12. Prove that when *dfs* is applied to a connected graph the edges of T form a tree.
13. Prove that when *bfs* is applied to a connected graph the edges of T form a tree.
14. An edge, (u, v), of a connected graph, G, is a *bridge iff* its deletion from G produces a graph that is no longer connected. In the graph of Figure 6.19, the edges (0, 1), (3, 5), (7, 8), and (7, 9) are bridges. Write a function that finds the bridges in a graph. Your function should have a time complexity of $O(n + e)$. (Hint: use *bicon* as a starting point.)
15. Using a complete graph with n vertices, show that the number of spanning trees is at least $2^{n-1} - 1$.

6.3 MINIMUM COST SPANNING TREES

The *cost* of a spanning tree of a weighted undirected graph is the sum of the costs (weights) of the edges in the spanning tree. A *minimum cost spanning tree* is a spanning tree of least cost. Three different algorithms can be used to obtain a minimum cost spanning tree of a connected undirected graph. All three use an algorithm design strategy called the *greedy method*. We shall refer to the three algorithms as Kruskal's, Prim's, and Sollin's algorithms, respectively.

In the greedy method, we construct an optimal solution in stages. At each stage, we make a decision that is the best decision (using some criterion) at this time. Since we cannot change this decision later, we make sure that the decision will result in a feasible solution. The greedy method can be applied to a wide variety of programming problems. Typically, the selection of an item at each stage is based on either a least cost or a highest profit criterion. A feasible solution is one which works within the constraints specified by the problem.

For spanning trees, we use a least cost criterion. Our solution must satisfy the following constraints:

(1) we must use only edges within the graph

(2) we must use exactly $n - 1$ edges

(3) we may not use edges that would produce a cycle.

6.3.1 Kruskal's Algorithm

Kruskal's algorithm builds a minimum cost spanning tree T by adding edges to T one at a time. The algorithm selects the edges for inclusion in T in nondecreasing order of their cost. An edge is added to T if it does not form a cycle with the edges that are already in T. Since G is connected and has $n > 0$ vertices, exactly $n - 1$ edges will be selected for inclusion in T.

Example 6.3: We will construct a minimum cost spanning tree of the graph of Figure 6.22(a). Figure 6.23 shows the order in which the edges are considered for inclusion, as well as the result and the changes (if any) in the spanning tree. For example, edge (0, 5) is the first considered for inclusion. Since it obviously cannot create a cycle, it is added to the tree. The result is the tree of Figure 6.22(c). Similarly, edge (2, 3) is considered next. It is also added to the tree, and the result is shown in Figure 6.22(d). This process continues until the spanning tree has $n-1$ edges (Figure 6.22(h)). The cost of the spanning tree is 99. □

Program 6.7 presents a formal description of Kruskal's algorithm. (We leave writing the C function as an exercise.) We assume that initially E is the set of all edges in G. To implement Kruskal's algorithm, we must be able to determine an edge with minimum cost and delete that edge. We can handle both of these operations efficiently if we maintain the edges in E as a sorted sequential list. As we shall see in Chapter 7, we can sort the edges in E in $O(e \log e)$ time. Actually, it is not necessary to sort the edges in E as long as we are able to find the next least cost edge quickly. Obviously a min heap is ideally suited for this task since we can determine and delete the next least cost edge in $O(\log e)$ time. Construction of the heap itself requires $O(e)$ time.

To check that the new edge, (v, w), does not form a cycle in T and to add such an edge to T, we may use the union-find operations discussed in Section 5.9. This means that we view each connected component in T as a set containing the vertices in that component. Initially, T is empty and each vertex of G is in a different set (see Figure 6.22(b)). Before we add an edge, (v, w), we use the find operation to determine if v and w are in the same set. If they are, the two vertices are already connected and adding the edge (v, w) would cause a cycle. For example, when we consider the edge (3, 2), the sets would be {0}, {1, 2, 3}, {5}, {6}. Since vertices 3 and 2 are already in the same set, the edge (3, 2) is rejected. The next edge examined is (1, 5). Since vertices 1 and 5 are in different sets, the edge is accepted. This edge connects the two components {1, 2, 3} and {5}. Therefore, we perform a union on these sets to obtain the set {1, 2, 3, 5}.

