MIT 6.02 **DRAFT** *Lecture Notes*
Fall 2011 (Last update: November 27, 2011)
Comments, questions or bug reports?
Please contact hari at mit.edu

CHAPTER 18
Network Routing - I Without Any Failures

This chapter and the next one discuss the key technical ideas in network routing. We start by describing the problem, and break it down into a set of sub-problems and solve them. The key ideas that you should understand by the end are:

1. Addressing.
2. Forwarding.
3. Distributed routing protocols: *distance-vector* and *link-state* protocols.
4. How routing protocols handle adapt to failures and find usable paths.

■ 18.1 The Problem

As explained in earlier chapters, sharing is fundamental to all practical network designs. We construct networks by interconnecting nodes (switches and end points) using point-to-point links and shared media. An example of a network topology is shown in Figure 18-1; the picture shows the "backbone" of the Internet2 network, which connects a large number of academic institutions in the U.S., as of early 2010. The problem we're going to discuss at length is this: what should the switches (and end points) in a packet-switched network do to ensure that a packet sent from some sender, S, in the network reaches its intended destination, D?

The word "ensure" is a strong one, as it implies some sort of guarantee. Given that packets could get lost for all sorts of reasons (queue overflows at switches, repeated collisions over shared media, and the like), we aren't going to worry about guaranteed delivery just yet.[1] Here, we are going to consider so-called *best-effort* delivery: i.e., the switches will "do their best" to try to find a way to get packets from S to D, but there are no guarantees. Indeed, we will see that in the face of a wide range of failures that we will encounter, providing even reasonable best-effort delivery will be hard enough.

[1]Subsequent chapters will address how to improve delivery reliability.

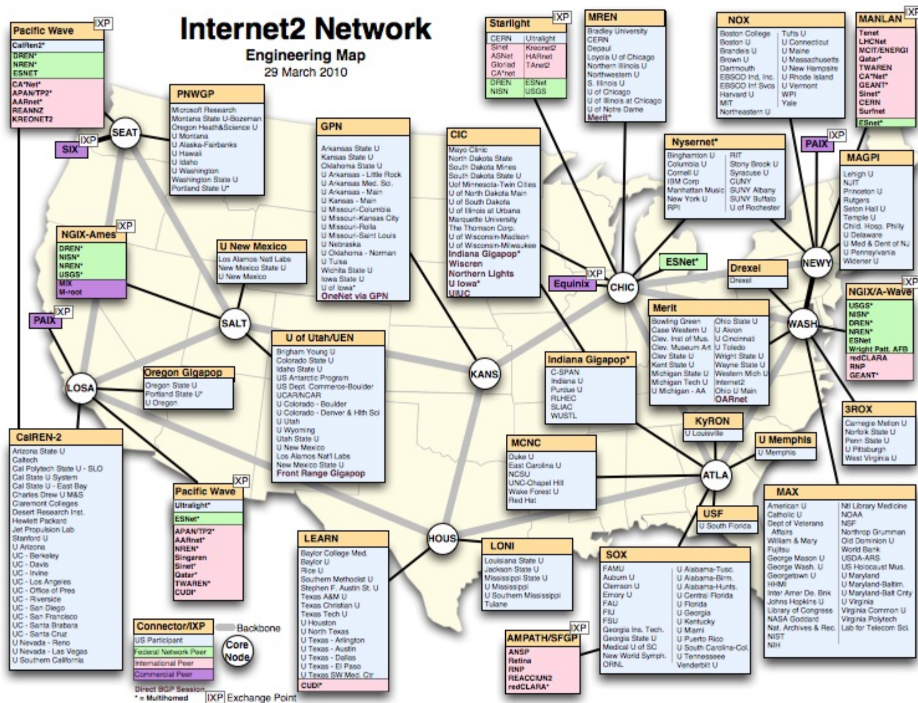

Figure 18-1: Topology of the Internet2 research and education network in the United States as of early 2010.

To solve this problem, we will model the network topology as a *graph*, a structure with nodes (vertices) connected by links (edges), as shown at the top of Figure 18-2. The nodes correspond to either switches or end points. The problem of finding paths in the network is challenging for the following reasons:

1. **Distributed information:** Each node only knows about its local connectivity, i.e., its immediate neighbors in the topology (and even determining that reliably needs a little bit of work, as we'll see). The network has to come up with a way to provide network-wide connectivity starting from this distributed information.

2. **Efficiency:** The paths found by the network should be reasonably good; they shouldn't be inordinately long in length, for that will increase the latency of packets. For concreteness, we will assume that links have costs (these costs could model link latency, for example), and that we are interested in finding a path between any source and destination that minimizes the total cost. We will assume that all link costs are non-negative. Another aspect of efficiency that we must pay attention to is the extra network bandwidth consumed by the network in finding good paths.

3. **Failures:** Links and nodes may fail and recover arbitrarily. The network should be able to find a path if one exists, without having packets get "stuck" in the network forever because of glitches. To cope with the churn caused by the failure and recovery of links and switches, as well as by new nodes and links being set up or removed, any solution to this problem must be dynamic and continually adapt to changing conditions.

In this description of the problem, we have used the term "network" several times while referring to the entity that solves the problem. The most common solution is for the network's switches to collectively solve the problem of finding paths that the end points' packets take. Although network designs where end points take a more active role in determining the paths for their packets have been proposed and are sometimes used, even those designs require the switches to do the hard work of finding a usable set of paths. Hence, we will focus on how switches can solve this problem. Clearly, because the information required for solving the problem is spread across different switches, the solution involves the switches cooperating with each other. Such methods are examples of *distributed computation*.

