also clockwise, resulting in costs shown in Figure 4.5(b). When the LS algorithm is run next, nodes *B*, *C*, and *D* all detect a zero-cost path to *A* in the counterclockwise direction, and all route their traffic to the counterclockwise routes. The next time the LS algorithm is run, *B*, *C*, and *D* all then route their traffic to the clockwise routes.

What can be done to prevent such oscillations (which can occur in any algorithm, not just a link state algorithm, that uses a congestion or delay-based link metric). One solution would be to mandate that link costs not depend on the amount of traffic carried—an unacceptable solution since one goal of routing is to avoid highly congested (for example, high-delay) links. Another solution is to ensure that not all routers run the LS algorithm at the same time. This seems a more reasonable solution, since we would hope that even if routers ran the LS algorithm with the same periodicity, the execution instance of the algorithm would not be the same at each node. Interestingly, researchers have recently noted that routers in the Internet can self-synchronize among themselves [Floyd Synchronization 1994]. That is, even though they initially execute the algorithm with the same period but at different instants of time, the algorithm execution instance can eventually become, and remain, synchronized at the routers. One way to avoid such self-synchronization is to introduce randomization purposefully into the period between execution instants of the algorithm at each node.

Having now studied the link state algorithm, let's next consider the other major routing algorithm that is used in practice today—the distance vector routing algorithm.

4.2.2 The Distance Vector Routing Algorithm

Whereas the LS algorithm is an algorithm using global information, the **distance vector (DV)** algorithm is iterative, *asynchronous, and distributed.* It is distributed in that each node receives some information from one or more of its *directly attached* neighbors, performs a calculation, and may then distribute the results of its calculation back to its neighbors. It is iterative in that this process continues on until no more information is exchanged between neighbors. (Interestingly, we will see that the algorithm is self-terminating—there is no "signal" that the computation should stop; it just stops.) The algorithm is asynchronous in that it does not require all of the nodes to operate in lockstep with each other. We'll see that an asynchronous, iterative, self-terminating, distributed algorithm is much more interesting and fun than a centralized algorithm!

The principal data structure in the DV algorithm is the **distance table** maintained at each node. Each node's distance table has a row for each destination in the network and a column for each of its directly attached neighbors. Consider a node X that is interested in routing to destination Y via its directly attached neighbor Z. Node X's **distance table entry,** $D^x(Y,Z)$ is the sum of the cost of the direct one-hop link between X and Z, $c(X,Z)$, plus neighbor Z's currently known minimum-cost path from itself (Z) to Y. That is:

$$D^x(Y,Z) = c(X,Z) + \min_w\{D^z(Y,w)\}$$

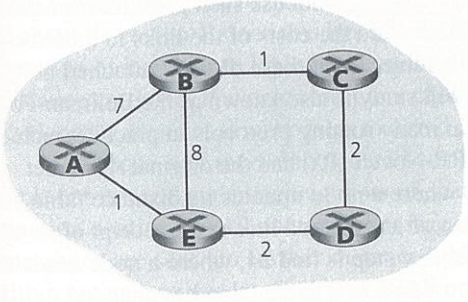
The $\min_w$ term in the equation is taken over all of Z's directly attached neighbors (including X, as we shall soon see).

The equation suggests the form of the neighbor-to-neighbor communication that will take place in the DV algorithm—each node must know the cost of the minimum-cost path of each of its neighbors to each destination. Thus, whenever a node computes a new minimum cost to some destination, it must inform its neighbors of this new minimum cost.

Before presenting the DV algorithm, let's consider an example that will help clarify the meaning of entries in the distance table. Consider the network topology and the distance table shown for node E in Figure 4.6. This is the distance table in node E once the DV algorithm has converged. Let's first look at the row for destination A.

