

## Chapter 8: An Introduction to Networks

Network models are an extremely important category of mathematical program that have numerous practical applications. Part of their appeal is the direct and intuitive mapping between the real world, the network diagram, and the underlying solution algorithms. Many network problems can be solved via linear programming, and in fact, special extremely fast variants of linear programming can be applied. The largest mathematical programs that are regularly solved in practice, e.g. airline crew scheduling problems, are usually network problems.

For many types of network problems there are also specialized non-LP solution algorithms. We will first look at some of the classic non-LP solution methods, and later return to the idea of solving networks via LP.

### Basic Definitions

Network models are created from two major building blocks: *arcs* (sometimes called *edges*), which are connecting lines, and *nodes*, which are the connecting points for the arcs. A *graph* is a structure that is built by interconnecting nodes and arcs. A *directed graph* (often called a *digraph*) is a graph in which the arcs have specified directions, as shown by arrowheads. Finally, a *network* is a graph (or more commonly a digraph) in which the arcs have an associated *flow*. Some example diagrams are given in Figure 8.1.

Here are some simple examples of networks:

nodes	arcs	flow
cities	highways	vehicles
call switching centers	telephone lines	telephone calls
pipe junctions	pipes	water

There are some further definitions associated with graphs and networks:

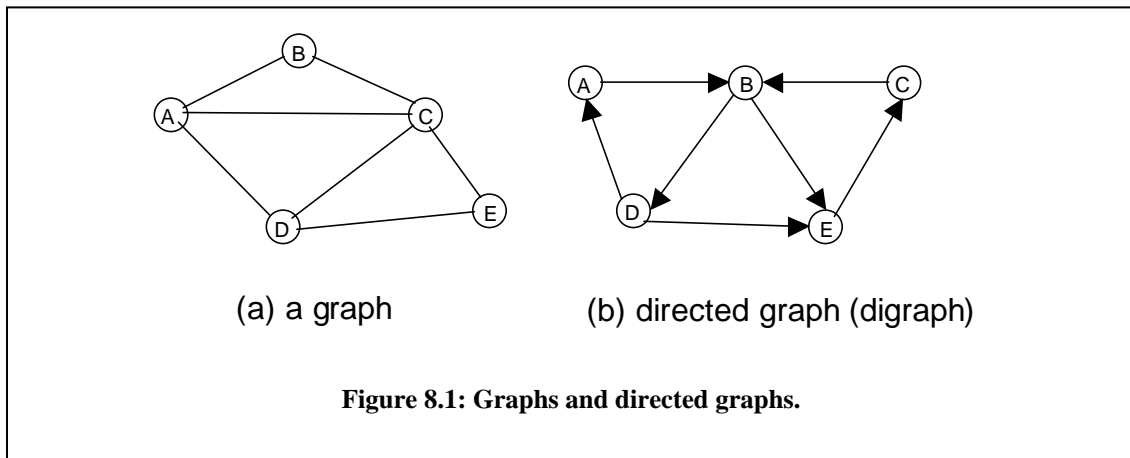
*chain*: a sequence of arcs connecting two nodes  $i$  and  $j$ . For example, in Figure 8.1(a), we might connect nodes A and E via the chains ABCE or ADCE.

*path*: a sequence of directed arcs connecting two nodes. In this case, you must respect the arc directions. For example, in Figure 8.1(b), a path from A to E might be ABDE, but the chain ABCE is *not* a path because it traverses arc BC in the wrong direction.

*cycle*: a chain that connecting a node to itself without any retracing. For example, in Figure 8.1(a), ABCEDA is a cycle, but ABCDECBA is *not* a cycle because of the double traversal of arcs AB and BC.

