To calculate correctly the weights of the paths flowing through each vertex in a network, we need first to calculate the total number of shortest paths from each vertex to s. This is actually quite straightforward to do: the shortest paths from s to a vertex i must pass through one or more neighbors of i and the total number of shortest paths to i is simply the sum of the numbers of shortest paths to each of those neighbors. We can calculate these sums as part of a modified breadth-first search process as follows.

Consider Fig. 10.3b and suppose we are starting at vertex s. We carry out the following steps:

1. Assign vertex s distance zero, to indicate that it is zero steps from itself, and set $d = 0$. Also assign s a weight $w_s = 1$ (whose purpose will become clear shortly).
2. For each vertex i whose assigned distance is d, follow each attached edge to the vertex j at its other end and then do one of the following three things:
   a) If j has not yet been assigned a distance, assign it distance $d + 1$ and weight $w_j = w_i$.
   b) If j has already been assigned a distance and that distance is equal to $d + 1$, then the vertex's weight is increased by w_i, that is $w_j \leftarrow w_j + w_i$.
   c) If j has already been assigned a distance less than $d + 1$, do nothing.
3. Increase d by 1.
4. Repeat from step 2 until there are no vertices that have distance d.

The resulting weights for the example of Fig. 10.3b are shown to the left of each vertex in the figure. Each weight is the sum of the ones above it in the "tree." (It may be helpful to work through this example yourself by hand to see how the algorithm arrives at these values for the weights.) Physically, the weight on a vertex i represents the number of distinct geodesic paths between the source vertex s and i.

Now if two vertices i and j are connected by a directed edge in the shortest path "tree" pointing from j to i, then the fraction of the paths to s that pass through (or starting at) j and that also pass through i is given by w_i/w_j.

Thus, and finally, to calculate the contribution to the betweenness from shortest paths starting at all vertices and ending at s, we need only carry out the following steps:

1. Find every "leaf" vertex t, i.e., a vertex such that no paths from s to other vertices go though t, and assign it a score of $x_t = 1$.
2. Now, starting at the bottom of the tree, work up towards s and assign to each vertex i a score $x_i = 1 + \sum_j x_j w_i/w_j$, where the sum is over the neighbors j immediately below vertex i.

3. Repeat from step 2 until vertex s is reached.

The resulting scores are shown to the right of each vertex in Fig. 10.3b. Now repeating this process for all n source vertices s and summing the resulting scores on the vertices gives us the total betweenness scores for all vertices in time $O(n(m+n))$.[11]

This algorithm again takes time $O(n(m+n))$ in general or $O(n^2)$ on a sparse network, which is the best known running time for any betweenness algorithm at the time of writing, and moreover seems unlikely to be beaten by any future algorithm given that the calculation of the betweenness necessarily requires us to find shortest paths between all pairs of vertices, which operation also has time complexity $O(n(m+n))$. Indeed, even if we want to calculate the betweenness of only a single vertex it seems unlikely we can do better given that such a calculation still requires us to find all shortest paths.

10.4 SHORTEST PATHS IN NETWORKS WITH VARYING EDGE LENGTHS

In Section 6.3 we discussed weighted networks, networks in which the edges have values or strengths representing, for instance, the traffic capacities of connections on the Internet or the frequencies of contacts between acquaintances in a social network. In some cases the values on edges can be interpreted as lengths for the edges. The lengths could be real lengths, such as distances along roads in a road network, or they could represent quantities that act like lengths, such as transmission delays for packets traveling along Internet connections. In other cases they might just be approximately length-like measures: one might say, for instance, that a pair of acquaintances in a social network are twice as far apart as another pair if they see one another half as often.

Sometimes with networks such as these we would like to calculate the shortest path between two vertices taking the lengths of the edges into account. For instance, we might want to calculate the shortest driving route from A to B via a road network or we might want to calculate the route across the Internet that gets a data packet to its destination in the shortest time. (In fact, this is exactly what many Internet routers do when routing data packets.)

But now we notice a crucial—and annoying—fact. The shortest path across a network when we take edge lengths into account may not the be same as the shortest path in terms of number of edges. Consider Fig. 10.4. The shortest path between s and t in this small network traverses four edges, but is

[11] As discussed in footnote 9, these scores give the betweenness as defined in Eq. (7.36). To get true path counts one would have to divide by two and add a half (or equivalently add one then divide by two) to correct for the double counting of paths between distinct vertices.

still shorter, in terms of total edge length, than the competing path with just two edges. Thus we cannot find the shortest path in such a network using standard breadth-first search, which finds paths with the minimum number of edges. For problems like this we need a different algorithm. We need *Dijkstra's algorithm*.