- Clearly the cost to get to A from E via the direct connection to A has a cost of 1. Hence $D^E(A,A) = 1$.
- Let's now consider the value of $D^E(A,D)$—the cost to get from E to A, given that the first step along the path is D. In this case, the distance table entry is the cost to get from E to D (a cost of 2) plus whatever the *minimum cost* is to get from D to A. Note that the minimum cost from D to A is 3—a path that passes right back through E! Nonetheless, we record the fact that the minimum cost from E to A given that the first step is via D has a cost of 5. We're left, though, with an uneasy feeling that the fact that the path from E via D loops back through E may be the source of problems down the road (it will!).
- Similarly, we find that the distance table entry via neighbor B is $D^E(A,B) = 14$. Note that the cost is *not* 15. (Why?)

A circled entry in the distance table gives the cost of the least-cost path to the corresponding destination (row). The column with the circled entry identifies the next node along the least-cost path to the destination. Thus, a node's **forwarding table** (which indicates which outgoing link should be used to forward packets to a given destination) is easily constructed from the node's distance table.



cost to destination via

destination	$D^E()$	A	B	D
	A	(1)	14	5
	B	7	8	(5)
	C	6	9	(4)
	D	4	11	(2)

Figure 4.6 ♦ Distance table for source node E

Distance Vector (DV) Algorithm

At each node, *X*:

```
1  Initialization:
2    for all adjacent nodes v:
3        D^X(*,v) = ∞          /* the * operator means "for all rows" */
4        D^X(v,v) = c(X,v)
5    for all destinations, y
6        send min_w D(y,w) to each neighbor /* w over all X's neighbors */
7
8  loop
9    wait (until I see a link cost change to neighbor V
10         or until I receive an update from neighbor V)
11
12   if (c(X,V) changes by d)
13      /* change cost to all dest's via neighbor v by d */
14      /* note: d could be positive or negative */
15      for all destinations y: D^X(y,V) = D^X(y,V) + d
16
17   else if (update received from V wrt destination Y)
18           /* shortest path from V to some Y has changed */
19           /* V has sent a new value for its min_w D^V(Y,w) */
20           /* call this received new value "newval" */
21           for the single destination y: D^X(Y,V) = c(X,V) + newval
22
23   if we have a new min_w D^X(Y,w)for any destination Y
24      send new value of min_w D^X(Y,w) to all neighbors
25
26 forever
```

In discussing the distance table entries for node *E* above, we informally took a global view, knowing the costs of all links in the network. The distance vector algorithm we will now present is *decentralized* and does not use such global information. Indeed, the only information a node will have are the costs of the links to its directly attached neighbors, and information it receives from these directly attached neighbors. The distance vector algorithm we will study is also known as the Bellman-Ford algorithm, after its inventors. It is used in many routing protocols in practice, including the Internet's RIP and BGP, ISO IDRP, Novell IP*X*, and the original ARPAnet.

The key steps are lines 15 and 21, where a node updates its distance table entries in response to either a change of cost of an attached link or the receipt of an update message from a neighbor. The other key step is line 24, where a node sends an update to its neighbors if its minimum-cost path to a destination has changed.

Figure 4.7 illustrates the operation of the DV algorithm for the simple three-node network shown at the top of the figure. The operation of the algorithm is illustrated in

a synchronous manner, where all nodes simultaneously receive messages from their neighbors, compute new distance table entries, and inform their neighbors of any changes in their new least-cost paths. After studying this example, you should convince yourself that the algorithm operates correctly in an asynchronous manner as well, with node computations and update generation/reception occurring at any time.

The circled distance table entries in Figure 4.7 show the current minimum path cost to a destination. A double-circled entry indicates that a new minimum cost has been computed (in either line 4 of the DV algorithm (initialization), line 15, or line 21). In such cases an update message will be sent (line 24 of the DV algorithm) to the node's neighbors, as represented by the arrows between columns in Figure 4.7. The leftmost column in Figure 4.7 shows the distance table entries for nodes X, Y, and Z after the initialization step.