Our solution will be in three parts: first, we need a way to name the different nodes in the network. This task is called **addressing**. Second, given a packet with the name of a destination in its header we need a way for a switch to send the packet on the correct outgoing link. This task is called **forwarding**. Finally, we need a way by which the switches can determine how to send a packet to any destination, should one arrive. This task is done in the background, and continuously, building and updating the data structures required for forwarding to work properly. This background task, which will occupy most of our time, is called **routing**.

■ 18.2 Addressing and Forwarding

Clearly, to send packets to some end point, we need a way to uniquely identify the end point. Such identifiers are examples of *names*, a concept commonly used in computer systems: names provide a handle that can be used to refer to various objects. In our context, we want to name end points and switches. We will use the term *address* to refer to the name of a switch or an end point. For our purposes, the only requirement is that addresses refer to end points and switches uniquely. In large networks, we will want to constrain how addresses are assigned, and distinguish between the unique identifier of a node and its addresses. The distinction will allow us to use an address to refer to each distinct network link (aka "interface") available on a node; because a node may have multiple links connected to it, the unique name for a node is distinct from the addresses of its interfaces (if you have a computer with multiple active network interfaces, say a wireless link and an Ethernet, then that computer will have multiple addresses, one for each active interface).

In a packet-switched network, each packet sent by a sender contains the address of the destination. It also usually contains the address of the sender, which allows applications and other protocols running at the destination to send packets back. All this information is in the packet's header, which also may include some other useful fields. When a switch gets a packet, it consults a table keyed by the destination address to determine which link to send the packet on in order to reach the destination. This process is also known as a *table lookup*, and the table in question is termed the **routing table**.[2] The selected link is called the *outgoing link*. The combination of the destination address and outgoing link is called

[2] In practice, in high-speed networks, the routing table is distinct from the *forwarding table*. The former contains both the route to use for any destination and other properties of the route, such as the cost. The latter is a table that contains only the route, and is usually placed in faster memory because it has to be consulted on every packet.

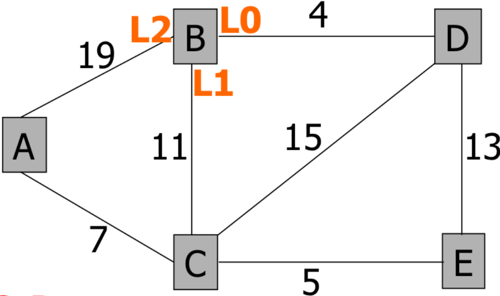

Table @ B

Destination	Link (next-hop)	Cost
A	ROUTE L1	18
B	'Self'	0
C	L1	11
D	L0	4
E	L1	16

Figure 18-2: A simple network topology showing the routing table at node B. The route for a destination is marked with an oval. The three links at node B are L0, L1, and L2; these names aren't visible at the other nodes but are internal to node B.

the **route** used by the switch for the destination. Note that the route is different from the *path* between source and destination in the topology; the sequence of routes at individual switches produces a sequence of links, which in turn leads to a path (assuming that the routing and forwarding procedures are working correctly). Figure 18-2 shows a routing table and routes at a node in a simple network.

Because data may be corrupted when sent over a link (uncorrected bit errors) or because of bugs in switch implementations, it is customary to include a checksum that covers the packet's header, and possibly also the data being sent.

These steps for forwarding work *as long as there are no failures* in the network. In the next chapter, we will expand these steps to combat problems caused by failures, packet losses, and other changes in the network that might cause packets to loop around in the network forever. We will use a "hop limit" field in the packet header to detect and discard packets that are being repeatedly forwarded by the nodes without finding their way to the intended destination.

18.3 Overview of Routing

If you don't know where you are going, any road will take you there.
—Lewis Carroll

Routing is the process by which the switches construct their routing tables. At a high level, most routing protocols have three components:

1. **Determining neighbors**: For each node, which directly linked nodes are currently both reachable and running? We call such nodes *neighbors* of the node in the topology. A node may not be able to reach a directly linked node either because the link has failed or because the node itself has failed for some reason. A link may fail to deliver all packets (e.g., because a backhoe cuts cables), or may exhibit a high packet loss rate that prevents all or most of its packets from being delivered. For now, we will assume that each node knows who its neighbors are. In the next chapter, we will discuss a common approach, called the *HELLO protocol*, by which each node determines who its current neighbors are. The basic idea if for each node to send periodic "HELLO" messages on all its live links; any node receiving a HELLO knows that the sender of the message is currently alive and a valid neighbor.

2. **Sending advertisements**: Each node sends *routing advertisements* to its neighbors. These advertisements summarize useful information about the network topology. Each node sends these advertisements periodically, for two reasons. First, in vector protocols, periodic advertisements ensure that over time the nodes all have all the information necessary to compute correct routes. Second, in both vector and link-state protocols, periodic advertisements are the fundamental mechanism used to overcome the effects of link and node failures (as well as packet losses).

3. **Integrating advertisements**: In this step, a node processes all the advertisements it has recently heard and uses that information to produce its version of the routing table.