Since the union-find operations require less time than choosing and deleting an edge (lines 3 and 4), the latter operations determine the total computing time of Kruskal's algorithm. Thus, the total computing time is $O(e \log e)$. Theorem 6.1 proves that Program 6.7 produces a minimum spanning tree of G.

Theorem 6.1: Let G be an undirected connected graph. Kruskal's algorithm generates a minimum cost spanning tree.

Proof: We shall show that:

(a) Kruskal's method produces a spanning tree whenever a spanning tree exists.

(b) The spanning tree generated is of minimum cost.

For (a), we note that Kruskal's algorithm only discards edges that produce cycles. We know that the deletion of a single edge from a cycle in a connected graph produces a

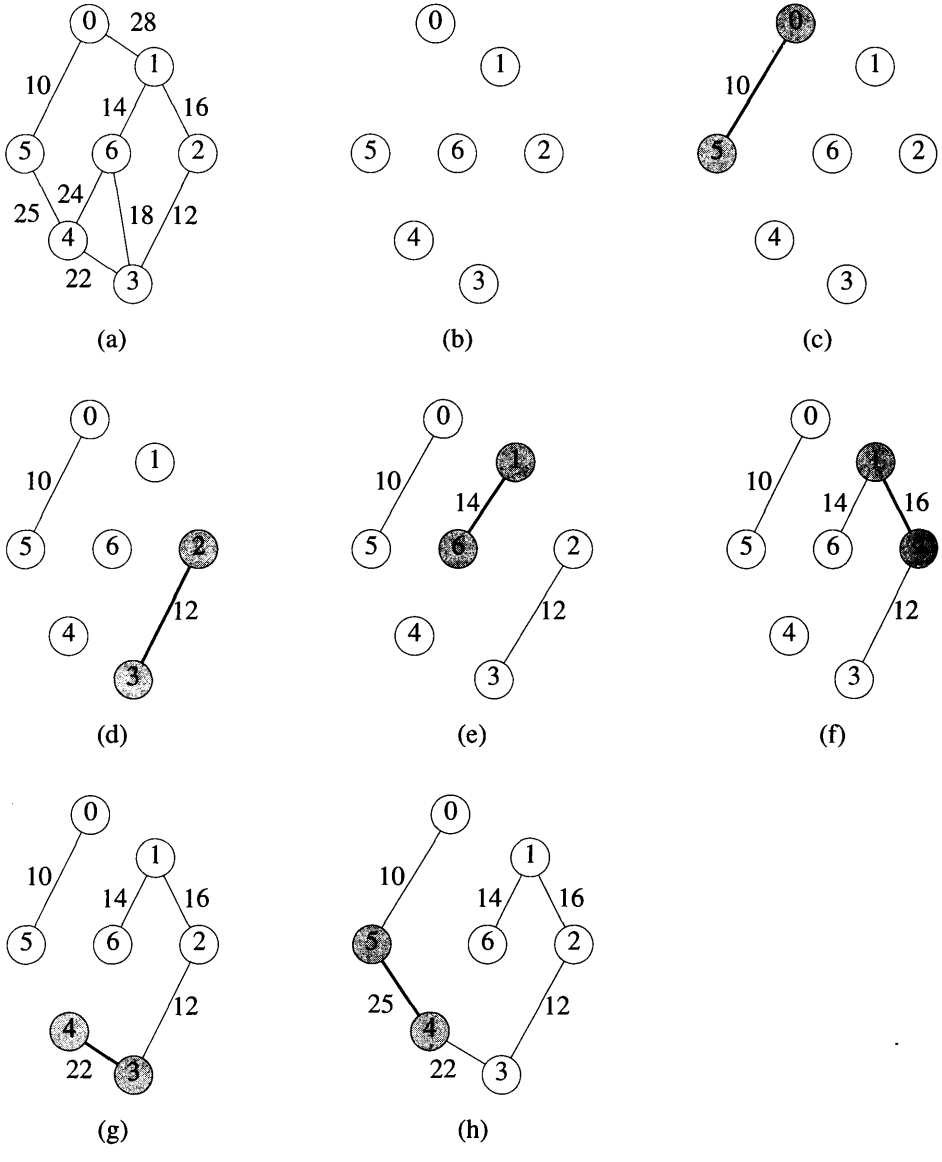

Figure 6.22: Stages in Kruskal's algorithm

Edge	Weight	Result	Figure
----	---	initial	Figure 6.22(b)
(0,5)	10	added to tree	Figure 6.22(c)
(2,3)	12	added	Figure 6.22(d)
(1,6)	14	added	Figure 6.22(e)
(1,2)	16	added	Figure 6.22(f)
(3,6)	18	discarded	
(3,4)	22	added	Figure 6.22(g)
(4,6)	24	discarded	
(4,5)	25	added	Figure 6.22(h)
(0,1)	28	not considered	

Figure 6.23: Summary of Kruskal's algorithm applied to Figure 6.22(a)

```
T = {};
while (T contains less than n-1 edges && E is not empty) {
   choose a least cost edge (v,w) from E;
   delete (v,w) from E;
   if ((v,w) does not create a cycle in T)
      add (v,w) to T;
   else
      discard (v,w);
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.7: Kruskal's algorithm

graph that is also connected. Therefore, if G is initially connected, the set of edges in T and E always form a connected graph. Consequently, if G is initially connected, the algorithm cannot terminate with $E = \{\}$ and $| T | < n - 1$.

Now let us show that the constructed spanning tree, T, is of minimum cost. Since G has a finite number of spanning trees, it must have at least one that is of minimum cost. Let U be such a tree. Both T and U have exactly $n - 1$ edges. If $T = U$, then T is of minimum cost and we have nothing to prove. So, assume that $T \neq U$. Let k, $k > 0$, be the number of edges in T that are not in U (k is also the number of edges in U that are not in

T).

We shall show that T and U have the same cost by transforming U into T. This transformation is done in k steps. At each step, the number of edges in T that are not in U is reduced by exactly 1. Furthermore, the cost of U is not changed as a result of the transformation. As a result, U after k transformation steps has the same cost as the initial U and contains exactly those edges that are in T. This implies that T is of minimum cost.