Let's now consider how node X computes the distance table shown in the middle column of Figure 4.7 after receiving updates from nodes Y and Z. As a result of receiving the updates from Y and Z, X computes in line 21 of the DV algorithm:

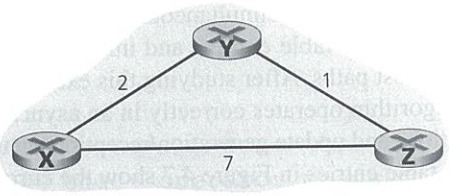
$$\begin{aligned} D^X(Y,Z) &= c(X,Z) + \min_w D^Z(Y,w) \\ &= 7 + 1 \\ &= 8 \end{aligned}$$

$$\begin{aligned} D^X(Z,Y) &= c(X,Y)\ 1\ \min_w D^Y(Z,w) \\ &= 2 + 1 \\ &= 3 \end{aligned}$$

It is important to note that the only reason that X knows about the terms $\min_w D^Z(Y,w)$ and $\min_w D^Y(Z,w)$ is because nodes Z and Y have sent those values to X (they are received by X in line 10 of the DV algorithm). As an exercise, verify the distance tables computed by Y and Z in the middle column of Figure 4.7.

The value $D^X(Z,Y) = 3$ in the second node X distance table in Figure 4.7 means that X's minimum cost to Z has changed from 7 to 3. Hence, X sends updates to Y and Z informing them of this new least cost to Z. Note that X need not update Y and Z about its cost to Y since this has not changed. Note also that although Y's recomputation of its distance table in the middle column of Figure 4.7 *does* result in new distance entries, it *does not* result in a change of Y's least-cost path to nodes X and Z. Hence Y does *not* send updates to X and Z.

The process of receiving updated costs from neighbors, recomputing distance table entries, and updating neighbors of changed costs of the least-cost path to a destination continues until no update messages are sent. At this point, since no update messages are sent, no further distance table calculations will occur and the algorithm enters a quiescent state; that is, all nodes are performing the wait in line 9 of the DV algorithm. The algorithm remains in the quiescent state until a link cost changes, as discussed next.



Node X's table

D^X	via Y	via Z
dest Y	2	∞
dest Z	∞	7

D^X	via Y	via Z
dest Y	2	8
dest Z	3	7

D^X	via Y	via Z
dest Y		
dest Z		

Node Y's table

D^Y	via X	via Z
dest X	2	∞
dest Z	∞	1

D^Y	via X	via Z
dest X	2	8
dest Z	9	1

D^Y	via X	via Z
dest X		
dest Z		

Node Z's table

D^Z	via X	via Y
dest X	7	∞
dest Y	∞	1

D^Z	via X	via Y
dest X	7	3
dest Y	9	1

D^Z	via X	via Y
dest X		
dest Y		

Time

Figure 4.7 ♦ Distance vector (DV) algorithm: Example

The Distance Vector Algorithm: Link-Cost Changes and Link Failure

When a node running the DV algorithm detects a change in the link cost from itself to a neighbor (line 12), it updates its distance table (line 15) and, if there's a change in the cost of the least-cost path, updates its neighbors (lines 23 and 24). Figure 4.8 illustrates this behavior for a scenario where the link cost from Y to X changes from 4 to 1. We focus here only on Y and Z's distance table entries to destination (row) X.

- At time t_0, Y detects the link-cost change (the cost has changed from 4 to 1) and informs its neighbors of this change since the cost of the minimum-cost path has changed.
- At time t_1, Z receives the update from Y and then updates its table. Since it computes a new least cost to X (it has decreased from a cost of 5 to a cost of 2), it informs its neighbors.
- At time t_2, Y receives Z's update and updates its distance table. Y's least costs have not changed (although its cost to X via Z has changed) and hence Y does *not* send any message to Z. The algorithm comes to a quiescent state.

In Figure 4.8, only two iterations are required for the DV algorithm to reach a quiescent state. The "good news" about the decreased cost between X and Y has propagated fast through the network.

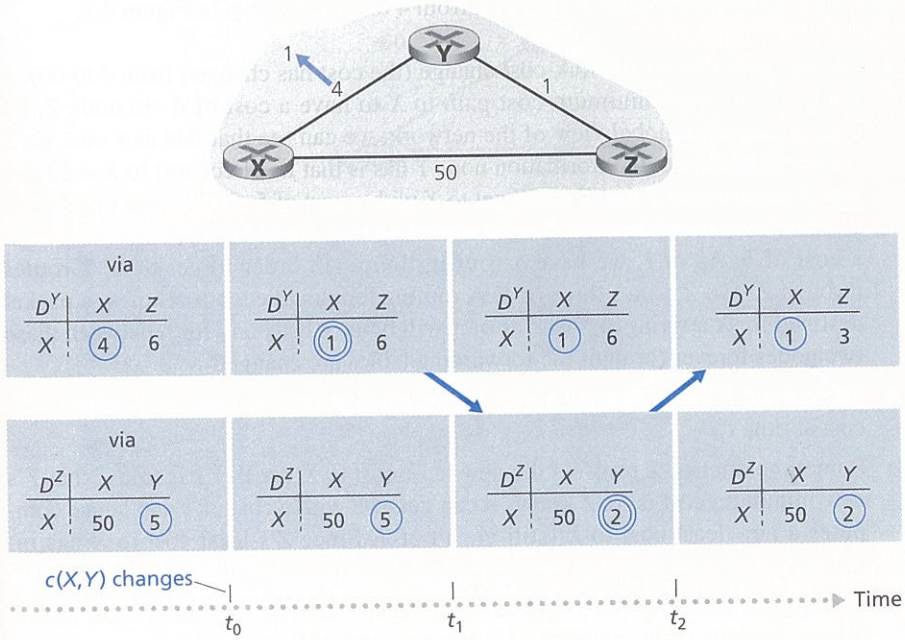


Figure 4.8 ♦ Link-cost change: Good news travels fast.

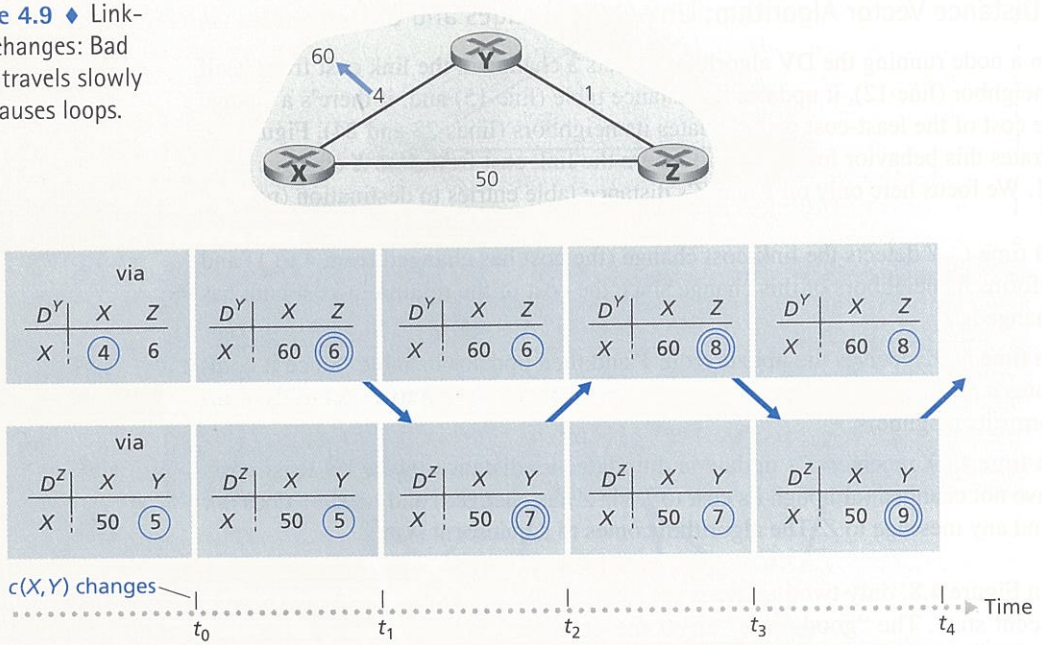


Figure 4.9 ♦ Link-cost changes: Bad news travels slowly and causes loops.

Let's now consider what can happen when a link cost *increases*. Suppose that the link cost between X and Y increases from 4 to 60, as shown in Figure 4.9.

- At time t_0, Y detects the link-cost change (the cost has changed from 4 to 60). Y computes its new minimum-cost path to X to have a cost of 6 via node Z. Of course, with our global view of the network, we can see that this new cost via Z is *wrong*. But the only information node Y has is that its direct cost to X is 60 and that Z has last told Y that Z could get to X with a cost of 5. So in order to get to X, Y would now route through Z, fully expecting that Z will be able to get to X with a cost of 5. As of t_1 we have a **routing loop**—in order to get to X, Y routes through Z, and Z routes through Y. A routing loop is like a black hole—a packet destined for X arriving at Y or Z as of t_1 will bounce back and forth between these two nodes forever (or until the forwarding tables are changed).
- Since node Y has computed a new minimum cost to X, it informs Z of this new cost at time t_1.
- Sometime after t_1, Z receives the new least cost to X via Y (Y has told Z that Y's new minimum cost is 6). Z knows it can get to Y with a cost of 1 and hence computes a new least cost to X (still via Y) of 7. Since Z's least cost to X has increased, it then informs Y of its new cost at t_2.
- In a similar manner, Y then updates its table and informs Z of a new cost of 8. Z then updates its table and informs Y of a new cost of 9, and so on.

How long will the process continue? You should convince yourself that the loop will persist for 44 iterations (message exchanges between Y and Z)—until Z eventually computes the cost of its path via Y to be greater than 50. At this point, Z will (finally!) determine that its least-cost path to X is via its direct connection to X. Y will then route to X via Z. The result of the "bad news" about the increase in link cost has indeed traveled slowly! What would have happened if the link cost $c(Y,X)$ had changed from 4 to 10,000 and the cost $c(Z,X)$ had been 9,999? Because of such scenarios, the problem we have seen is sometimes referred to as the "count-to-infinity" problem.

Distance Vector Algorithm: Adding Poisoned Reverse

The specific looping scenario illustrated in Figure 4.9 can be avoided using a technique known as *poisoned reverse*. The idea is simple—if Z routes through Y to get to destination X, then Z will advertise to Y that its (Z's) distance to X is infinity. Z will continue telling this "little white lie" to Y as long as it routes to X via Y. Since Y believes that Z has no path to X, Y will never attempt to route to X via Z, as long as Z continues to route to X via Y (and lies about doing so).

Figure 4.10 illustrates how poisoned reverse solves the particular looping problem we encountered before in Figure 4.9. As a result of the poisoned reverse, Y's distance table indicates an infinite cost when routing to X via Z (the result of

