NETWORK ROUTING ALGORITHMS

In most communication networks (except for fully connected networks), the network resources (e.g., links and buffer space) must be shared among several users. The way this is done is determined by the switching technique employed and greatly affects the performance and the characteristics of the network. The switching techniques most often used in practice are *circuit switching, message switching,* and *packet switching* (1).

In circuit-switched networks (such as telephone networks), a portion of the capacity on a path from the source to the destination is explicitly allocated to a user for its exclusive use. Circuit switching has a number of desirable characteristics, but it tends to be inefficient for data networks. This is because in data networks the traffic is bursty, and if circuit switching is used, the capacity allocated to a user may stay idle if the user does not have data to send, even if other users could use it. Message switching is a store-and-forward technology, wherein the entire message (e.g., a complete unit of data, such as a bank transaction) is stored temporarily at each intermediate node before being forwarded to the next node on the path. This was the pervasive method for data communications in the 1960s and 1970s. Message switching utilizes capacity efficiently but has several drawbacks, such as the large storage required at the nodes, the large end-to-end delay since a message has to be fully received at a node before it can start transmission on the next link, and the need for retransmitting the entire message if buffer overflow occurs. Because of these deficiencies, the industry began to move toward packet switching in the 1970s. In this switching format, messages are divided into smaller pieces, called packets, which carry protocol control information (headers), and are routed through the network independently. In both message and packet switching the capacity of the links is shared dynamically among the users, on a demand basis, since no advance allocation of the capacity is made. Packet switching has become the prevalent approach for data networks and for control networks in telephone systems.

Packet switching is further distinguished in *datagram* and *virtual circuit switching.* In a datagram network, packets are forwarded through the network independently of each other [Fig. 1(a)]. Packets that have the same source and destination may travel along different routes and arrive at the destination out of order. Each packet must carry with it a full description of the destination address, which for large networks can be quite long. In a virtual circuit network, a path is first set up end to end through the network, and all packets sent from the source to the destination follow that path [Fig. 1(b)]. Packets are switched at each node based on a virtual circuit number contained in their header, and arrive at the destination node in the order in which they are transmitted. Virtual circuits are generally used in networks whose primary service is connection oriented.

In all the switching formats described previously, a path or set of paths must be selected to connect a given source and a given destination. We will mainly focus on packet switching since the selection of paths in message and circuit switching is similar to the selection of the paths in datagram and virtual circuit switching, respectively. The algorithms used to select and establish the appropriate path from the source machine to the destination machine are called *routing algorithms.* Routing is the main function of the *network layer,* which is the third layer in the OSI architecture (2). In most networks, packets require multiple hops to make the journey to the destination, and the selection of an appropriate path typically requires the collaboration among different nodes of the network. The routing path(s) chosen is commonly mapped into entries in the routing table at each node along the path, indicating which outgoing link a given packet should be transmitted on. A packet with a specified routing tag (e.g., the destination address or the virtual circuit number) can then be