Because the network topology can change and because new information can become available, these three steps must run continuously, discovering the current set of neighbors, disseminating advertisements to neighbors, and adjusting the routing tables. This continual operation implies that the *state* maintained by the network switches is *soft*: that is, it refreshes periodically as updates arrive, and adapts to changes that are represented in these updates. This soft state means that the path used to reach some destination could change at any time, potentially causing a stream of packets from a source to destination to arrive reordered; on the positive side, however, the ability to refresh the route means that the system can adapt by "routing around" link and node failures. We will study how the routing protocol adapts to failures in the next chapter.

A variety of routing protocols have been developed in the literature and several different ones are used in practice. Broadly speaking, protocols fall into one of two categories depending on what they send in the advertisements and how they integrate advertisements to compute the routing table. Protocols in the first category are called **vector protocols** because each node, n, advertises to its neighbors a *vector*, with one component per destination, of information that tells the neighbors about n's route to the corresponding

destination. For example, in the simplest form of a vector protocol, *n* advertises its *cost* to reach each destination as a vector of destination:cost tuples. In the integration step, each recipient of the advertisement can use the advertised cost from each neighbor, together with some other information (the cost of the link from the node to the neighbor) known to the recipient, to calculate its own cost to the destination. A vector protocol that advertises such costs is also called a **distance-vector protocol**.[3]

Routing protocols in the second category are called **link-state protocols**. Here, each node advertises information about the link to its current neighbors on all its links, and each recipient re-sends this information on all of *its* links, *flooding* the information about the links through the network. Eventually, all nodes know about all the links and nodes in the topology. Then, in the integration step, each node uses an algorithm to compute the minimum-cost path to every destination in the network.

We will compare and contrast distance-vector and link-state routing protocols at the end of the next chapter, after we study how they work in detail. For now, keep in mind the following key distinction: in a distance-vector protocol (in fact, in any vector protocol), the route computation is itself distributed, while in a link-state protocol, the route computation process is done independently at each node and the dissemination of the topology of the network is done using *distributed flooding*.

The next two sections discuss the essential details of distance-vector and link-state protocols. In this chapter, *we will assume that there are no failures of nodes or links in the network*; we will assume that the only changes that can occur in the network are *additions of either nodes or links*. We will relax this assumption in the next chapter.

We will assume that all links in the network are *bi-directional* and that the costs in each direction are symmetric (i.e., the cost of a link from *A* to *B* is the same as the cost of the link from *B* to *A*, for any two directly connected nodes *A* and *B*).

■ 18.4 A Simple Distance-Vector Protocol

The best way to understand any routing protocol is in terms of how the two distinctive steps—sending advertisements and integrating advertisements—work. In this section, we explain these two steps for a simple distance-vector protocol that achieves minimum-cost routing.

■ 18.4.1 Distance-vector Protocol Advertisements

The advertisement in a distance-vector protocol is simple, consisting of a set of tuples as shown below:

[(dest1, cost1), (dest2, cost2), (dest3, cost3), ...]

Here, each "dest" is the address of a destination known to the node, and the corresponding "cost" is the cost of the current best path known to the node. Figure 18-3 shows an example of a network topology with the distance-vector advertisements sent by each node in *steady state*, after all the nodes have computed their routing tables. During the

[3]The actual costs may have nothing to do with physical distance, and the costs need not satisfy the triangle inequality. The reason for using the term "distance-vector" rather than "cost-vector" is historic.

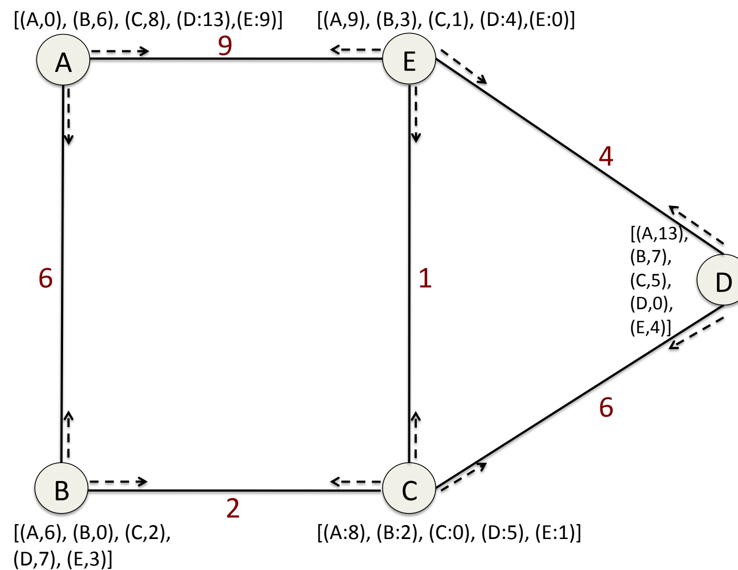

Figure 18-3: In *steady state*, each node in the the topology in this picture sends out the distance-vector advertisements shown near the node,along each link at the node.

process of computing the tables, each node advertises its *current* routing table (i.e., the destination and cost fields from the table), allowing the neighbors to make changes to their tables and advertise updated information.

What does a node do with these advertised costs? The answer lies in how the advertisements from all the neighbors are integrated by a node to produce its routing table.