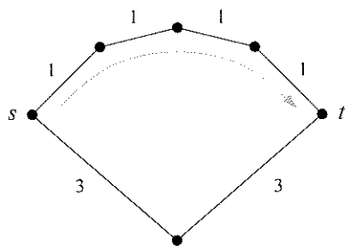

Figure 10.4: The shortest path in a network with varying edge lengths. The numbers on the edges in this network represent their lengths. The shortest path between s and t, taking the lengths into account, is the upper path marked with the arrow (which has total length 4), even though it traverses more edges than the alternative, lower path (which has length 6).

Dijkstra's algorithm, like breadth-first search, finds the shortest distance from a given source vertex s to every other vertex in the same component of a network, but does so taking the lengths of edges into account.[12] It works by keeping a record of the shortest distance it has found so far to each vertex and updating that record whenever a shorter one is found. It can be shown that, at the end of the algorithm, the shortest distance found to each vertex is in fact the shortest distance possible by any route. In detail the algorithm is as follows.

We start by creating an array of n elements to hold our current estimates of the distances from s to every vertex. At all times during the running of the algorithm these estimates are upper bounds on the true shortest distances. Initially we set our estimate of the distance from s to itself to be zero, which is trivially correct, and from s to every other vertex to be ∞, which is clearly a safe upper bound.

We also create another array of n elements in which we record whether we are certain that the distance we have to a given vertex is the smallest possible distance. For instance, we might use an integer array with 1s to indicate the distances we are sure about and 0s for the distances that are just our best current estimate. Initially, we put a 0 in every element of this array. (You might argue that we know for certain that the distance from s to itself is zero and hence that we should put a 1 in the element corresponding to vertex s. Let us, however, pretend that we don't know this to begin with, as it makes the algorithm work out more neatly.)

Now we do the following.

1. We find the vertex v in the network that has the smallest estimated distance from s, i.e., the smallest distance about which we are not yet certain.

[12] We assume that the lengths are all non-negative. If lengths can be negative, which happens in some cases, then the problem is much harder, falling in the class of "NP-complete" computational problems, for which even the best known algorithms take an amount of time exponential in n to finish, in the worst case [8]. Indeed, if edges are allowed to have negative lengths, there may not be any shortest path between a pair of vertices, since one can have a loop in the network that has negative length, so that one can reduce the length of a path arbitrarily by going around the loop repeatedly.

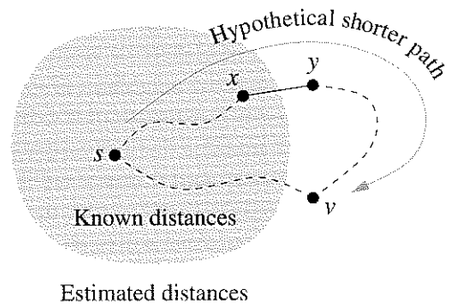

Figure 10.5: Paths in Dijkstra's algorithm. If v is the vertex with the smallest estimated (i.e., not certain) distance from s then that estimated distance must in fact be the true shortest distance to v. If it were not and there were a shorter path $s, \ldots, x, y, \ldots, v$ then all points along that path must have shorter distances from s than v's estimated distance, which means that y has a smaller estimated distance than v, which is impossible.

2. We mark this distance as being certain.
3. We calculate the distances from s via v to each of the neighbors of v by adding to v's distance the lengths of the edges connecting v to each neighbor. If any of the resulting distances is smaller than the current estimated distance to the same neighbor, the new distance replaces the older one.
4. We repeat from step 1 until the distances to all vertices are flagged as being certain.

Simple though it is to describe, it's not immediately obvious that this algorithm does what it is supposed to do and finds true shortest paths. The crucial step is step 2 where we declare the current smallest estimated distance in fact to be certain. That is, we claim that among vertices for which we don't yet definitely know the distance, the smallest distance recorded to any vertex is in fact the smallest possible distance to that vertex.

To see why this is true consider such a vertex, which we'll again call v, and consider a hypothetical path from s to v that has a shorter length than the current estimated distance recorded for v. The situation is illustrated in Fig. 10.5. Since this hypothetical path is shorter than the estimated distance to v, the distance along the path to each vertex in the path must also be less than that estimated distance.

Furthermore, there must exist somewhere along the path a pair of adjacent vertices x, y such that x's distance is known for certain and y's is not. Vertex x need not necessarily be distinct from vertex s (although we have drawn it that

way in the figure), but vertex y must be distinct from v: if y and v were the same vertex, so that v was a neighbor of x, then we would already have found the shorter path to v when we explored the neighbors of x in step 3 above and we would accordingly have revised our estimate of v's distance downward. Since this hasn't happened, y and v must be distinct vertices.

But notice now that y's current estimated distance will be at most equal to its distance from s *along the path* because that distance is calculated in step 3 above when we explore x's neighbors. And since, as we have said, all distances along the path are necessarily less than the current estimated distance to v, it follows that y's estimated distance must be less than v's and we have a contradiction, because v is by hypothesis the vertex with the shortest estimated distance. Hence there is no path to vertex v with length less than v's current estimated distance, so we can safely mark that distance as being certain, as in step 2 above.

Thus on each step the algorithm correctly flags one additional distance as being known exactly and when all distances have been so flagged the algorithm has done its job.

As with breadth-first search, the running time of Dijkstra's algorithm depends on how it is implemented. The simplest implementation is one that searches through all vertices on each round of the algorithm to find the one that has the smallest estimated distance. This search takes time $O(n)$. Then we must calculate a new estimated distance to each of the neighbors of the vertex we find, of which there are $O(m/n)$ on average. To leading order, one round thus takes time $O(m/n + n)$ and the whole algorithm, which runs (in the worst case of a network with a single component) for n rounds, takes time $O(m + n^2)$ to find the distance from s to every other vertex.

But we can do better than this. If we store the estimated distances in a binary heap (see Section 9.7) then we can find the smallest one and remove it from the heap in time $O(\log n)$. The operation of replacing an estimated distance with a new and better estimate (which in the worst case we have to do an average of $O(m/n)$ times per round) also takes $O(\log n)$ time, and hence a complete round of the algorithm takes time $O((m/n)\log n + \log n)$ and all n rounds then take $O((m + n)\log n)$, or $O(n \log n)$ on a sparse network with $m \propto n$. This is very nearly the best running time known for this problem,[13] and close to, though not quite as good as, the $O(m + n)$ for the equivalent problem on an unweighted network (factors of $\log n$ being close to constant given that

[13] In theory one can achieve a slightly better running time of $O(m + n \log n)$ using a data structure known as a Fibonacci heap [81], but in practice the operation of the Fibonacci heap is so complicated that the calculation usually ends up running slower.

the logarithm is a very slowly growing function of its argument).

As we have described it, Dijkstra's algorithm finds the shortest distance from a vertex s to every other in the same component but, like breadth-first search, it can be modified also to find the actual paths that realize those distances. The modification is very similar to the one for breadth-first search. We maintain a shortest path tree, which is initially empty and to which we add directed edges pointing from the vertices along the first step of their shortest path to s. We create such a directed edge when we first assign a vertex an estimated distance less than ∞ and move the edge to point to a new vertex every time we find a new estimated distance that is less than the current one. The last position in which an edge comes to rest indicates the true first step in the shortest path. If a new estimate of the distance to a vertex is ever exactly the same as the current estimate then we put two directed edges in the shortest path tree indicating the two alternative paths that give the shortest distance. When the algorithm is finished the shortest path tree, like those in Fig. 10.2, can be used to reconstruct the shortest paths themselves, or to calculate other quantities such as a weighted version of betweenness centrality (which could be used for instance as a measure of traffic flow in a network where traffic always takes the shortest weighted path).

10.5 MAXIMUM FLOWS AND MINIMUM CUTS

In Section 6.12 we discussed the ideas of connectivity, independent paths, cut sets, and maximum flows in networks. In particular, we defined two paths that connect the same vertices s and t to be edge-independent if they share none of the same edges and vertex-independent if they share none of the same vertices except for s and t themselves. And the edge or vertex connectivity of the vertices is the number of edge- or vertex-independent paths between them. We also showed that the edge or vertex connectivity is equal to the size of the minimum edge or vertex cut set—the minimum number of edges or vertices that need to be removed from the network to disconnect s from t. Connectivity is thus a simple measure of the robustness of the connection between a pair of vertices. Finally, we showed that the edge-connectivity is also equal to the maximum flow that can pass from s to t if we think of the network as a network of pipes, each of which can carry one unit of flow.

In this section we look at algorithms for calculating maximum flows between vertices on networks. As we will see, there is a simple algorithm, the Ford–Fulkerson or augmenting path algorithm, that calculates the maximum flow between two vertices in average time $O((m + n)m/n)$. Once we have this maximum flow, then we also immediately know the number of edge-