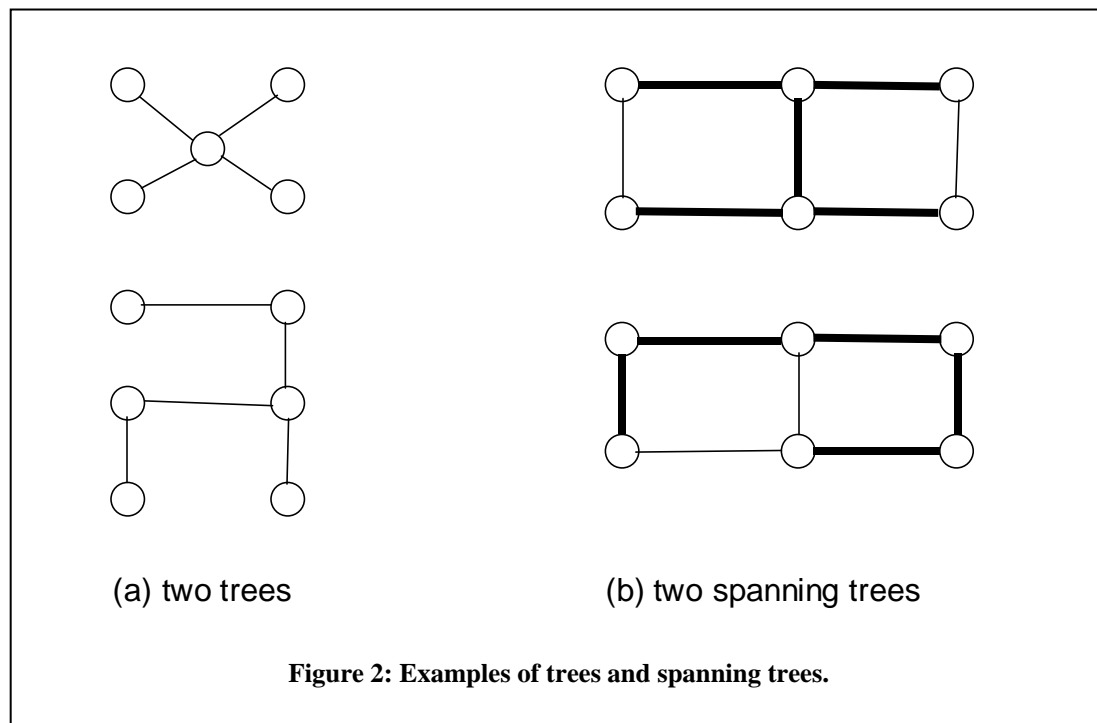
*connected graph (or connected network)*: has just one part. In other words you can reach any node in the graph or network via a chain from any other node. It is sometimes

important to know whether a graph is connected and there are efficient computer algorithms for checking this.



*tree*: a connected graph having no cycles. Some examples are shown in Figure 8.2(a).

*spanning tree*: normally a tree selected from among the arcs in a graph or network so that all of the nodes in the tree are connected. See Figure 8.2(b). Spanning trees have interesting applications in services layout, for example, finding a way to lay out the computer cable connecting all of the buildings on a campus (nodes) by selecting from among the possible inter-building connections (arcs).



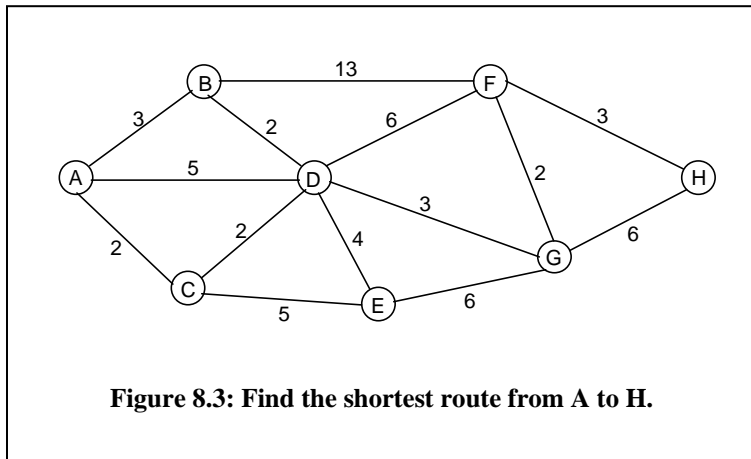
*flow capacity*: an upper (and sometimes lower) limit on the amount of flow in an arc in a network. For example, the maximum flow rate of water in a pipe, or the maximum simultaneous number of calls on a telephone connection.