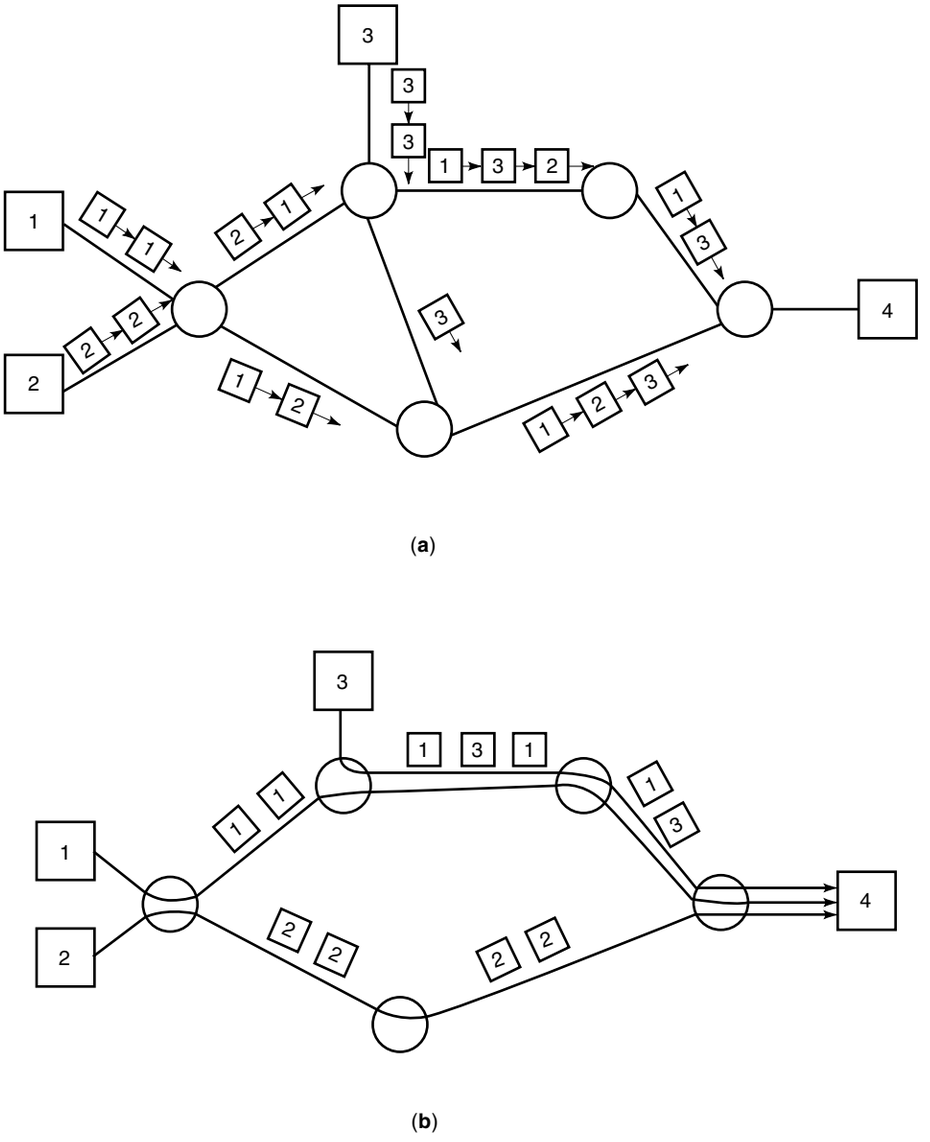


Figure 1. Routing in packet-switching networks. The integer of a packet represents the ID of its source. (a) Routing in a datagram network. (b) Routing in a virtual-circuit network.

directed by an entry in the routing table to an appropriate outgoing link.

Delay and throughput are the two main measures of interest in evaluating network performance, where delay is the time required for a packet to arrive at its destination and throughput is the number of packets that are delivered by the network per unit of time (2). These two measures are substantially affected by the routing and flow control algorithms used (2). The purpose of the routing algorithms is to keep average delay as low as possible for any level of throughput, while the purpose of flow control algorithms is to control the insertion of packets into the network so as to strike a balance between throughput and delay. When the traffic load offered by the external sites is relatively low, it is fully accepted into the network so that throughput is equal to the offered load; when the offered load is excessive, a portion of it will be rejected by the flow control algorithm and the throughput will be equal to the offered load minus the rejected load. Figure 2 shows the delay-throughput curves for good and poor routing algorithms.

TYPES OF ROUTING ALGORITHMS

Routing algorithms can be classified into static routing and adaptive routing algorithms, according to whether or not the selection of the paths takes into account the traffic conditions in the network.

Static Routing

In *static routing schemes* (also called *nonadaptive routing schemes*), the routing decisions are not based on measurements or estimates of the current traffic. The routes are computed off-line for all source-destination pairs before the net-

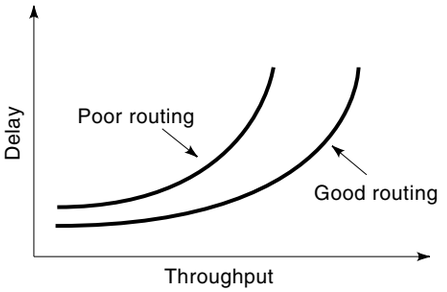
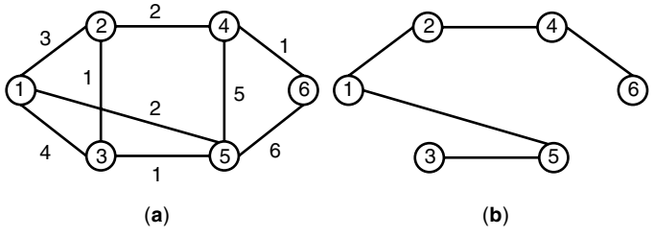


Figure 2. Delay-throughput curves for good and poor routing.

work is booted. Each node in the network maintains a *routing table,* which contains routing information for each destination. Whenever a message arrives for a given node, the outgoing link is selected from the routing table and the message is placed in the appropriate link queue. Figure 3(c) shows an example of such a routing table. The routing table is usually updated at regular intervals (e.g., weekly or monthly), or whenever nodes are added to or removed from the network, or when some obvious problems with the current routes occur. It is also usual to provide an alternative route to each destination so that when a link or node along the primary route fails, the message can still be delivered. Although static routing is the simplest routing scheme, its performance may suffer when the actual traffic on the network links is considerably different from the predicted traffic or when the traffic patterns change substantially with time.

Adaptive Routing

With *adaptive routing,* network traffic is continually estimated, and routing decisions (on a per packet, per message, or per virtual circuit basis) are made at each node according to the current traffic conditions in the network. Therefore, the route to be taken cannot be determined in advance. There are three basic approaches to adaptive routing: centralized, distributed, and isolated; these differ in the way an estimated best route is determined.



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

(c)

Figure 3. Shortest path routing with source node 1. (a) A six-node network and its link cost. (b) The shortest path tree rooted at node 1. (c) The routing table for node 1.

Centralized Routing. Centralized routing relies on a *network control center* (also called a *routing control center*), which makes traffic decisions on behalf of the network nodes. The network center periodically receives status reports (e.g., a list of a node's links or neighbors that are functional, queue lengths, amount of traffic processed) from each network node. It then analyzes these reports based on its knowledge of the entire network and provides each node with regularly updated routing information.

Centralized routing schemes have several advantages:

- The selection of the routes takes into account the traffic of the network and, therefore, improves the delay-throughput curve.
- The statistics on network performance are continually gathered, which may be valuable to network administration.

However, they also have some disadvantages:

- Additional traffic is generated between the network nodes and the control center, which competes for network resources.
- The status reports at the network control center may be outdated, because network traffic tends to fluctuate over short time intervals and the delay required to communicate with the control center may not be small enough.
- The routing information at the network control center is most needed when the traffic is not light, while the delay for getting the information is usually large when the traffic is heavy.

TYMNET is a commercial network that was originally used for terminals to log into remote computers, and it is the most notable example of a network that uses centralized adaptive routing. The nodes in the TYMNET network send to the network control center information about their status, and the control center records this information. When a new user logs in, a packet is sent to the control center. The control center then computes the best route and sends a *needle packet* back to the network node to which the user is connected, containing a description of the route. The needle packet "threads" its way through the subnet, setting the node tables along the path and, thus, setting up the virtual circuit.

Distributed Routing. The problems with centralized routing suggest that decentralized routing schemes may be preferable. In *distributed routing,* first used in the ARPANET network, neighboring nodes periodically exchange routing information (e.g., their own estimates of the time required to reach a particular destination), and nodes make routing decisions on the basis of this information, without being instructed by a central location. Typically, each node maintains a routing table that contains an entry for each of the other nodes in the network. Each entry records the preferred outgoing link for a certain destination and an estimate of the cost to reach that destination. The cost metric used may be the distance, estimated delay, bandwidth available on the path, or something similar. The details of the distributed routing algorithm used in ARPANET are presented later in this article. Distributed routing provides more network resiliency, since each node

makes its own decisions without regard to a centralized network control center, but requires more complicated network nodes and is more complex to implement.

Isolated Routing. *Isolated routing* is a particularly simple form of decentralized adaptive routing. In isolated routing, each network node makes routing decisions based on its local information. Network nodes do not exchange routing information with each other, but they still try to adapt to the current traffic conditions and topology.

Flooding and Baran's *hot potato routing* (3) are two simple examples of isolated routing. The central idea of hot potato routing is that nodes try to get rid of their stored packets as soon as possible. To do this, in Baran's method, a packet is transmitted on the link that currently has the shortest queue, which does not necessarily bring it closer to its destination. When several options are available, a link belonging to the shortest path is selected. Such routing schemes are important for networks where the storage space available at each node is limited.

A variant of this idea is to combine static routing with hot potato routing. When a packet comes in, the routing algorithm takes into account both the static costs of the links and the current lengths of the queues. A possible method is to use the link whose distance to the destination plus its queue length is the smallest.

Backward learning (3) is another isolated routing algorithm proposed and investigated by Baran and his coworkers at Rand Corp. One way to implement backward learning is to include a counter and the ID of the source node in each packet. The counter is increased by one on each hop. If a network node receives a packet on link l from node V with hop count 3, it knows that node V is at most 3 hops away from link l. If its current best route to node V has more than 3 hops, it marks link l as the entry for packets whose destination is V and records the estimated distance from node V as 3. When this algorithm is executed for a while, each node learns the outgoing link on the shortest path to every other node. The counter can also be increased by the delay, the queue length found by an arriving packet, or some other similar metric, whenever the packet is forwarded. Therefore, backward routing algorithms can adapt to traffic patterns without the need for additional information exchange among the nodes.

ROUTING ALGORITHMS

Shortest Path Routing

The routing algorithms used in most practical systems are based on the shortest path idea, in one form or another (4). In such algorithms, each communication link is assigned a positive number, called its *cost* or *length,* which can be different for each direction of the link. The cost (or length) of a path is defined as the sum of the costs of the links on the path. A shortest path routing algorithm is an algorithm that routes each packet along a shortest (minimum-cost) path between the source and the destination of the packet. Several shortest path algorithms will be presented later. An example of shortest path routing is given in Fig. 3.

It is apparent that the selection of routes for different classes of traffic should be made according to different optimization criteria. For example, bursty interactive traffic requires relatively small delay and should avoid large propagation-delay links (like satellite links). This is readily accomplished using shortest path routing, by assigning larger costs to links that have large propagation delays. Traffic that requires a high level of reliability or security must be transmitted over reliable links, which can be accomplished by assigning link costs appropriately (e.g., the cost of a link could be chosen as $-\ln p$, where p is the probability that the link is operational). To make shortest path routing algorithms adaptive to traffic conditions, we could make the link cost an increasing function of the queue length, as discussed previously.

Bifurcated (Multipath) Routing

In practice, there may be several paths for a given source-destination pair that are almost equally good. Smaller networkwide average delay may be achieved in such networks by splitting the traffic over several paths, so as to minimize the load on each communication link and the queueing delay incurred when crossing the link. Routing algorithms that use multiple routes between a given pair of nodes are called *bifurcated routing algorithms* or *multipath routing algorithms.*

Bifurcated routing is applicable to both datagram networks and virtual circuit networks. For datagram networks, a choice is made among the various "good" paths for each packet separately. For virtual circuit networks, a route is chosen when a virtual circuit is set up, but different virtual circuits with the same source and destination may choose different routes.

Optimal Routing

Shortest path routing may limit the achievable throughput of the network because a single path is used for a given source-destination pair. Also, the capability of shortest path routing to adapt to changes in the traffic conditions is limited by its susceptibility to oscillations, as described later in the context of the ARPANET routing algorithm. *Optimal routing algorithms,* based on the optimization of average delay, can eliminate the aforementioned drawbacks by shifting traffic gradually between alternative paths. A detailed treatment of such algorithms can be found in Ref. 2.

Flooding

Flooding is an extreme case of isolated routing. In a flooding algorithm, copies of the packet are sent on all outgoing links of a node, except for the link over which the packet was received, so that every possible path between the source and destination nodes is used. An advantage of this approach is that the first copy of the packet that arrives at the destination will have followed a shortest-delay path, which is one of the major goals of routing. Another advantage of flooding is that it is highly resilient to node and link failures, since a copy of the packet is guaranteed to arrive at the destination as long as a path between the source and the destination exists. For this reason, several military networks (e.g., the United States Defense Communication Agency Defense Switched Network) use such routing algorithms due to their robustness. However, the *traffic multiplication effect,* which indicates that a packet generates multiple identical copies, is quite severe with flooding and the traffic load is proportional to the num-

ber of alternative paths between the source and destination nodes.

The traffic multiplication effect can be mitigated by adding several additional rules. A simple rule is for a node to discard packets that it has already forwarded. This process is called *packet die-out* or *packet kill* and substantially reduces the traffic multiplication effect. *Selective flooding,* where a packet is sent only on to the links that are going approximately in the right direction, is another practical variant of flooding.

SHORTEST PATH ALGORITHMS

In this section we discuss three popular algorithms for computing shortest paths in data networks.

Dijkstra's Algorithm

This algorithm requires that all link lengths (or costs) are nonnegative, which is the case for most network applications. It is also known as "Algorithm A" presented in Ref. 4. The basic idea of this algorithm is to find the shortest paths from a given node (called the source) to all the other nodes of the network in order of increasing path length. At step k, $k = 1, 2, \ldots, N-1$, we obtain the set P of k nodes that are closest to the source node 1, where N is the number of nodes in the network. At step $k+1$, we add a new node to the set P, whose distance to the source is the shortest of the remaining nodes not included in the set P. Let $D(i)$ denote the distance from the source to node i along the shortest path traversing nodes within the set P, and $l(i, j)$ denote the length of the link from node i to node j, where $D(i)$ [or $l(i, j)$] is equal to ∞ if no such path (or link) exists. Initially, $P = \{1\}$, $D(1) = 0$, and $D(j) = l(1, j)$ for $j \neq 1$. The algorithm can be presented as follows:

Step 1. Find the next closest node $i \notin P$ such that

$$D(i) = \min_{j \notin P} D(j)$$

Set $P = P \cup \{i\}$. If P contains all nodes, then the algorithm is complete.

Step 2. For all the remaining nodes $j \notin P$, update the distances

$$D(j) = \min(D(j), D(i) + l(i, j))$$

Go to Step 1.

Application of Dijkstra's algorithm to the network of Fig. 3(a) is shown in Fig. 4(a). The resultant tree of shortest paths and a routing table for node 1 is shown in Figs. 3(b) and 3(c).

Since there are $N-1$ iterations and the number of arithmetic operations per iteration is $O(N)$, the time required by the algorithm is at most $O(N^2)$. With proper implementation, the worst-case computation time can be considerably reduced (5).

Bellman–Ford Algorithm

This algorithm requires that the lengths (or costs) of all cycles are nonnegative. It is also known as "Algorithm B" presented in Ref. 4. The basic idea of the algorithm is to iterate on the number of arcs in a path. At step k of the algorithm, $D(i)$ records the distance from node i to the destination node 1 through the shortest path that consists of at most k arcs, where $D(i)$ is equal to ∞ if no such path exists. Initially, $D(1) = 0$, and $D(i) = \infty$ for $i \neq 1$. The algorithm can be presented as follows: For all the nodes $i \neq 1$,

Step	P	D(2)	D(3)	D(4)	D(5)	D(6)
Initial	{1}	3	4	∞	2	∞
1	{1,5}	3	3	7	2	8
2	{1,2,5}	3	3	5	2	8
3	{1,2,3,5}	3	3	5	2	8
4	{1,2,3,4,5}	3	3	5	2	6
5	{1,2,3,4,5,6}					

(**a**)

Step	D(2)	D(3)	D(4)	D(5)	D(6)
Initial	∞	∞	∞	∞	∞
1	3	4	5	2	6
2	3	3	5	2	6

(**b**)

Figure 4. Shortest path routing algorithms. (a) The Dijkstra's algorithm for the network in Fig. 2(a). (b) The Bellman–Ford algorithm for the network in Fig. 2(a).

Step 1. Find the adjacent node w of node i such that

$$D(w) + l(i, w) = \min_j (D(j) + l(i, j))$$

where the minimization is performed over all nodes j that are neighbors of node i.

Step 2. Update the distances

$$D(j) = D(w) + l(j, w)$$

Repeat this process until no changes are made.

Application of the Bellman–Ford algorithm to the network of Fig. 3(a) is shown is Fig. 4(b). The resultant tree of shortest paths and a routing table for node 1 is shown in Figs. 3(b) and 3(c).

In the worst case, N iterations have to be executed, each of which involves $N-1$ nodes, and for each node there are no more than $N-1$ alternatives for minimization. Therefore, a simple upper bound on the time required for the Bellman–Ford algorithm is at most $O(N^3)$, which is considerably worse than the time required by Dijkstra's algorithm. However, we can also show that the estimate of the computation time required by the Bellman–Ford algorithm can be reduced to $O(hA)$, where A is the number of links in the network and h is the maximum number of links in a shortest path. In a network that is not very dense, we have $A \ll N^2$; also, in many networks we have $h \ll N$, and the Bellman–Ford algorithm terminates in very few iterations. Therefore, the time required for the Bellman–Ford algorithm may be considerably less than $O(N^3)$. Generally speaking, efficiently implemented variants of the Bellman–Ford and Dijkstra's algorithms appear to be equally competitive (5).