■ 18.4.2 Distance-Vector Protocol: Integration Step

The key idea in the integration step uses an old observation about finding shortest-cost paths in graphs, originally due to Bellman and Ford. Consider a node n in the network and some destination d. Suppose that n hears from each of its neighbors, i, what its cost, c_i, to reach d is. Then, if n were to use the link n-i as its route to reach d, the corresponding cost would be $c_i + l_i$, where l_i is the cost of the n-i link. Hence, from n's perspective, it should choose the neighbor (link) for which the advertised cost *plus* the cost of the link from n to that neighbor is smallest. More formally, the lowest-cost path to use would be via the neighbor j, where

$$j = \arg\min_i (c_i + l_i). \quad (18.1)$$

The beautiful thing about this calculation is that it does not require the advertisements from the different neighbors to arrive synchronously. They can arrive at arbitrary times, and in any order; moreover, the integration step can run each time an advertisement arrives. The algorithm will eventually end up computing the right cost and finding the correct route (i.e., it will *converge*).

Some care must be taken while implementing this algorithm, as outlined below:

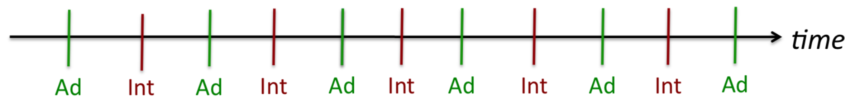

Figure 18-4: Periodic integration and advertisement steps at each node.

1. A node should update its cost and route if the new cost is smaller than the current estimate, *or* if the cost of the route currently being used changes. One question you might have is what the initial value of the cost should be before the node hears any advertisements for a destination. clearly, it should be large, a number we'll call "infinity". Later on, when we discuss failures, we will find that "infinity" for our simple distance-vector protocol can't actually be all that large. Notice that "infinity" does needs to be larger than the cost of the longest minimum-cost path in the network for routing between any pair of nodes to work correctly, because a path cost of "infinity" between some two nodes means that there is no path between those two nodes.

2. In the advertisement step, each node should make sure to advertise the current best (lowest) cost along all its links.

The implementor must take further care in these steps to correctly handle packet losses, as well as link and node failures, so we will refine this step in the next chapter.

Conceptually, we can imagine the advertisement and integration processes running periodically, for example as shown in Figure 18-4. On each advertisement, a node sends the destination:cost tuples from its current routing table. In the integration step that follows, the node processes all the information received in the most recent advertisement from each neighbor to produce an updated routing table, and the subsequent advertisement step uses this updated information. Eventually, assuming no packet losses or failures or additions, the system reaches a steady state and the advertisements don't change.

■ 18.4.3 Correctness and Performance

These two steps are enough to ensure correctness in the absence of failures. To see why, first consider a network where each node has information about only itself and about no other nodes. At this time, the only information in each node's routing table is its own, with a cost of 0. In the advertisement step, a node sends that information to each of its neighbors (whose liveness is determined using the HELLO protocol). Now, the integration step runs, and each node's routing table has a set of new entries, one per neighbor, with the route set to the link along which the advertisement arrived and a path cost equal to the cost of the link.

The next advertisement sent by each node includes the node-cost pairs for each routing table entry, and the information is integrated into the routing table at a node if, and only if, the cost of the current path to a destination is larger than (or larger than or equal to) the advertised cost *plus* the cost of the link on which the advertisement arrived.

One can show the correctness of this method by induction on the length of the path. It is easy to see that if the minimum-cost path has length 1 (i.e., 1 hop), then the algorithm finds it correctly. Now suppose that the algorithm correctly computes the minimum-cost

path from a node s to any destination for which the minimum-cost path is $\leq \ell$ hops. Now consider a destination, d, whose minimum-cost path is of length $\ell + 1$. It is clear that this path may be written as $s, t, \ldots, d$, where t is a neighbor of s and the sub-path from t to d has length ℓ. By the inductive assumption, the sub-path from t to d is a path of length ℓ and therefore the algorithm must have correctly found it. The Bellman-Ford integration step at s processes all the advertisements from s's neighbors and picks the route whose link cost plus the advertised path cost is smallest. Because of this step, and the assumption that the minimum-cost path has length $\ell + 1$, the path $s, t, \ldots, d$ must be a minimum-cost route that is correctly computed by the algorithm. This completes the proof of correctness.

How well does this protocol work? In the absence of failures, and for small networks, it's quite a good protocol. It does not consume too much network bandwidth, though the size of the advertisements grows linearly with the size of the network. How long does it take for the protocol to *converge*, assuming no packet losses or other failures occur? The next chapter will discuss what it means for a protocol to "converge"; briefly, what we're asking here is the time it takes for each of the nodes to have the correct routes to every other destination. To answer this question, observe that after every integration step, assuming that advertisements and integration steps occur at the same frequency, every node obtains information about potential minimum-cost paths that are one hop longer compared to the previous integration step. This property implies that after H steps, each node will have correct minimum-cost paths to all destinations for which the minimum-cost paths are $\leq H$ hops. Hence, the convergence time in the absence of packet losses of other is equal to the length (i.e., number of hops) of the longest minimum-cost path in the network.

In the next chapter, when we augment the protocol to handle failures, we will calculate the bandwidth consumed by the protocol and discuss some of its shortcomings. In particular, we will discover that when link or node failures occur, this protocol behaves poorly. Unfortunately, it will turn out that many of the solutions to this problem are a two-edged sword: they will solve the problem, but do so in a way that does not work well as the size of the network grows. As a result, a distance vector protocol is limited to small networks. For these networks (tens of nodes), it is a good choice because of its relative simplicity. In practice, some examples of distance-vector protocols include RIP (Routing Information Protocol), the first distributed routing protocol ever developed for packet-switched networks; EIGRP, a proprietary protocol developed by Cisco; and a slew of wireless mesh network protocols (which are variants of the concepts described above) including some that are deployed in various places around the world.