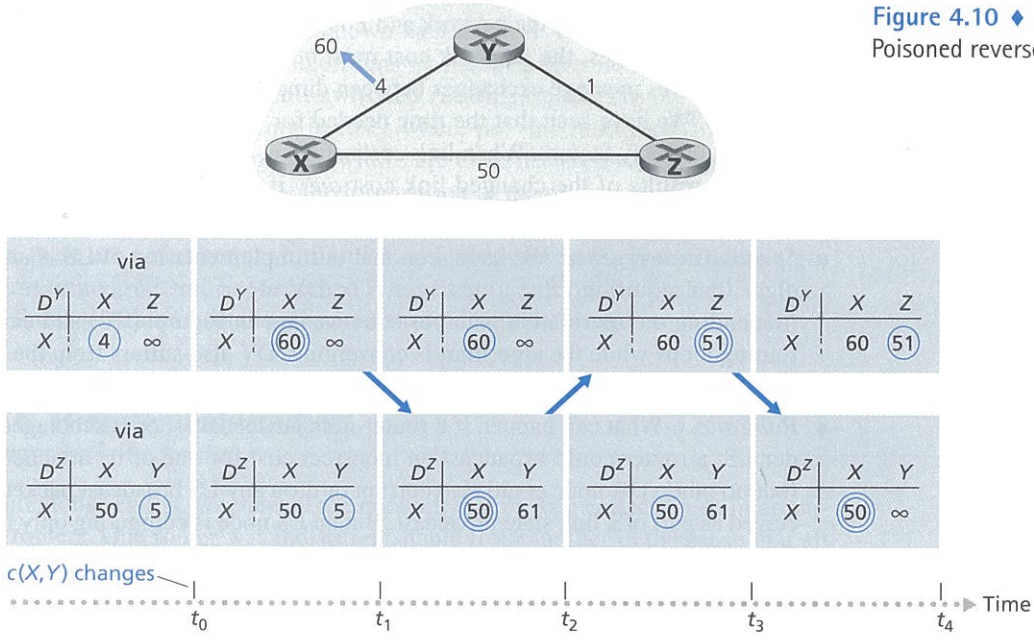


Figure 4.10 ♦ Poisoned reverse

Z having informed Y that Z's cost to X was infinity). When the cost of the XY link changes from 4 to 60 at time t_0, Y updates its table and continues to route directly to X, albeit at a higher cost of 60, and informs Z of this change in cost. After receiving the update at t_1, Z immediately shifts its route to X to be via the direct ZX link at a cost of 50. Since this is a new least-cost path to X, and since the path no longer passes through Y, Z informs Y of this new least-cost path to X at t_2. After receiving the update from Z, Y updates its distance table to route to X via Z at a least cost of 51. Also, since Z is now on Y's least-cost path to X, Y poisons the reverse path from Z to X by informing Z at time t_3 that it (Y) has an infinite cost to get to X. The algorithm becomes quiescent after t_4, with distance table entries for destination X shown in the rightmost column in Figure 4.10.

Does poison reverse solve the general count-to-infinity problem? It does not. You should convince yourself that loops involving three or more nodes (rather than simply two immediately neighboring nodes, as we saw in Figure 4.10) will not be detected by the poison reverse technique.

A Comparison of Link State and Distance Vector Routing Algorithms

Let's conclude our study of link state and distance vector algorithms with a quick comparison of some of their attributes.

- *Message complexity.* We have seen that LS requires each node to know the cost of each link in the network. This requires $O(nE)$ messages to be sent, where n is the number of nodes in the network and E is the number of links. Also, whenever a link cost changes, the new link cost must be sent to *all* nodes. The DV algorithm requires message exchanges between directly connected neighbors at each iteration. We have seen that the time needed for the algorithm to converge can depend on many factors. When link costs change, the DV algorithm will propagate the results of the changed link cost *only* if the new link cost results in a changed least-cost path for one of the nodes attached to that link.
- *Speed of convergence.* We have seen that our implementation of LS is an $O(n^2)$ algorithm requiring $O(nE)$ messages. The DV algorithm can converge slowly (depending on the relative path costs, as we saw in Figure 4.10) and can have routing loops while the algorithm is converging. DV also suffers from the count-to-infinity problem.
- *Robustness.* What can happen if a router fails, misbehaves, or is sabotaged? Under LS, a router could broadcast an incorrect cost for one of its attached links (but no others). A node could also corrupt or drop any LS broadcast packets it received as part of a link state broadcast. But an LS node is computing only its own forwarding tables; other nodes are performing the similar calculations for themselves. This means route calculations are somewhat separated under LS, providing a degree of robustness. Under DV, a node can advertise incorrect least-cost paths to any or all destinations. (Indeed, in 1997, a malfunctioning router in a