Floyd–Warshall Algorithm

This algorithm finds the shortest paths between all pairs of nodes together and requires that the lengths (or costs) of all cycles are nonnegative. The basic idea of the algorithm is to iterate on the set of nodes that are allowed as intermediate nodes. At step k, $D(i, j)$ records the distance from node i to node j through the shortest path with the constraint that only nodes 1, 2, . . ., k can be intermediate nodes. The algorithm can be presented as follows: Initially,

$$D(i, j) = l(i, j) \text{ for all } i, j, i \neq j$$

For $k = 1, 2, \ldots, N$,

$$D(i, j) = \min(D(i, j), D(i, k) + D(k, j)), \text{ for all } i \neq j$$

Since each of the N steps must be executed for each pair of nodes, the time required for the Floyd–Warshall algorithm is $O(N^3)$.

MINIMUM WEIGHT SPANNING TREE ALGORITHMS

In previous sections we considered unicast communication, where each packet has to be sent to a single destination. Many applications, however, require sending a packet from a single source to multiple destinations (e.g., update messages for the network topology); such communication is called broadcast communication. One way to broadcast a packet is to use a flooding algorithm. But too many messages will be generated, and the routing paths involved will not be optimal according to any criterion. Another way to perform a broadcast is to send copies of the packet along a *minimum weight spanning tree.* The total cost involved in such a broadcast is optimal.

A minimum weight spanning tree, also called *minimum spanning tree,* is a tree that spans all nodes of the network and has the minimum sum of link *weights* (also called costs or lengths) over all such trees. A link weight represents the communication cost for transmitting a message along a link in either direction, and the total weight of the spanning tree represents the cost for broadcasting a message to all nodes in the network. Note that a minimum weight spanning tree is not, in general, a shortest path spanning tree [e.g., the tree of Fig. 3(b)], and vice versa. The Kruskal's algorithm and the Prim–Dijkstra algorithm are well-known algorithms for computing minimum weight spanning trees and are described next.

Kruskal's Algorithm

The Kruskal's algorithm starts with N sets of nodes, each of which consists of an isolated node, and then successively combines two of the current sets of nodes (which are two partial spanning trees) into one partial spanning tree by using the link that has minimum weight. This can be implemented by first sorting the edges into nondecreasing order by their weights and then examining each edge in turn and including it if this does not generate a cycle (or, equivalently, if it connects nodes in different sets). Figure 5 shows an example of the construction of a minimum weight spanning tree using Kruskal's algorithm. An implementation of Kruskal's algorithm that requires $O(A \log A)$ time is given in Ref. 6.

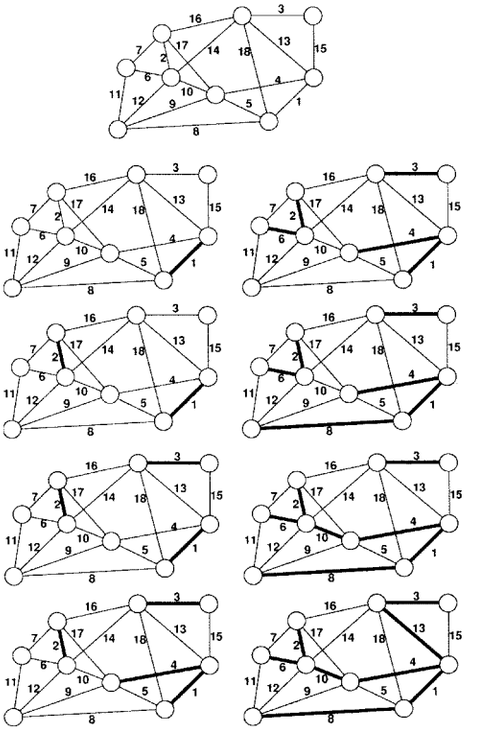


Figure 5. Successive iterations of the Kruskal's algorithm.

Prim–Dijkstra Algorithm

The Prim–Dijkstra algorithm restricts the growth to a single partial spanning tree. This approach starts with an arbitrary node and builds the spanning tree by successively adding a minimum weight outgoing link to it. Figure 6 shows an example for constructing a minimum weight spanning tree using the Prim–Dijkstra algorithm. An implementation of the Prim–Dijkstra algorithm that requires $O(A \log N)$ time is given in Ref. 6.

EXAMPLES OF ROUTING ALGORITHMS USED IN PRACTICE

The ARPANET Routing Algorithms

The ARPANET network, implemented by the Department of Defense (DOD) in 1969, has played an important role in the development of routing algorithms. The original ARPANET routing algorithm was a distributed adaptive routing algorithm, based on the Bellman–Ford iteration. This algorithm stimulated considerable research on routing and distributed computation. It had, however, some fundamental flaws and was replaced in 1979 by a new version, which is a variant of Dijkstra's algorithm. Both ARPANET algorithms are based on the shortest path idea, where the link costs are taken to be equal to the measured or estimated link delays.

In the original routing algorithm, each node was aware only of the status of its neighboring nodes. In particular, each

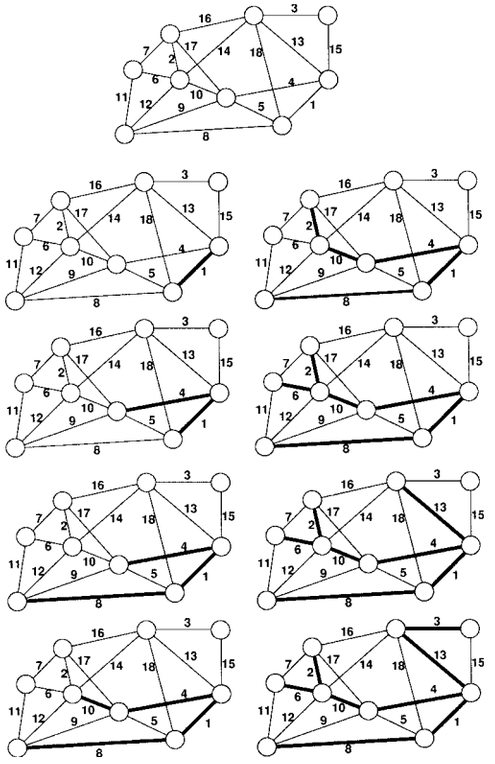


Figure 6. Successive iterations of the Prim–Dijkstra algorithm.

node i maintained two vectors: a delay vector $D(i, j), j = 1, 2, \ldots, N$, and a successor node vector $N(i, j)$ for all the destination nodes. The delay vector $D(i, j)$ contained the current estimate of the minimum delay from node i to node j; the successor node vector $N(i, j)$ contained the next node in the current minimum-delay route from node i to node j. Neighboring nodes exchanged the estimated delay every 625 ms. The algorithm updated the shortest distance estimate $D(i, j)$ according to the Bellman–Ford method

$$D(i, j) = \min_{v}(D(v, j) + l(i, v))$$

where the link length $l(i, v)$ was taken to be a function of the queue length of link (i, j).

Routing decisions were made on a packet by packet basis, and each packet was forwarded on the link along the shortest path estimated by the node at the time of transmission. Since link costs were changing very rapidly and a packet might require multiple hops to reach its destination, the estimated shortest path could change several times during the routing of a packet in the network. This had two undesirable effects. First, packets of a given message might arrive at their destination out of order, in which case they would have to be reordered upon arrival. (This was not a major disadvantage since ARPANET was a datagram network and did not make an effort to keep the order of packets anyway.) Second, the routing algorithm was prone to oscillation. More precisely, the selection of routes in an area of the network increased the load and therefore the costs of the corresponding links, so that the next selection of routes tended to use links in different areas. This, in turn, made the first area desirable for the subsequent routing decisions. The feedback effect between link costs and routing decisions resulted in oscillations and was an important flaw of the original ARPANET routing algorithm. To stabilize oscillations, a large constant was added to the link costs. Unfortunately, this reduced the adaptability of the algorithm to the changes of traffic conditions. These problems caused difficulties over several years and led to several modifications of the routing algorithm (7) and eventually to its replacement.

In the newer ARPANET routing algorithm, each node computes its routing tables using the *shortest path first algorithm* (8), which is essentially Dijkstra's algorithm with some modifications. Since Dijkstra's algorithm requires global network information at the node making routing decisions, this algorithm can be viewed as a "partially" centralized method.

Link costs in the newer ARPANET routing algorithm are a function of the delay incurred by the packets when crossing the links. Each link cost is updated every 10 s and is taken to be equal to the average packet delay on the link during the preceding 10 s period. The delay of a packet is defined as the time between the arrival of the packet at a node and the delivery of the packet at the next node, which includes the time required for processing, queueing, transmitting, retransmitting, and propagating. Each node evaluates the cost of its outgoing links and broadcasts these costs to all other nodes using a flooding algorithm. To reduce the communication overhead, the updated costs are transmitted only when the change in the link delay exceeds a certain threshold, which is reduced when the time increases since the last transmission. The total communication overhead required for the link cost updates is less than 1%.

Upon reception of a new link cost, a node recalculates the shortest path tree rooted at itself, using an incremental version of the Dijkstra's algorithm. In this algorithm, each node stores the most recently updated tree as a basis for future updates. When a single link delay changes or when a link or node is added to or deleted from the network, each node performs a partial computation to restructure its shortest path tree. The algorithm is asynchronous in the sense that link cost update messages are neither transmitted nor received simultaneously at all nodes. It turns out that this asynchronism improves the stability properties of the algorithm.

The routing update at a node required several milliseconds on the average. The algorithm responded fairly rapidly (e.g., within 100 ms) to topological changes, which solved the slow-response problems of the original ARPANET algorithm. Experimentation showed that the algorithm usually selected a minimum-hop path, but also spread traffic over links with excess capacity when the traffic was heavy. Although packets occasionally traveled in loops, looping rarely persisted.

In the current algorithm implemented in 1987, the range of possible link costs and the rate at which these costs changes have been considerably reduced, leading to a substantial improvement of the algorithm's dynamic behavior (9,10).

The TYMNET Routing Algorithm

TYMNET is a computer-communication network developed in 1970 by Tymshare Inc. It was originally designed for time-sharing purposes but also handled general data network functions later on and is classified as a value-added carrier. The

TYMNET routing algorithm is based on the shortest path method and is adaptive like the ARPANET algorithms. However, the implementations are quite different in these two networks.

The TYMNET routing algorithm is centralized and runs at a network control center called the *supervisor.* Since TYMNET uses virtual circuits, a routing path is set up each time a user establishes a connection and remains unchanged during the duration of the connection. Upon receiving a virtual circuit request, the supervisor computes a shortest path from the source to the destination. Details of the algorithm used appear in Ref. 11.

The routing path construction carried out by the supervisor incorporates the concept of class of service, which is also used in the SNA network. Link costs vary depending on the type of traffic transmitted. For example, low-speed interactive users are steered away from satellite links by increasing the link cost for such users. For file transfer between computers, on the other hand, wider-band satellite links are chosen instead of low-bandwidth terrestrial links. The link costs are also taken to depend on the utilization and the error conditions of the links.

Once the supervisor has computed the path to be used by a virtual circuit, it informs the intermediate nodes and records the relevant information in the routing table of each node on the path. A virtual circuit is assigned a channel number on each link it traverses. The routing table at a node maps the channel numbers on the incoming links to corresponding channel numbers on the outgoing links. In the original version of TYMNET, called TYMNET I, the supervisor maintained the routing tables of all nodes and read and wrote the tables explicitly. In the current version of TYMNET, called TYMNET II, each node maintains its own routing table. The supervisor establishes a new virtual circuit by sending a needle packet to the source node. The needle packet contains routing information and threads its way through the network, building the virtual circuit as it goes, followed by the user data. In addition to the nodes on the routing path, the needle packet also contains some flags that indicate the circuit class. When a needle packet arrives at a node, its contents are checked. If the node is not the destination, the routing table is updated and the packet is sent to the next node on the path; otherwise, the packet is sent to the appropriate external site. If a link or node failure prevents the setup from reaching the destination, the setup packet is sent back to the source node, followed by the data packets that have already left the source.

The TYMNET algorithm is well designed and has performed quite well. Although the TYMNET routing algorithm is adaptive to changes in the traffic conditions, it does not have the oscillation problems of the ARPANET routing algorithms. Several of its ideas were implemented in the Codex network (2).

DEADLOCKS AND DEADLOCK AVOIDANCE

Deadlock (also called *lockup*) is an undesirable configuration where every packet in a set of packets is requesting resources (e.g., buffers or channels) held by other packets while at the same time holding resources requested by other packets in the same set. In such a situation no packet can move if the resources held by another packet are not released, and the packets are blocked forever.

There are three strategies for handling deadlocks: *deadlock prevention, deadlock avoidance,* and *deadlock recovery.* In deadlock prevention, resources are granted to packets in a way that never leads to deadlock. For example, this can be achieved by reserving all the required resources before the packet is inserted in the network, as is the case with circuit switching when backtracking is allowed. In deadlock avoidance, resources are requested as a packet traverses the network, but the resources are granted to a packet only if the resultant global state does not lead to deadlock. This is not easy to achieve in a distributed and efficient manner. In deadlock recovery, resources are allocated to a packet without checking for the possibility of deadlock. If a deadlock is detected, some resources are deallocated and granted to other packets. Then packets holding these resources may be dropped.

Deadlock prevention may lead to lower resource utilization, since a packet has to reserve all the required resources before it starts transmissions. Deadlock recovery can only be used when deadlocks are rare and can be tolerated. In what follows, we describe several types of deadlock in communication networks and then discuss deadlock avoidance in some more detail.

Deadlocks

A simple example of a deadlock is illustrated in Fig. 7(a), where we assume that all of the buffers of each node are occupied by packets going to the other node. Since the buffers are full, no incoming packets can be accepted, and none of the buffers will become available because the packets occupying them cannot be transmitted. This situation will go on forever and is called *direct store-and-forward lockup.*

A similar situation can happen with a larger set of nodes, whose buffer spaces are occupied by packets going to other nodes in the same set, as illustrated in Fig. 7(b). The channels required to route the packets form a cycle, and the packets are waiting for resources that will never be granted because they are held by other packets in the set. This situation is called *indirect store-and-forward lockup.*

In networks that use datagram switching but have to deliver packets to the host machine in the correct order, a slightly different deadlock situation, called *reassembly lockup,* is also possible. Here, the reassembly buffers at a destination, used for storing packets until all packets of a given message have arrived, may be filled with packets of partially reassembled messages. These packets cannot be released to their re-

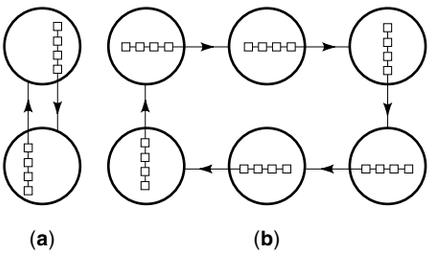


Figure 7. Store-and-forward deadlocks. (a) Direct lockup. (b) Indirect lockup.

spective hosts until correctly assembled, but the packets required to complete reassembly cannot reach the node due to the lack of buffer space. Therefore, a deadlock occurs at that node, causing the buffers of neighboring nodes to fill up.

Deadlock Avoidance

As a simple example of a deadlock avoidance scheme, consider a network that has N nodes, each having at least $D + 1$ buffers, where D is the maximum number of hops in a routing path. A packet from the host can be admitted to the network only when buffer 0 of the source node is empty; the packet can then be transmitted to an adjacent node only when its buffer 1 is empty. A packet currently in buffer i of a node, $i = 1, 2, 3, \ldots, D - 1$, can be transmitted to a neighboring node only when buffer $i + 1$ of the neighboring node is empty. Since the routes for packets traverse buffers 0, 1, 2, . . ., in ascending order, the dependency on buffers cannot form a cycle and the routing algorithm is deadlock free.

Merlin and Schweitzer have proposed a solution to store-and-forward lockups based on directed graphs (12). In the directed graph approach, each buffer is represented by a vertex, and vertices corresponding to buffers of the same node or adjacent nodes are connected by an edge. If packets are allowed to move from buffer to buffer along the edges of the directed graph, no deadlock can occur. The deadlock avoidance scheme of the previous paragraph corresponds to the following subgraph, called a *buffer graph,* of the directed graph. The buffer graph is constructed by drawing edges from buffer 0 of all the source nodes to buffer 1 in their neighboring nodes, and from buffer i each node to buffer $i + 1$ of its neighboring nodes, for $i = 0, 1, 2, \ldots, D - 1$. Since all packets move along the edges in the directed graph, the routing algorithm is deadlock free.

Blazewicz, Brzezinski, and Gambosi have proposed a deadlock avoidance scheme based on timestamps (13). In this scheme, each packet carries a stamp containing the time it was generated in the high-order bits and the machine number in the low-order bits. The clocks at different nodes of the network need not be synchronized. The algorithm requries each network node to reserve a buffer per input link as a special receive buffer. All other buffers can be used for transit packets, queued in timestamp order in a separate queue for each output link. When a node V has packets for node U while node U has no packets for node V, the timestamps of the oldest packets in nodes V and U are compared. If the oldest packet in node V is older than the oldest packet in node U, node U is forced to send a packet to node V (preferably the one that is going in the same general direction), and the oldest packet in node V is sent to node U. We can easily show that this algorithm is deadlock free by noting that as time passes, a packet will either be delivered before it becomes the oldest, or it will eventually become the oldest, in which case it has priority and is guaranteed to be delivered.

The ARPANET uses buffer allocation to prevent reassembly lockup. Before transmitting a multipacket message, a source has to send a request for reassembly space to the destination node, which then replies by sending an allocate message. Transmission of the multipacket message does not begin until this allocate message is received at the source node. To reduce the control overhead, the destination node may automatically allocate assembly buffers after delivering a multipacket message to its host. An allocate message is then sent to the source node, often piggybacked on the reverse traffic. If the source node does not have other multipacket messages for transmission to that destination, it sends a give back message to release the reassembly space.

BIBLIOGRAPHY

1. U. D. Black, *Data Networks: Concepts, Theory, and Practice,* Englewood Cliffs, NJ: Prentice-Hall, 1989.
2. D. P. Bertsekas and R. Gallager, *Data Networks,* Englewood Cliffs, NJ: Prentice-Hall, 1992.
3. P. Baran, On distributed communication networks, *IEEE Trans Commun. Syst.,* **CS-12**: 1–9, 1964.
4. M. Schwartz and T. E. Stern, Routing techniques used in computer communication networks, *IEEE Trans. Commun.,* **COM-28**: 539–552, 1980.
5. D. P. Bertsekas, *Linear Network Optimization: Algorithms and Codes,* Cambridge, MA: MIT Press, 1991.
6. B. M. E. Moret and H. D. Shapiro, *Algorithms from P to NP,* Redwood City, CA: Benjamin/Cummings, 1991.
7. J. M. McQuillan, G. Falk, and I. Richer, A review of the development and performance of the ARPANET routing algorithm, *IEEE Trans. Commun.,* **COM-26**: 1802–1811, 1978.
8. J. M. McQuillan, I. Richer, and E. C. Rosen, The new routing algorithm for the ARPANET, *IEEE Trans. Commun.,* **COM-28**: 711–719, 1980.
9. A. Khanna and J. Zinky, The revised ARPANET routing metric, *Comput. Commun. Rev.,* **19** (4): 45–56, 1989.
10. J. Zinky, G. Vichniac, and A. Khanna, Performance of the revised routing metric in the ARPANET and MILNET, *Proc. MILCOM'89.* Vol. 1, 1989, pp. 219–224.
11. L. W. Tymes, Routing and flow control in TYMNET, *IEEE Trans. Commun.,* **COM-29**: 392–398, 1981.
12. P. M. Merlin and P. J. Schweitzer, Deadlock avoidance in store-and-forward networks I: Store-and-forward deadlock, *IEEE Trans. Commun.,* **COM-28**: 345–354, 1980.
13. J. Blazewicz, J. Brzezinski, and G. Gambosi, Time-stamp approach to store-and-forward deadlock prevention, *IEEE Trans. Commun.,* **COM-35**: 490–495, 1987.

EMMANOUEL (MANOS) VARVARIGOS
CHI-HSIANG YEH
University of California

NETWORKS. See COMPUTER NETWORKS.

NETWORKS, ALL-PASS. See ALL-PASS FILTERS.

NETWORKS, ATM. See ASYNCHRONOUS TRANSFER MODE NETWORKS.

NETWORKS, COMPUTER COMMUNICATION. See MULTIPLE ACCESS SCHEMES.

NETWORKS FOR DATA COMMUNICATION. See DATA COMMUNICATION.

NETWORKS, INTELLIGENT. See INTELLIGENT NETWORKS.

NETWORKS, PARALLEL COMPUTER. See INTERCONNECTION NETWORKS FOR PARALLEL COMPUTERS.

NETWORKS, TOKEN RING. See TOKEN RING NETWORKS.

NETWORK SURVIVABILITY. See NETWORK RELIABILITY AND FAULT-TOLERANCE.

NETWORKS WITH VSATS. See VSAT NETWORKS.

NETWORK SYNCHRONIZATION. See CLOCKS IN TELECOMMUNICATIONS.