■ 18.5 A Simple Link-State Routing Protocol

A link-state protocol may be viewed as a counter-point to distance-vector: whereas a node advertised only the best cost to each destination in the latter, in a link state protocol, a node advertises information about *all* its neighbors and the link costs to them in the advertisement step (note again: a node does not advertise information about its routes to various destinations). Moreover, upon receiving the advertisement, a node *re-broadcasts* the advertisement along all its links.[4] This process is termed *flooding*.

As a result of this flooding process, each node has a map of the entire network; this map

[4] We'll assume that the information is re-broadcast even along the link on which it came, for simplicity.

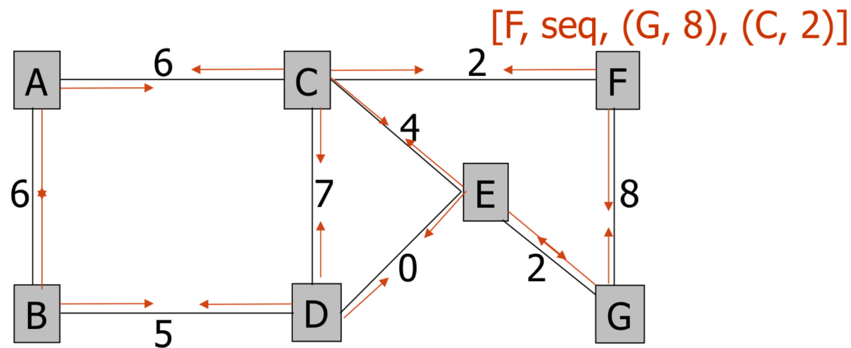

Figure 18-5: Link-state advertisement from node F in a network. The arrows show the same advertisement being re-broadcast (at different points in time) as part of the flooding process once per node, along all of the links connected to the node. The link state is shown in this example for one node; in practice, there is one of these originating from each node in the network, and re-broadcast by the other nodes.

consists of the nodes and currently working links (as evidenced by the HELLO protocol at the nodes). Armed with the complete map of the network, each node can independently run a *centralized* computation to find the shortest routes to each destination in the network. As long as all the nodes optimize the same metric for each destination, the resulting routes at the different nodes will correspond to a valid path to use. In contrast, in a distance-vector protocol, the actual computation of the routes is distributed, with no node having any significant knowledge about the topology of the network. A link-state protocol distributes information about the state of each link (hence the name) and node in the topology to all the nodes, and *as long as the nodes have a consistent view of the topology and optimize the same metric, routing will work as desired.*

■ 18.5.1 Flooding link-state advertisements

Each node uses the HELLO protocol (mentioned earlier, and which we will discuss in the next chapter in more detail) to maintain a list of current neighbors. Periodically, every `ADVERT_INTERVAL`, the node constructs a *link-state advertisement (LSA)* and sends it along all its links. The LSA has the following format:

[origin_addr, seq, (nbhr1, linkcost1), (nbhr2, linkcost2), (nbhr3, linkcost3), ...]

Here, "origin_addr" is the address of the node constructing the LSA, each "nbhr" refers to a currently active neighbor (the next chapter will describe more precisely what "currently active" means), and the "linkcost" refers to the cost of the corresponding link. An example is shown in Figure 18-5.

In addition, the LSA has a sequence number, "seq", that starts at 0 when the node turns on, and increments by 1 each time the node sends an LSA. This information is used by the flooding process, as follows. When a node receives an LSA that originated at another node, s, it first checks the sequence number of the last LSA from s. It uses the "origin_addr" field of the LSA to determine who originated the LSA. If the current sequence number is greater than the saved value for that originator, then the node *re-broadcasts the LSA on all its links*, and updates the saved value. Otherwise, it silently discards the LSA, because that same

or later LSA *must* have been re-broadcast before by the node. There are various ways to improve the performance of this flooding procedure, but we will stick to this simple (and correct) process.

For now, let us assume that a node sends out an LSA every time it discovers a new neighbor or a new link gets added to the network. The next chapter will refine this step to send advertisements periodically, in order to handle failures and packet losses, as well as changes to the link costs.

■ 18.5.2 Integration step: Dijkstra's shortest path algorithm

> *The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.*
> —Edsger W. Dijkstra, in *The Humble Programmer,* CACM 1972

> *You probably know that arrogance, in computer science, is measured in nanodijkstras.*
> —Alan Kay, 1997

The final step in the link-state routing protocol is to compute the minimum-cost paths from each node to every destination in the network. Each node independently performs this computation on its version of the network topology (map). As such, this step is quite straightforward because it is a centralized algorithm that doesn't require any inter-node coordination (the coordination occurred during the flooding of the advertisements).

Over the past few decades, a number of algorithms for computing various properties over graphs have been developed. In particular, there are many ways to compute minimum-cost path between any two nodes. For instance, one might use the Bellman-Ford method developed in Section 18.4. That algorithm is well-suited to a distributed implementation because it iteratively converges to the right answer as new updates arrive, but applying the algorithm on a complete graph is slower than some alternatives.

One of these alternatives was developed a few decades ago, a few years after the Bellman-Ford method, by a computer scientist named Edsger Dijkstra. Most link-state protocol implementations use Dijkstra's shortest-paths algorithm (and numerous extensions to it) in their integration step. One crucial assumption for this algorithm, which is fortunately true in most networks, is that the link costs must be non-negative.