*source (or source node)*: a node which introduces flow into a network. This happens at the boundary between the network under study and the external world.

*sink (or sink node)*: a node which removes from the network. This happens at the boundary between the network under study and the external world.

## The Shortest Route Problem

Here is a technical statement of the shortest route problem: given a graph in which each arc is labeled with the distance between the two nodes that it connects, what is the shortest route between some node  $i$  and some other node  $j$  in the graph? For example, consider the graph in Figure 8.3: what is the shortest route between node A and node H? What is the length of the shortest route?



In a graph this small, it is possible to solve the problem by simple inspection: try this for the graph in Figure 8.3. However, if the graph is quite large, a more organized approach is needed. Brute force enumeration of all of the possible routes is impractical in large graphs because the number of possible routes is combinatorially explosive. Try tracing out all of the routes between nodes A and H in Figure 8.3 for example: there are quite a few!

Note that “shortest route problem” is also the name applied when the arc labels are not distances, but perhaps travel time (quickest route problem) or travel cost (cheapest route problem). The solution methods are identical in all cases.

We will look first at one of the most popular methods for finding the shortest route in a graph or network: *Dijkstra’s algorithm*. It is an iterative method. At the  $n$ th iteration you will find the  $n$ th closest node to the start node (the *origin*), and the shortest route to that node. You stop iterating when the destination node is reached, even if you have not visited all of the nodes in the network.

All of the nodes in the network are initially “unsolved”. A node is “solved” at each iteration when the shortest distance and route to that node is found. The length of the shortest path from the origin to a node is the “distance”: a node is solved when the distance is determined. By definition (and common sense), the distance from the origin

node to the origin node is zero. Arcs which may form part of the shortest route are gradually added to the “arc set” as the method proceeds; the arc set is initially empty. Note that the method ventures up some blind alleys as it proceeds: not all of the arcs that are added to the arc set will feature in the final shortest route!

Here is a statement of Dijkstra’s algorithm:

0. To initialize:
  - a. the origin is the only solved node.
  - b. the distance to the origin is 0.
  - c. the arc set is empty.
1. Find all the unsolved nodes that are directly connected by a single arc to any solved node. For each unsolved node, calculate all of the “candidate distances” (there may be several of these for one unsolved node because it may directly connect to more than one solved node):
  - a. choose an arc connecting the unsolved node directly to a solved node.
  - b. the “candidate distance” is (distance to solved node) + (length of arc directly connecting the solved and the unsolved node)
2. Choose the smallest candidate distance:
  - a. add the corresponding unsolved node to the solved set.
  - b. distance to the newly solved node is the candidate distance.
  - c. add the corresponding arc to the arc set.
3. If the newly solved node is not the destination node, then go to step 1. Else, stop and recover the solution:
  - a. length of shortest route from origin to destination is the distance to the destination node.
  - b. shortest route is found by tracing backwards from the destination node to the origin (or the reverse) using the arcs in the arc set. It is usually easiest to trace backwards because each node is reached from exactly one other node, but may have outward arcs to several other nodes.

You can keep track of the progress of the algorithm by constructing a table with headings such as iteration number, unsolved node connected to solved node, candidate distances and arcs, selected node, distance and arc. However, for relatively small problems it is easier to keep track by making notations directly on the diagram. We will use the following conventions for tracking the algorithm on a diagram:

- mark the solved nodes in boldface and label them with the distance,
- mark the arcs in the arc set in boldface.

Let us follow the progress of Dijkstra's algorithm as we apply it to the network shown in Figure 8.3. We will omit the initial diagram in which only node A, the origin, is solved, and so shown in boldface and labeled with the distance 0. At this point, there are 3 unsolved nodes to consider (B, C, and D) because they are directly connected to the only solved node (A). The candidate distances are AB ( $0+3=3$ ), AC ( $0+2=2$ ), and AD ( $0+5=5$ ). The smallest candidate distance is AC, so C is now a solved node with a distance of 2 as shown in Figure 8.4.

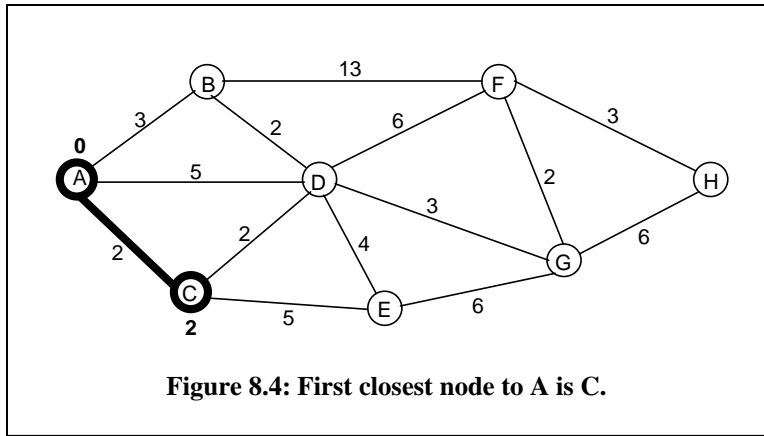


Figure 8.4: First closest node to A is C.

Now there are 3 unsolved nodes to consider (B, D, and E). Candidate distances are AB ( $0+3=3$ ), AD ( $0+5=5$ ), CD ( $2+2=4$ ), and CE ( $2+5=7$ ). The smallest candidate distance is AB at 3, so B is now a solved node, as shown in Figure 8.5.

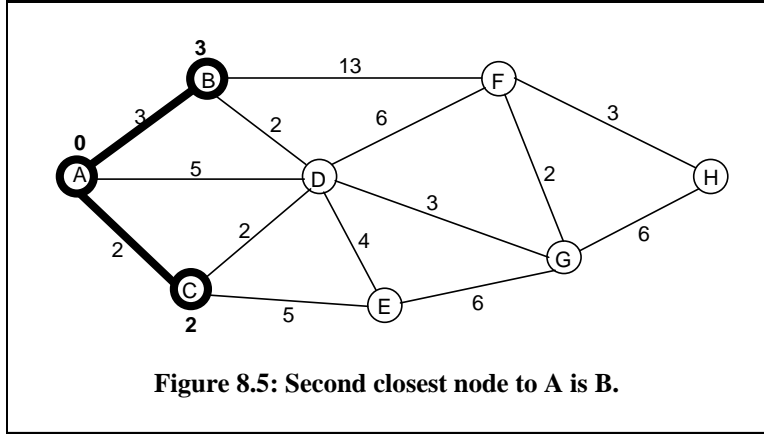


Figure 8.5: Second closest node to A is B.

There are again 3 unsolved nodes to consider (F, D, and E). Candidate distances are BF ( $3+13=16$ ), BD ( $3+2=5$ ), AD ( $0+5=5$ ), CD ( $2+2=4$ ), and CE ( $2+5=7$ ). The smallest candidate distance is CD at 4, so D is now a solved node, as shown in Figure 8.6.

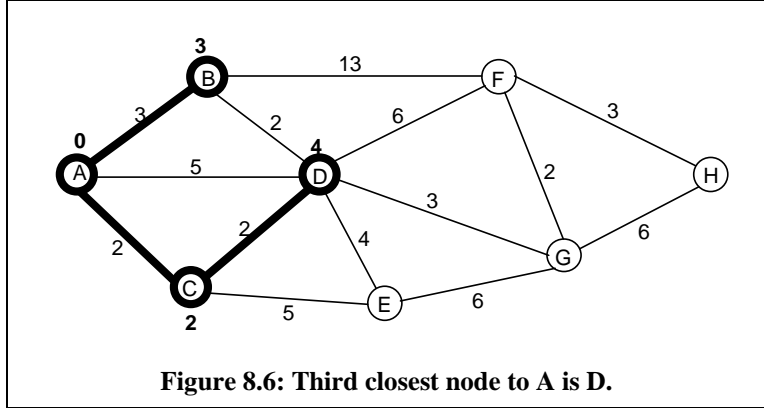


Figure 8.6: Third closest node to A is D.

Now the unsolved nodes to consider are F, G, and E. Candidate distances are BF ( $3+13=16$ ), DF ( $4+6=10$ ), DG ( $4+3=7$ ), DE ( $4+4=8$ ), and CE ( $2+5=7$ ). The smallest candidate distance is 7 with a

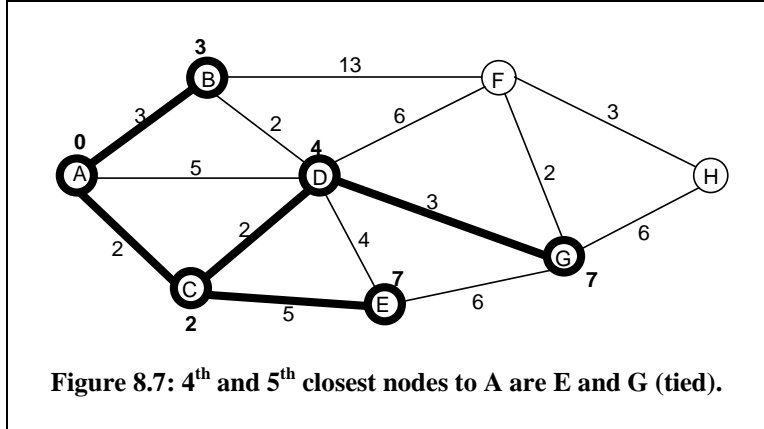
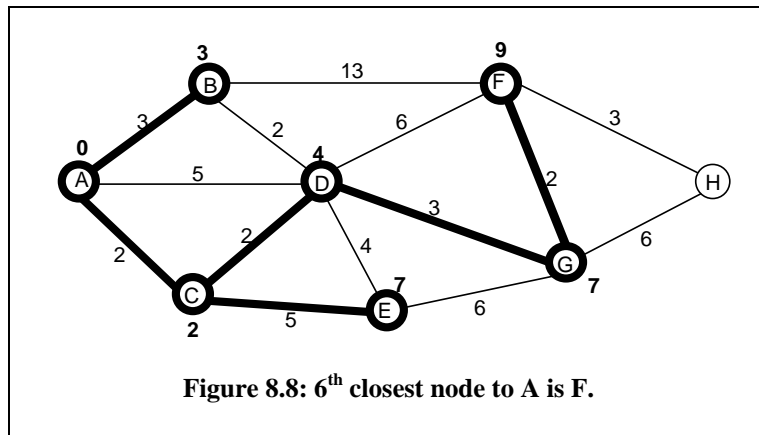


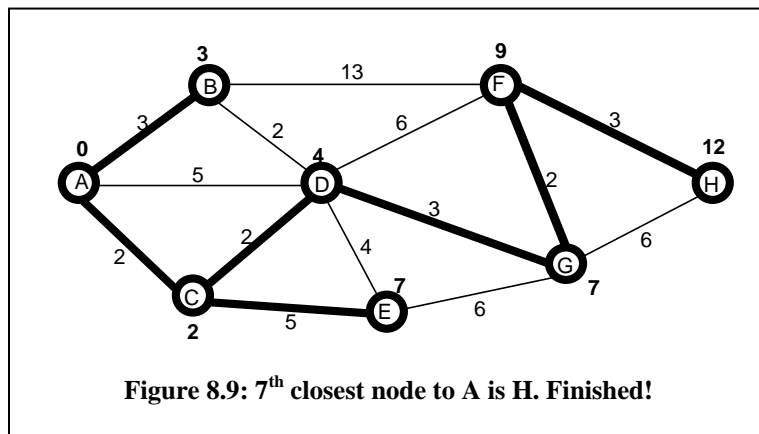
Figure 8.7: 4<sup>th</sup> and 5<sup>th</sup> closest nodes to A are E and G (tied).

tie between DG and CE. We can choose either candidate arbitrarily and update the diagram accordingly, however we know that the candidate not chosen this time will definitely be chosen in the next iteration, so we might as well update both candidates at once, as shown in Figure 8.7.



Now there are only two unsolved nodes to consider: F and H. The candidate distances are BF ( $3+13=16$ ), DF ( $4+6=10$ ), GF ( $7+2=9$ ), and GH ( $7+6=13$ ). The smallest candidate distance is 9, so F is now a solved node, as shown in Figure 8.8.

Finally there is only one remaining unsolved node: H. The candidate distances are FH ( $9+3=12$ ) and GH ( $7+6=13$ ). So H becomes a solved node with a distance of 12, as shown in Figure 8.9. Since H is the destination node, the iterations stop. We visited every node in the network in this small example, but this will not always happen.



At this point, we know that the length of the shortest route from A to H is 12 units (kilometers perhaps). But how do we determine the route? Simply trace backwards from H to A, and then reverse the route that you find in this manner. In Figure 8.9, the reverse route is HFGDCA, and when given in the forward order this is ACDGFH. The reason for finding the route in this manner is that working backwards avoids the blind alleys that exist in the forward direction.

Note that you may sometimes find a tie in the routing to an intermediate node. For example, if arc BD had a length of 1, then there would be two routes from A to D with length 4: ACD as selected in the diagram, and ABD. It doesn't matter which one you use. In fact, you can mark both choices on the diagram as you proceed, which will remind you of the alternate routes.

What makes Dijkstra's algorithm efficient is that it calculates the lengths of only a very small subset of all of the possible routes through the graph. Once a node has been solved, the shortest path to the node from the origin is known. All paths that follow on from that

node then have the benefit of the shortest path to that point. The method is actually a specific implementation of a dynamic programming algorithm (covered later).

It is easy to modify Dijkstra's algorithm for use on directed graphs. You simply need to respect the arc directions when selecting candidate connections between solved and unsolved nodes. Think of it as finding the shortest route through a system of one-way streets.

## ***The Minimum Spanning Tree Problem***

The technical statement of the minimum spanning tree problem is simple: given a graph in which the arcs are labeled with the distances between the nodes that they connect, find a spanning tree which has the minimum total length. Recall that a spanning tree connects all of the nodes in the graph, and has no cycles.

As for the shortest route problem, the arc labels could as well be related to time or cost. There are many examples of applications of the minimum spanning tree problem:

- Find the least cost set of roadways among the possible set of roadways to connect a set of locations.
- Find the shortest total length of sewer to lay among the buildings in a planned subdivision, given the set of possible inter-building sewer routes.
- Find the minimum total length of telephone cable to connect all of the offices in a building, given the possible routings of cable between offices.

The algorithm for solving this problem is probably the simplest that you will ever learn in mathematical programming:

0. Initialize: all nodes are unsolved, no arcs are in the arc set.
1. Select any node arbitrarily and label it as solved. Connect this node to its nearest neighbour: label the neighbour solved and add the connecting arc to the arc set.
2. Identify the unsolved node that is closest to a solved node. Label this node as solved and add the connecting arc to the arc set.
3. If all nodes are solved, then stop. Else go to Step 2.

Upon termination of the algorithm, the minimum spanning tree is given by the arcs in the arc set, and the length of the minimum spanning tree is found by summing the lengths of the arcs in the arc set.

In the case of a tie for the next solved node, choose arbitrarily. You will obtain the same final optimum solution, but it indicates that there are multiple optima. You will need to go back and enumerate them all if you need them.

There is again a simple diagrammatic convention for keeping track of the steps of the algorithm: show the solved nodes and the arcs in the arc set in boldface as you proceed. Let us apply this convention while solving the minimum spanning tree problem for the graph in Figure 8.3 that we already used for the shortest route problem. Let us begin arbitrarily at node E, and label the nodes with the order in which they are solved, while keeping track of all of the notations on a copy of the graph in Figure 8.10.

The closest node to node E is node D at a distance of 4 units, so node D is the 2<sup>nd</sup> solved node.

Candidate unsolved nodes are now B (BD=2), A (AD=5), C (CD=2, CE=5), F (FD=6), and G (GD=3, GE=6). The smallest connecting length is 2 (CD or BD), so we choose CD arbitrarily. Node C is the 3<sup>rd</sup> solved node.

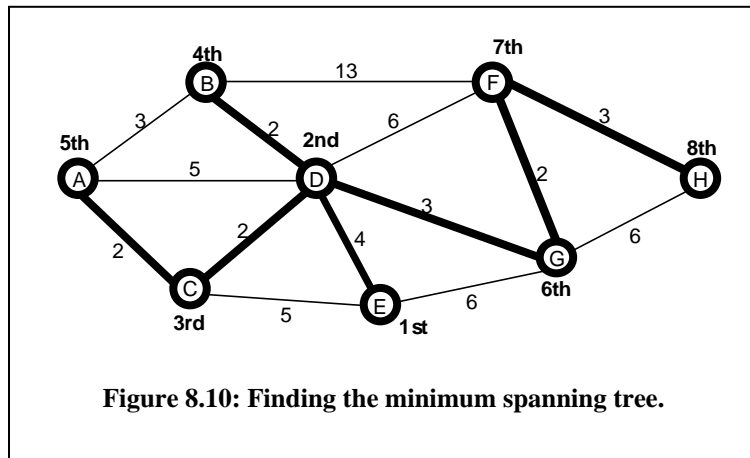


Figure 8.10: Finding the minimum spanning tree.

Nodes E, D, and C are now solved. Candidate unsolved nodes are A (AC=2, AD=5), B (BD=2), F (FD=6), and G (GD=3, GE=6). The smallest connecting length is 2 (AC or BD), so we choose BD arbitrarily. Node B is the 4<sup>th</sup> solved node.

Nodes E, D, C and B are now solved. Candidate unsolved nodes are A (AB=3, AD=5, AC=2), F (FB=13, FD=6) and G (GD=3, GE=6). The smallest connecting length is 2 (AC), so this is chosen. Node A is the 5<sup>th</sup> solved node.

Nodes E, D, C, B and A are now solved. Candidate unsolved nodes are F (FB=13, FD=6), and G (GD=3, GE=6). The smallest connecting length is 3 (GD), so node G is the 6<sup>th</sup> solved node.

Nodes E, D, C, B, A and G are now solved. Candidate unsolved nodes are F (FB=13, FD=6, FG=2), and H (HG=6). The smallest connecting length is 2 (FG), so F is the 7<sup>th</sup> solved node.

Finally, all nodes are solved except node H. The candidate connecting lengths are HF=3 and HG=6. HF is chosen, so H is the 8<sup>th</sup> and last solved node.

Now the solution can be recovered. The minimum spanning tree itself is *all* of the arcs in the arc set (i.e. all of the arcs shown in bold in Figure 8.10). Note how this differs from the shortest route solution in which not all of the arcs in the arc set are part of the final solution. The total length of the minimum spanning tree is found by summing the lengths of all of the arcs in the arc set: ED + DC + DB + CA + DG + GF + FH = 4 + 2 + 2 + 2 + 2 + 3 + 2 + 3 = 18. The total length of the minimum spanning tree is 18 units.

Note that you will get the identical total of 18 units no matter which node you choose as the initial node (try it yourself). There is more than one spanning tree that gives this same result, however. We know this because of the arbitrary choice we made in solving the 3<sup>rd</sup> node.

The solution method for the minimum spanning tree problem is an example of a *greedy algorithm*. A greedy algorithm does whatever is best at the current step, without ever considering what the impact might be on the overall problem. This is usually a bad idea in optimization because it leads to a solution that is less than optimal overall. However, just choosing the closest unsolved nodes leads to an overall optimum solution in this special case.

The algorithm for a *maximum* spanning tree is obvious: simply choose the *longest* solved-to-unsolved node connection at each step. You might want a maximum spanning tree in a case where profit is involved, for example, choosing television cable routings to connect a set of locations. The extension to directed graphs is also straightforward: you can only select from among arcs that connect in the direction that you are working (either away from the initial node, or towards the initial node).