capacities C_{ij} and link flows F_{ij} (determined by the routing algorithm as in 2 above) result in a delay D given by the $M/M/1$ formula (5.40) (or a similar formula) that is below a certain threshold.
4. There is a reliability constraint that must be met. As an example, it is common to require that the network be 2-connected (*i.e.*, after a single node failure, all other nodes remain connected). More generally, it may be required that after $k-1$ nodes

fail, all other nodes remain connected—such a network is said to be k-connected. We postpone a discussion of how to evaluate the reliability of a network for later.

5. There is a cost criterion according to which different topologies are ranked.

The objective is to find a topology that meets the delay and reliability constraints as per 3 and 4 above, and has as small a cost as possible as per 5 above. We describe a prototype iterative heuristic method for addressing the problem. At the start of each iteration, there is available a *current best topology* and a *trial topology*. The former topology satisfies the delay and reliability constraints and is the best one in terms of cost that has been found so far, while the latter topology is the one that will be evaluated in the current iteration. We assume that these topologies are chosen initially by some ad hoc method—for example, by establishing an ample number of links to meet the delay and reliability constraints. The steps of the iteration are as follows:

Step 1: (Assign Flows). Calculate the link flows F_{ij} for the trial topology by means of some routing algorithm as per assumption 2.

Step 2: (Check Delay). Evaluate the average delay per packet D [as given, for example, by the $M/M/1$ formula (5.40)] for the trial topology. If

$$D \leq T$$

where T is a given threshold, go to step 3; else go to step 5.

Step 3: (Check Reliability). Test whether the trial topology meets the reliability constraints. (Methods for carrying out this step will be discussed shortly.) If the constraints are not met, go to step 5; else go to step 4.

Step 4: (Check Cost Improvement). If the cost of the trial topology is less than the cost of the current best topology, replace the current best topology with the trial topology.

Step 5: (Generate a New Trial Topology). Use some heuristic to change one or more capacities of the current best topology, thereby obtaining a trial topology that has not been considered before. Go to step 1.

Note that a trial topology is adopted as the current best topology in step 4 only if it satisfies the delay and reliability constraints in steps 2 and 3 and improves the cost (step 4). The algorithm terminates when no new trial topology can be generated or when substantial further improvement is deemed unlikely. Naturally, there is no guarantee that the final solution is optimal. One possibility to attempt further improvement is to repeat the algorithm with a different starting topology. Another possibility is to modify step 4 so that on occasion a trial topology is accepted as a replacement of the current best topology even if its cost is greater than the best found so far. A popular scheme of this type, known as *simulated annealing*, can be used to find a globally optimal solution under mild assumptions ([KGV83], [Haj88], and [Tsi89]). The amount of computing time required, however, may be excessive. For a method that uses this approach see [FeA89].

There are a number of heuristic rules proposed in the literature for generating new trial topologies (step 5). One possibility is simply to lower the capacity of a link that seems underutilized (low F_{ij}/C_{ij} and F_{ji}/C_{ji}) or to eliminate the link altogether. Another possibility is to increase the capacity of some overutilized link when the delay constraint is not met. A combination of these possibilities is the *branch exchange heuristic*, whereby one link is deleted and another link is added. A useful way to choose the links to be deleted and added is the so-called *saturated cut method*. The idea here is to identify a partition (or cut) of the nodes into two sets N_1 and N_2 such that the links joining N_1 and N_2 are highly utilized. It then seems plausible that adding a link between a node in N_1 and a node in N_2 will help reduce the high level of utilization across the cut. The method works as follows (see Fig. 5.55):

1. Prepare a list of all undirected links (i, j), sorted in order of decreasing utilization as measured by max $\{F_{ij}/C_{ij}, F_{ji}/C_{ji}\}$.
2. Find a link k such that (a) if all links above k on the list are eliminated, the network remains connected, and (b) if k is eliminated together with all links above it on the list, the network separates in two disconnected components N_1 and N_2.
3. Remove the most underutilized link in the network, and replace it with a new link that connects a node in the component N_1 and a node in the component N_2.

There are a number of variations of the scheme described above. For example, in selecting the link to remove, one may take into account the cost of capacity for the link. Needless to say, prior knowledge about the network and the cost structure for link capacity can be effectively incorporated into the heuristics. In addition, the design process can be enhanced by using an interactive program that can be appropriately guided by the designer. We now turn our attention to methods for evaluating reliability of a network.

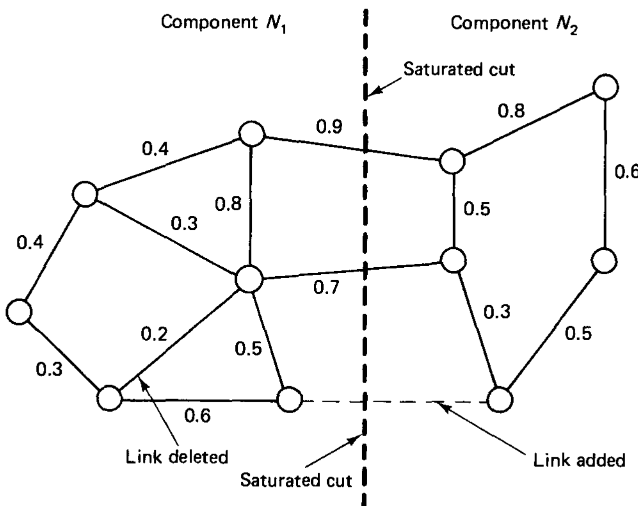


Figure 5.55 Illustration of the cut saturation method for performing a branch exchange heuristic. The number next to each arc is the link utilization. Links of high utilization are temporarily removed until the network separates into two components, N_1 and N_2. A link is then added that connects a node in N_1 with a node in N_2 while a link of low utilization is deleted.

Network reliability issues. We have already mentioned that a common reliability requirement for a network is that it must be k-connected. We now define k-connectivity more formally.

We say that *two nodes i and j* in an undirected graph are *k-connected* if there is a path connecting i and j in every subgraph obtained by deleting $(k-1)$ nodes other than i and j together with their adjacent arcs from the graph. We say that *the graph is k-connected* if every pair of nodes is k-connected. These definitions are illustrated in Fig. 5.56. Another notion of interest is that of *arc connectivity*, which is defined as the minimum number of arcs that must be deleted before the graph becomes disconnected. A lower bound k on arc connectivity can be established by checking for k-(node) connectivity in an expanded graph where each arc (i, j) is replaced by two arcs (i, n) and (n, j), where n is a new node. This is evident by noting that failure of node n in the expanded graph can be associated with failure of the arc (i, j) in the original graph.

It is possible to calculate the number of paths connecting two nodes i and j which are node-disjoint in the sense that any two of them share no nodes other than i and j. This can be done by setting up and solving a max-flow problem, a classical combinatorial problem, which is treated in detail in many sources (*e.g.*, [PaS82] and [Ber91]). We do not discuss here the max-flow problem, since it is somewhat tangential to the purposes of this section. The reader who is already familiar with this problem can see its connection with the problem of k-connectivity of two nodes from Fig. 5.57. Using the theory of the max-flow problem, it can be shown that i and j are k-connected if and only if either i and j are connected with an arc or there are at least k node-disjoint paths connecting i and j.

One way to check k-connectivity of a graph is to check k-connectivity of every pair of nodes. There are, however, more efficient methods. One particular method, due to Kleitman [Kle69], operates as follows:

Choose an arbitrary node n_0 and check k-connectivity between that node and every other node. Delete n_0 and its adjacent arcs from the graph, choose another node n_1, and check $(k-1)$-connectivity between that node and every other node. Continue in this manner until either node n_{k-1} is checked to be 1-connected to every remaining node, or $(k-i)$-connectivity of some node $n_i, i = 0, 1, \ldots, k-1$, to every remaining node

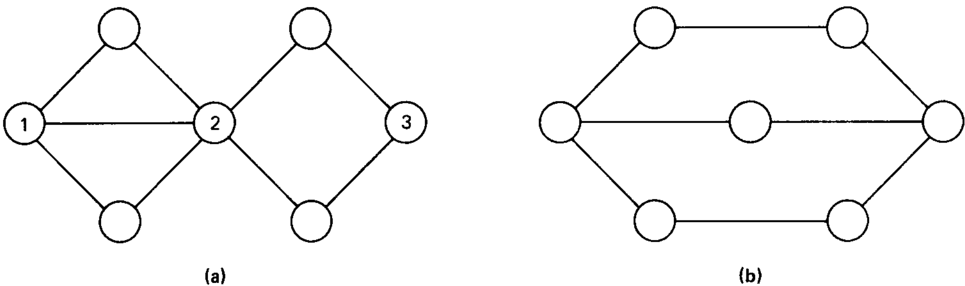


Figure 5.56 Illustration of the definition of k-connectivity. In graph (a), nodes 1 and 2 are 6-connected, nodes 2 and 3 are 2-connected, and nodes 1 and 3 are 1-connected. Graph (a) is 1-connected. Graph (b) is 2-connected.

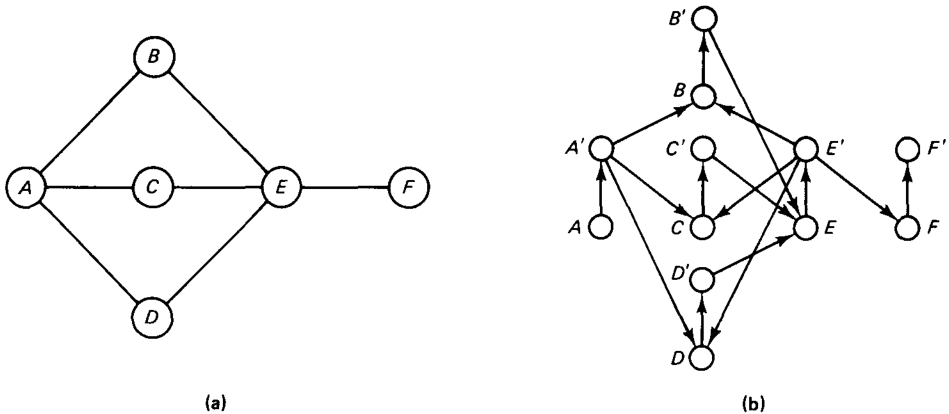


Figure 5.57 Illustration of test of k-connectivity for those familiar with the max-flow problem. To find the number of node-disjoint paths from node A to node F in graph (a), do the following: (1) Replace each node i with two nodes i and i' and a directed arc (i, i'). (2) Replace each arc (i, j) of graph (a) with two directed arcs (i', j) and (j', i). (3) Delete all arcs incoming to A and outgoing from F', thereby obtaining graph (b). (4) Assign infinite capacity to arcs (A, A') and (F, F'), and unit capacity to all other arcs. Then the maximum flow that can be sent from node A to node F' in graph (b) equals the number of node-disjoint paths connecting nodes A and F in graph (a).

cannot be verified. In the latter case, it is evident that the graph is not k-connected: If node n_i is not $(k - i)$-connected to some other remaining node, $(k - i - 1)$ nodes can be found such that their deletion along with nodes $n_0, \ldots, n_{i-1}$ disconnects the graph. The process is demonstrated in Fig. 5.58.

To establish the validity of Kleitman's algorithm, it must be shown that when nodes $n_0, \ldots, n_{k-1}$ can be found as described above, the graph is k-connected. The argument is by contradiction. If the graph is not k-connected, there must exist nodes i and j, and

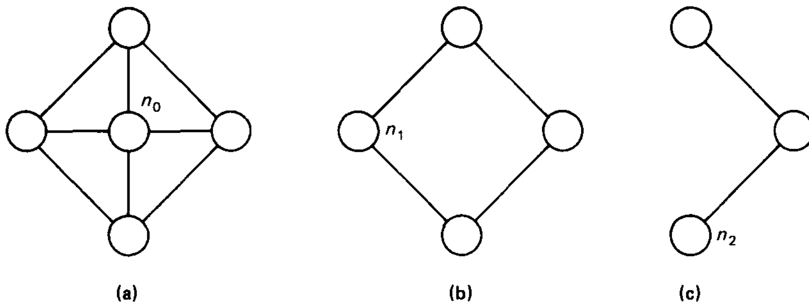


Figure 5.58 Verifying 3-connectivity of graph (a) by Kleitman's algorithm. An arbitrary node n_0 is chosen, and 3-connectivity between n_0 and every other node is verified. Then node n_0 and its adjacent arcs are deleted, thereby obtaining graph (b). An arbitrary node n_1 of (b) is chosen, and 2-connectivity between n_1 and every other node in (b) is verified. Then node n_1 and its adjacent arcs are deleted, thereby obtaining graph (c). Since graph (c) is connected, it follows that the original graph (a) is 3-connected.

a set of $(k-1)$ nodes $\mathcal{F}_0$ ($i \notin \mathcal{F}_0, j \notin \mathcal{F}_0$), such that deletion of $\mathcal{F}_0$ leaves no path connecting i and j. Then n_0 must belong to $\mathcal{F}_0$ because if $n_0 \notin \mathcal{F}_0$, then since n_0 is k-connected to every other node, it follows that there are paths P_i and P_j connecting n_0, i, and j even after the set $\mathcal{F}_0$ is deleted. Therefore, a path from i to j can be constructed by first using P_i to go from i to n_0, and then using P_j to go from n_0 to j. Next, consider the sets $\mathcal{F}_1 = \mathcal{F}_0 - \{n_0\}$, $\mathcal{F}_2 = \mathcal{F}_1 - \{n_1\}, \ldots$. Similarly, n_1 must belong to $\mathcal{F}_1$, n_2 must belong to $\mathcal{F}_2$, and so on. After this argument is used $(k-1)$ times, it is seen that $\mathcal{F}_0 = \{n_0, n_1, \ldots, n_{k-2}\}$. Therefore, after $\mathcal{F}_0$ is deleted from the graph, node n_{k-1} is still intact, and by definition of n_{k-1}, so are the paths from n_{k-1} to i and j. These paths can be used to connect i and j, contradicting the hypothesis that deletion of $\mathcal{F}_0$ leaves i and j disconnected.

Kleitman's algorithm requires a total of $\sum_{i=1}^{k}(N-i) = kN - k(k+1)/2$ connectivity tests of node pairs, where N is the number of nodes. There is another algorithm which requires roughly N connectivity tests; see [Eve75] for details.

The preceding formulation of the reliability problem addresses the worst case where the most unfavorable combination of k node failures occurs. An alternative formulation is to assign failure probabilities to all network elements (nodes and links) and evaluate the probability that a given pair of nodes becomes disconnected. To get an appreciation of the difficulties involved in this approach, suppose that there are n failure-prone elements in the network. Then the number of all distinct combinations of surviving elements is 2^n. Let s_k denote the kth combination and p_k denote the probability of its occurrence. The probability that the network remains connected is

$$P_C = \sum_{s_k \in C} p_k \tag{5.41}$$

where C is the set of combinations of surviving elements for which the network remains connected. Thus, evaluation of P_C involves some form of implicit or explicit enumeration of the set C. This is in general a very time-consuming procedure. However, shortcuts and approximations have been found making the evaluation of the survival probability P_C feasible for some realistic networks ([BaP83] and [Ben86]). An alternative approach evaluates a lower bound for P_C by ordering the possible combinations of surviving elements according to their likelihood, and then adding p_k over a subset of most likely combinations from C ([LaL86] and [LiS84]). A similar approach using the complement of C gives an upper bound to P_C.

Spanning tree topology design. For some networks where reliability is not a serious issue, a spanning tree topology may be appropriate. Using such a topology is consistent with the idea of concentrating capacity on just a few links to reduce the average delay per packet (cf. the earlier discussion on capacity assignment).

It is possible to design a spanning tree topology by assigning a weight w_{ij} to each possible link (i, j) and by using the minimum weight spanning tree (MST) algorithms of Section 5.2.2. A potential difficulty arises when there is a constraint on the amount of traffic that can be carried by any one link. This gives rise to the *constrained MST problem*, where a matrix of input traffic from every node to every other node is given

and an MST is to be designed subject to the constraint that the flow on each link will not exceed a given upper bound. Because of this constraint, the problem has a combinatorial character and is usually addressed using heuristic methods.

One possibility is to modify the Kruskal or Prim–Dijkstra algorithms, described in Section 5.2.2, so that at each iteration, when a new link is added to the current fragments, a check is made to see if the flow constraints on the links of the fragments are satisfied. If not, the link is not added and another link is considered. This type of heuristic can also be combined with some version of the branch exchange heuristic discussed earlier in connection with the general subnet design problem.

A special case of the constrained MST problem can be addressed using the *Essau–Williams algorithm* [EsW66]. In this case, there is a central node denoted 0, and N other nodes. All traffic must go through the central node, so it can be assumed that there is input traffic only from the noncentral nodes to the central node and the reverse. Problems of this type arise also in the context of the local access design problem, with the central node playing the role of a traffic concentrator.

The Essau–Williams algorithm is, in effect, a branch exchange heuristic. One starts with the spanning tree where the central node is directly connected with each of the N other nodes. (It is assumed that this is a feasible solution.) At each successive iteration, a link $(i, 0)$ connecting some node i with the central node 0 is deleted from the current spanning tree, and a link (i, j) is added. These links are chosen so that:

1. No cycle is formed.
2. The capacity constraints of all the links of the new spanning tree are satisfied.
3. The saving $w_{i0} - w_{ij}$ in link weight obtained by exchanging $(i, 0)$ with (i, j) is positive and is maximized over all nodes i and j for which 1 and 2 are satisfied.

The algorithm terminates when there are no nodes i and j for which requirements 1 to 3 are satisfied when $(i, 0)$ is exchanged with (i, j). The operation of the algorithm is illustrated in Fig. 5.59.

5.4.3 Local Access Network Design Problem

Here we assume that a communication subnet is available, and we want to design a network that connects a collection of terminals with known demands to the subnet. This problem is often addressed in the context of a hierarchical strategy, whereby groups of terminals are connected, perhaps through local area networks, to various types of concentrators, which are in turn connected to higher levels of concentrators, and so on. It is difficult to recommend a global design strategy without knowledge of the given practical situation. However, there are a few subproblems that arise frequently. We will discuss one such problem, known as the *concentrator location problem*.

In this problem, there are n sources located at known geographical points. For example, a source may be a gateway of a local area network that connects several terminals within a customer's facility; or it could be a host computer through which several time-sharing terminals access some network. The nature of the sources is not

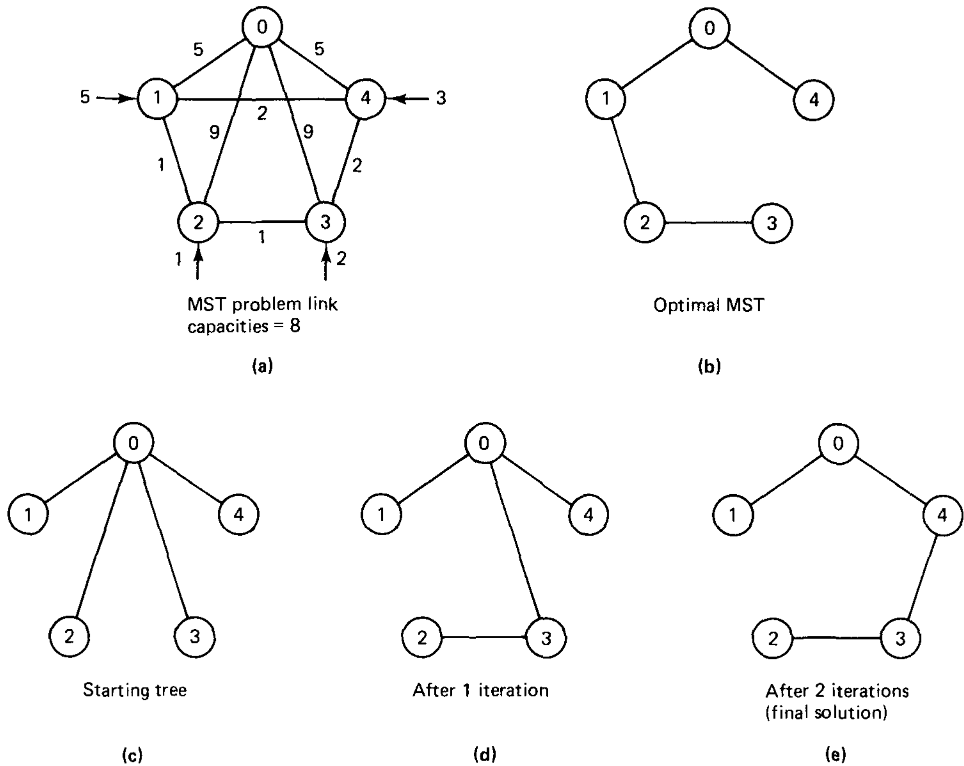


Figure 5.59 Illustration of the Essau–Williams, constrained MST heuristic algorithm. The problem data are given in (a). Node 0 is the central node. The link weights are shown next to the links. The input flows from nodes 1, 2, 3, and 4 to the central node are shown next to the arrows. The flow on each link is constrained to be no more than 8. The algorithm terminates after two iterations, with the tree (e) having a total weight of 13. Termination occurs because when link (1,0) or (4,0) is removed and a link that is not adjacent to node 0 is added, some link capacity constraint (link flow ≤ 8) will be violated. The optimal tree, shown in (b), has a total weight of 12.

material to the discussion. The problem is to connect the n sources to m concentrators that are themselves connected to a communication subnet (see Fig. 5.60). We denote by a_{ij} the cost of connecting source i to concentrator j. Consider the variables x_{ij} where

$$x_{ij} = \begin{cases} 1, & \text{if source } i \text{ is connected to concentrator } j \\ 0, & \text{otherwise} \end{cases}$$

Then the total cost of a source assignment specified by a set of variables $\{x_{ij}\}$ is

$$\text{cost} = \sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij} x_{ij} \,. \tag{5.42}$$

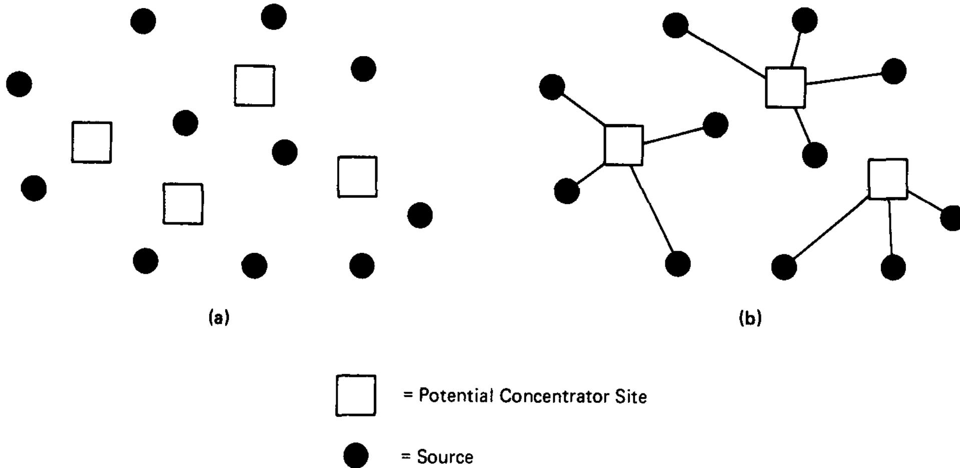


Figure 5.60 (a) Collection of sources and potential concentrator locations. (b) Possible assignment of sources to concentrators.

We assume that each source is connected to only one concentrator, so that there is the constraint

$$\sum_{j=1}^{m} x_{ij} = 1, \qquad \text{for all sources } i \tag{5.43}$$

Also, there is a maximum number of sources K_j that can be handled by concentrator j. This is expressed by the constraint

$$\sum_{i=1}^{n} x_{ij} \leq K_j, \qquad \text{for all concentrators } j \tag{5.44}$$

The problem is to select the variables x_{ij} so as to minimize the cost (5.42) subject to the constraints (5.43) and (5.44).

Even though the variables x_{ij} are constrained to be 0 or 1, the problem above is not really a difficult combinatorial problem. If the integer constraint

$$x_{ij} = 0 \text{ or } 1$$

is replaced by the noninteger constraint

$$0 \leq x_{ij} \leq 1 \tag{5.45}$$

the problem becomes a linear transportation problem that can be solved by very efficient algorithms such as the simplex method. It can be shown that the solution obtained from the simplex method will be integer (*i.e.*, x_{ij} will be either 0 or 1; see [Dan63] and [Ber91]). Therefore, solving the problem without taking into account the integer constraints automatically obtains an integer optimal solution. There are methods other than simplex that can be used for solving the problem ([Ber85], [Ber91], [BeT89], [FoF62], and [Roc84]).

Next, consider a more complicated problem whereby the location of the concentrators is not fixed but instead is subject to optimization. There are m potential concentrator sites, and there is a cost b_j for locating a concentrator at site j. The cost then becomes

$$\text{cost} = \sum_{i=1}^{n}\sum_{j=1}^{m} a_{ij}x_{ij} + \sum_{j=1}^{m} b_j y_j \tag{5.46}$$

where

$$y_j = \begin{cases} 1, & \text{if a concentrator is located at site } j \\ 0, & \text{otherwise} \end{cases}$$

The constraint (5.44) now becomes

$$\sum_{i=1}^{n} x_{ij} \le K_j y_j, \qquad \text{for all concentrators } j \tag{5.47}$$

Thus, the problem is to minimize the cost (5.46) subject to the constraints (5.43) and (5.47) and the requirement that x_{ij} and y_j are either 0 or 1.

This problem has been treated extensively in the operations research literature, where it is known as the warehouse location problem. It is considered to be a difficult combinatorial problem that can usually be solved only approximately. A number of exact and heuristic methods have been proposed for its solution. We refer the reader to the literature for further details ([AkK77], [AlM76], [CFN77], [KeH63], [Khu72], and [RaW83]).

5.5 CHARACTERIZATION OF OPTIMAL ROUTING

We now return to the optimal routing problem formulated at the beginning of Section 5.4. The main objective in this section is to show that optimal routing directs traffic exclusively along paths which are shortest with respect to some link lengths that depend on the flows carried by the links. This is an interesting characterization which motivates algorithms discussed in Sections 5.6 and 5.7.

Recall the form of the cost function

$$\sum_{(i,j)} D_{ij}(F_{ij}) \tag{5.48}$$

Here F_{ij} is the total flow (in data units/sec) carried by link (i, j) and given by

$$F_{ij} = \sum_{\substack{\text{all paths } p \\ \text{containing } (i,j)}} x_p \tag{5.49}$$

where x_p is the flow (in data units/sec) of path p. For every OD pair w, there are the constraints

$$\sum_{p \in P_w} x_p = r_w \tag{5.50}$$

$$x_p \ge 0, \qquad \text{for all } p \in P_w \tag{5.51}$$

where r_w is the given traffic input of the OD pair w (in data units/sec) and P_w is the set of directed paths of w. In terms of the unknown path flow vector $x = \{x_p \mid p \in P_w, w \in W\}$, the problem is written as

$$\begin{aligned} &\text{minimize} \sum_{(i,j)} D_{ij}\left[\sum_{\substack{\text{all paths } p\\ \text{containing } (i,j)}} x_p\right] \\ &\text{subject to} \sum_{p \in P_w} x_p = r_w, \qquad \text{for all } w \in W \\ &\qquad x_p \geq 0, \qquad \text{for all } p \in P_w, w \in W \end{aligned} \tag{5.52}$$

In what follows we will characterize an optimal routing in terms of the first derivatives D'_{ij} of the functions D_{ij}. We assume that each D_{ij} is a differentiable function of F_{ij} and is defined in an interval $[0, C_{ij})$, where C_{ij} is either a positive number (typically representing the capacity of the link) or else infinity. Let x be the vector of path flows x_p. Denote by $D(x)$ the cost function of the problem of Eq. (5.52),

$$D(x) = \sum_{(i,j)} D_{ij}\left[\sum_{\substack{\text{all paths } p\\ \text{containing } (i,j)}} x_p\right]$$

and by $\partial D(x)/\partial x_p$ the partial derivative of D with respect to x_p. Then

$$\frac{\partial D(x)}{\partial x_p} = \sum_{\substack{\text{all links } (i,j)\\ \text{on path } p}} D'_{ij}$$

where the first derivatives D'_{ij} are evaluated at the total flows corresponding to x. It is seen that $\partial D/\partial x_p$ *is the length of path p when the length of each link* (i, j) *is taken to be the first derivative* D'_{ij} *evaluated at* x. Consequently, in what follows $\partial D/\partial x_p$ is called the *first derivative length of path p.*

Let $x^* = \{x^*_p\}$ be an optimal path flow vector. Then if $x^*_p > 0$ for some path p of an OD pair w, we must be able to shift a small amount $\delta > 0$ from path p to any other path p' of the same OD pair without improving the cost; otherwise, the optimality of x^* would be violated. To first order, the change in cost from this shift is

$$\delta\frac{\partial D(x^*)}{\partial x_{p'}} - \delta\frac{\partial D(x^*)}{\partial x_p}$$

and since this change must be nonnegative, we obtain

$$x^*_p > 0 \qquad \Rightarrow \qquad \frac{\partial D(x^*)}{\partial x_{p'}} \geq \frac{\partial D(x^*)}{\partial x_p}, \qquad \text{for all } p' \in P_w \tag{5.53}$$

In words, *optimal path flow is positive only on paths with a minimum first derivative length.* Furthermore, at an optimum, the paths along which the input flow r_w of OD pair w is split must have *equal* length (and less or equal length to that of all other paths of w).

The condition (5.53) is a necessary condition for optimality of x^*. It can also be shown to be sufficient for optimality if the functions D_{ij} are convex; for example, when the second derivatives D''_{ij} exist and are positive in $[0, C_{ij})$, the domain of definition of D_{ij} (see Problem 5.24).

Example 5.7

Consider the two-link network shown in Fig. 5.61, where nodes 1 and 2 are the only origin and destination, respectively. The given input r is to be divided into the two path flows x_1 and x_2 so as to minimize a cost function based on the $M/M/1$ approximation

$$D(x) = D_1(x_1) + D_2(x_2)$$

where for $i = 1, 2$,

$$D_i(x_i) = \frac{x_i}{C_i - x_i}$$

and C_i is the capacity of link i. For the problem to make sense, we must assume that r is less than the maximum throughput $C_1 + C_2$ that the network can sustain.

At the optimum, the shortest path condition (5.53) must be satisfied as well as the constraints

$$x_1^* + x_2^* = r, \quad x_1^* \geq 0, \quad x_2^* \geq 0$$

Assume that $C_1 \geq C_2$. Then, from elementary reasoning, the optimal flow x_1 cannot be less than x_2 (it makes no sense to send more traffic on the slower link). The only possibilities are:

1. $x_1^* = r$ and $x_2^* = 0$. According to the shortest path condition (5.53), we must have

$$\frac{dD_1(r)}{dx_1} \leq \frac{dD_2(0)}{dx_2}$$

Equivalently [since the derivative of $x/(C - x)$ is $C/(C - x)^2$],

$$\frac{C_1}{(C_1 - r)^2} \leq \frac{1}{C_2}$$

or

$$r \leq C_1 - \sqrt{C_1 C_2} \tag{5.54}$$

2. $x_1^* > 0$ and $x_2^* > 0$. In this case, the shortest path condition (5.53) implies that the lengths of paths 1 and 2 are equal, that is,

$$\frac{dD_1(x_1^*)}{dx_1} = \frac{dD_2(x_2^*)}{dx_2}$$

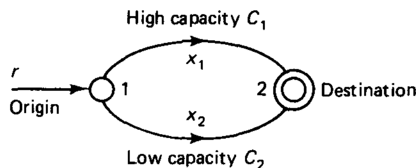


Figure 5.61 Example routing problem involving a single OD pair and two paths.

or, equivalently,

$$\frac{C_1}{(C_1 - x_1^*)^2} = \frac{C_2}{(C_2 - x_2^*)^2} \tag{5.55}$$

This equation together with the constraint $x_1^* + x_2^* = r$ determine the values of x_1^* and x_2^*. A straightforward calculation shows that the solution is

$$x_1^* = \frac{\sqrt{C_1}\left[r - \left(C_2 - \sqrt{C_1 C_2}\right)\right]}{\sqrt{C_1} + \sqrt{C_2}}$$

$$x_2^* = \frac{\sqrt{C_2}\left[r - \left(C_1 - \sqrt{C_1 C_2}\right)\right]}{\sqrt{C_1} + \sqrt{C_2}}$$

The optimal solution is shown in Fig. 5.62 for r in the range $[0, C_1 + C_2)$ of possible inputs. It can be seen that for

$$r \leq C_1 - \sqrt{C_1 C_2}$$

the faster link 1 is used exclusively. When r exceeds the threshold value $C_1 - \sqrt{C_1 C_2}$ for which the first derivative lengths of the two links are equalized and relation (5.54) holds as an equation, the slower link 2 is also utilized. As r increases, the flows on both links increase while the equality of the first derivative lengths, as in Eq. (5.55), is maintained. This behavior is typical of optimal routing when the cost function is based on the $M/M/1$ approximation; *for low input traffic, each OD pair tends to use only one path for routing (the fastest in terms of packet transmission time), and as traffic input increases, additional paths are used to avoid overloading the fastest path.* More generally, this type of behavior is associated with link cost functions D_{ij} with the property that the derivative $D'_{ij}(0)$ at zero flow depends only on the link capacity and decreases as the link capacity increases (see Problem 5.36).

We were able to solve the preceding example analytically only because of its extreme simplicity. Unfortunately, in more complex problems we typically have to resort to a computational solution—either centralized or distributed. The following sections deal with some of the possibilities.

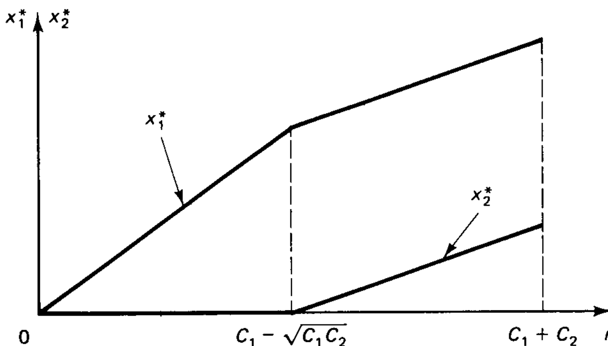


Figure 5.62 Optimal path flows for the routing example. When the traffic input is low, only the high-capacity link is used. As the input increases beyond the threshold $C_1 - \sqrt{C_1 C_2}$, some traffic is routed on the low-capacity link.

Finally, note that while the optimal routing problem was formulated in terms of a fixed set of input rates, the optimal solution $\{x_p^*\}$ can be implemented naturally even when the input rates are time varying. This can be done by working with the *fractions* of flow

$$\xi_p = \frac{x_p^*}{r_w}, \qquad \text{for all } p \in P_w$$

and by requiring each OD pair w to divide traffic (packets or virtual circuits) among the available paths according to these fractions. In virtual circuit networks, where the origin nodes are typically aware of the detailed description of the paths used for routing, it is easy to implement the routing process in terms of the fractions ξ_p. The origins can estimate the data rate of virtual circuits and can route new virtual circuits so as to bring about a close match between actual and desired fractions of traffic on each path. In some datagram networks or in networks where the detailed path descriptions are not known at the origin nodes, it may be necessary to maintain in a routing table at each node i a routing variable $\phi_{ik}(j)$ for each link (i,k) and destination j. This routing variable is defined as the fraction of all flow arriving at node i, destined for node j, and routed along link (i,k). Mathematically,

$$\phi_{ik}(j) = \frac{f_{ik}(j)}{\sum_m f_{im}(j)}, \qquad \text{for all } (i,k) \text{ and } j \tag{5.56}$$

where $f_{ik}(j)$ is the flow that travels on link (i,k) and is destined for node j. Given an optimal solution of the routing problem in terms of the path flow variables $\{x_p^*\}$, it is possible to determine the corresponding link flow variables $f_{ik}(j)$ and, by Eq. (5.56), the corresponding optimal routing variables $\phi_{ik}(j)$. A direct formulation and distributed computational solution of the optimal routing problem in terms of the variables $\phi_{ik}(j)$ is given in references [Gal77], [Ber79a], [Gaf79], and [BGG84].

5.6 FEASIBLE DIRECTION METHODS FOR OPTIMAL ROUTING

In Section 5.5 it was shown that optimal routing results only if flow travels along minimum first derivative length (MFDL) paths for each OD pair. Equivalently, a set of path flows is strictly suboptimal only if there is a positive amount of flow that travels on a non-MFDL path. This suggests that suboptimal routing can be improved by shifting flow to an MFDL path from other paths for each OD pair. The adaptive shortest path method for datagram networks of Section 5.2.5 does that in a sense, but shifts *all* flow of each OD pair to the shortest path, with oscillatory behavior resulting. It is more appropriate to shift only *part* of the flow of other paths to the shortest path. This section considers methods based on this idea. Generally, these methods solve the optimal routing problem computationally by decreasing the cost function through incremental changes in path flows.

Given a feasible path flow vector $x = \{x_p\}$ (*i.e.*, a vector x satisfying the constraints of the problem), consider changing x along a direction $\Delta x = \{\Delta x_p\}$. There are two requirements imposed on the direction Δx:

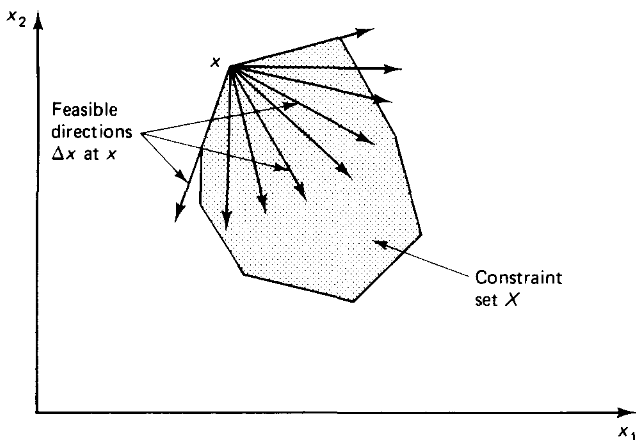


Figure 5.63 Feasible directions Δx at a feasible point x. Δx is a feasible direction if changing x by a small amount along the direction Δx still maintains feasibility.

1. The first requirement is that Δx should be a *feasible direction* in the sense that small changes along Δx maintain the feasibility of the path flow vector x (see Fig. 5.63). Mathematically, it is required that for some $\overline{\alpha} > 0$ and all $\alpha \in [0, \overline{\alpha}]$, the vector $x + \alpha \Delta x$ is feasible or equivalently,

$$\sum_{p \in P_w} \Delta x_p = 0, \quad \text{for all } w \in W \tag{5.57}$$

$$\Delta x_p \geq 0, \quad \text{for all } p \in P_w,\ w \in W \text{ for which } x_p = 0 \tag{5.58}$$

Equation (5.57) follows from the feasibility requirement

$$\sum_{p \in P_w} (x_p + \alpha \Delta x_p) = r_w$$

and the fact that x is feasible, which implies that

$$\sum_{p \in P_w} x_p = r_w$$

It simply expresses that to maintain feasibility, all increases of flow along some paths must be compensated by corresponding decreases along other paths of the same OD pair. One way to obtain feasible directions is to select another feasible vector $\overline{x}$ and take

$$\Delta x = \bar{x} - x$$

In fact, a little thought reveals that all feasible directions can be obtained in this way up to scalar multiplication.

2. The second requirement is that Δx should be a *descent direction* in the sense that the cost function can be decreased by making small movements along the direction Δx starting from x (see Fig. 5.64). Since the gradient vector $\nabla D(x)$ is normal to the equal cost surfaces of the cost function D, it is clear from Fig. 5.64 that the

descent condition translates to the condition that the inner product of $\nabla D(x)$ and Δx is negative, that is,

$$\sum_{w \in W} \sum_{p \in P_w} \frac{\partial D(x)}{\partial x_p} \Delta x_p < 0 \tag{5.59}$$

For a mathematical verification, note that the inner product in Eq. (5.59) is equal to the first derivative of the function $G(\alpha) = D(x + \alpha \Delta x)$ at $\alpha = 0$, so Eq. (5.59) is equivalent to $G(\alpha)$ being negative for sufficiently small positive α. Note that the partial derivative $\partial D(x)/\partial x_p$ is given by

$$\frac{\partial D(x)}{\partial x_p} = \sum_{\substack{\text{all links } (i,j) \\ \text{on path } p}} D'_{ij}(F_{ij}),$$

and can be viewed as the first derivative length of path p [the length of link (i,j) is $D'_{ij}(F_{ij})$, cf. Section 5.5]. One way to satisfy the descent condition of Eq. (5.59), which is in fact commonly used in algorithms, is to require that Δx satisfies the conservation of flow condition $\sum_{p \in P_w} \Delta x_p = 0$ [cf. Eq. (5.57)], and that

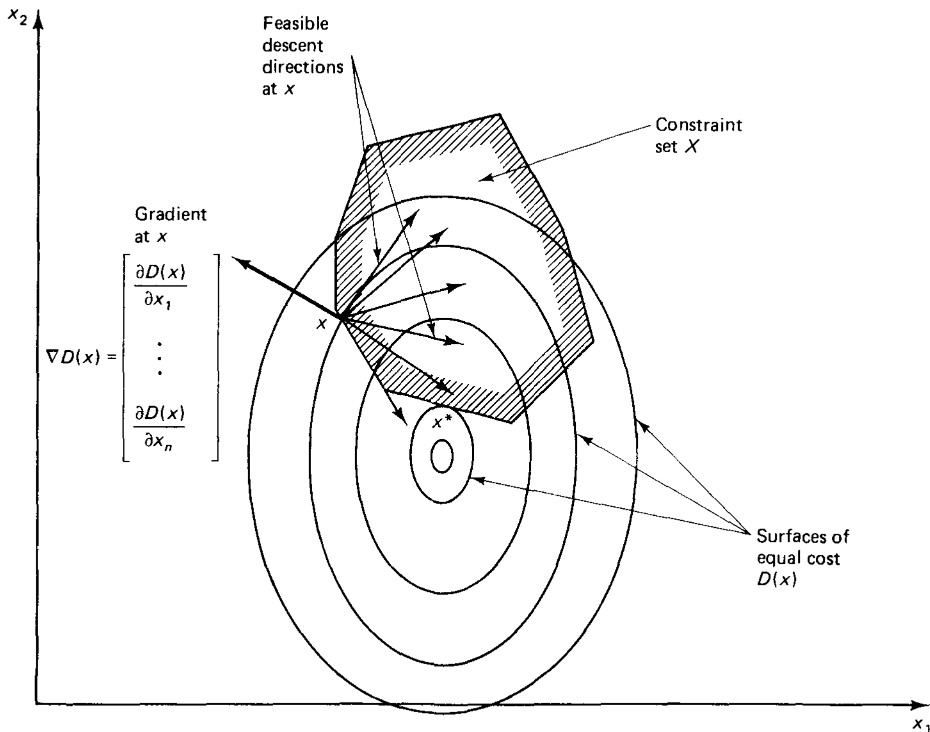


Figure 5.64 Descent directions at a feasible point x form an angle greater than 90° with the gradient $\nabla D(x)$. At the optimal point x^*, there are no feasible descent directions.

$$\Delta x_p \leq 0, \quad \text{for all paths } p \text{ that are nonshortest in the sense } \partial D(x)/\partial x_p > \partial D(x)/\partial x_{\overline{p}} \text{ for some path } \overline{p} \text{ of the same OD pair} \tag{5.60}$$

$$\Delta x_p < 0, \quad \text{for at least one nonshortest path } p.$$

In words, the conditions above together with the condition $\sum_{p \in P_w} \Delta x_p = 0$ state that some positive flow is shifted from nonshortest paths to the shortest paths [with respect to the lengths $D'_{ij}(F_{ij})$], and no flow is shifted from shortest paths to nonshortest ones. Since $\partial D(x)/\partial x_p$ is minimal for those (shortest) paths $\overline{p}$ for which $\Delta x_{\overline{p}} > 0$, it is seen that these conditions imply the descent condition (5.59).

We thus obtain a broad class of iterative algorithms for solving the optimal routing problem. Their basic iteration is given by

$$x := x + \alpha \Delta x$$

where Δx is a feasible descent direction [*i.e.*, satisfies conditions (5.57) to (5.59) or (5.57), (5.58), and (5.60)], and α is a positive stepsize chosen so that the cost function is decreased, that is,

$$D(x + \alpha \Delta x) < D(x)$$

and the vector $x + \alpha \Delta x$ is feasible. The stepsize α could be different in each iteration. It can be seen that a feasible descent direction can be found if x violates the shortest path optimality condition (5.53) of section 5.5. Figure 5.65 illustrates the operation of such an algorithm.

5.6.1 The Frank–Wolfe (Flow Deviation) Method

Here is one way to implement the philosophy of incremental changes along feasible descent directions:

Given a feasible path flow vector $x = \{x_p\}$, find a minimum first derivative length (MFDL) path for each OD pair. (The first derivatives D'_{ij} are evaluated, of course, at the current vector x.) Let $\overline{x} = \{\overline{x}_p\}$ be the vector of path flows that would result if all input r_w for each OD pair $w \in W$ is routed along the corresponding MFDL path. Let α^* be the stepsize that minimizes $D[x + \alpha(\overline{x} - x)]$ over all $\alpha \in [0, 1]$, that is,

$$D\big[x + \alpha^*(\overline{x} - x)\big] = \min_{\alpha \in [0,1]} D\big[x + \alpha(\overline{x} - x)\big] \tag{5.61}$$

The new set of path flows is obtained by

$$x_p := x_p + \alpha^*(\overline{x}_p - x_p), \qquad \text{for all } p \in P_w, w \in W \tag{5.62}$$

and the process is repeated.

The algorithm above is a special case of the so-called *Frank–Wolfe method* for solving general, nonlinear programming problems with convex constraint sets (see [Zan69]). It has been called the *flow deviation method* (see [FGK73]), and can be shown to reduce the value of the cost function to its minimum in the limit (see Problem 5.31) although its convergence rate near the optimum tends to be very slow.

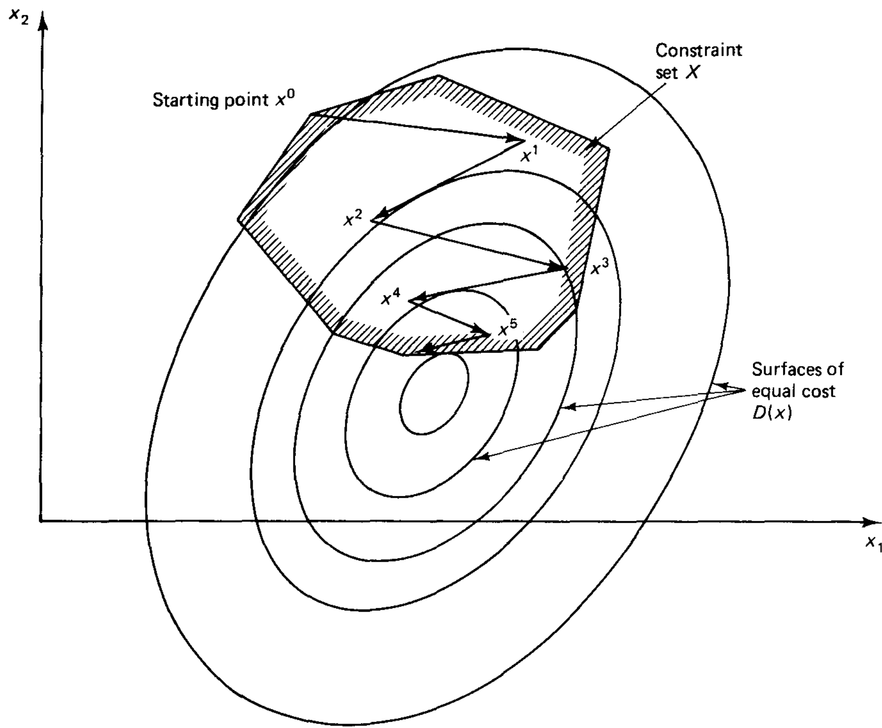


Figure 5.65 Sample path of an iterative descent method based on feasible descent directions. At each iteration, a feasible point of lower cost is obtained.

Note that the feasible direction used in the Frank–Wolfe iteration (5.62) is of the form (5.60); that is, a proportion α^* of the flow of all the nonshortest paths (those paths for which $x_p = 0$) is shifted to the shortest path (the one for which $\overline{x}_p = r_w$) for each OD pair w. The characteristic property here is that flow is shifted from the nonshortest paths in *equal* proportions. This distinguishes the Frank–Wolfe method from the gradient projection methods discussed in the next section. The latter methods also shift flow from the nonshortest paths to the shortest paths; however, they do so in generally unequal proportions.

Here is a simple example illustrating the Frank–Wolfe method:

Example 5.8

Consider the three-link network with one origin and one destination, shown in Fig. 5.66. There are three paths with corresponding flows x_1, x_2, and x_3, which must satisfy the constraints

$$x_1 + x_2 + x_3 = 1, \quad x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0$$

The cost function is

$$D(x) = \frac{1}{2}\left(x_1^2 + x_2^2 + 0.1x_3^2\right) + 0.55x_3$$

(The linear term $0.55x_3$ can be attributed to a large processing and propagation delay on link 3.)

This is an easy problem that can be solved analytically. It can be argued (or shown using the optimality condition of Section 5.5) that at an optimal solution $x^* = (x_1^*, x_2^*, x_3^*)$, we must have by symmetry $x_1^* = x_2^*$, so there are two possibilities:

(a) $x_3^* = 0$ and $x_1^* = x_2^* = 1/2$.

(b) $x_3^* = \beta > 0$ and $x_1^* = x_2^* = (1-\beta)/2$.

Case (b) is not possible because according to the optimality condition of Section 5.5, if $x_3^* > 0$, the length of path 3 [$= \partial D(x^*)/\partial x_3 = 0.1\beta + 0.55$] must be less or equal to the lengths of paths 1 and 2 [$= \partial D(x^*)/\partial x_1 = (1-\beta)/2$], which is clearly false. Therefore, the optimal solution is $x^* = (1/2, 1/2, 0)$.

Consider now application of the Frank–Wolfe iteration (5.68) at a feasible path flow vector $x = (x_1, x_2, x_3)$. The three paths have first derivative lengths

$$\frac{\partial D(x)}{\partial x_1} = x_1, \quad \frac{\partial D(x)}{\partial x_2} = x_2, \quad \frac{\partial D(x)}{\partial x_3} = 0.1x_3 + 0.55$$

It is seen, using essentially the same argument as above, that the shortest path is either 1 or 2, depending on whether $x_1 \leq x_2$ or $x_2 < x_1$, and the corresponding shortest-path flows are $\overline{x} = (1,0,0)$ and $\overline{x} = (0,1,0)$, respectively. (The tie is broken arbitrarily in favor of path 1 in case $x_1 = x_2$.) Therefore, the Frank–Wolfe iteration takes the form

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} := \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \alpha^* \begin{bmatrix} 1 - x_1 \\ -x_2 \\ -x_3 \end{bmatrix} = \begin{bmatrix} x_1 + \alpha^*(x_2 + x_3) \\ (1-\alpha^*)x_2 \\ (1-\alpha^*)x_3 \end{bmatrix}, \quad \text{if } x_1 \leq x_2$$

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} := \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \alpha^* \begin{bmatrix} -x_1 \\ 1 - x_2 \\ -x_3 \end{bmatrix} = \begin{bmatrix} (1-\alpha^*)x_1 \\ x_2 + \alpha^*(x_1 + x_3) \\ (1-\alpha^*)x_3 \end{bmatrix}, \quad \text{if } x_2 < x_1$$

Thus, at each iteration, a proportion α^* of the flows of the nonshortest paths is shifted to the shortest path.

The stepsize α^* is obtained by line minimization over [0,1] [cf. Eq. (5.67)]. Because $D(x)$ is quadratic, this minimization can be done analytically. We have

$$D\big[x + \alpha(\overline{x} - x)\big] = \frac{1}{2}\Big(\big[x_1 + \alpha(\overline{x}_1 - x_1)\big]^2 + \big[x_2 + \alpha(\overline{x}_2 - x_2)\big]^2$$
$$+0.1\big[x_3 + \alpha(\overline{x}_3 - x_3)\big]^2\Big) + 0.55\big[x_3 + \alpha(\overline{x}_3 - x_3)\big]$$

Differentiating with respect to α and setting the derivative to zero we obtain the unconstrained minimum $\overline{\alpha}$ of $D\big[x + \alpha(\overline{x} - x)\big]$ over α. This is given by

$$\overline{\alpha} = -\frac{x_1(\overline{x}_1 - x_1) + x_2(\overline{x}_2 - x_2) + (0.1x_3 + 0.55)(\overline{x}_3 - x_3)}{(\overline{x}_1 - x_1)^2 + (\overline{x}_2 - x_2)^2 + 0.1(\overline{x}_3 - x_3)^2}$$

where $\overline{x} = (\overline{x}_1, \overline{x}_2, \overline{x}_3)$ is the current shortest path flow vector [(1,0,0) or (0,1,0), depending on whether $x_1 \leq x_2$ or $x_2 < x_1$]. Since $(\overline{x} - x)$ is a descent direction, we have $\overline{\alpha} \geq 0$. Therefore, the stepsize α^*, which is the constrained minimum over [0,1] [cf. Eq. (5.67)], is given by

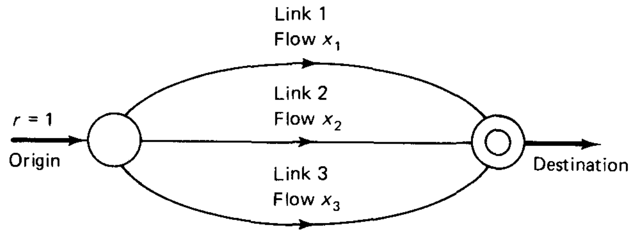
$$\alpha^* = \min[1, \overline{\alpha}]$$

(In this example, it can be verified that $\overline{\alpha} < 1$, implying that $\alpha^* = \overline{\alpha}$.)

Figures 5.66 and 5.67 show successive iterates of the method. It can be seen that the rate of convergence becomes slow near the optimal solution. The reason is that as x^* is approached, the directions of search tend to become orthogonal to the direction leading from the current iterate x^k to x^*. In fact, it can be seen from Fig. 5.66 that the ratio of successive cost errors

$$\frac{D(x^{k+1}) - D(x^*)}{D(x^k) - D(x^*)}$$

while always less than 1, actually converges to 1 as $k \to \infty$. This is known as *sublinear* convergence rate [Ber82d], and is typical of the Frank–Wolfe method ([CaC68] and [Dun79]).



Iteration # k	0	10	20	40	80	160	320
x_1	0.4	0.4593	0.4702	0.4795	0.4866	0.4917	0.4950
x_2	0.3	0.4345	0.4562	0.4717	0.4823	0.4893	0.4938
x_3	0.3	0.1061	0.0735	0.0490	0.0310	0.0189	0.0110
Cost	0.2945	0.2588	0.2553	0.2532	0.2518	0.2510	0.2506
Error Ratio $\frac{D(x^{k+1}) - D(x^*)}{D(x^k) - D(x^*)}$	0.7164	0.9231	0.9576	0.9774	0.9882	0.9939	0.9969

Figure 5.66 Example problem and successive iterates of the Frank–Wolfe method. The cost function is

$$D(x) = \frac{1}{2}\left(x_1^2 + x_2^2 + x_3^2\right) + 0.55x_3$$

The optimal solution is $x^* = \left(\frac{1}{2}, \frac{1}{2}, 0\right)$. As the optimal solution is approached, the method becomes slower. In the limit, the ratio of successive errors $D(x^{k+1}) - D(x^*)/D(x^k) - D(x^*)$ tends to unity.

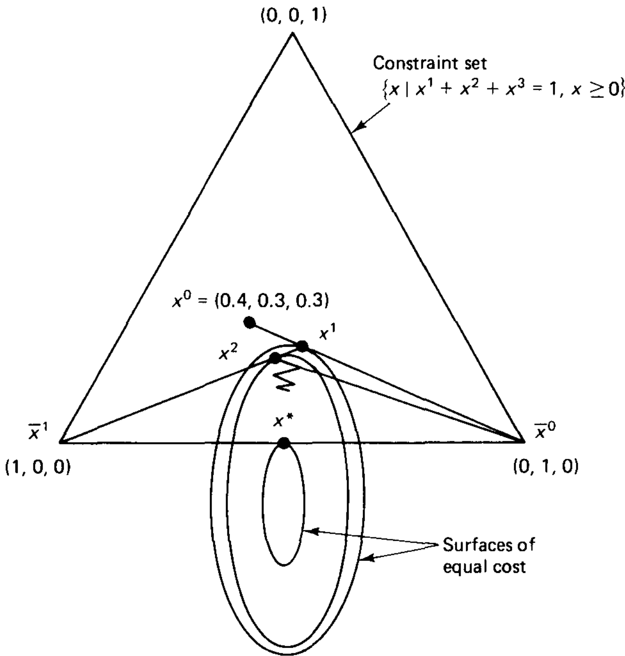


Figure 5.67 Successive iterates of the Frank–Wolfe method in the two-node, three-link example. Slow convergence is due to the fact that as the optimal solution x^* is approached, the directions of search tend to become orthogonal to the direction leading to the optimum.

The determination of an optimal stepsize α^* satisfying Eq. (5.61) requires a one-dimensional minimization over [0,1] which can be carried out through any one of several existing methods (see [Lue84]). A simpler method is to choose the stepsize α^* by means of

$$\alpha^* = \min \left[1, -\frac{\sum_{(i,j)} (\overline{F}_{ij} - F_{ij}) D'_{ij}}{\sum_{(i,j)} (\overline{F}_{ij} - F_{ij})^2 D''_{ij}} \right] \tag{5.63}$$

Here $\{F_{ij}\}$ and $\{\overline{F}_{ij}\}$ are the sets of total link flows corresponding to $\{x_p\}$ and $\{\overline{x}_p\}$, respectively [*i.e.*, F_{ij} (or $\overline{F}_{ij}$) is obtained by adding all flows x_p (or $\overline{x}_p$) of paths p traversing the link (i, j)]. The first and second derivatives, D'_{ij} and D''_{ij}, respectively, are evaluated at F_{ij}. The formula for α^* in Eq. (5.63) is obtained by making a second-order Taylor approximation $\tilde{G}(\alpha)$ of $G(\alpha) = D\big[x + \alpha(\overline{x} - x)\big]$ around $\alpha = 0$:

$$\tilde{G}(\alpha) = \sum_{(i,j)} \left\{ D_{ij}(F_{ij}) + \alpha D'_{ij}(F_{ij})(\overline{F}_{ij} - F_{ij}) + \frac{\alpha^2}{2} D''_{ij}(F_{ij})(\overline{F}_{ij} - F_{ij})^2 \right\}$$

and by minimizing $\tilde{G}(\alpha)$ with respect to α over the interval [0,1] as in the earlier example.

It can be shown that the algorithm with the choice of Eq. (5.63) for the stepsize converges to the optimal set of total link flows provided that the starting set of total link flows is sufficiently close to the optimal. For the type of cost functions used in routing problems [*e.g.*, the $M/M/1$ delay formula; cf. Eq. (5.30) in Section 5.4], it appears that the simple stepsize choice of Eq. (5.63) typically leads to convergence even when the starting total link flows are far from optimal. This stepsize choice is also well suited for distributed implementation. However, even with this rule, the method appears inferior for distributed

application to the projection method of the next section primarily because it is essential for the network nodes to synchronize their calculations in order to obtain the stepsize.

The Frank–Wolfe method has an insightful geometrical interpretation. Suppose that we have a feasible path flow vector $x = \{x_p\}$, and we try to find a path flow variation $\Delta x = \{\Delta x_p\}$ which is feasible in the sense of Eqs. (5.57) and (5.58), that is,

$$\sum_{p \in P_w} \Delta x_p = 0, \qquad \text{for all } w \in W$$
$$x_p + \Delta x_p \geq 0, \qquad \text{for all } p \in P_w, w \in W \tag{5.64}$$

and along which the initial rate of change [cf. Eq. (5.59)]

$$\sum_{w \in W} \sum_{p \in P_w} \frac{\partial D(x)}{\partial x_p} \Delta x_p$$

of the cost function D is most negative. Such a variation Δx is obtained by solving the optimization problem

$$\text{minimize} \sum_{w \in W} \sum_{p \in P_w} \frac{\partial D(x)}{\partial x_p} \Delta x_p$$
$$\text{subject to} \quad \sum_{p \in P_w} \Delta x_p = 0, \qquad \text{for all } w \in W \tag{5.64}$$
$$x_p + \Delta x_p \geq 0, \qquad \text{for all } p \in P_w, w \in W$$

and it is seen that $\Delta x = \overline{x} - x$ is an optimal solution, where $\overline{x}$ is the shortest path flow vector generated by the Frank–Wolfe method. The process of finding Δx can be visualized as in Fig. 5.68. It can be seen that $\overline{x}$ is a vertex of the polyhedron of feasible path flow vectors that lies farthest out along the negative gradient direction $-\nabla D(x)$, thereby minimizing the inner product of $\nabla D(x)$ and Δx subject to the constraints (5.64).

Successive iterations of the Frank–Wolfe method are shown in Fig. 5.69. At each iteration, a vertex of the feasible set (*i.e.*, the shortest path flow $\overline{x}$) is obtained by solving a shortest path problem; then the next path flow vector is obtained by a search along the line joining the current path flow vector with the vertex. The search ensures that the new path flow vector has lower cost than the preceding one.

We finally mention an important situation where the Frank–Wolfe method has an advantage over other methods. Suppose that one is not interested in obtaining optimal path flows, but that the only quantities of interest are the optimal total link flows F_{ij} or just the value of optimal cost. (This arises in topological design studies; see Section 5.4.2.) Then the Frank–Wolfe method can be implemented in a way that only the current total link flows together with the current shortest paths for all OD pairs are maintained in memory at each iteration. This can be done by computing the total link flows corresponding to the shortest path flow vector $\overline{x}$ and executing the stepsize minimization indicated in Eq. (5.61) in the space of total link flows. The amount of storage required in this implementation is relatively small, thereby allowing the solution of very large network problems. If, however, one is interested in optimal path flows as well as total

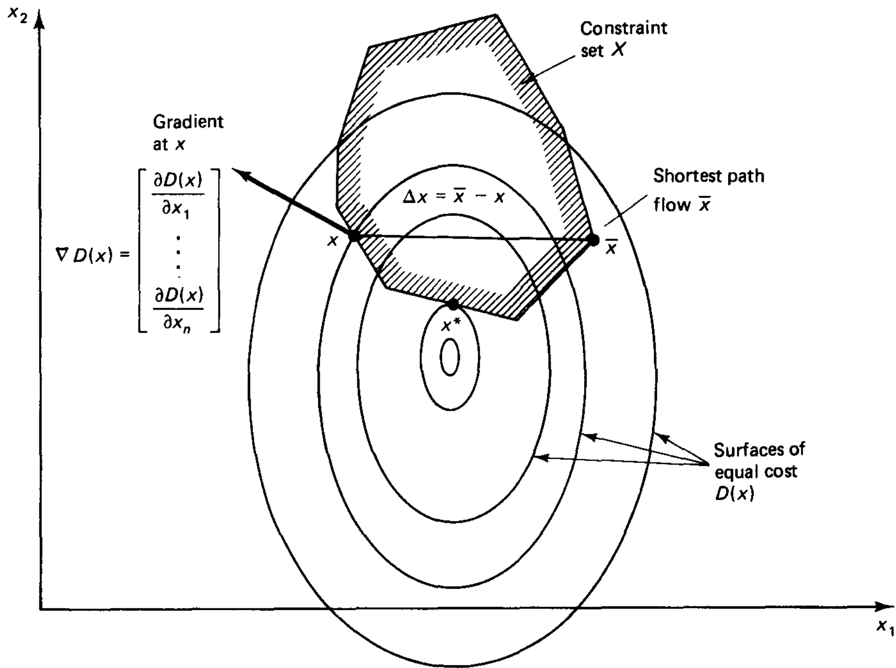


Figure 5.68 Finding the feasible descent direction $\Delta x = \bar{x} - x$ at a point x in the Frank–Wolfe method. $\bar{x}$ is the shortest path flow (or extreme point of the feasible set X) that lies farthest out along the negative gradient direction $-\nabla D(x)$.

link flows, the storage requirements of the Frank–Wolfe method are about the same as those of other methods, including the projection methods discussed in the next section.

5.7 PROJECTION METHODS FOR OPTIMAL ROUTING

We now consider a class of feasible direction algorithms for optimal routing that are faster than the Frank–Wolfe method and lend themselves more readily to distributed implementation. These methods are also based on shortest paths and determine a minimum first derivative length (MFDL) path for every OD pair at each iteration. An increment of flow change is calculated for each path on the basis of the relative magnitudes of the path lengths and, sometimes, the second derivatives of the cost function. If the increment is so large that the path flow becomes negative, the path flow is simply set to zero (*i.e.*, it is "projected" back onto the positive orthant). These routing algorithms may be viewed as constrained versions of common, unconstrained optimization methods, such as steepest descent and Newton's method, which are given in nonlinear programming texts (*e.g.*, [Zan69], [Pol71], [Ber82d], and [Lue84]). We first present these methods briefly in a general, nonlinear optimization setting, and subsequently specialize them to the routing problem.

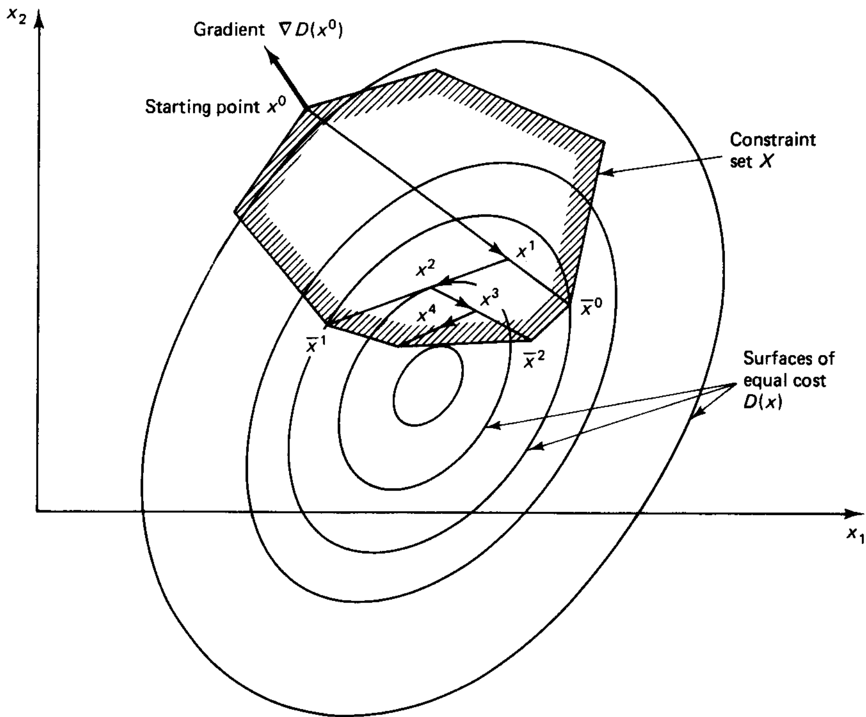


Figure 5.69 Successive iterations of the Frank–Wolfe method. At each iteration, a vertex of the feasible set (a shortest path flow) is found. The next path flow vector is obtained by a search on the line segment connecting the current path flow vector with the vertex.

5.7.1 Unconstrained Nonlinear Optimization

Let f be a twice differentiable function of the n-dimensional vector $x = (x_1, \ldots, x_n)$, with a gradient and Hessian matrix at any x denoted $\nabla f(x)$ and $\nabla^2 f(x)$:

$$\nabla f(x) = \begin{bmatrix} \dfrac{\partial f(x)}{\partial x_1} \\ \vdots \\ \dfrac{\partial f(x)}{\partial x_n} \end{bmatrix}, \qquad \nabla^2 f(x) = \begin{bmatrix} \dfrac{\partial^2 f(x)}{(\partial x_1)^2} & \cdots & \dfrac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \dfrac{\partial^2 f(x)}{\partial x_n \partial x_1} & \cdots & \dfrac{\partial^2 f(x)}{(\partial x_n)^2} \end{bmatrix}$$

We assume that for all x, $\nabla^2 f(x)$ is a positive semidefinite matrix that depends continuously on x.* The *method of steepest descent* for finding an unconstrained minimum of

*A symmetric $n \times n$ matrix A with elements A_{ij} is said to be *positive semidefinite* if the quadratic form

$$\sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} z_i z_j$$

is nonnegative for all vectors $z = (z_1, \ldots, z_n)$. The matrix A is said to be *positive definite* if this quadratic form is positive for all vectors $z \neq 0$.

f starts with some initial guess x^0 and proceeds according to the iteration

$$x^{k+1} = x^k - \alpha^k \nabla f(x^k), \qquad k = 0, 1, \ldots \tag{5.65}$$

where α^k is a positive scalar stepsize, determined according to some rule. The idea here is similar to the one discussed in the preceding section, namely changing x^k along a descent direction. In the preceding iteration, the change is made along the negative gradient, which is a descent direction since it makes a negative inner product with the gradient vector [unless $\nabla f(x^k) = 0$, in which case the vector x^k is optimal]. Common choices for α^k are the minimizing stepsize determined by

$$f\left[x^k - \alpha^k \nabla f(x^k)\right] = \min_{\alpha > 0} f\left[x^k - \alpha \nabla f(x^k)\right] \tag{5.66}$$

and a constant positive stepsize $\overline{\alpha}$

$$\alpha^k \equiv \overline{\alpha}, \qquad \text{for all } k \tag{5.67}$$

There are a number of convergence results for steepest descent methods. For example, if f has a unique unconstrained minimizing point, it may be shown that the sequence $\{x^k\}$ generated by the steepest descent method of Eqs. (5.65) and (5.66) converges to this minimizing point for every starting x^0 (see [Ber82d] and [Lue84]). Also, given any starting vector x^0, the sequence generated by this steepest descent method converges to the minimizing point provided that $\overline{\alpha}$ is chosen sufficiently small. Unfortunately, however, the speed of convergence of $\{x^k\}$ can be quite slow. It can be shown ([Lue84], p. 218) that for the case of the line minimization rule of Eq. (5.66), if f is a positive definite quadratic function, there holds

$$\frac{f(x^{k+1}) - f^*}{f(x^k) - f^*} \le \left(\frac{M - m}{M + m}\right)^2 \tag{5.68}$$

where $f^* = \min_x f(x)$, and M, m are the largest and smallest eigenvalues of $\nabla^2 f(x)$, respectively. Furthermore, there exist starting points x^0 such that Eq. (5.68) holds with equality for every k. So if the ratio M/m is large (this corresponds to the equal-cost surfaces of f being very elongated ellipses), the rate of convergence is slow. Similar results can be shown for the steepest descent method with the constant stepsize (5.67), and these results also hold in a qualitatively similar form for functions f with an everywhere continuous and positive definite Hessian matrix.

The rate of convergence of the steepest descent method can be improved by premultiplying the gradient by a suitable positive definite scaling matrix B^k, thereby obtaining the iteration

$$x^{k+1} = x^k - \alpha^k B^k \nabla f(x^k), \qquad k = 0, 1, \ldots \tag{5.69}$$

This method is also based on the idea of changing x^k along a descent direction. [The direction of change $-B^k \nabla f(x^k)$ makes a negative inner product with $\nabla f(x^k)$, since B^k is a positive definite matrix.] From the point of view of rate of convergence, the best

method is obtained with the choice

$$B^k = \left[\nabla^2 f(x^k)\right]^{-1} \tag{5.70}$$

[assuming that $\nabla^2 f(x^k)$ is invertible]. This is *Newton's method*, which can be shown to have a very fast (superlinear) rate of convergence near the minimizing point when the stepsize α^k is taken as unity. Unfortunately, this excellent convergence rate is achieved at the expense of the potentially substantial overhead associated with the inversion of $\nabla^2 f(x^k)$. It is often useful to consider other choices of B^k which approximate the "optimal" choice $[\nabla^2 f(x^k)]^{-1}$ but do not require as much overhead. A simple choice that often works well is to take B^k as a diagonal approximation to the inverse Hessian, that is,

$$B^k = \begin{bmatrix} \left[\dfrac{\partial^2 f(x^k)}{(\partial x_1)^2}\right]^{-1} & 0 & \cdots & 0 \\ 0 & \left[\dfrac{\partial^2 f(x^k)}{(\partial x_2)^2}\right]^{-1} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \left[\dfrac{\partial^2 f(x^k)}{(\partial x_n)^2}\right]^{-1} \end{bmatrix} \tag{5.71}$$

With this choice, the scaled gradient iteration (5.69) can be written in the simple form

$$x_i^{k+1} = x_i^k - \alpha^k \left[\frac{\partial^2 f(x^k)}{(\partial x_i)^2}\right]^{-1} \frac{\partial f(x^k)}{\partial x_i}, \quad i = 1, \ldots, n \tag{5.72}$$

5.7.2 Nonlinear Optimization over the Positive Orthant

Next consider the problem of minimizing the function f subject to the nonnegativity constraints $x_i \geq 0$, for $i = 1, \ldots, n$, that is, the problem

$$\begin{aligned} &\text{minimize } f(x) \\ &\text{subject to } x \geq 0 \end{aligned} \tag{5.73}$$

A straightforward analog of the steepest descent method known as the *gradient projection method* ([Gol64] and [LeP65]), is given by

$$x^{k+1} = \left[x^k - \alpha^k \nabla f(x^k)\right]^+, \qquad k = 0, 1, \ldots \tag{5.74}$$

where for any vector z, we denote by $[z]^+$ the projection of z onto the positive orthant

$$[z]^+ = \begin{bmatrix} \max\{0, z_1\} \\ \max\{0, z_2\} \\ \vdots \\ \max\{0, z_n\} \end{bmatrix} \tag{5.75}$$

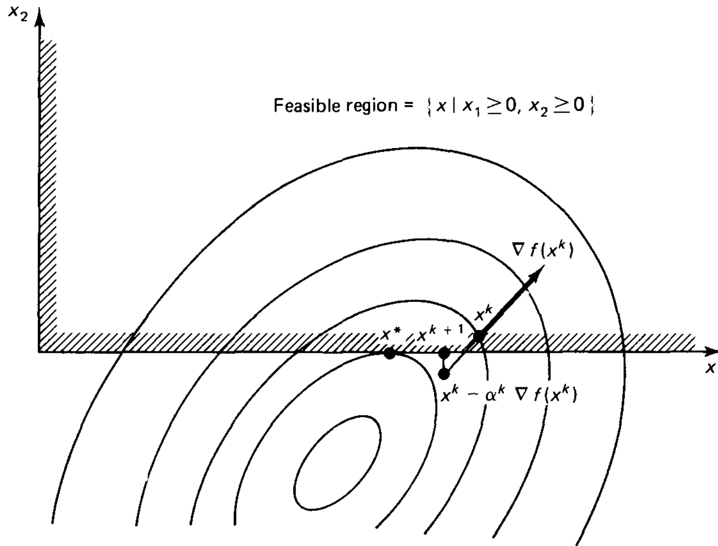


Figure 5.70 Operation of the gradient projection method. A step is made in the direction of the negative gradient, and the result is orthogonally projected on the positive orthant. It can be shown that the step $(x^{k+1} - x^k)$ is a descent direction [makes a negative inner product with $\nabla f(x^k)$], and if the stepsize α is sufficiently small, the iteration improves the cost [*i.e.*, $f(x^{k+1}) < f(x^k)$].

An illustration of how this method operates is given in Fig. 5.70. It can be shown that the convergence results mentioned earlier in connection with the unconstrained steepest descent method of Eq. (5.65) also hold true for its constrained analog of Eq. (5.74) (see Problem 5.32 and [Ber76]). The same is true for the method

$$x^{k+1} = \left[x^k - \alpha^k B^k \nabla f(x^k)\right]^+$$

where B^k is a *diagonal*, positive definite scaling matrix (see Problem 5.33). When B^k is given by the diagonal inverse Hessian approximation of Eq. (5.71), the following iteration is obtained:

$$x_i^{k+1} = \max\left\{0, x_i^k - \alpha^k \left[\frac{\partial^2 f(x^k)}{(\partial x_i)^2}\right]^{-1} \frac{\partial f(x^k)}{\partial x_i}\right\}, \qquad i = 1, \ldots, n \qquad (5.76)$$

5.7.3 Application to Optimal Routing

Consider now the optimal routing problem

$$\text{minimize } D(x) \stackrel{\Delta}{=} \sum_{(i,j)} D_{ij}(F_{ij})$$
$$\text{subject to } \sum_{p \in P_w} x_p = r_w, \quad x_p \geq 0, \qquad \text{for all } p \in P_w, w \in W \qquad (5.77)$$

where each total link flow F_{ij} is expressed in terms of the path flow vector $x = \{x_p\}$ as the sum of path flows traversing the link (i, j). Assume that the second derivatives of D_{ij}, denoted by $D''_{ij}(F_{ij})$, are positive for all F_{ij}. Let $x^k = \{x_p^k\}$ be the path flow vector obtained after k iterations, and let $\{F_{ij}^k\}$ be the corresponding set of total link flows. For each OD pair w, let $\overline{p}_w$ be an MFDL path [with respect to the current link lengths $D'_{ij}(F_{ij}^k)$].

The optimal routing problem (5.77) can be converted (for the purpose of the next iteration) to a problem involving only positivity constraints by expressing the flows of the MFDL paths $\overline{p}_w$ in terms of the other path flows, while eliminating the equality constraints

$$\sum_{p \in P_w} x_p = r_w$$

in the process. For each w, $x_{\overline{p}_w}$ is substituted in the cost function $D(x)$ using the equation

$$x_{\overline{p}_w} = r_w - \sum_{\substack{p \in P_w \\ p \neq \overline{p}_w}} x_p \tag{5.78}$$

thereby obtaining a problem of the form

$$\begin{aligned} &\text{minimize } \tilde{D}(\tilde{x}) \\ &\text{subject to} \quad x_p \geq 0, \qquad \text{for all } w \in W,\ p \in P_w,\ p \neq \overline{p}_w \end{aligned} \tag{5.79}$$

where $\tilde{x}$ is the vector of all path flows which are *not* MFDL paths.

We now calculate the derivatives that will be needed to apply the scaled projection iteration (5.76) to the problem of Eq. (5.79). Using Eq. (5.78) and the definition of $\tilde{D}(\tilde{x})$, we obtain

$$\frac{\partial \tilde{D}(\tilde{x}^k)}{\partial x_p} = \frac{\partial D(x^k)}{\partial x_p} - \frac{\partial D(x^k)}{\partial x_{\overline{p}_w}}, \qquad \text{for all } p \in P_w,\ p \neq \overline{p}_w \tag{5.80}$$

for all $w \in W$. In Section 5.5 it was shown that $\partial D(x)/\partial x_p$ is the first derivative length of path p, that is,

$$\frac{\partial D(x^k)}{\partial x_p} = \sum_{\substack{\text{all links} \\ (i,j) \text{ on path } p}} D'_{ij}(F_{ij}^k), \tag{5.81}$$

Regarding second derivatives, a straightforward differentiation of the first derivative expressions (5.80) and (5.81) shows that

$$\frac{\partial^2 \tilde{D}(\tilde{x}^k)}{(\partial x_p)^2} = \sum_{(i,j) \in L_p} D''_{ij}(F_{ij}^k), \qquad \text{for all } w \in W,\ p \in P_w,\ p \neq \overline{p}_w \tag{5.82}$$

where, for each p,

L_p = Set of links belonging to either p, or the corresponding MFDL path $\bar{p}_w$, but not both

Expressions for both the first and second derivatives of the "reduced" cost $\tilde{D}(\tilde{x})$, are now available and thus the scaled projection method of Eq. (5.76) can be applied. The iteration takes the form [cf. Eqs. (5.76), (5.80), and (5.82)]

$$x_p^{k+1} = \max\{0, x_p^k - \alpha^k H_p^{-1}(d_p - d_{\overline{p}_w})\}, \qquad \text{for all } w \in W,\ p \in P_w,\ p \neq \overline{p}_w \quad (5.83)$$

where d_p and $d_{\overline{p}_w}$ are the first derivative lengths of the paths p and $\overline{p}_w$ given by [cf. Eq. (5.81)]

$$d_p = \sum_{\substack{\text{all links } (i,j) \\ \text{on path } p}} D'_{ij}(F_{ij}^k), \qquad d_{\overline{p}_w} = \sum_{\substack{\text{all links } (i,j) \\ \text{on path } \overline{p}_w}} D'_{ij}(F_{ij}^k), \quad (5.84)$$

and H_p is the "second derivative length"

$$H_p = \sum_{(i,j)\in L_p} D''_{ij}(F_{ij}^k) \quad (5.85)$$

given by Eq. (5.82). The stepsize α^k is some positive scalar which may be chosen by a variety of methods. For example, α^k can be constant or can be chosen by some form of line minimization. Stepsize selection will be discussed later. We refer to the iteration of Eqs. (5.83) to (5.85) as the *projection algorithm*.

The following observations can be made regarding the projection algorithm:

1. Since $d_p \geq d_{\overline{p}_w}$ for all $p \neq \overline{p}_w$, all the nonshortest path flows that are positive will be reduced with the corresponding increment of flow being shifted to the MFDL path $\overline{p}_w$. If the stepsize α^k is large enough, all flow from the nonshortest paths will be shifted to the shortest path. Therefore, the projection algorithm may also be viewed as a generalization of the adaptive routing method based on shortest paths of Section 5.2.5, with α^k, H_p, and $(d_p - d_{\overline{p}_w})$ determining the amounts of flow shifted to the shortest path.
2. Those nonshortest path flows x_p, $p \neq \overline{p}_w$ that are zero will stay at zero. Therefore, the path flow iteration of Eq. (5.83) should only be carried out for paths that carry positive flow.
3. Only paths that carried positive flow at the starting flow pattern or were MFDL paths at some previous iteration can carry positive flow at the beginning of an iteration. This is important since it tends to keep the number of flow carrying paths small, with a corresponding reduction in the amount of calculation and bookkeeping needed at each iteration.

There are several ways to choose the stepsize α^k. One possibility is to keep α^k constant ($\alpha^k \equiv \alpha$, for all k). With this choice it can be shown that given any starting set of path flows, there exists $\overline{\alpha} > 0$ such that if $\alpha \in (0, \overline{\alpha}]$, a sequence generated by the projection algorithm converges to the optimal cost of the problem (see Problem 5.32). A crucial question has to do with the magnitude of the constant stepsize. It is known from

nonlinear programming experience and analysis that a stepsize equal to 1 usually works well with Newton's method as well as diagonal approximations to Newton's method [cf. Eq. (5.71)] that employ scaling based on second derivatives ([Lue84] and [Ber82d]).

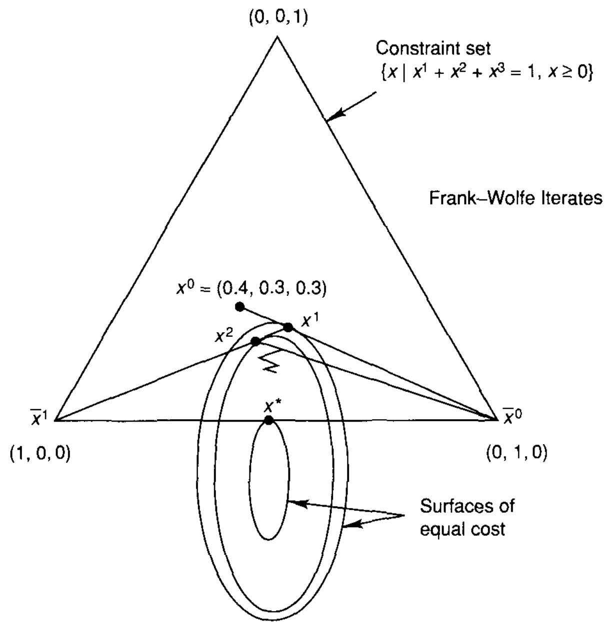
Experience has verified that choosing α^k in the path flow iteration (5.83) close to 1 typically works quite well regardless of the values of the input flows r_w [BGV79]. Even better performance is usually obtained if the iteration is carried out *one OD pair (or one origin) at a time*; that is, first carry out the iteration with $\alpha^k = 1$ for a single OD pair (or origin), adjust the corresponding total link flows to account for the effected change in the path flows of this OD pair (or origin), and continue with the next OD pair until all OD pairs are taken up cyclically. The rationale for this is that dropping the off-diagonal terms of the Hessian matrix [cf. Eqs. (5.69) and (5.71)], in effect, neglects the interaction between the flows of different OD pairs. In other words, the path flow iteration (5.83) is based to some extent on the premise that each OD pair will adjust its own path flows while the other OD pairs will keep theirs unchanged. Carrying out this iteration one OD pair at a time reduces the potentially detrimental effect of the neglected off-diagonal terms and increases the likelihood that the unity stepsize is appropriate and effective.

Using a constant stepsize is well suited for distributed implementation. For centralized computation, it is possible to choose α^k by a simple form of line search. For example, start with a unity stepsize, evaluate the corresponding cost, and if no reduction is obtained over $D(x^k)$, successively reduce the stepsize until a cost reduction $D(x^{k+1}) < D(x^k)$ is obtained. (It is noted for the theoretically minded that this scheme cannot be shown to converge to the optimal solution. However, typically, it works well in practice. Similar schemes with better theoretical convergence properties are described in [Ber76] and [Ber82c].) Still another possibility for stepsize selection is discussed in Problem 5.32.

The projection algorithm typically yields rapid convergence to a neighborhood of an optimal solution. Once it comes near a solution (how "near" depends on the problem), it tends to slow down. Its progress is often satisfactory near a solution and usually far better than that of the Frank–Wolfe method. Figure 5.71 provides a comparison of the projection method and the Frank–Wolfe method for the example of the preceding section

To obtain faster convergence near an optimal solution, it is necessary to modify the projection algorithm so that the off-diagonal terms of the Hessian matrix are taken into account. Surprisingly, it is possible to implement sophisticated methods of this type (see [BeG83]), but we will not go into this further. Suffice to say that these methods are based on a more accurate approximation of a constrained version of Newton's method (using the conjugate gradient method) near an optimal solution. However, when far from a solution, their speed of convergence is usually only slightly superior to that of the projection algorithm. So, if one is only interested in getting fast near an optimal solution in few iterations, but the subsequent rate of progress is of little importance (as is often the case in practical routing problems), the projection algorithm is typically satisfactory.

We finally note that the projection algorithm is well suited for distributed implementation. The most straightforward possibility is for all nodes i to broadcast to all other nodes the current total flows F_{ij}^k of their outgoing links (i, j), using, for example, a flooding algorithm or the SPTA of Section 5.3.3. Each node then computes the MFDL



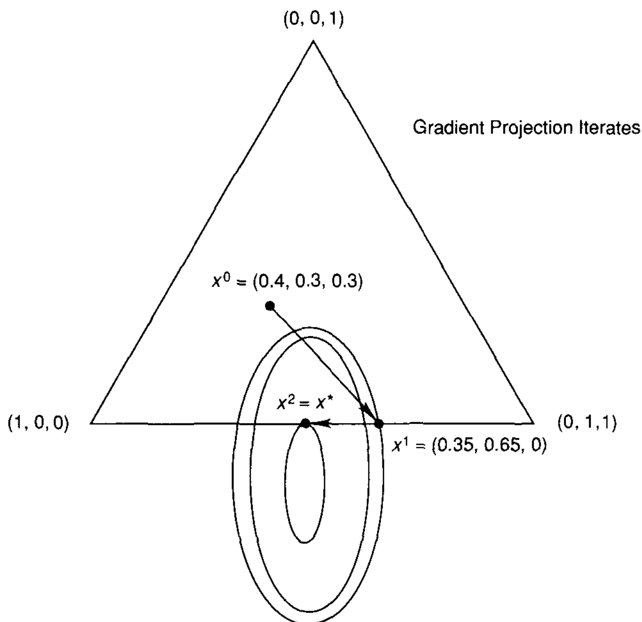


Figure 5.71 Iterates of (a) the Frank–Wolfe and (b) the gradient projection method for the example of the preceding section. The gradient projection method converges much faster. It sets the flow x_3 to the correct value 0 in one iteration and (because the Hessian matrix here is diagonal and the cost is quadratic) requires one more iteration to converge.

paths of OD pairs for which it is the origin and executes the path flow iteration (5.83) for some fixed stepsize. This corresponds to an "all OD pairs at once" mode of implementation. The method can also be implemented in an asynchronous, distributed format, whereby the computation and information reception are not synchronized at each node. The validity of the method under these conditions is shown in [TsB86] and [BeT89].

We close this section with an example illustrating the projection algorithm:

Example 5.9

Consider the network shown in Fig. 5.72. There are only two OD pairs (1,5) and (2,5) with corresponding inputs $r_1 = 4$ and $r_2 = 8$. Consider the following two paths for each OD pair:

Paths of OD pair (1, 5)

$$p_1(1) = \{1,4,5\}$$

$$p_2(1) = \{1,3,4,5\}$$

Paths of OD pair (2, 5)

$$p_1(2) = \{2,4,5\}$$

$$p_2(2) = \{2,3,4,5\}$$

Consider the instance of the routing problem where the link cost functions are all identical and are given by

$$D_{ij}(F_{ij}) = \frac{1}{2}(F_{ij})^2, \qquad \text{for all } (i,j)$$

Consider an initial path flow pattern whereby each OD pair input is routed through the middle link (3,4). This results in a flow distribution given in Tables 5.1 and 5.2.

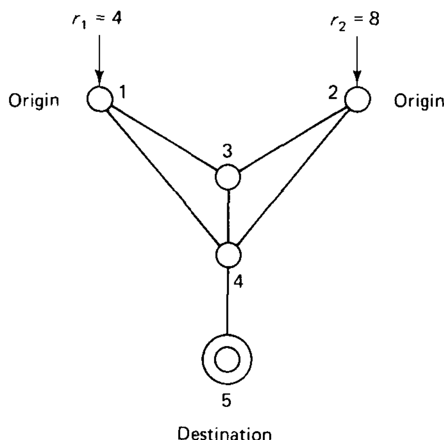


Figure 5.72 The network, for the example, where node 5 is the only destination and there are two origins, nodes 1 and 2.

TABLE 5.1 INITIAL PATH FLOWS FOR THE ROUTING EXAMPLE

OD pair	Path	Path flow
(1,5)	$p_1(1) = \{1, 4, 5\}$	0
	$p_2(1) = \{1, 3, 4, 5\}$	4
(2,5)	$p_1(2) = \{2, 4, 5\}$	0
	$p_2(2) = \{2, 3, 4, 5\}$	8

TABLE 5.2 INITIAL TOTAL LINK FLOWS FOR THE ROUTING EXAMPLE

Link	Total link flow
(1,3)	4
(1,4)	0
(2,3)	8
(2,4)	0
(3,4)	12
(4,5)	12
Others	0

The first derivative length of each link is given by

$$D'_{ij}(F_{ij}) = F_{ij}$$

so the total link flows given in Table 5.2 are also the link lengths for the current iteration. The corresponding first derivative lengths of paths are given in Table 5.3. Therefore, the shortest paths for the current iteration are $p_1(1)$ and $p_1(2)$ for OD pairs (1,5) and (2,5), respectively.

The form of the projection algorithm of Eqs. (5.83) to (5.85) is illustrated for the first OD pair. Here, for the nonshortest path $p = p_2(1)$ and the shortest path $\overline{p} = p_1(1)$ we have $d_p = 28$ and $d_{\overline{p}} = 12$. Also, $H_p = 3$ [each link has a second derivative length $D''_{ij} = 1$, and there are three links that belong to either $p_1(1)$ or $p_2(1)$, but not to both; cf. Eqs. (5.82) and (5.85). Therefore, the path flow iteration (5.83) takes the form

$$x_p := \max\left\{0, 4 - \frac{\alpha^k}{3}(28 - 12)\right\} = \max\left\{0, 4 - \frac{16\alpha^k}{3}\right\} = 4 - \min\left\{4, \frac{16\alpha^k}{3}\right\}$$

and

$$x_{\overline{p}} := r_1 - x_p$$

TABLE 5.3 INITIAL FIRST DERIVATIVE LENGTHS FOR THE ROUTING EXAMPLE

OD pair	Path	First derivative length
(1,5)	$p_1(1) = \{1, 4, 5\}$	12
	$p_2(1) = \{1, 3, 4, 5\}$	28
(2,5)	$p_1(2) = \{2, 4, 5\}$	12
	$p_2(2) = \{2, 3, 4, 5\}$	32

More generally, let $x_1(1), x_2(1), x_1(2)$, and $x_2(2)$ denote the flows along the paths $p_1(1)$, $p_2(1)$, $p_1(2)$, and $p_2(2)$, respectively, at the beginning of an iteration. The corresponding path lengths are

$$d_{p_1(1)} = x_1(1) + r_1 + r_2$$
$$d_{p_2(1)} = 2x_2(1) + x_2(2) + r_1 + r_2$$
$$d_{p_1(2)} = x_1(2) + r_1 + r_2$$
$$d_{p_2(2)} = 2x_2(2) + x_2(1) + r_1 + r_2$$

The second derivative length H_p of Eq. (5.85) equals 3. The projection algorithm (5.83) to (5.85) takes the following form. For $i = 1, 2$,

$$x_1(i) := \begin{cases} x_1(i) - \min\left[x_1(i), \frac{\alpha^k}{3}[d_{p_1(i)} - d_{p_2(i)}]\right], & \text{if } d_{p_1(i)} > d_{p_2(i)} \\ x_1(i) + \min\left[x_2(i), \frac{\alpha^k}{3}[d_{p_2(i)} - d_{p_1(i)}]\right], & \text{otherwise} \end{cases}$$

$$x_2(i) := r_i - x_1(i)$$

Notice that the presence of link (4,5) does not affect the form of the iteration, and indeed this should be so since the total flow of link (4,5) is always equal to $r_1 + r_2$ independent of the routing.

Table 5.4 gives sequences of successive cost function values obtained by the algorithm for different stepsizes and the "all OD pairs at once" and "one OD pair at a time" modes of implementation. The difference between these two implementation modes is that in the all-at-once mode, the OD pairs are processed simultaneously during an iteration using the link and path flows obtained at the end of the preceding iteration. In the one-at-a-time mode the OD pairs are processed sequentially, and following the iteration of one OD pair, the link flows are adjusted to reflect the results of the iteration before carrying out an iteration for the other OD pair. The stepsize is chosen to be constant at one of three possible values ($\alpha^k \equiv 0.5$, $\alpha^k \equiv 1$, and $\alpha^k \equiv 1.8$). It can be seen that for a unity stepsize, the convergence to a neighborhood of a solution is very fast in both the one-at-a-time and the all-at-once modes. As the stepsize is increased, the danger of divergence increases, with divergence typically occurring first for the all-at-once mode. This can be seen from the table, where for $\alpha^k \equiv 1.8$, the algorithm converges (slowly) in the one-at-a-time mode but diverges in the all-at-once mode.

TABLE 5.4 SEQUENCE OF COSTS GENERATED BY THE GRADIENT PROJECTION METHOD FOR THE ROUTING EXAMPLE AND FOR A VARIETY OF STEPSIZES

Iteration	All-at-once mode			One-at-a-time mode		
k	$\alpha^k \equiv 0.5$	$\alpha^k \equiv 1.0$	$\alpha^k \equiv 1.8$	$\alpha^k \equiv 0.5$	$\alpha^k \equiv 1.0$	$\alpha^k \equiv 1.8$
0	184.00	184.00	184.00	184.00	184.00	184.00
1	110.88	104.00	112.00	114.44	101.33	112.00
2	102.39	101.33	118.72	103.54	101.00	109.15
3	101.28	101.03	112.00	101.63		101.79
4	101.09	101.00	118.72	101.29		101.56
5	101.03		112.00	101.08		101.33
6	101.01		118.72	101.03		101.18
7	101.00		112.00	101.01		101.12
8			118.72	101.00		101.09
9			112.00			101.06
10			118.72			101.03

5.8 ROUTING IN THE CODEX NETWORK

In this section the routing system of a network marketed by Codex, Inc., is discussed. References [HuS86] and [HSS86] describe the system in detail. The Codex network uses virtual circuits internally for user traffic and datagrams for system traffic (accounting, control, routing information, etc.). It was decided to use datagrams for system traffic because the alternative, namely establishing a virtual circuit for each pair of nodes, was deemed too expensive in terms of network resources. Note that for each user session, there are two separate virtual circuits carrying traffic in opposite directions, but not necessarily over the same set of links. Routing decisions are done separately in each direction.

There are two algorithms for route selection. The first, used for datagram routing of internal system traffic, is a simple shortest path algorithm of the type discussed in Section 5.2. The shortest path calculations are done as part of the second algorithm, which is used for selecting routes for new virtual circuits and for rerouting old ones. We will focus on the second algorithm, which is more important and far more sophisticated than the first.

The virtual circuit routing algorithm has much in common with the gradient projection method for optimal routing of the preceding section. There is a cost function D_{ij} for each link (i, j) that depends on the link flow, but also on additional parameters, such as the link capacity, the processing and propagation delay, and a priority factor for the virtual circuits currently on the link. Each node monitors the parameters of its adjacent links and broadcasts them periodically to all other nodes. If all virtual circuits have the same priority, the link cost function has the form

$$D_{ij}(F_{ij}) = \frac{F_{ij}}{C_{ij} - F_{ij}} + d_{ij}F_{ij}$$

where F_{ij} is the total data rate on the link, d_{ij} is the processing and propagation delay, and C_{ij} is the link capacity. This formula is based on the $M/M/1$ delay approximation; see the discussion of Sections 3.6 and 5.4. When there are more than one, say M, priority levels for virtual circuits, the link cost function has the form

$$D_{ij}(F_{ij}^1, \ldots, F_{ij}^M) = \left(\sum_{k=1}^{M} p_k F_{ij}^k\right)\left(\frac{1}{C_{ij} - \sum_{k=1}^{M} F_{ij}^k} + d_{ij}\right)$$

where F_{ij}^k is the total link flow for virtual circuits of priority k, and p_k is a positive weighting factor. The form of this expression is motivated by the preceding formula, but otherwise has no meaningful interpretation based on a queueing analysis.

Routing of a new virtual circuit is done by assignment on a path that is shortest for the corresponding OD pair. This path is calculated at the destination node of the virtual circuit on the basis of the network topology, and the flow and other parameters last received for each network link. The choice of link length is motivated by the fact that the communication rate of a virtual circuit may not be negligible compared with the total flow of the links on its path. The length used is the link cost difference between when the virtual circuit is routed through the link and when it is not. Thus, if the new

virtual circuit has an estimated data rate Δ and priority class k, the length of link (i, j) is

$$D_{ij}\left(F_{ij}^1, \ldots, F_{ij}^k + \Delta, \ldots, F_{ij}^M\right) - D_{ij}(F_{ij}^1, \ldots, F_{ij}^M) \tag{5.86}$$

As a result, routing of a new virtual circuit on a shortest path with respect to the link length (5.86) results in a minimum increase in total cost. When all virtual circuits have the same priority, the link length (5.86) becomes

$$D_{ij}(F_{ij} + \Delta) - D_{ij}(F_{ij}) \tag{5.87}$$

If Δ is very small, the link lengths (5.87) are nearly equal to $\Delta D'_{ij}(F_{ij})$. Otherwise (by the mean value theorem of calculus), they are proportional to the corresponding derivatives of D_{ij} at some intermediate point between the current flow and the flow resulting when the virtual circuit is routed through (i, j). Thus, the link lengths (5.87), divided by Δ, are approximations to the first derivatives of link costs that are used in the gradient projection method. The link lengths (5.86) admit a similar interpretation (see Problem 5.27).

Rerouting of old virtual circuits is done in a similar manner. If there is a link failure, virtual circuits crossing the link are rerouted as if they are new. Rerouting with the intent of alleviating congestion is done by all nodes gradually once new link flow information is received. Each node scans the virtual circuits terminating at itself. It selects one of the virtual circuits as a candidate for rerouting and calculates the length of each link as the difference of link cost with and without that virtual circuit on the link. The virtual circuit is then rerouted on the shortest path if it does not already lie on the shortest path. An important parameter here is the number of virtual circuits chosen for rerouting between successive link flow broadcasts. This corresponds to the stepsize parameter in the gradient projection method, which determines how much flow is shifted on a shortest path at each iteration. The problem, of course, is that too many virtual circuits may be rerouted simultaneously by several nodes acting without coordination, thereby resulting in oscillatory behavior similar to that discussed in Section 5.2.5. The Codex network uses a heuristic rule whereby only a fraction of the existing virtual circuits are (pseudorandomly) considered for rerouting between successive link flow broadcasts.

SUMMARY

Routing is a sophisticated data network function that requires coordination between the network nodes through distributed protocols. It affects the average packet delay and the network throughput.

Our main focus was on methods for route selection. The most common approach, shortest path routing, can be implemented in a variety of ways, as exemplified by the ARPANET and TYMNET algorithms. Depending on its implementation, shortest path routing may cause low throughput, poor response to traffic congestion, and oscillatory behavior. These drawbacks are more evident in datagram than in virtual circuit networks. A more sophisticated alternative is optimal routing based on flow models. Several algorithms were given for computation of an optimal routing, both centralized and distributed.

As the statistics of the input arrival processes change more rapidly, the appropriateness of the type of optimal routing we focused on diminishes. In such cases it is difficult to recommend routing methods that are simultaneously efficient and practical.

Another interesting aspect of the routing problem relates to the dissemination of routing-related information over failure-prone links. We described several alternative algorithms based on flooding ideas.

Finally, routing must be taken into account when designing a network's topology, since the routing method determines how effectively the link capacities are utilized. We described exact and heuristic methods for addressing the difficulties of topological design.

NOTES, SOURCES, AND SUGGESTED READING

Section 5.1. Surveys of routing, including descriptions of some practical routing algorithms, can be found in [Eph86] and [ScS80]. Routing in the ARPANET is described in [McW77], [MRR78], and [MRR80]. The TYMNET system is described in [Tym81]. Routing in SNA is described in [Ahu79] and [Atk80]. A route selection method for SNA networks is described in [GaH83]. Descriptions of other routing systems may be found in [Wec80] and [SpM81]. Topology broadcast algorithms are discussed in [HuS88]. Extensive discussions of interconnected local area networks can be found in a special issue of *IEEE Network* [IEE88].

Section 5.2. There are many sources for the material on graphs, spanning trees, and shortest paths (*e.g.*, [Ber91] and [PaS82]). Particularly fast shortest path algorithms are described in [Pap74], [DGK79], and [GaP88].

An analysis of some asynchronous min-hop algorithms was first given in [Taj77]. The material on asynchronous shortest path algorithms given here is taken from [Ber82a]. For an extensive treatment of asynchronous distributed algorithms, see [BeT89].

For more on stability issues of shortest path routing algorithms, see [MRR78], [Ber79b], [Ber82b], and [GaB83].

Section 5.3. The difficulties with the ARPANET flooding algorithm and some remedies are described in [Ros81] and [Per83]. The SPTA appeared in the thesis [Spi85]; see also [SpG89]. For related work, see [EpB81], [Fin79], [GaB81], [Seg83], and [SoH86].

Section 5.4. There is considerable literature on routing based on queue state information. These works consider data networks but also relate to routing and control problems in other contexts. See [EVW80], [FoS78], [HaO84], [MoS82], [Ros86], [SaH87], [SaO82], [Sar82], [StK85], and [Yum81]. A survey of the application of control and optimization methods in communication network problems is given in [EpV89].

Surveys on topological design of data networks are provided in [BoF77], [GeK77], [KeB83], [McS77], and [MoS86]. The special issue [IEE89] describes several recent works on the subject. For further material on network reliability, see [BaP83], [Bal79], [Ben86], [LaL86], [LiS84], [PrB84], and the references quoted therein. For some al-

ternative approaches to the problem of capacity assignment and routing, see [Gav85], [NgH87], and [LeY89].

Section 5.5. The optimal routing problem considered is a special case of a multicommodity network flow problem. The problem arises also in the context of transportation networks in a form that is very similar to the one discussed here; see [AaM81], [Daf71], [Daf80], [DeK81], [FlN74], and [LaH84].

Section 5.6. The Frank–Wolfe method [FrW56] has been proposed in the context of the routing problem in [FGK73]. Problems 5.31 and 5.32 give a self-contained presentation of the convergence properties of this method as well as those of the gradient projection method.

Section 5.7. The gradient projection method was applied to multicommodity flow problems based on a path flow formulation in [Ber80]. Extensions were described in [BeG82], [BeG83], and [GaB84a]. A public domain FORTRAN implementation of the gradient projection method is given in [BGT84]. Distributed asynchronous versions of the algorithm are analyzed in [TsB86] and [Tsa89]. The optimal routing algorithm of [Gal77], which adjusts link flow fractions (see the end of Section 5.5), bears a relationship to the gradient projection method (see [BGG84]). Extensions of the algorithm are given in [Ber79a], [Gaf79], and [BGG84]. Computational and simulation results relating to this algorithm and its extensions may be found in [BGV79] and [ThC86].

Section 5.8. The comments of P. Humblet, one of the principal designers of the Codex network, were very helpful in preparing this section. Additional material can be found in [HuS86] and [HSS86].

PROBLEMS

5.1 Find a minimum weight spanning tree of the graph in Fig. 5.73 using the Prim–Dijkstra and the Kruskal algorithms.

5.2 Find the shortest path tree from every node to node 1 for the graph of Fig. 5.74 using the Bellman–Ford and Dijkstra algorithms.

5.3 The number shown next to each link of the network in Fig. 5.75 is the probability of the link failing during the lifetime of a virtual circuit from node A to node B. It is assumed that links fail independently of each other. Find the most reliable path from A to B, that

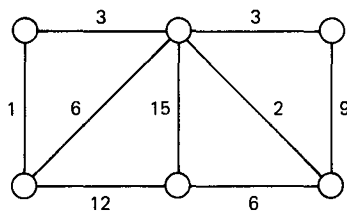


Figure 5.73 Graph for Problem 5.1.

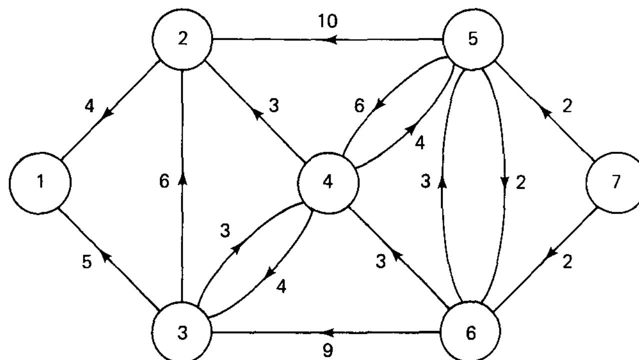


Figure 5.74 Graph for Problem 5.2.

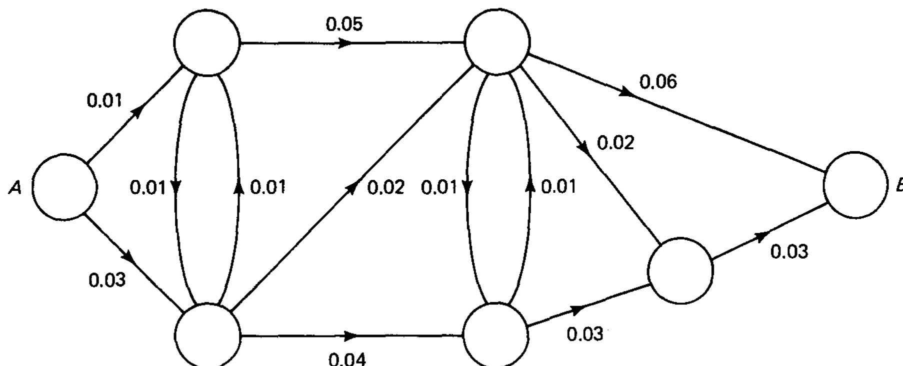


Figure 5.75 Graph for Problem 5.3.

is, the path for which the probability that all its links stay intact during the virtual circuit's lifetime is maximal. What is this probability?

5.4 A shortest path spanning tree (Section 5.2.3) differs from a minimum weight spanning tree in that the arc lengths refer to directed arcs, while in the minimum weight spanning tree problem, the arcs weights refer to undirected arcs or, equivalently, the arc weights are assumed equal in both directions. However, even if all arcs have equal length in both directions, a minimum weight spanning tree (for arc weights equal to the corresponding lengths) need not be a shortest path spanning tree. To see this, consider a network with three nodes A, B, and C, and three bidirectional arcs AB, BC, and CA. Choose a weight for each arc so that the minimum weight spanning tree is not a shortest path tree with root node C. (The length of an arc in each of its two directions is equal to the arc weight.)

5.5 Consider Example 5.4.

(a) Repeat the example with $N = 6$ instead of $N = 16$. (Nodes $1, 2, 4$, and 5 send 1 unit to node 6, while node 3 sends ϵ with $0 < \epsilon \ll 1$.)

(b) Repeat part (a) with the difference that the length d_{ij} of link (i, j) is $\alpha + F_{ij}$ for $\alpha = 1$ (instead of F_{ij}). Consider all possible choices of initial routing.

(c) What is the minimum value of α for which the shortest paths of all nodes except node 3 eventually stay constant regardless of the choice of initial routing?

(d) Repeat part (a) with the difference that at each iteration after the first, the length of each link is the average of the link arrival rates corresponding to the current and the preceding routings (instead of the arrival rates at just the current routing).

5.6 Consider the shortest path problem assuming that all cycles not containing node 1 have nonnegative length. Let $\tilde{D}_j$ be the shortest path distances corresponding to a set of link lengths $\tilde{d}_{ij}$, and let d_{ij} be a new set of link lengths. For $k = 1, 2, \ldots$, define

$$N_1 = \{i \neq 1 \mid \text{for some arc } (i,j) \text{ with } d_{ij} > \tilde{d}_{ij} \text{ we have } \tilde{D}_i = \tilde{d}_{ij} + \tilde{D}_j\}$$

$$N_{k+1} = \{i \neq 1 \mid \text{for some arc } (i,j) \text{ with } j \in N_k \text{ we have } \tilde{D}_i = \tilde{d}_{ij} + \tilde{D}_j\}$$

(a) Show that the Bellman–Ford algorithm starting from the initial conditions $D_1^0 = 0$, $D_i^0 = \tilde{D}_i$ for $i \notin \cup_k N_k \cup \{1\}$, and $D_i^0 = \infty$ for $i \in \cup_k N_k$ terminates after at most N iterations.

(b) Based on part (a), suggest a heuristic method for alleviating the "bad news phenomenon" of Fig. 5.36, which arises in the asynchronous Bellman–Ford algorithm when some link lengths increase. *Hint*: Whenever the length of a link on the current shortest path increases, the head node of the link should propagate an estimated distance of ∞ along the shortest path tree in the direction away from the destination.

5.7 Consider the Bellman–Ford algorithm assuming that all cycles not containing node 1 have nonnegative length. For each node $i \neq 1$, let (i, j_i) be an arc such that j_i attains the minimum in the equation

$$D_i^{h_i} = \min_j \left[d_{ij} + D_j^{h_i - 1}\right]$$

where h_i is the largest h such that $D_i^h \neq D_i^{h-1}$. Consider the subgraph consisting of the arcs (i, j_i), for $i \neq 1$, and show that it is a spanning tree consisting of shortest paths. *Hint*: Show that $h_i > h_{j_i}$.

5.8 Consider the shortest path problem. Show that if there is a cycle of zero length not containing node 1, but no cycles of negative length not containing node 1, Bellman's equation has more than one solution.

5.9 Suppose that we have a directed graph with no directed cycles. We are given a length d_{ij} for each directed arc (i, j) and we want to compute a shortest path to node 1 from all other nodes, assuming there exists at least one such path. Show that nodes $2, 3, \ldots, N$ can be renumbered so that there is an arc from i to j only if $i > j$. Show that once the nodes are renumbered, Bellman's equation can be solved with $O(N^2)$ operations at worst.

5.10 Consider the shortest path problem from every node to node 1 and Dijkstra's algorithm.

(a) Assume that a positive lower bound is known for all arc lengths. Show how to modify the algorithm to allow more than one node to enter the set of permanently labeled nodes at each iteration.

(b) Suppose that we have already calculated the shortest path from every node to node 1, and assume that a single arc length *increases*. Modify Dijkstra's algorithm to recalculate as efficiently as you can the shortest paths.

5.11 *A Generic Single-Destination Multiple-Origin Shortest Path Algorithm*. Consider the shortest path problem from all nodes to node 1. The following algorithm maintains a list of nodes V and a vector $D = (D_1, D_2, \ldots, D_N)$, where each D_j is either a real number or ∞.

Initially,

$$V = \{1\}$$

$$D_1 = 0, \qquad D_i = \infty, \qquad \text{for all } i \neq 1$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (assuming that V is nonempty) is as follows:

Remove a node j from V. For each incoming arc $(i, j) \in \mathcal{A}$, with $i \neq 1$, if $D_i > d_{ij} + D_j$, set

$$D_i = d_{ij} + D_j$$

and add i to V if it does not already belong to V.

(a) Show that at the end of each iteration: (1) if $D_j < \infty$, then D_j is the length of some path starting at j and ending at 1, and (2) if $j \notin V$, then either $D_j = \infty$ or else

$$D_i \leq d_{ij} + D_j, \qquad \text{for all } i \text{ such that } (i, j) \in \mathcal{A}$$

(b) If the algorithm terminates, then upon termination, for all $i \neq 1$ such that $D_i < \infty$, D_i is the shortest distance from i to 1 and

$$D_i = \min_{(i,j) \in \mathcal{A}} [d_{ij} + D_j]$$

Furthermore, $D_i = \infty$ for all i such that there is no path from i to 1.

(c) If the algorithm does not terminate, there exist paths of arbitrarily small length from at least one node i to node 1.

(d) Show that if at each iteration the node j removed from V satisfies $D_j = \min_{i \in V} D_i$, the algorithm is equivalent to Dijkstra's algorithm.

5.12 *A Generic Single-Destination Single-Origin Shortest Path Algorithm.* Consider the problem of finding a shortest path from a given node t to node 1. Suppose that for all i we have an underestimate u_i of the shortest distance from t to i. The following algorithm maintains a list of nodes V and a vector $D = (D_1, D_2, \ldots, D_N)$, where each D_i is either a real number or ∞. Initially,

$$V = \{1\}$$

$$D_1 = 0, \qquad D_i = \infty, \qquad \text{for all } i \neq 1$$

The algorithm proceeds in iterations and terminates when V is empty. The typical iteration (assuming that V is nonempty) is as follows:

Remove a node j from V. For each incoming arc $(i, j) \in \mathcal{A}$, with $i \neq 1$, if $\min\{D_i, D_t - u_i\} > d_{ij} + D_j$, set

$$D_i = d_{ij} + D_j$$

and add i to V if it does not already belong to V.

(a) If the algorithm terminates, then upon termination, either $D_t < \infty$, in which case D_t is the shortest distance from t to 1, or else there is no path from t to 1.

(b) If the algorithm does not terminate, there exist paths of arbitrarily small length from at least one node i to node 1.

5.13 Consider the second flooding algorithm of Section 5.3.2. Suppose that there is a known upper bound on the time an update message requires to reach every node connected with the originating node. Devise a scheme based on an age field to supplement the algorithm so that it works correctly even if the sequence numbers can wrap around due to a memory or

communication error. *Note*: Packets carrying an age field should be used only in exceptional circumstances after the originating node detects an error.

5.14 Modify the SPTA so that it can be used to broadcast one-directional link information other than status throughout the network (cf. the remarks at the end of Section 5.3.3). The link information is collected by the start node of the link. Justify the modified algorithm. *Hint*: Add to the existing algorithm additional main and port tables to hold the directional link information. Formulate update rules for these tables using the node labels provided by the main topology update algorithm of Section 5.3.3.

5.15 **(a)** Give an example where the following algorithm for broadcasting topological update information fails. Initially, all nodes know the correct status of all links.

Update rules:

1. When an adjacent link changes status, the node sends the new status in a message on all operating adjacent links.
2. When a node receives a message about a nonadjacent link which differs from its view of the status of the link, it changes the status of the link in its topology table. It also sends the new status in a message on all operating adjacent links *except* the one on which it received the message.
3. When a node receives a message about an adjacent link which differs from its view of the status of the link, it sends the correct status on the link on which it received the message.

Hint: Failure occurs with simple network topologies, such as the one shown in Fig. 5.43.

(b) Give an example of failure when the word "except" in rule 2 is changed to "including."

5.16 Describe what, if anything, can go wrong in the topology broadcast algorithms of Section 5.3 if the assumption that messages are received across links in the order transmitted is relaxed. Consider the ARPANET flooding algorithm, the two algorithms of Section 5.3.2, and the SPTA of Section 5.3.3.

5.17 *Distributed Computation of the Number of Nodes in a Network.* Consider a strongly connected communication network with N nodes and A (bidirectional) links. Each node knows its identity and the set of its immediate neighbors but not the network topology. Node 1 wishes to determine the number of nodes in the network. As a first step, it initiates an algorithm for finding a *directed, rooted spanning tree with node 1 as the root.* By this we mean a tree each link (i, k) of which is directed and oriented toward node 1 along the unique path on the tree leading from i to 1 (see the example tree shown in Fig. 5.76).

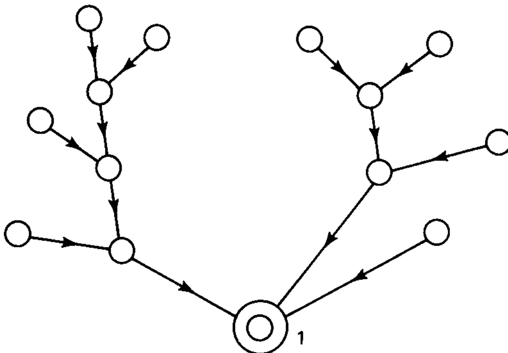


Figure 5.76 Example tree for Problem 5.17.

(a) Devise a distributed algorithm involving exchange of messages between nodes that constructs such a tree. The algorithm is initiated by node 1 and should involve no more than $O(A)$ message transmissions. Communication along any link is assumed error-free. At the end of the algorithm, the end nodes of each link should know whether the link is part of the tree, and, if so, they should know its direction.

(b) Supplement the algorithm derived in part (a) by another algorithm involving no more than $O(N)$ message transmissions by means of which node 1 gets to know N.

(c) Assuming that each message transmission takes an equal amount of time T, derive an upper bound for the time needed to complete the algorithms in parts (a) and (b).

5.18 Consider the minimum weight spanning tree problem. Show that the following procedure implements the Prim–Dijkstra algorithm and that it requires $O(N^2)$ arithmetic operations. Let node 1 be the starting node, and initially set $P = \{1\}$, $T = \emptyset$, $D_1 = 0$, and $D_j = w_{1j}$, $a_j = 1$ for $j \neq 1$.

Step 1. Find $i \notin P$ such that

$$D_i = \min_{j \notin P} D_j$$

and set $T := T \cup \{(j, a_i)\}$, $P := P \cup \{i\}$. If P contains all nodes, stop.

Step 2. For all $j \notin P$, if $w_{ij} < D_j$, set $D_j := w_{ij}$, $a_j := i$. Go to step 1.

Note: Observe the similarity with Dijkstra's algorithm for shortest paths.

5.19 Use Kleitman's algorithm to prove or disprove that the network shown in Fig. 5.77 is 3-connected. What is the maximum k for which the network is k-connected?

5.20 Suppose you had an algorithm that would find the maximum k for which two nodes in a graph are k-connected (*e.g.*, the max-flow algorithm test illustrated in Fig. 5.57). How would you modify Kleitman's algorithm to find the maximum k for which the graph is k-connected? Apply your modified algorithm to the graph of Fig. 5.77.

5.21 Use the Essau–Williams heuristic algorithm to find a constrained MST for the network shown in Fig. 5.78. Node 0 is the central node. The link weights are shown next to the links. The input flows from each node to the concentrator are shown next to the arrows. All link capacities are 10.

5.22 *MST Algorithm [MaP88].* We are given an undirected graph $G = (N, A)$ with a set of nodes $N = \{1, 2, \ldots, n\}$, and a weight a_{ij} for each arc (i, j) $(a_{ij} = a_{ji})$. We refer to the maximum over all the weights of the arcs contained in a given walk as the *critical weight* of the walk.

(a) Show that an arc (i, j) belongs to a minimum weight spanning tree (MST) if and only if the critical weight of every walk starting at i and ending at j is greater or equal to a_{ij}.

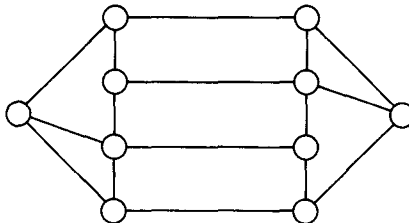


Figure 5.77 Network for Problem 5.19.

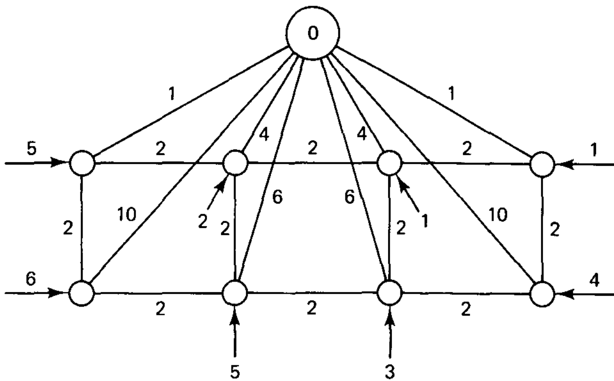


Figure 5.78 Network for Problem 5.21.

(b) Consider the following iterative algorithm:

$$x_{ij}^0 = \begin{cases} a_{ij}, & \text{if } (i,j) \in A, \\ \infty, & \text{otherwise,} \end{cases}$$

$$x_{ij}^{k+1} = \begin{cases} \min\left\{x_{ij}^k,\ \max\left\{x_{i(k+1)}^k, x_{(k+1)j}^k\right\}\right\}, & \text{if } j \neq i, \\ \infty, & \text{otherwise.} \end{cases}$$

Show that x_{ij}^k is the minimum over all critical weights of walks that start at i, end at j, and use only nodes 1 to k as intermediate nodes.

5.23 *Suboptimal Selective Broadcasting.* We are given a connected undirected graph $G = (N, A)$ with a nonnegative weight a_{ij} for each arc $(i,j) \in A$. Consider a tree in G which spans a given subset of nodes $\tilde{N}$ and has minimum weight over all such trees. Let W^* be the weight of this tree. Consider the graph $I(G)$, which has node set $\tilde{N}$ and is complete (has an arc connecting every pair of its nodes). Let the weight for each arc (i,j) of $I(G)$ be equal to the shortest distance in the graph G from the node $i \in \tilde{N}$ to the node $j \in \tilde{N}$. [Here, G is viewed as a directed graph with bidirectional arcs and length a_{ij} for each arc $(i,j) \in A$ in each direction.] Let T be a minimum weight spanning tree of $I(G)$.

(a) Show that the weight of T is no greater than $2W^*$. *Hint*: A *tour* in a graph is a cycle with no repeated nodes that passes through all the nodes. Consider a minimum weight tour in $I(G)$. Show that the weight of this tour is no less than the weight of T and no more than $2W^*$.

(b) Provide an example where the weight of T lies strictly between W^* and $2W^*$.

(c) Develop an algorithm for selective broadcasting from one node of $\tilde{N}$ to all other nodes of $\tilde{N}$ using T. Show that this algorithm comes within a factor of 2 of being optimal based on the given weights of the arcs of G.

5.24 Show that the necessary optimality condition of Section 5.5,

$$x_p^* > 0 \quad \Rightarrow \quad \frac{\partial D(x^*)}{\partial x_{p'}} \geq \frac{\partial D(x^*)}{\partial x_p}, \qquad \text{for all } p' \in P_w$$

implies that for all feasible path flow vectors $x = \{x_p\}$, we have

$$\sum_{p \in P_w} \left(x_p - x_p^*\right) \frac{\partial D(x^*)}{\partial x_p} \geq 0, \qquad \text{for all OD pairs } w$$

[The latter condition is sufficient for optimality of x^* when $D(x)$ is a convex function, that is,

$$D\big(\alpha x + (1-\alpha)x'\big) \leq \alpha D(x) + (1-\alpha)D(x')$$

for all feasible x and x' and all α with $0 \leq \alpha \leq 1$. To see this, note that if $D(x)$ is convex, we have

$$D(x) \geq D(x^*) + \sum_{w \in W} \sum_{p \in P_w} \left(x_p - x_p^*\right) \frac{\partial D(x^*)}{\partial x_p}$$

for all x ([Lue84], p. 178), so when the shortest path condition above holds, x^* is guaranteed to be optimal.] *Hint*: Let $D_w^* = \min_{p \in P_w} \partial D(x^*)/\partial x_p$. We have for every feasible x,

$$0 = \sum_{p \in P_w} (x_p - x_p^*) D_w^* \leq \sum_{\{p \in P_w | x_p > x_p^*\}} (x_p - x_p^*) \frac{\partial D(x^*)}{\partial x_p} + \sum_{\{p \in P_w | x_p < x_p^*\}} (x_p - x_p^*) D_w^*$$

5.25 Consider the network shown in Fig. 5.79 involving one OD pair with a given input rate r and three links with capacities C_1, C_2, and C_3, respectively. Assume that $C_1 = C_3 = C$, and $C_2 > C$, and solve the optimal routing problem with cost function based on the $M/M/1$ delay approximation and r between 0 and $2C + C_2$.

5.26 Consider the problem of finding the optimal routing of one unit of input flow in a single-OD-pair, three-link network for the cost function

$$D(x) = \frac{1}{2}\left[(x_1^2) + 2(x_2^2) + (x_3^2)\right] + 0.7x_3$$

Mathematically, the problem is

$$\text{minimize } D(x)$$

$$\text{subject to } x_1 + x_2 + x_3 = 1$$

$$x_1, x_2, x_3 \geq 0$$

(a) Show that $x_1^* = 2/3$, $x_2^* = 1/3$, and $x_3^* = 0$ is the optimal solution.

(b) Carry out several iterations of the Frank–Wolfe method and the projection method starting from the point (1/3, 1/3, 1/3). Do enough iterations to demonstrate a clear trend in rate of convergence. (Use a computer for this if you wish.) Plot the successive iterates on the simplex of feasible flows.

5.27 *Extensions of the Gradient Projection Method*

(a) Show how the optimality condition of Section 5.5 and the gradient projection method can be extended to handle the situation where the cost function is a general twice differentiable function of the path flow vector x.

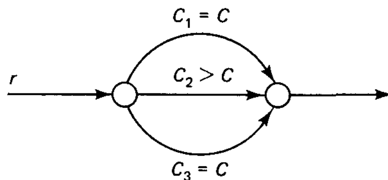


Figure 5.79 Network for Problem 5.25.

(b) Discuss the special case where the cost of link (i, j) is $D_{ij}(\tilde{F}_{ij}, F_{ij})$ where

$$\tilde{F}_{ij} = \sum_{k=1}^{M} p_k F_{ij}^k, \qquad F_{ij} = \sum_{k=1}^{M} F_{ij}^k$$

p_k is a positive scalar weighting factor for priority class k, and F_{ij}^k is the total flow of priority class k crossing link (i, j) (cf. the Codex algorithm cost function).

5.28 *Optimal Broadcast Routing along Spanning Trees.* Consider the optimal routing problem of Section 5.4. Suppose that in addition to the regular OD pair input rates r_w that are to be routed along directed paths, there is additional traffic R to be broadcast from a single node, say 1, to all other nodes along one or more directed spanning trees rooted at node 1. The spanning trees to be used for routing are to be selected from a given collection T, and the portion of R to be broadcast along each is subject to optimization.

(a) Characterize the path flow rates and the portions of R broadcast along spanning trees in T that minimize a total link cost, such as the one given by Eq. (5.29) in Section 5.4.

(b) Extend the gradient projection method to solve this problem.

(c) Extend your analysis to the case where there are several root nodes and there is traffic to be broadcast from each of these nodes to all other nodes.

Hint: Consider the first derivative total weights of spanning trees in addition to the first derivative lengths of paths.

5.29 *Optimal Routing with VCs Using the Same Links in Both Directions.* Consider the optimal routing problem of Section 5.4 with the additional restriction that the traffic originating at node i and destined for node i' must use the same path (in the reverse direction) as the traffic originating at i' and destined for i. In particular, assume that if w and w' are the OD pairs (i, i') and (i', i), respectively, there is a proportionality constant $c_{ww'}$ such that if $p \in P_w$ is a path and $p' \in P_{w'}$ is the reverse path, and x_p and $x_{p'}$ are the corresponding path flows, then $x_p = c_{ww'} x_{p'}$. Derive the conditions characterizing an optimal routing, and appropriately extend the gradient projection method.

5.30 *Use of the $M/M/1$-Based Cost Function with the Gradient Projection Method.* When the cost function

$$\sum_{(i,j)} D_{ij}(F_{ij}) = \sum_{(i,j)} \frac{F_{ij}}{C_{ij} - F_{ij}}$$

is minimized using the gradient projection method, there is potential difficulty due to the fact that some link flow F_{ij} may exceed the capacity C_{ij}. One way to get around this is to replace D_{ij} with a function $\tilde{D}_{ij}$ that is identical with D_{ij} for F_{ij} in the interval $[0, \rho C_{ij}]$, where ρ is some number less than one (say, $\rho = 0.99$), and is quadratic for $F_{ij} > \rho C_{ij}$. The quadratic function is chosen so that D_{ij} and $\tilde{D}_{ij}$ have equal values and first two derivatives at the point $F_{ij} = \rho C_{ij}$. Derive the formula for $\tilde{D}_{ij}$. Show that if the link flows F_{ij}^* that minimize

$$\sum_{(i,j)} D_{ij}(F_{ij})$$

result in link utilizations not exceeding ρ [*i.e.*, $F_{ij}^* \leq \rho C_{ij}$ for all (i, j)], then F_{ij}^* also minimize

$$\sum_{(i,j)} \tilde{D}_{ij}(F_{ij})$$

5.31 *Convergence of the Frank–Wolfe Algorithm.* The purpose of this problem and the next one is to guide the advanced reader through convergence proofs of the Frank–Wolfe and the gradient projection methods. Consider the problem

$$\text{minimize } f(x)$$
$$\text{subject to } x \in X$$

where f is a differentiable function of the n-dimensional vector x and X is a closed, bounded, convex set. Assume that the gradient of f satisfies

$$|\nabla f(x) - \nabla f(y)| \le L|x - y|, \quad \text{for all } x, y \in X$$

where L is a positive constant and $|z|$ denotes the Euclidean norm of a vector z, that is,

$$|z| = \sqrt{\textstyle\sum_{i=1}^{n} (z_i)^2}$$

(This assumption can be shown to hold if $\nabla^2 f$ exists and is continuous over X.)

(a) Show that for all vectors x, Δx, and scalars α, such that $x \in X$, $x + \Delta x \in X$, $\alpha \in [0, 1]$,

$$f(x + \alpha\Delta x) \le f(x) + \alpha \nabla f(x)^T \Delta x + \frac{\alpha^2 L}{2} |\Delta x|^2$$

where the superscript T denotes transpose. *Hint*: Use the Taylor formula

$$f(x + y) = f(x) + \nabla f(x)^T y + \int_0^1 \left[\nabla f(x + ty) - \nabla f(x)\right]^T y \, dt$$

(b) Use part (a) to show that if Δx is a descent direction at x [*i.e.*, $\nabla f(x)^T \Delta x < 0$], then

$$\min_{\alpha \in [0,1]} f(x + \alpha \Delta x) \le f(x) + \delta$$

where

$$\delta = \begin{cases} \frac{1}{2} \nabla f(x)^T \Delta x, & \text{if} \quad \nabla f(x)^T \Delta x + L|\Delta x|^2 < 0 \\ -\frac{|\nabla f(x)^T \Delta x|^2}{2LR^2}, & \text{otherwise} \end{cases}$$

where R is the diameter of X, that is,

$$R = \max_{x, y \in X} |x - y|$$

Hint: Minimize over $\alpha \in [0, 1]$ in both sides of the inequality of part (a).

(c) Consider the Frank–Wolfe method

$$x^{k+1} = x^k + \alpha^k \Delta x^k$$

where $x^0 \in X$ is the starting vector, Δx^k solves the problem

$$\text{minimize } \nabla f(x^k)^T \Delta x$$
$$\text{subject to } x^k + \Delta x \in X$$

and α^k is a minimizing stepsize

$$f(x^k + \alpha^k \Delta x^k) = \min_{\alpha \in [0,1]} f(x^k + \alpha \Delta x^k)$$

Show that every limit x^* of the sequence $\{x^k\}$ satisfies the optimality condition

$$\nabla f(x^*)^T (x - x^*) \geq 0, \qquad \text{for all } x \in X$$

Hint: Argue that if $\{x^k\}$ has a limit point and δ^k corresponds to x^k as in part (b), then $\delta^k \to 0$, and therefore also $\nabla f(x^k)^T \Delta x^k \to 0$. Taking the limit in the relation $\nabla f(x^k)^T \Delta x \leq \nabla f(x^k)^T (x - x^k)$ for all $x \in X$, argue that a limit point x^* must satisfy $0 \leq \nabla f(x^*)^T (x - x^*)$ for all $x \in X$.

5.32 *Convergence of the Gradient Projection Method.* Consider the minimization problem and the assumptions of Problem 5.31. Consider also the iteration

$$x^{k+1} = x^k + \alpha^k (\bar{x}^k - x^k)$$

where $\bar{x}^k$ is the projection of $x^k - s\nabla f(x^k)$ on X and s is some fixed positive scalar, that is, $\bar{x}^k$ solves

$$\text{minimize } \left| x - x^k + s\nabla f(x^k) \right|^2$$

$$\text{subject to } x \in X$$

(a) Assume that a^k is a minimizing stepsize

$$f\left[x^k + \alpha^k (\bar{x}^k - x^k)\right] = \min_{\alpha \in [0,1]} f\left[x^k + \alpha (\bar{x}^k - x^k)\right]$$

and show that every limit point x^* of $\{x^k\}$ satisfies the optimality condition

$$\nabla f(x^*)^T (x - x^*) \geq 0, \qquad \text{for all } x \in X$$

Hint: Follow the hints of Problem 5.31. Use the following necessary condition satisfied by the projection

$$\left[x^k - s\nabla f(x^k) - \bar{x}^k\right]^T (x - \bar{x}^k) \leq 0, \qquad \text{for all } x \in X$$

to show that $s\nabla f(x^k)^T (\bar{x}^k - x^k) \leq -|\bar{x}^k - x^k|^2$.

(b) Assume that $a^k = 1$ for all k. Show that if $s < 2/L$, the conclusion of part (a) holds.

5.33 *Gradient Projection Method with a Diagonal Scaling Matrix.* Consider the problem

$$\text{minimize } f(x)$$

$$\text{subject to } x \geq 0$$

of Section 5.7. Show that the iteration

$$x_i^{k+1} = \max\left\{0, x_i^k - \alpha^k b_i \frac{\partial f(x^k)}{\partial x_i}\right\}, \qquad i = 1, \ldots, n$$

with b_i some positive scalars, is equivalent to the gradient projection iteration

$$y_i^{k+1} = \max\left\{0, y_i^k - \alpha^k \frac{\partial h(y^k)}{\partial y_i}\right\}, \qquad i = 1, \ldots, n$$

where the variables x_i and y_i are related by $x_i = \sqrt{b_i}y_i$, and $h(y) = f(Ty)$, and T is the diagonal matrix with the ith diagonal element equal to $\sqrt{b_i}$.

5.34 This problem illustrates the relation between optimal routing and adaptive shortest path routing for virtual citcuits; see also Section 5.2.5 and [GaB83]. Consider a virtual circuit (VC) routing problem involving a single origin–destination pair and two links as shown in the figure. Assume that a VC arrives at each of the times $0, \tau, 2\tau, \ldots,$ and departs exactly H time units later. Each VC is assigned to one of the two links and remains assigned on that link for its entire duration. Let $N_i(t)$, $i = 1, 2$, be the number of VCs on link i at time t. We assume that initially the system is empty, and that an arrival and/or departure at time t is not counted in $N_i(t)$, so, for example, $N_i(0) = 0$. The routing algorithm is of the adaptive shortest path type and works as follows: For $kT \le t < (k+1)T$, a VC that arrives at time t is routed on link 1 if $\gamma_1 N_1(kT) \le \gamma_2 N_2(kT)$ and is routed on link 2 otherwise. Here T, γ_1, and γ_2 are given positive scalars.

Assume that $T \le H$ and that τ divides evenly H. Show that:

(a) $N_1(t) + N_2(t) = H/\tau$ for all $t > H$.

(b) For all $t > H$ we have

$$\frac{|N_1(t) - N_1^*|}{N_1^*} \le \frac{\gamma_1 + \gamma_2}{\gamma_2}\frac{T}{H}, \qquad \frac{|N_2(t) - N_2^*|}{N_2^*} \le \frac{\gamma_1 + \gamma_2}{\gamma_1}\frac{T}{H}$$

where (N_1^*, N_2^*) solve the optimal routing problem

$$\text{minimize } \gamma_1 N_1^2 + \gamma_2 N_2^2$$
$$\text{subject to } N_1 + N_2 = H/\tau, \quad N_1 \ge 0,\ N_2 \ge 0$$

Hint: Show that for $t > H$ we have

$$\gamma_1 N_1(t) \le \gamma_2 N_2(t) \quad \Longleftrightarrow \quad N_1(t) \le N_1^*$$

5.35 *Virtual Circuit Routing by Flooding.* In some networks where it may be hard to keep track of the topology as it changes (*e.g.*, packet radio networks), it is possible to use flooding to set up virtual circuits. The main idea is that a node m, which wants to set up a virtual circuit to node n, should flood an "exploratory" packet through the network. A node that rebroadcasts this packet stamps its ID number on it so that when the destination node n receives an exploratory packet, it knows the route along which it came. The destination node can then choose one of the potentially many routes carried by the copies of the exploratory packet received and proceed to set up the virtual circuit. Describe one or more flooding protocols that will make this process workable. Address the issues of indefinite or excessive message circulation in the network, appropriate numbering of exploratory packets, unpredictable link failures and delays, potential confusion between the two end nodes of virtual circuits, and so on.

5.36 Consider the optimal routing problem of Section 5.5. Assume that each link cost is chosen to be the same function $D(F)$ of the link flow F, where the first link cost derivative at zero flow $D'(0)$ is positive.

(a) For sufficiently small values of origin–destination pair input rates r_w show that optimal routings use only minimum-hop paths from origins to destinations.

(b) Construct an example showing that optimal routings do not necessarily have the minimum-hop property for larger values of r_w.

5.37 Consider the Frank–Wolfe method with the stepsize of Eq. (5.63) in Section 5.6. Describe an implementation as a distributed synchronous algorithm involving communication from

origins to links and the reverse. At each iteration of this algorithm, each origin should calculate the stepsize of Eq. (5.63) and update accordingly the path flows that originate at itself.

5.38 *Optimal Dynamic Routing Based on Window Ratio Strategies.* Consider the optimal routing problem of Section 5.4 in a datagram network. Suppose that we want to implement a set of path flows $\{x_p\}$ calculated on the basis of a nominal set of traffic inputs $\{r_w\}$. The usual solution, discussed in Section 5.5, is to route the traffic of each OD pair w in a way that matches closely the actual fractions of packets routed on the paths $p \in P_w$ with the desired fractions x_p/r_w. With this type of implementation each OD pair w takes into account changes in its own input traffic r_w, but makes no effort to adapt to changes in the input traffic of other OD pairs or to queue lengths inside the network. This problem considers the following more dynamic alternative.

Suppose that each OD pair w calculates the average number of packets N_p traveling on each path $p \in P_w$ by means of Little's theorem

$$N_p = x_p T_p, \qquad \text{for all } p \in P_w, w \in W$$

where T_p is the estimated average round-trip packet delay on path p (time between introduction of the packet into the network and return of an end-to-end acknowledgment for the packet). The origin of each OD pair w monitors the *actual* number of routed but unacknowledged packets $\tilde{N}_p$ on path p and routes packets in a way that roughly equalizes the ratios $\tilde{N}_p/N_p$ for all paths $p \in P_w$. Note that this scheme is sensitive to changes in delay on the paths of an OD pair due to statistical fluctuations of the input traffic and/or the routing policies of other OD pairs. The difficulty with this scheme is that the delays T_p are unknown when the numbers N_p are calculated, and therefore T_p must be estimated somehow. The simplest possibility is to use the measured value of average delay during a preceding time period. You are asked to complete the argument in the following analysis of this scheme for a simple special case.

Suppose that there is a single OD pair and only two paths with flows and path delays denoted x_1, x_2 and $T_1(x)$, $T_2(x)$, respectively, where $x = (x_1, x_2)$. The form of the path delay functions $T_1(x)$, $T_2(x)$ is unknown. We assume that $x_1 + x_2 = r$ for some constant r. Suppose that we measure x_1, x_2, $T_1(x)$, $T_2(x)$, then calculate new nominal path flows $\overline{x}_1$, $\overline{x}_2$ using some unspecified algorithm, and then implement them according to the ratio scheme described above.

(a) Argue that the actual path flow vector $\tilde{x} = (\tilde{x}_1, \tilde{x}_2)$ is determined from $\tilde{x}_1 + \tilde{x}_2 = r$ and the relation

$$\frac{\tilde{x}_1 T_1(\tilde{x})}{\overline{x}_1 T_1(x)} = \frac{\tilde{x}_2 T_2(\tilde{x})}{\overline{x}_2 T_2(x)}$$

(b) Assume that T_1 and T_2 are monotonically increasing in the sense that if $z = (z_1, z_2)$ and $z' = (z_1', z_2')$ are such that $z_1 > z_1'$, $z_2 < z_2'$ and $z_1 + z_2 = z_1' + z_2'$, then $T_1(z) > T_1(z')$ and $T_2(z) < T_2(z')$. Show that the path flows $\tilde{x}_1$, $\tilde{x}_2$ of part (a) lie in the interval between x_1 and $\overline{x}_1$ and x_2 and $\overline{x}_2$, respectively.

(c) Consider a convex cost function of the form $\sum_{(i,j)} D_{ij}(F_{ij})$. Under the assumption in part (b) show that if $\overline{x}$ gives a lower cost than x, the same is true for $\tilde{x}$.

5.39 *Routing in Networks with Frequently Changing Topology [GaB81], [BeT89].* In some situations (*e.g.*, mobile packet radio networks) topological changes are so frequent that topology broadcast algorithms are somewhat impractical. The algorithms of this problem are designed to cope with situations of this type. Consider a connected undirected graph with a special

node referred to as the *destination*. Consider the collection C of directed acyclic graphs obtained by assigning a unique direction to each of the undirected links. A graph G in C is said to be *rooted at the destination* if for every node there is a directed path in G starting at the node and ending at the destination. Show that the following two distributed asynchronous algorithms will yield a graph in C that is rooted at the destination starting from any other graph in C.

Algorithm A. A node other than the destination with no outgoing links reverses the direction of all its adjacent links. (This is done repeatedly until every node other than the destination has an outgoing link.)

Algorithm B. Every node i other than the destination keeps a list of its neighboring nodes j that have reversed the direction of the corresponding links (i, j). At each iteration, each node i that has no outgoing link reverses the directions of the links (i, j), for all j that do not appear on its list, and empties the list. If no such j exists (*i.e.*, the list is full), node i reverses the directions of all incoming links and empties the list. Initially, all lists are empty.

Hint: For algorithm A, assign to each node a distinct number. With each set of node numbers, associate the graph in C where each link is directed from a higher to a lower number. Consider an algorithm where each node reverses link directions by changing its number. Use a similar idea for algorithm B.

5.40 *Verifying the Termination of the Distributed Bellman–Ford Algorithm [DiS80], [BeT89].* The objective of the following algorithm (based on the Bellman–Ford method) is to compute in a distributed way a shortest path from a single origin (node 1) to all other nodes *and* to notify node 1 that the computation has terminated. Assume that all links (i, j) are bidirectional, have nonnegative length d_{ij}, maintain the order of messages sent on them, and operate with no errors or failures. Each node i maintains an estimate D_i of the shortest distance from node 1 to itself. Initially, $D_i = \infty$ for all nodes $i \neq 1$, and $D_1 = 0$. Node 1 initiates the computation by sending the estimate $D_1 + d_{1j}$ to all neighbor nodes j. The algorithmic rules for each node $j \neq 1$ are as follows:

1. When node j receives an estimate $D_i + d_{ij}$ from some other node i, after some unspecified finite delay (but before processing a subsequently received estimate), it does the following:
   - **(a)** If $D_j \leq D_i + d_{ij}$, node j sends an ACK to node i.
   - **(b)** If $D_j > D_i + d_{ij}$, node j sets $D_j = D_i + d_{ij}$, marks node i as its best current predecessor on a shortest path, sends an ACK to its previous best predecessor (if any), and sends the estimate $D_j + d_{jk}$ to each neighbor k.
2. Node j sends an ACK to its best current predecessor once it receives an ACK for each of the latest estimates sent to its neighbors.

We assume that each ACK is uniquely associated with a previously sent estimate, and that node 1 responds with an ACK to any length estimate it receives.

(a) Show that eventually node 1 will receive an ACK from each of its neighbors, and at that time D_i will be the correct shortest distance for each node i.

(b) What are the advantages and disadvantages of this algorithm compared with the distributed, asynchronous Bellman–Ford algorithm of Section 5.2.4?