Dijkstra's algorithm uses the following property of shortest paths: *if a shortest path from node X to node Y goes through node Z, then the sub-path from X to Z must also be a shortest path*. It is easy to see why this property must hold. If the sub-path from X to Z is not a shortest path, then one could find a shorter path from X to Y that uses a different, and shorter, sub-path from X to Z instead of the original sub-path, and then continue from Z to Y. By the same logic, the sub-path from Z to Y must also be a shortest path in the network. As a result, shortest paths can be concatenated together to form a shortest path between the nodes at the ends of the sub-paths.

This property suggests an iterative approach toward finding paths from a node, n, to all the other destinations in the network. The algorithm maintains two disjoint sets of nodes, S and $X = V - S$, where V is the set of nodes in the network. Initially S is empty. In each step, we will add one more node to S, and correspondingly remove that node from X. The node, v, we will add satisfies the following property: it is the node in X that has the shortest path from n. Thus, the algorithm adds nodes to S in non-decreasing order of

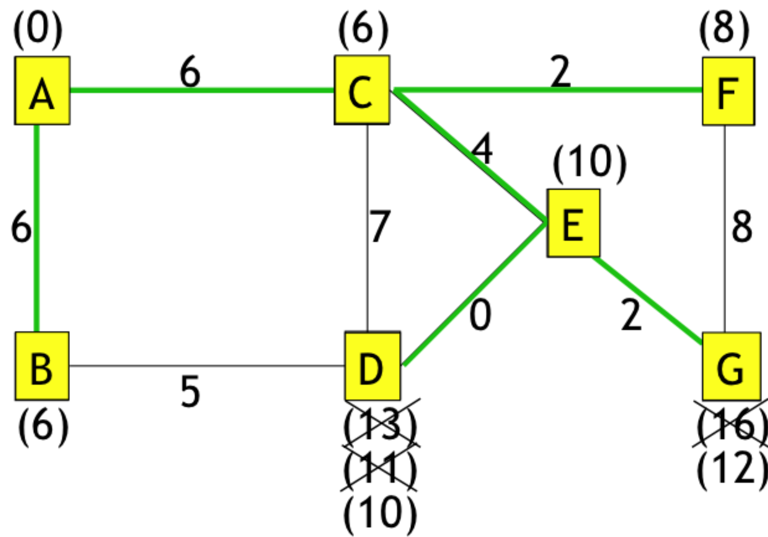

Figure 18-6: Dijkstra's shortest paths algorithm in operation, finding paths from A to all the other nodes. Initially, the set S of nodes to which the algorithm knows the shortest path is empty. Nodes are added to it in non-decreasing order of shortest path costs, with ties broken arbitrarily. In this example, nodes are added in the order (A, C, B, F, E, D, G). The numbers in parentheses near a node show the current value of `spcost` of the node as the algorithm progresses, with old values crossed out.

shortest-path costs. The first node we will add to S is n itself, since the cost of the path from n to itself is 0 (and not larger than the path to any other node, since the links all have non-negative weights). Figure 18-6 shows an example of the algorithm in operation.

Fortunately, there is an efficient way to determine the next node to add to S from the set X. As the algorithm proceeds, it maintains the current shortest-path costs, `spcost`(v), for each node v. Initially, `spcost`$(v) = \infty$ (some big number in practice) for all nodes, except for n, whose `spcost` is 0. Whenever a node u is added to S, the algorithm checks each of u's neighbors, w, to see if the current value of `spcost`(w) is larger than `spcost`(u) + `linkcost`(uw). If it is, then update `spcost`(w). Clearly, we don't need to check if the `spcost` of any other node that isn't a neighbor of u has changed because u was added to S—it couldn't have. Having done this step, we check the set X to find the next node to add to S; as mentioned before, the node with the smallest `spcost` is selected (we break ties arbitrarily).

The last part is to remember that what the algorithm needs to produce is a *route* for each destination, which means that we need to maintain the outgoing link for each destination. To compute the route, observe that what Dijkstra's algorithm produces is a *shortest path tree* rooted at the source, n, traversing all the destination nodes in the network. (A tree is a graph that has no cycles and is connected, i.e., there is exactly one path between any two nodes, and in particular between n and every other node.) There are three kinds of nodes in the shortest path tree:

1. n itself: the route from n to n is not a link, and we will call it "Self".
2. A node v directly connected to n in the tree, whose *parent* is n. For such nodes, the route is the link connecting n to v.

3. All other nodes, w, which are not directly connected to n in the shortest path tree. For such nodes, the route to w is the same as the route to w's parent, which is the node one step closer to n along the (reverse) path in the tree from w to n. Clearly, this route will be one of n's links, but we can just set it to w's parent and rely on the second step above to determine the link.
   We should also note that just because a node w is directly connected to n, it doesn't imply that the route from n is the direct link between them. If the cost of that link is larger than the path through another link, then we would want to use the route (outgoing link) corresponding to that better path.

■ Acknowledgments

Thanks to Sari Canelake and Katrina LaCurts for many helpful comments, and to Fred Chen and Eduardo Lisker for bug fixes.

■ Problems and Questions

1. Consider the network shown in Figure 18-7. The number near each link is its cost. We're interested in finding the shortest paths (taking costs into account) from S to every other node in the network.