For each transformation step, we add one edge, e, from T to U and remove one edge, f, from U. We select the edges e and f in the following way:

(1) Let e be the least cost edge in T that is not in U. Such an edge must exist because $k > 0$.

(2) When we add e to U, we create a unique cycle. Let f be any edge on this cycle that is not in T. We know that at least one of the edges on this cycle is not in T because T contains no cycles.

Given the way e and f are selected, it follows that $V = U + \{e\} - \{f\}$ is a spanning tree and that T has exactly $k - 1$ edges that are not in V. We need to show that the cost of V is the same as the cost of U. Clearly, the cost of V is the cost of U plus the cost of the edge e minus the cost of the edge f. The cost of e cannot be less than the cost of f since this would mean that the spanning tree V has a lower cost than the tree U. This is impossible. If e has a higher cost than f, then f is considered before e by Kruskal's algorithm. Since it is not in T, Kruskal's algorithm must have discarded this edge at this time. Therefore, f together with the edges in T having a cost less than or equal to the cost of f must form a cycle. By the choice of e, all these edges are also in U. Thus, U must contain a cycle. However, since U is a spanning tree it cannot contain a cycle. So the assumption that e is of higher cost than f leads to a contradiction. This means that e and f must have the same cost. Hence, V has the same cost as U. □

6.3.2 Prim's Algorithm

Prim's algorithm, like Kruskal's, constructs the minimum cost spanning tree one edge at a time. However, at each stage of the algorithm, the set of selected edges forms a tree. By contrast, the set of selected edges in Kruskal's algorithm forms a forest at each stage. Prim's algorithm begins with a tree, T, that contains a single vertex. This may be any of the vertices in the original graph. Next, we add a least cost edge (u, v) to T such that $T \cup \{(u, v)\}$ is also a tree. We repeat this edge addition step until T contains $n - 1$ edges. To make sure that the added edge does not form a cycle, at each step we choose the edge (u, v) such that exactly one of u or v is in T. Program 6.8 contains a formal description of Prim's algorithm. T is the set of tree edges, and TV is the set of tree vertices, that is, vertices that are currently in the tree. Figure 6.24 shows the progress of Prim's algorithm on the graph of Figure 6.22(a).

```
T = {};
TV = {0}; /* start with vertex 0 and no edges */
while (T contains fewer than n-1 edges) {
   let (u, v) be a least cost edge such that u ∈ TV and
   v ∉ TV;
   if (there is no such edge)
      break;
   add v to TV;
   add (u, v) to T;
}
if (T contains fewer than n-1 edges)
   printf("No spanning tree\n");
```

Program 6.8: Prim's algorithm

To implement Prim's algorithm, we assume that each vertex v that is not in TV has a companion vertex, *near(v)*, such that *near(v)* $\in$ TV and *cost(near(v), v)* is minimum over all such choices for *near(v)*. (We assume that *cost(v, w)* = ∞ if $(v, w) \notin E$). At each stage we select v so that *cost(near(v), v)* is minimum and $v \notin TV$. Using this strategy we can implement Prim's algorithm in $O(n^2)$, where n is the number of vertices in G. Asymptotically faster implementations are also possible. One of these results from the use of Fibonacci heaps which we examine in Chapter 9.

6.3.3 Sollin's Algorithm

Unlike Kruskal's and Prim's algorithms, Sollin's algorithm selects several edges for inclusion in T at each stage. At the start of a stage, the selected edges, together with all n graph vertices, form a spanning forest. During a stage we select one edge for each tree in the forest. This edge is a minimum cost edge that has exactly one vertex in the tree. Since two trees in the forest could select the same edge, we need to eliminate multiple copies of edges. At the start of the first stage the set of selected edges is empty. The algorithm terminates when there is only one tree at the end of a stage or no edges remain for selection.

Figure 6.25 shows Sollin's algorithm applied to the graph of Figure 6.22(a). The initial configuration of zero selected edges is the same as that shown in Figure 6.22(b). Each tree in this forest is a a single vertex. At the next stage, we select edges for each of the vertices. The edges selected are (0, 5), (1, 6), (2, 3), (3, 2), (4, 3), (5, 0), (6, 1). After eliminating the duplicate edges, we are left with edges (0, 5), (1, 6), (2, 3), and (4, 3). We add these edges to the set of selected edges, thereby producing the forest of Figure

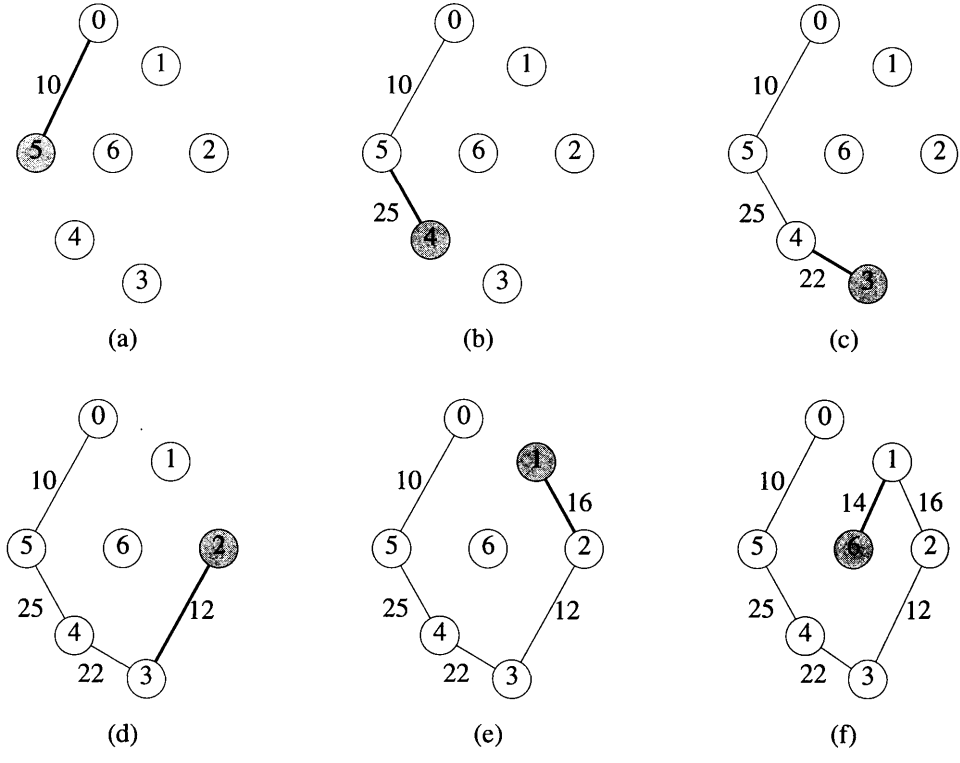

Figure 6.24: Stages in Prim's algorithm

6.25(a). In the next stage, the tree with vertex set {0, 5} selects edge (5, 4), and the two remaining trees select edge (1, 2). After these two edges are added, the spanning tree is complete, as shown in Figure 6.25(b). We leave the development of Sollin's algorithm into a C function and its correctness proof as exercises.

EXERCISES

1. Prove that Prim's algorithm finds a minimum cost spanning tree for every undirected connected graph.
2. Refine Prim's algorithm (Program 6.8) into a C function that finds a minimum cost spanning tree. The complexity of your function should be $O(n^2)$, where n is the number of vertices in the graph. Show that this is the case.

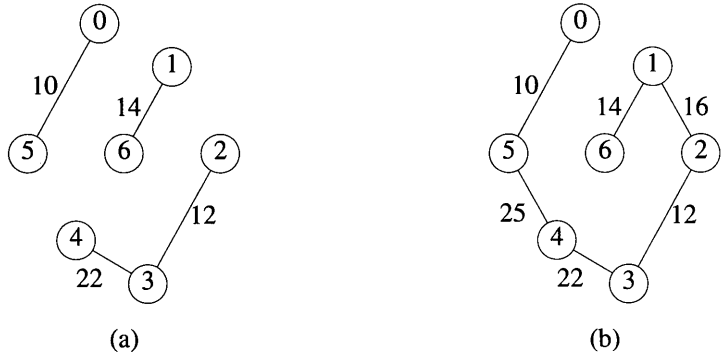

Figure 6.25: Stages in Sollin's algorithm

3. Prove that Sollin's algorithm finds a minimum cost spanning tree for every connected undirected graph.
4. What is the maximum number of stages in Sollin's algorithm? Give this as a function of the number of vertices, n, in the graph.
5. Write a C function that finds a minimum cost spanning tree using Sollin's algorithm. What is the complexity of your function?
6. Write a C function that finds a minimum cost spanning tree using Kruskal's algorithm. You may use the *union* and *find* functions from Chapter 5 and the *sort* function from Chapter 1 or the min heap functions from Chapter 5.
7. Show that if T is a spanning tree for an undirected graph G, then the addition of an edge e, $e \notin E(T)$ and $e \in E(G)$, to T creates a unique cycle.

6.4 SHORTEST PATHS AND TRANSITIVE CLOSURE

MapQuest, Google Maps, Yahoo! Maps, and MapNation are a few of the many Web systems that find a path between any two specified locations in the country. Path finding systemes generally use a graph to represent the highway system of a state or a country. In this graph, the vertices represent cities and the edges represent sections of the highway. Each edge has a weight representing the distance between the two cities connected by the edge. Alternatively, the weight could be an estimate of the time it takes to travel between the two cities. A motorist wishing to drive from city A to city B would be interested in answers to the following questions: