cally, an edge's cost ... physical length of the corresponding link (for example, a transoceanic link might have a higher cost than a short-haul terrestrial link), the link speed, or the monetary cost associated with a link. For our purposes, we'll simply take the edge costs as a given and won't worry about how they are determined. For any edge $(x,y)$ in $E$, we denote $c(x,y)$ as the cost of the edge between nodes $x$ and $y$. If the pair $(x,y)$ does not belong to $E$, we set $c(x,y) = \infty$. Also, throughout we consider only undirected graphs (i.e., graphs whose edges do not have a direction), so that edge $(x,y)$ is the same as edge $(y,x)$ and that $c(x,y) = c(y,x)$. Also, a node $y$ is said to be a **neighbor** of node $x$ if $(x,y)$ belongs to $E$.

Given that costs are assigned to the various edges in the graph abstraction, a natural goal of a routing algorithm is to identify the least costly paths between sources and destinations. To make this problem more precise, recall that a **path** in a graph $G = (N,E)$ is a sequence of nodes $(x_1, x_2, \ldots, x_p)$ such that each of the pairs $(x_1,x_2)$, $(x_2,x_3), \ldots, (x_{p-1},x_p)$ are edges in $E$. The cost of a path $(x_1,x_2,\ldots,x_p)$ is simply the sum of all the edge costs along the path, that is, $c(x_1,x_2) + c(x_2,x_3) + \ldots + c(x_{p-1},x_p)$. Given any two nodes $x$ and $y$, there are typically many paths between the two nodes, with each path having a cost. One or more of these paths is a **least-cost path**. The least-cost problem is therefore clear: Find a path between the source and destination that has least cost. In Figure 4.27, for example, the least-cost path between source node $u$ and destination node $w$ is $(u, x, y, w)$ with a path cost of 3. Note that if all edges in the

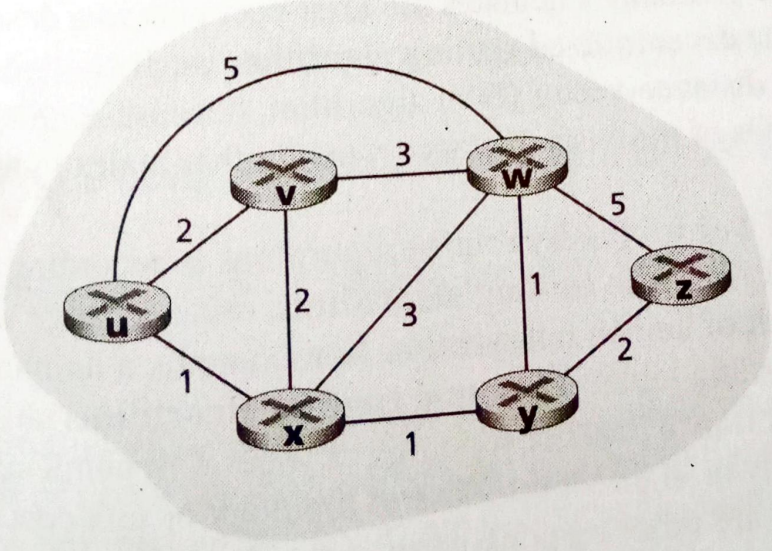



Figure 4.27 ♦ Abstract graph model of a computer network

sensitive algorithm, link costs vary dynamically to reflect the current level of congestion in the underlying link. If a high cost is associated with a link that is currently congested, a routing algorithm will tend to choose routes around such a congested link. While early ARPAnet routing algorithms were load-sensitive [McQuillan 1980], a number of difficulties were encountered [Huitema 1998]. Today's Internet routing algorithms (such as RIP, OSPF, and BGP) are **load-insensitive**, as a link's cost does not explicitly reflect its current (or recent past) level of congestion.

### 4.5.1 The Link-State (LS) Routing Algorithm

Recall that in a link-state algorithm, the network topology and all link costs are known, that is, available as input to the LS algorithm. In practice this is accomplished by having each node broadcast link-state packets to *all* other nodes in the network, with each link-state packet containing the identities and costs of its attached links. In practice (for example, with the Internet's OSPF routing protocol, discussed in Section 4.6.1) this is often accomplished by a **link-state broadcast** algorithm [Perlman 1999]. We'll cover broadcast algorithms in Section 4.7. The result of the nodes' broadcast is that all nodes have an identical and complete view of the network. Each node can then run the LS algorithm and compute the same set of least-cost paths as every other node.

The link-state routing algorithm we present below is known as *Dijkstra's algorithm*, named after its inventor. A closely related algorithm is Prim's algorithm; see [Cormen 2001] for a general discussion of graph algorithms. Dijkstra's algorithm computes the least-cost path from one node (the source, which we will refer to as $u$) to all other nodes in the network. Dijkstra's algorithm is iterative and has the property that after the $k$th iteration of the algorithm, the least-cost paths are known to $k$ destination nodes, and among the least-cost paths to all destination nodes, these $k$ paths will have the $k$ smallest costs. Let us define the following notation:

- $D(v)$: cost of the least-cost path from the source node to destination $v$ as of this iteration of the algorithm.
- $p(v)$: previous node (neighbor of $v$) along the current least-cost path from the source to $v$.
- $N'$ : subset of nodes; $v$ is in $N'$ if the least-cost path from the source to $v$ is definitively known.

The global routing algorithm consists of an initialization step followed by a loop. The number of times the loop is executed is equal to the number of nodes in the network. Upon termination, the algorithm will have calculated the shortest paths from the source node $u$ to every other node in the network.

| step | N′ | D(v),p(v) | D(w),p(w) | D(x),p(x) | D(y),p(y) | D(z),p(z) |
|---|---|---|---|---|---|---|
| 0 | u | 2,u | 5,u | 1,u | ∞ | ∞ |
| 1 | ux | 2,u | 4,x | | 2,x | ∞ |
| 2 | uxy | 2,u | 3,y | | | 4,y |
| 3 | uxyv | | 3,y | | | 4,y |
| 4 | uxyvw | | | | | 4,y |
| 5 | uxyvwz | | | | | |

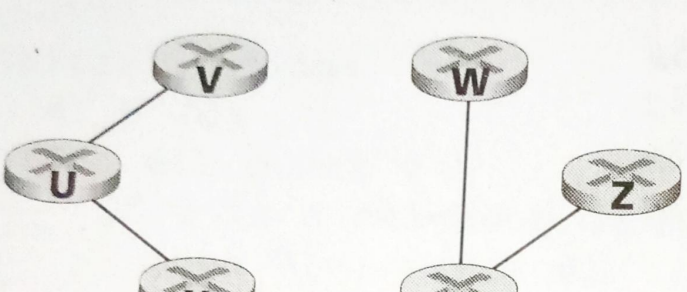
Table 4.3 • Running the link-state algorithm on the network in Figure 4.27

are updated via line 12 of the LS algorithm, yielding the results shown in the third row in the Table 4.3.

And so on. . . .

When the LS algorithm terminates, we have, for each node, its predecessor along the least-cost path from the source node. For each predecessor, we also have *its* predecessor, and so in this manner we can construct the entire path from the source to all destinations. The forwarding table in a node, say node $u$, can then be constructed from this information by storing, for each destination, the next-hop node on the least-cost path from $u$ to the destination. Figure 4.28 shows the resulting least-cost paths and forwarding table in $u$ for the network in Figure 4.27.

What is the computational complexity of this algorithm? That is, given $n$ nodes (not counting the source), how much computation must be done in the worst case to find the least-cost paths from the source to all destinations? In the first iteration, we need to search through all $n$ nodes to determine the node, $w$, not in $N'$ that has the minimum cost. In the second iteration, we need to check $n - 1$ nodes to determine the minimum cost; in the third iteration $n - 2$ nodes, and so on. Overall, the total number of nodes we need to search through over all the iterations is $n(n + 1)/2$, and thus we say that the preceding implementation of the LS algorithm




| Destination | Link |
|---|---|
| v | (u, v) |
| w | (u, x) |
| x | (u, x) |
| y | (u, x) |
| z | (u, x) |

**Figure 4.28** • Least costs paths and forwarding table for nodule u

unit of traffic destined for *w*, and node *y* injects an amount of traffic equal to *e*, also destined for *w*. The initial routing is shown in Figure 4.29(a) with the link costs corresponding to the amount of traffic carried.

When the LS algorithm is next run, node *y* determines (based on the link costs shown in Figure 4.29(a)) that the clockwise path to *w* has a cost of 1, while the counterclockwise path to *w* (which it had been using) has a cost of 1 + *e*. Hence *y*'s least-cost path to *w* is now clockwise. Similarly, *x* determines that its new least-cost path to *w* is also clockwise, resulting in costs shown in Figure 4.29(b). When the LS algorithm is run next, nodes *x*, *y*, and *z* all detect a zero-cost path to *w* in the counterclockwise direction, and all route their traffic to the counterclockwise routes. The next time the LS algorithm is run, *x*, *y*, and *z* 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 an LS 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