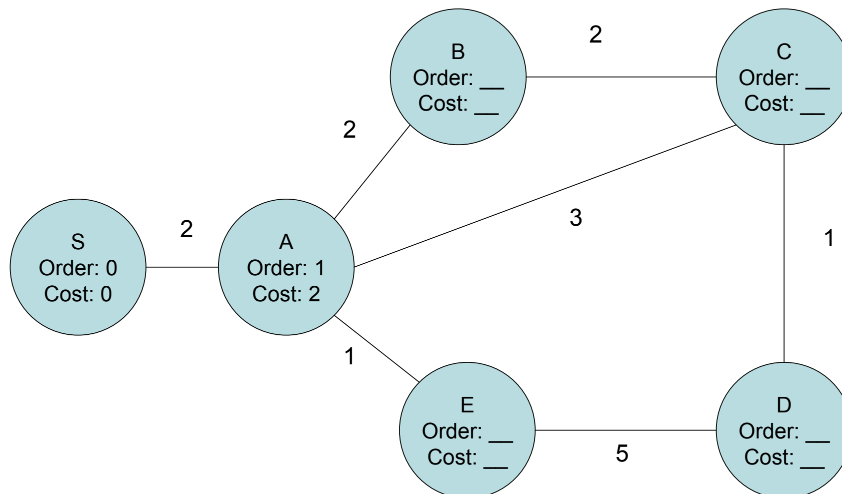

Figure 18-7: Topology for problem 1.

What is the result of running Dijkstra's shortest path algorithm on this network? To answer this question, near each node, list a pair of numbers: The first element of the pair should be the *order*, or the iteration of the algorithm in which the node is picked. The second element of each pair should be the *shortest path cost* from S to that node.

2. Alice and Bob are responsible for implementing Dijkstra's algorithm at the nodes in a network running a link-state protocol. On her nodes, Alice implements a minimum-cost algorithm. On his nodes, Bob implements a "shortest number of hops" algorithm. Give an example of a network topology with 4 or more nodes in which a routing loop occurs with Alice and Bob's implementations running simultaneously in the same network. Assume that there are no failures.

   (Note: A routing loop occurs when a group of $k \geq 1$ distinct nodes, $n_0, n_1, n_2, \ldots, n_{k-1}$ have routes such that n_i's next-hop (route) to a destination is $n_{i+1} \bmod k$.)

3. Consider any two graphs(networks) G and G' that are identical except for the costs of the links.

   (a) The cost of link l in graph G is $c_l > 0$, and the cost of the same link l in Graph G' is kc_l, where $k > 0$ is a constant. Are the shortest paths between any two nodes in the two graphs identical? Justify your answer.

   (b) Now suppose that the cost of a link l in G' is $kc_l + h$, where $k > 0$ and $h > 0$ are constants. Are the shortest paths between any two nodes in the two graphs identical? Justify your answer.

4. Eager B. Eaver implements distance vector routing in his network in which the links all have arbitrary positive costs. In addition, there are at least two paths between any two nodes in the network. One node, u, has an erroneous implementation of the integration step: it takes the advertised costs from each neighbor and picks the route corresponding to the minimum advertised cost to each destination as its route to that destination, **without adding the link cost to the neighbor**. It breaks any ties arbitrarily. All the other nodes are implemented correctly.

   Let's use the term "correct route" to mean the route that corresponds to the minimum-cost path. Which of the following statements are true of Eager's network?

   (a) Only u may have incorrect routes to any other node.

   (b) Only u and u's neighbors may have incorrect routes to any other node.

   (c) In some topologies, all nodes may have correct routes.

   (d) Even if no HELLO or advertisements packets are lost and no link or node failures occur, a routing loop may occur.

5. Alyssa P. Hacker is trying to reverse engineer the trees produced by running Dijkstra's shortest paths algorithm at the nodes in the network shown in Figure 18-8 **on the left**. She doesn't know the link costs, but knows that they are all positive. All link costs are symmetric (the same in both directions). She also knows that there is exactly one minimum-cost path between any pair of nodes in this network.

   She discovers that the routing tree computed by Dijkstra's algorithm at node **A** looks like the picture in Figure 18-8 **on the right**. Note that the exact order in which the nodes get added in Dijkstra's algorithm is not obvious from this picture.

   (a) Which of A's links has the highest cost? If there could be more than one, tell us what they are.

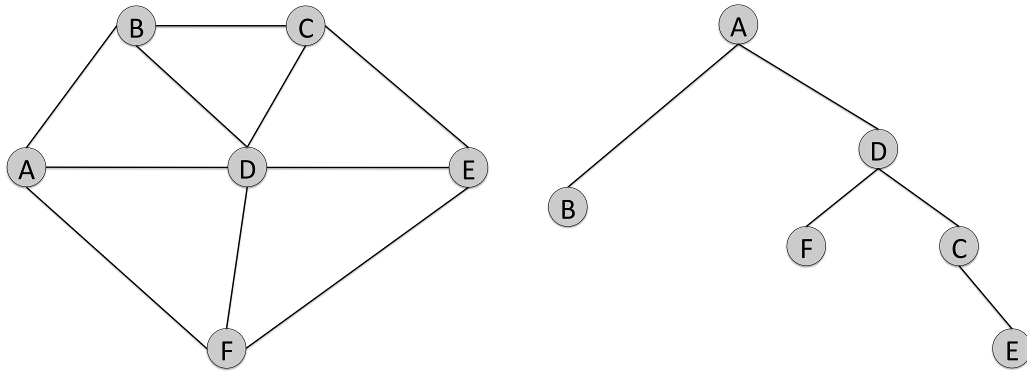

Figure 18-8: Topology for problem 5.

(b) Which of A's links has the lowest cost? If there could be more than one, tell us what they are.

Alyssa now inspects node **C**, and finds that it looks like Figure 18-9. She is sure that the bold (not dashed) links belong to the shortest path tree from node **C**, but is not sure of the dashed links.

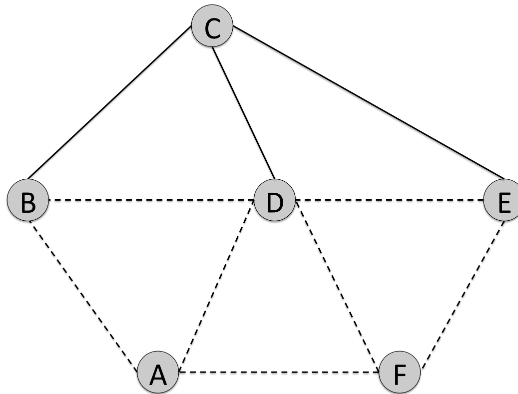

Figure 18-9: Picture for problems 5(c) and 5(d).

(c) List all the dashed links in Figure 18-9 that are **guaranteed to be** on the routing tree at node **C**.

(d) List all the dashed links in Figure 18-9 that are **guaranteed not to be** (i.e., surely not) on the routing tree at node **C**.

6. Consider a network implementing minimum-cost routing using the distance-vector protocol. A node, S, has k neighbors, numbered 1 through k, with link cost c_i to neighbor i (all links have symmetric costs). Initially, S has no route for destination D. Then, S hears advertisements for D from each neighbor, with neighbor i advertising a

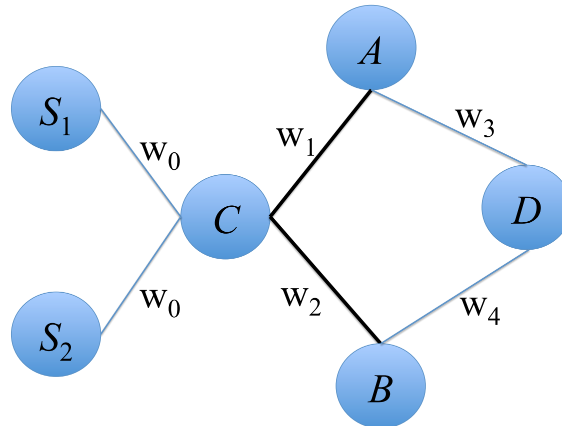

Figure 18-10: Fishnet topology for problem 6.

cost of p_i. The node integrates these k advertisements. What is the cost for destination D in S's routing table after the integration?

7. Ben Bitdiddle is responsible for routing in FishNet, shown in Figure 18-10. He gets to pick the costs for the different links (the w's shown near the links). All the costs are non-negative.

   Goal: To ensure that the links connecting C to A and C to B, shown as darker lines, carry **equal traffic load**. All the traffic is generated by S_1 and S_2, in some unknown proportion. The rate (offered load) at which S_1 and S_2 together generate traffic for destinations A, B, and D are r_A, r_B, and r_D, respectively. Each network link has a bandwidth higher than $r_A + r_B + r_D$. There are no failures.

   Protocol: FishNet uses link-state routing; each node runs Dijkstra's algorithm to pick minimum-cost routes.

   (a) If $r_A + r_D = r_B$, then what constraints (equations or inequalities) must the link costs satisfy for the goal to be met? Explain your answer. If it's impossible to meet the goal, say why.

   (b) If $r_A = r_B = 0$ and $r_D > 0$, what constraints must the link costs satisfy for the goal to be met? Explain your answer. If it's impossible to meet the goal, say why.

8. Consider the network shown in Figure 18-11. Each node implements Dijkstra's shortest paths algorithm using the link costs shown in the picture.

   (a) Initially, node **B**'s routing table contains only one entry, for itself. When **B** runs Dijkstra's algorithm, in what order are nodes added to the routing table? **List all possible answers.**

   (b) Now suppose the link cost for one of the links changes but all costs remain non-negative. For each change in link cost listed below, **state whether it is**

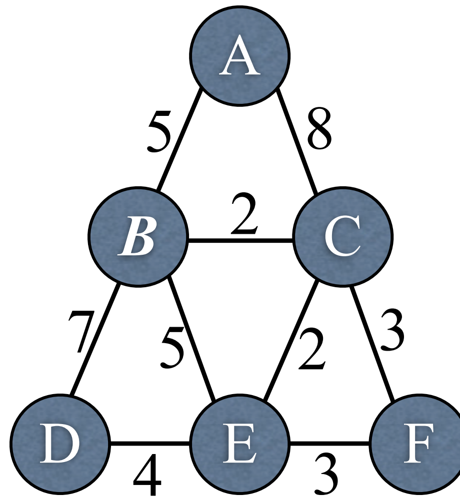

Figure 18-11: Topology for Problem 8.

possible for the *route* at node B (i.e., the link used by **B**) for any destination to change, **and if so, name the destination(s) whose routes may change**.

i. The cost of link(A, C) increases:
ii. The cost of link(A, C) decreases:
iii. The cost of link(B, C) increases:
iv. The cost of link(B, C) decreases: