⋆**17.** Notice that Algorithm 7.1.1 is optimal for the denominations

$$d_1 = 50, \quad d_2 = 10, \quad d_3 = 5, \quad d_4 = 1,$$

and that

$$d_i \text{ divides } d_{i-2} - d_{i-1}, \qquad 3 \le i \le n.$$

Show, by giving counterexamples that, nevertheless, the preceding condition is neither necessary nor sufficient for Algorithm 7.1.1 to be optimal.

18. Given coins of denominations

$$1 = d[1] < d[2] < \cdots < d[n],$$

prove or disprove whether the following sets $c[i][j]$ equal to the minimum number of coins needed to make change for an amount j, $1 \le j \le m$, using only the denominations

$$d[1], \ldots, d[i].$$

```
for i = 0 to n
  c[i][0] = 0
for j = 1 to m
  c[0][j] = 0
for i = 1 to n
  for j = 1 to m
    if (d[i] divides j)
      c[i][j] = j/d[i]
    else
      c[i][j] = min(c[i][j - 1] + 1, c[i - 1][j])
```

7.2 Kruskal's Algorithm

Figure 7.2.1 shows six cities and the costs (in hundreds of thousands of dollars) of rebuilding roads between them. The road commission has decided

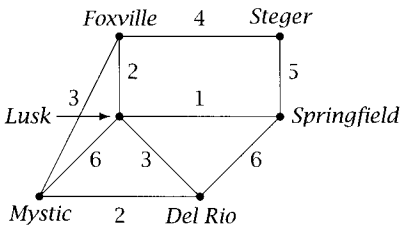


Figure 7.2.1 Cities and the costs (in hundreds of thousands of dollars) of rebuilding roads between them.

to rebuild enough roads so that each pair of cities will be connected, either directly or by going through other cities, by rebuilt roads. Figure 7.2.2 shows one possibility that costs

$$2 + 6 + 1 + 5 + 4 = 18.$$

The road commission needs an algorithm to find a minimum-cost set of roads meeting its criterion.

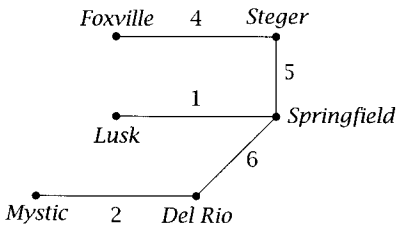


Figure 7.2.2 A subset of the roads of Figure 7.2.1. If these roads are rebuilt, each pair of cities will be connected, either directly or by going through other cities, by rebuilt roads. The cost of rebuilding these roads is 18.

Roads, cities, and costs can be modeled as a weighted graph where the vertices represent the cities, the edges represent the roads, and the weights represent the costs (see Figure 7.2.3). A **spanning tree** for a graph G is a subgraph of G that is a tree containing all of G's vertices. A **minimal spanning tree** is a spanning tree of minimum weight. Every connected graph has a spanning tree and, therefore, a minimal spanning tree. Thus, the problem of finding a minimum-cost set of roads directly or indirectly connecting all of the cities is the problem of finding a minimal spanning tree.

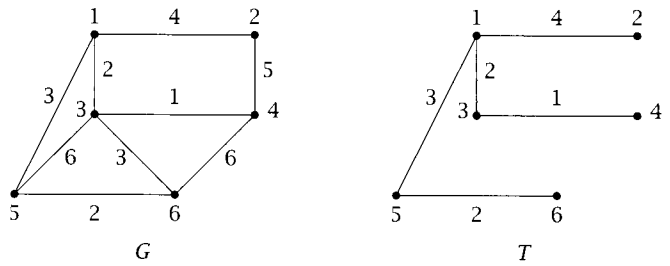


Figure 7.2.3 The graph G of Figure 7.2.1 with integers replacing the city names and a spanning tree T. No other spanning tree has a smaller weight, so T is a *minimal* spanning tree.

Example 7.2.1. In Figure 7.2.3, the weight of the spanning tree T for the graph G is 12. No other spanning tree for G has weight less than 12. [The only set of five edges whose weights sum to less than 12 is

$$\{(3,4),(5,6),(1,3),(3,6),(1,5)\},$$

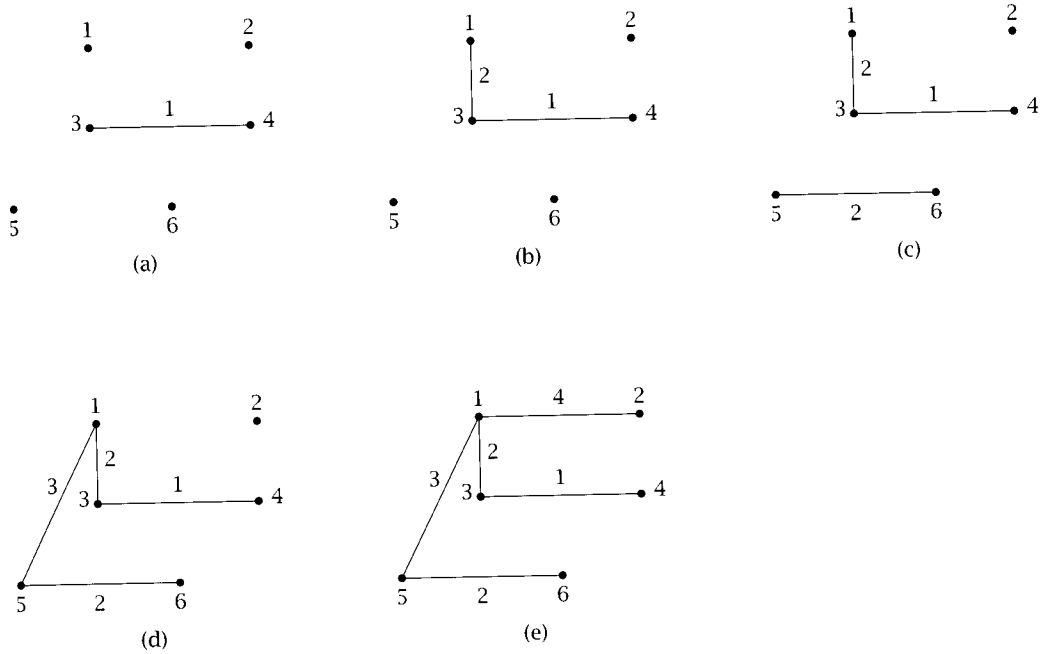


Figure 7.2.4 Kruskal's algorithm with input the graph G of Figure 7.2.3. The algorithm begins with all of the vertices and no edges. It then repeatedly adds an edge of minimum weight that does not make a cycle. In this case, it first selects edge (3,4) since it has minimum weight [graph (a)]. It next selects an edge of minimum weight 2; we assume that it selects edge $(1,3)$ [graph (b)]. It next selects edge (5,6) since it has minimum weight [graph (c)]. It next selects an edge of minimum weight 3; we assume that it selects edge $(1,5)$ [graph (d)]. Finally, it selects edge $(1,2)$ since it has minimum weight [graph(e)] to complete the minimal spanning tree.

but these edges do not form a tree.] Thus, T is a minimal spanning tree.

Since G models the road system of Figure 7.2.1, the minimal spanning tree T is a solution to the problem of finding a minimum-cost subset of roads directly or indirectly connecting all of the cities. □

WWW In this section and the next, we discuss the problem of finding a minimal spanning tree in a connected, weighted graph. Unless specified otherwise, all of the weights are assumed to be positive. **Kruskal's algorithm** is a greedy algorithm for finding a minimal spanning tree in a graph G. The algorithm begins with all of the vertices of G and no edges. It then applies the greedy rule: Add an edge of minimum weight that does not make a cycle.

Example 7.2.2. We show how Kruskal's algorithm finds a minimal spanning tree for the graph G in Figure 7.2.3. Kruskal's algorithm first selects edge $(3,4)$ since it has minimum weight [see Figure 7.2.4(a)].

Kruskal's algorithm next selects edge $(1,3)$ or $(5,6)$; either edge has minimum weight 2 and neither makes a cycle when added to $\{(3,4)\}$. When

more than one edge has the same minimum weight, any can be selected. Different spanning trees may result, but all will be minimal. Suppose that we arbitrarily select edge $(1,3)$ [see Figure 7.2.4(b)].

Kruskal's algorithm next selects edge $(5,6)$ since it has minimum weight 2 and does not make a cycle when added to $\{(3,4),(1,3)\}$ [see Figure 7.2.4(c)].

Kruskal's algorithm next selects edge $(1,5)$ or $(3,6)$; both have minimum weight 3 and neither makes a cycle when added to

$$\{(3,4),(1,3),(5,6)\}.$$

Suppose that we arbitrarily select edge $(1,5)$ [see Figure 7.2.4(d)].

Kruskal's algorithm next considers selecting edge $(3,6)$, which has minimum weight 3. Since $(3,6)$ makes a cycle when added to

$$\{(3,4),(1,3),(5,6),(1,5)\},$$

it does *not* select $(3,6)$.

Finally, Kruskal's algorithm selects edge $(1,2)$ because it has minimum weight 4 and does not make a cycle when added to

$$\{(3,4),(1,3),(5,6),(1,5)\}$$

[see Figure 7.2.4(e)].

Since we now have a spanning tree, Kruskal's algorithm terminates with the minimal spanning tree shown in Figure 7.2.4(e). □

To implement Kruskal's algorithm, several issues need to be addressed. First, we must represent the graph. Since we are selecting edges by weight, we represent the graph as a list of edges and their weights. Second, we must select the edges in nondecreasing order of weight. We can sort the edges in nondecreasing order by weight and then examine them in sorted order. Third, we must be able to determine whether adding an edge would create a cycle. We observe that adding edge (v,w) creates a cycle when there is a path between v and w formed by edges already selected, that is, when v and w are in the same *component* (see Definition 2.5.25) of the graph of edges already selected. We keep track of components by recording the set of vertices belonging to each component.

Example 7.2.3. Consider the graph G of Figure 7.2.3. Its representation is

$$(1,2,4)\ (1,3,2)\ (1,5,3)\ (2,4,5)\ (3,4,1)\ (3,5,6)\ (3,6,3)\ (4,6,6)\ (5,6,2),$$

where (v_1,v_2,w) is interpreted as edge (v_1,v_2) of weight w.

We first sort the edges in nondecreasing order by weight:

$$(3,4,1)\ (1,3,2)\ (5,6,2)\ (1,5,3)\ (3,6,3)\ (1,2,4)\ (2,4,5)\ (3,5,6)\ (4,6,6).$$

When Kruskal's algorithm starts, no edges have been selected, so each vertex belongs to a component consisting of itself:

$$\{1\}\quad\{2\}\quad\{3\}\quad\{4\}\quad\{5\}\quad\{6\}.$$

The first edge $(3,4)$ is selected, and the components to which vertices 3 and 4 belong are merged; the components become

$$\{1\} \quad \{2\} \quad \{3,4\} \quad \{5\} \quad \{6\}.$$

Next edge $(1,3)$ is selected, and the components to which vertices 1 and 3 belong are merged; the components become

$$\{1,3,4\} \quad \{2\} \quad \{5\} \quad \{6\}.$$

Next edge $(5,6)$ is selected, and the components to which vertices 5 and 6 belong are merged; the components become

$$\{1,3,4\} \quad \{2\} \quad \{5,6\}.$$

Next edge $(1,5)$ is selected, and the components to which vertices 1 and 5 belong are merged; the components become

$$\{1,3,4,5,6\} \quad \{2\}.$$

Next edge $(3,6)$ is examined but rejected because its vertices belong to the same component $\{1,3,4,5,6\}$. Finally, edge $(1,2)$ is selected, and the components to which vertices 1 and 2 belong are merged; the components become

$$\{1,2,3,4,5,6\},$$

and Kruskal's algorithm terminates. □

The algorithms to manage disjoint sets (see Section 3.6) can be used to handle the components. Algorithm *makeset* can be used to initialize each vertex to its own component;

$$findset(v) == findset(w)$$

can be used to test whether vertices v and w belong to the same component; and $union(v,w)$ can be used to merge the components to which vertices v and w belong. Algorithm 7.2.4 formally states Kruskal's algorithm.

Algorithm 7.2.4 Kruskal's Algorithm. Kruskal's algorithm finds a minimal spanning tree in a connected, weighted graph with vertex set $\{1,\ldots,n\}$. The input to the algorithm is *edgelist*, an array of *edge*, and n. The members of *edge* are

- v and w, the vertices on which the edge is incident.
- *weight*, the weight of the edge.

The output lists the edges in a minimal spanning tree. The function *sort* sorts the array *edgelist* in nondecreasing order of *weight*.

Input Parameters: *edgelist*, n
Output Parameters: None

```
kruskal(edgelist, n) {
  sort(edgelist)
  for i = 1 to n
    makeset(i)
  count = 0
  i = 1
  while (count < n − 1) {
    if (findset(edgelist[i].v) != findset(edgelist[i].w)) {
      println(edgelist[i].v + " " + edgelist[i].w)
      count = count + 1
      union(edgelist[i].v, edgelist[i].w)
    }
    i = i + 1
  }
}
```

After Kruskal's algorithm adds $n - 1$ edges, an acyclic, $(n-1)$-edge subgraph of the original n-vertex graph is obtained. By Theorem 2.6.5, the subgraph is a tree and, therefore, a spanning tree.

There are n *makeset* operations, at most $2m$ *findset* operations, and $n-1$ *union* operations. Because the graph input to Kruskal's algorithm is connected, $m \geq n - 1$. Thus the number of union and findset operations is $O(m)$. Using union by rank alone, or union by rank and path compression, these operations take time $O(m \lg m)$ (see Section 3.6). In the worst case, comparison-based sorting takes time $\Theta(m \lg m)$. Thus the worst-case time of Algorithm 7.2.4 is $\Theta(m \lg m)$.

In Section 7.1, we noted that greedy algorithms may or may not be optimal. Fortunately, Kruskal's algorithm is optimal. We deduce this from a slightly stronger result.

Theorem 7.2.5. *Let G be a connected, weighted graph, and let G' be a subgraph of a minimal spanning tree of G. Let C be a component of G', and let S be the set of all edges with one vertex in C and the other not in C. If we add a minimum weight edge in S to G', the resulting graph is also contained in a minimal spanning tree of G.*

Before proving Theorem 7.2.5, we show how it implies the correctness of Kruskal's algorithm (Algorithm 7.2.4).

Theorem 7.2.6 Correctness of Kruskal's Algorithm. *Kruskal's algorithm (Algorithm 7.2.4) is correct; that is, it finds a minimal spanning tree.*

Proof. We use induction to show that at each iteration of Kruskal's algorithm, the subgraph constructed is contained in a minimal spanning tree. It then follows that, at the termination of Kruskal's algorithm, the subgraph constructed *is* a minimal spanning tree.

When we begin, the subgraph, which consists of no edges, is contained in every minimal spanning tree. Thus the basis step is true.

Turning to the inductive step, let G' denote the subgraph constructed by Kruskal's algorithm prior to another iteration of the algorithm. The inductive assumption is that G' is contained in a minimal spanning tree. Let (v, w) be the next edge selected by Kruskal's algorithm, and let C be the component of G' to which v belongs. Edge (v, w) is a minimum weight edge with one vertex in C and one not in C because it is a minimum weight edge from *any* component to any other. Therefore, by Theorem 7.2.5, when (v, w) is added to G', the resulting graph is also contained in a minimal spanning tree. The inductive step is complete and the theorem is proved. ■

We conclude by proving Theorem 7.2.5.

Proof of Theorem 7.2.5. Let G be a connected, weighted graph, and let G' be a subgraph of G that is contained in a minimal spanning tree T of G. Let C be a component of G', and let (v, w) be a minimum weight edge with v in C and w not in C. We must show that the graph obtained by adding (v, w) to G' is contained in a minimal spanning tree of G.

If T also contains (v, w), the proof is complete; so, suppose that T does not contain (v, w). If we add the edge (v, w) to T and remove an edge from the cycle S created by adding (v, w), the resulting subgraph T' is also a spanning tree. We choose the edge to remove as follows.

Let w' be the first vertex on S, going from v to w, that is not in C, and let v' be the vertex on S just before w' (v' is in C) (see Figure 7.2.5). Add (v, w) to T and remove (v', w') from T to obtain T'. Since (v, w) is a minimum weight edge with one vertex in C and the other not in C,

$$\text{weight}(v', w') \geq \text{weight}(v, w).$$

Therefore

$$\text{weight}(T) \geq \text{weight}(T').$$

Since T is a *minimal* spanning tree, we must have

$$\text{weight}(T) = \text{weight}(T').$$

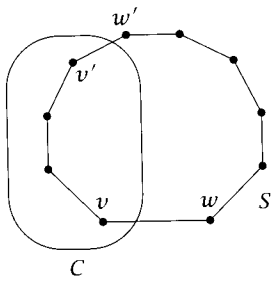


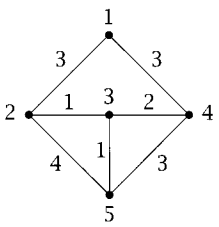
Figure 7.2.5 The proof of Theorem 7.2.5. Vertex w' is the first vertex on cycle S, going from v to w, that is not in C. Vertex v' is the vertex on S just before w'. The spanning tree T is modified by adding edge (v, w) and removing edge (v', w'). The tree T' obtained is also a minimal spanning tree.

Therefore T' is a minimal spanning tree. Since T' contains all of the edges of G' as well as (v, w), the proof is complete. ■

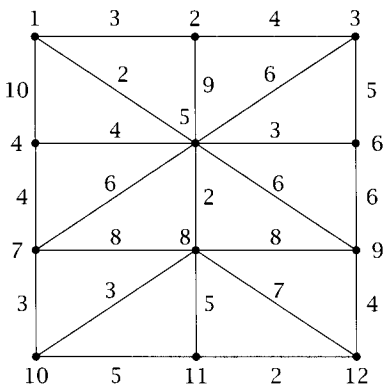
Exercises

Trace Kruskal's algorithm for each graph in Exercises 1–3.

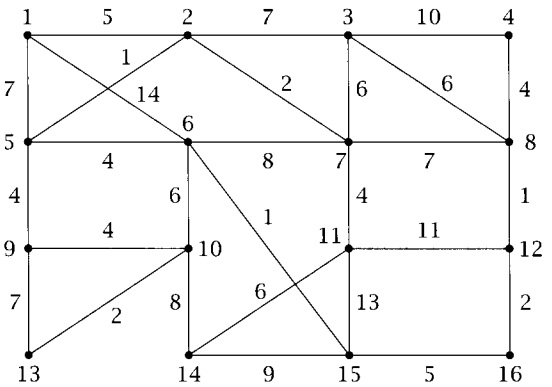
1S.



2.



3.



4S. What is the worst-case time (in terms of n) of Kruskal's algorithm when the input is the complete graph on n vertices?

5. In Kruskal's algorithm, sorting all of the edges will be more work than necessary if not all of the edges are examined for possible inclusion in the minimal spanning tree; so, suppose that instead of sorting the edges, we place them in a binary minheap and remove them from the minheap as needed. Analyze the worst-case time of this implementation of Kruskal's algorithm in terms of the number of edges, the number of vertices, and the number of edges examined for possible inclusion in a minimal spanning tree.

6. Consider a possible divide-and-conquer approach to finding a minimal spanning tree in a connected, weighted graph G. Suppose that we divide the vertices of G into two disjoint subsets V_1 and V_2. We then find a minimal spanning tree T_1 for V_1 and a minimal spanning tree T_2 for V_2. Finally, we find a minimum weight edge e connecting T_1 and T_2. We then let T be the graph obtained by combining T_1, T_2, and e.

   (a) Is T always a spanning tree?

   (b) If T is a spanning tree, is it always a minimal spanning tree?

7S. Let T be a minimal spanning tree for a graph G, let e be an edge in T, and let T' be T with e removed. Show that e is a minimum weight edge between components of T'.

8. Let G be a connected, weighted graph, let v be a vertex in G, and let e be an edge of minimum weight incident on v. Show that e is contained in some minimal spanning tree.

9. Let G be a connected, weighted graph, and let v be a vertex in G. Suppose that the weights of the edges incident on v are distinct. Let e be the edge of minimum weight incident on v. Show that e is contained in every minimal spanning tree.

10S. Let T and T' be two spanning trees of a connected graph G. Suppose that an edge e is in T but not in T'. Show that there is an edge e' in T', but not in T, such that $(T - \{e\}) \cup \{e'\}$ and $(T' - \{e'\}) \cup \{e\}$ are spanning trees of G.

In Exercises 11–13, tell whether the statement is true or false. If the statement is true, prove it; otherwise, give a counterexample. In each exercise, G is a connected, weighted graph.

11S. If all of the weights in G are distinct, distinct spanning trees of G have distinct weights.

12. If all of the weights in G are distinct, G has a unique minimal spanning tree.

13. If e is an edge in G whose weight is less than the weight of every other edge, e is in every minimal spanning tree of G.

7.3 Prim's Algorithm

WWW **Prim's algorithm** is another greedy algorithm that finds a minimal spanning tree in a connected, weighted graph. Unless specified otherwise, all of the weights are assumed to be positive. Unlike Kruskal's algorithm, whose partial solutions are not necessarily connected, a partial solution in Prim's algorithm is a tree.

Prim's algorithm begins with a start vertex and no edges and then applies the greedy rule: Add an edge of minimum weight that has one vertex in the current tree and the other not in the current tree.

Example 7.3.1. We show how Prim's algorithm finds a minimal spanning tree for the graph G in Figure 7.2.3 assuming that the start vertex is 5.

Prim's algorithm first selects edge $(5,6)$ since, among all of the edges incident on the start vertex 5, it has minimum weight. Edge $(5,6)$ has one vertex in the current tree and the other not in the current tree.

At the next iteration, possible edges to add are

$$(5,1),(5,3),(6,3),(6,4)$$

since each has one vertex in the current tree and the other not in the current tree [see Figure 7.3.1(a)]. Edges $(5,1)$ and $(6,3)$ each have minimum weight 3 and either can be selected; different spanning trees will result, but each will have minimum weight. We arbitrarily assume that $(5,1)$ is selected.

At the next iteration, possible edges to add are

$$(1,2),(1,3),(5,3),(6,3),(6,4)$$

[see Figure 7.3.1(b)]. Edge $(1,3)$, which has minimum weight 2, is selected.

At the next iteration, possible edges to add are

$$(1,2),(3,4),(6,4)$$

[see Figure 7.3.1(c)]. Notice that $(5,3)$ and $(6,3)$ are no longer candidates for selection because vertices 3, 5, and 6 are all in the tree. Edge $(3,4)$, which has minimum weight 1, is selected.

At the final iteration, possible edges to add are

$$(1,2),(2,4)$$

[see Figure 7.3.1(d)]. Edge $(1,2)$, which has minimum weight 4, is selected. We obtain the minimal spanning shown in Figure 7.3.1(e). □

To implement Prim's algorithm, we must keep track of candidate edges to add to the current tree. We can simplify this task if we retain only *one* minimum weight edge from each vertex not in the current tree to the current tree. For example, Figure 7.3.1(b) shows three edges from vertex 3, which is not in the current tree, to the tree: $(3,1)$ of weight 2, $(3,5)$ of weight 6, and $(3,6)$ of weight 3. We would not select $(3,5)$ or $(3,6)$ because the

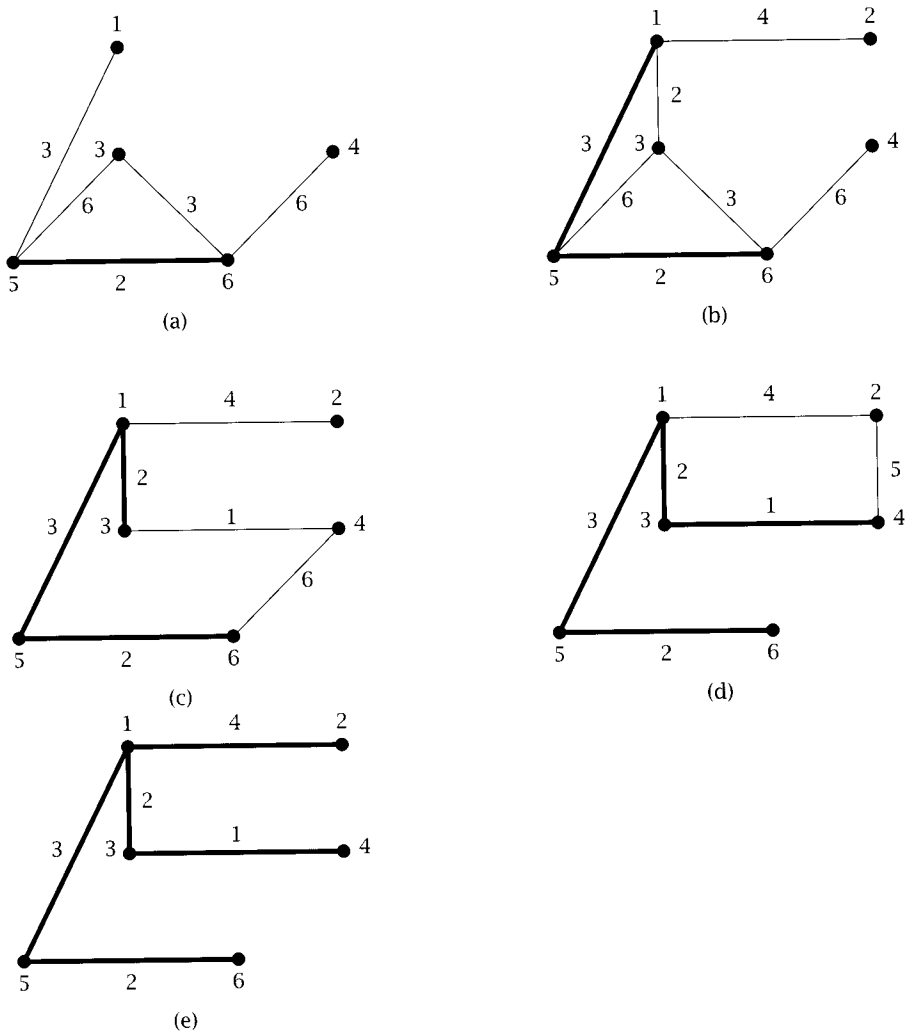


Figure 7.3.1 Prim's algorithm. The start vertex is 5. Edges chosen are shown as thick lines. Candidate edges for selection at the next iteration are shown as thin lines. Edge $(5,6)$ is selected first because, among the edges incident on the start vertex 5, $(5,6)$ has minimum weight. Next [see (a)], among the candidate edges, $(5,1)$, which has minimum weight, is selected. Next [see (b)], among the candidate edges, $(1,3)$, which has minimum weight, is selected. Next [see (c)], among the candidate edges, $(3,4)$, which has minimum weight, is selected. Finally [see (d)], among the candidate edges, $(1,2)$, which has minimum weight, is selected yielding the minimal spanning tree (e).

weight of each exceeds the weight of $(3,1)$. Therefore, we retain only $(3,1)$ as a candidate edge from vertex 3 to the current tree. If we retain only one minimum weight edge from each vertex not in the current tree to the current tree, Figure 7.3.1(b) becomes Figure 7.3.2.

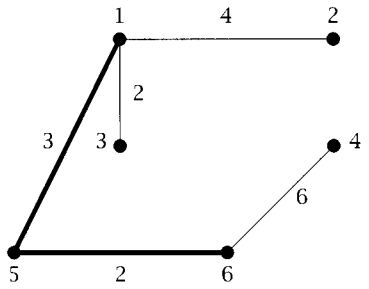


Figure 7.3.2 Modifying Figure 7.3.1(b) to implement Prim's algorithm. Instead of considering *all* edges from vertices not in the current tree to the tree, we consider only a *least weight* edge from each vertex not in the current tree to the tree.

We keep a list h of vertices v not in the tree and the minimum weight of an edge from v to a vertex in the tree. We also maintain an array *parent* that tells us which edges give minimum weights. If (v,w) is an edge of minimum weight where v is not in the tree and w is in the tree, then $parent[v] = w$.

Example 7.3.2. For Figure 7.3.2, the following table shows the list h and the parent of each vertex in h

h		$parent[v]$
Vertex (v)	*Minimum Weight from* v *to Tree*	
2	4	1
3	2	1
4	6	6

The *parent* array is

$$parent[2] = 1, \quad parent[3] = 1, \quad parent[4] = 6. \qquad \square$$

After the vertex v with a minimum weight edge to the tree is deleted from the h list, the vertices still in the h list may need their weights adjusted. For example, if, in the original h list, the weight corresponding to vertex w was 10 but there is an edge from w to v of weight less than 10, say 5, the adjusted weight corresponding to w becomes 5 (see Figure 7.3.3). Thus, after selecting vertex v, we examine each vertex w not in the tree adjacent to v. If the weight of edge (v,w) is less than the weight in the h list corresponding to w, we update the weight corresponding to w to the weight of edge (v,w). We also update $parent[w]$ to v. In order to perform this updating efficiently, we represent the graph using adjacency lists.

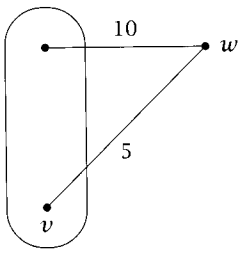


Figure 7.3.3 Updating a vertex's weight entry in the h list. Before vertex v was added to the tree, the least weight edge from w to the tree was 10. Since edge (v, w) has weight 5, after v is added to the tree, the weight corresponding to w becomes 5. Also, $parent[w]$ becomes v.

Example 7.3.3. We show a trace of Prim's algorithm for the graph G of Figure 7.2.3. We assume that the start vertex is 5.

Initially, the h list and parents of vertices in h are

h		$parent[v]$
Vertex (v)	*Minimum Weight from* v *to Tree*	
1	3	5
2	∞	-
3	6	5
4	∞	-
6	2	5

As shown, if there is no edge from a vertex not in the tree to the tree, we set its weight to ∞. The *parent* array is

$$parent[1] = 5, \quad parent[3] = 5, \quad parent[5] = 0, \quad parent[6] = 5.$$

As shown, to indicate the start vertex, we set its parent to zero.

We then select the minimum weight 2 in the h list and delete this entry from the h list, which corresponds to selecting edge $(5, 6)$. We then examine all of the vertices adjacent to 6 not in the tree to determine whether any entries in the h list need updating. In this case, vertices 3 and 4 in the h list have their weights adjusted. Since the edge $(3, 6)$ has weight 3, but 3's old weight in the h list was 6, vertex 3's weight entry is updated to 3. Similarly, since there is an edge from 4 to 6 of weight 6, 4's weight entry is updated to 6. The h list and parents of vertices in h become

h		$parent[v]$
Vertex (v)	*Minimum Weight from* v *to Tree*	
1	3	5
2	∞	-
3	3	6
4	6	6

The *parent* array becomes

$$parent[1] = 5, \quad parent[3] = 6, \quad parent[4] = 6,$$
$$parent[5] = 0, \quad parent[6] = 5.$$

Next, we select the minimum weight 3 in the h list and delete this entry from the h list. Since there is a tie, we could select the entry corresponding to either vertex 1 or 3. We arbitrarily choose vertex 1, which corresponds to selecting edge $(1, 5)$. We then examine all of the vertices adjacent to 1 not in the tree to determine whether any entries in the h list need updating. In this case, vertices 2 and 3 in the h list have their weights adjusted. The h list and parents of vertices in h become

h		$parent[v]$
Vertex (v)	*Minimum Weight from* v *to Tree*	
2	4	1
3	2	1
4	6	6

and the *parent* array becomes

$$parent[1] = 5, \quad parent[2] = 1, \quad parent[3] = 1,$$
$$parent[4] = 6, \quad parent[5] = 0, \quad parent[6] = 5.$$

Next, we select the minimum weight 2 in the h list and delete this entry from the h list, which corresponds to selecting edge $(1, 3)$. We then examine all of the vertices adjacent to 3 not in the tree to determine whether any entries in the h list need updating. In this case, vertex 4 in the h list has its weight adjusted. The h list and parents of vertices in h become

h		$parent[v]$
Vertex (v)	*Minimum Weight from* v *to Tree*	
2	4	1
4	1	3

and the *parent* array becomes

$$parent[1] = 5, \quad parent[2] = 1, \quad parent[3] = 1,$$
$$parent[4] = 3, \quad parent[5] = 0, \quad parent[6] = 5.$$

Next, we select the minimum weight 1 in the h list and delete this entry from the h list, which corresponds to selecting edge $(3, 4)$. We then examine all of the vertices adjacent to 4 not in the tree to determine whether any entries in the h list need updating. In this case, no vertex has its weight adjusted. The h list and parent of the vertex in h become

h		$parent[v]$
Vertex (v)	*Minimum Weight from* v *to Tree*	
2	4	1

and the *parent* array becomes

$$parent[1] = 5, \quad parent[2] = 1, \quad parent[3] = 1,$$
$$parent[4] = 3, \quad parent[5] = 0, \quad parent[6] = 5.$$

We select the remaining weight 4 in the h list and delete this entry from the h list, which corresponds to selecting edge $(1, 2)$. The h list becomes empty and the *parent* array is unchanged. Prim's algorithm terminates with the minimal spanning tree T shown in Figure 7.2.3. □

In Prim's algorithm, we assume that h is an abstract data type that supports the following operations: If *key* is an array of size n, the expression

$h.init(key, n)$

initializes h to the values in *key*. The expression

$h.del()$

deletes the item in h with the smallest weight and returns the corresponding vertex. The expression

$h.isin(w)$

returns true if vertex w is in h and false otherwise. The expression

$h.keyval(w)$

returns the weight corresponding to vertex w. Finally, the expression

$h.decrease(w, wgt)$

changes the weight corresponding to vertex w to *wgt* (a smaller value).

Algorithm 7.3.4 Prim's Algorithm. This algorithm finds a minimal spanning tree in a connected, weighted, n-vertex graph. The graph is represented using adjacency lists; $adj[i]$ is a reference to the first node in a linked list of nodes representing the vertices adjacent to vertex i. Each node has members *ver*, the vertex adjacent to i; *weight*, representing the weight of edge (i, ver); and *next*, a reference to the next node in the linked list or null, for the last node in the linked list. The start vertex is *start*. In the minimal spanning tree, the parent of vertex $i \neq start$ is $parent[i]$, and $parent[start] = 0$. The value ∞ is the largest available integer value.

Input Parameters: *adj*, *start*
Output Parameter: *parent*

```
prim(adj, start, parent) {
  n = adj.last
  // key is a local array
  for i = 1 to n
    key[i] = ∞
  key[start] = 0
  parent[start] = 0
  // the following statement initializes the
  // container h to the values in the array key
```

```
    h.init(key, n)
    for i = 1 to n {
      v = h.del()
      ref = adj[v]
      while (ref != null) {
        w = ref.ver
        if (h.isin(w) && ref.weight < h.keyval(w)) {
          parent[w] = v
          h.decrease(w, ref.weight)
        }
        ref = ref.next
      }
    }
}
```

There are several ways to implement the abstract data type h in Prim's algorithm (Algorithm 7.3.4). One efficient way is to use a binary minheap (see Section 3.5). We analyze the worst-case time of Prim's algorithm assuming that h is implemented using a binary minheap and that the graph has n vertices and m edges. The worst-case time of the various heap operations involved are summarized in Figure 7.3.4 (see Section 3.5 for details).

Operation	*Worst-Case Time*
init(key, n)	$\Theta(n)$
del()	$\Theta(\lg n)$
isin(w)	$\Theta(1)$
keyval(w)	$\Theta(1)$
decrease$(w, ref.weight)$	$\Theta(\lg n)$

Figure 7.3.4 The worst-case time for binary minheap operations.

Each for loop takes time $\Theta(n)$. The *init* operation takes time $\Theta(n)$. The delete operation *del*, which takes time $O(\lg n)$, is in a for loop whose time is $\Theta(n)$; thus, the total worst-case time for the delete operations is $O(n \lg n)$. The *total* time for the while loop is $\Theta(m)$ since each iteration of the while loop inspects another node on some adjacency list and there are $2m$ nodes altogether. Each *isin* and *keyval* operation takes constant time to evaluate. The decrease operation *decrease*, which takes time $O(\lg n)$, is in the while loop whose total time is $\Theta(m)$; thus, the total worst-case time for the decrease operations is $O(m \lg n)$. Since $m \geq n - 1$, the dominant term is $m \lg n$ and the worst-case time is $O(m \lg n)$. [In the following subsection, we show that this estimate is sharp; that is, the worst-case time is $\Theta(m \lg n)$.]

If, instead of using a binary heap to implement Prim's algorithm, we use a Fibonacci heap (see Fredman, 1987), we can improve the worst-case time of Prim's algorithm to $\Theta(m + n \lg n)$.

The proof of correctness of Prim's algorithm is similar to the proof of correctness of Kruskal's algorithm.

Theorem 7.3.5 Correctness of Prim's Algorithm. *Prim's algorithm (Algorithm 7.3.4) is correct; that is, it finds a minimal spanning tree.*

Proof. We use induction to show that, at each iteration of Prim's algorithm, the tree constructed is contained in a minimal spanning tree. It then follows that at the termination of Prim's algorithm, the tree constructed *is* a minimal spanning tree.

When we begin, the tree consists of no edges and is contained in every minimal spanning tree. Thus the basis step is true.

Turning to the inductive step, let T denote the tree constructed by Prim's algorithm prior to another iteration of the algorithm. The inductive assumption is that T is contained in a minimal spanning tree. Let (v, w) be the next edge selected by Prim's algorithm, where v is in T and w is not in T. Let G' be T together with all of the vertices not in T. Then T is a component of G' and (v, w) is a minimum weight edge with one vertex in T and one not in T. By Theorem 7.2.5, when (v, w) is added to G', the resulting graph is also contained in a minimal spanning tree. The inductive step is complete and the theorem is proved. ■

†Lower Bound Time Estimate

In this subsection, we show that the worst-case time for Prim's algorithm using a binary heap is $\Theta(m \lg n)$, where m denotes the number of edges and n denotes the number of vertices.

The bottleneck is the decrease operation

$h.decrease(w, ref.weight)$

To obtain worst-case time, we must construct a graph in which the decrease operation takes time $\Theta(\lg i)$ sufficiently often when the heap contains i vertices. We can guarantee such behavior if the next vertex's key to decrease has the maximum key in the heap which is then decreased so that it becomes the smallest key in the heap (in which case the vertex moves from a terminal node in the heap to the root). We construct such a graph with $n \geq 4$ vertices and $m \geq 4n$ edges as follows. [If $m < 4n$, for any graph the worst-case time T satisfies

$$T \geq Cn \lg n \geq \frac{C}{4} m \lg n = \Omega(m \lg n).$$

The first inequality results from the fact that any implementation of Prim's algorithm that uses comparisons of weights can sort an array of size $\Theta(n)$, and, so, has worst-case time $\Omega(n \lg n)$ (see Exercise 15).]

Our graph G has vertices $1, 2, \ldots, n$. For $i = 1, \ldots, n-1$, we construct edges $(i, i+1)$ of weight 1 (see Figure 7.3.5). We next construct edges

$$(1, n), (1, n-1), \ldots, (1, 4), (1, 3)$$

of decreasing weight. We next construct edges

$$(2, n), (2, n-1), \ldots, (2, 5), (2, 4)$$

†This subsection can be omitted without loss of continuity.

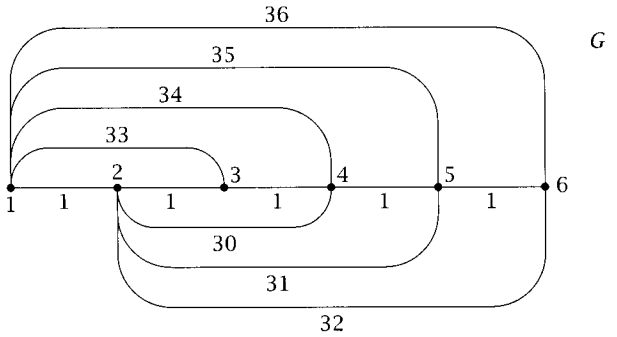


Figure 7.3.5 Part of the graph G with $n = 6$ vertices for input to Prim's algorithm. The edges shown as curves are given the weights $n^2, n^2 - 1$, $n^2 - 2, \ldots$, which here become $36, 35, \ldots$. This graph produces worst-case time $\Theta(m \lg n)$. After vertices 1 and 2 are deleted from the heap, the heap contains keys $33, 34, 35, 36$ (the original minimum weights of edges from vertices $3, 4, 5, 6$ to vertex 1). We then examine the edges $(2, 6)$, $(2, 5)$, $(2, 4)$, $(2, 3)$ in this order. Since the weight of $(2, 6)$ is 32, 6's key is decreased from 36 to 32. Since 36 was the largest key and 32 will become the smallest key, vertex 6 moves from a terminal vertex in the heap to the root, which takes time at least $C \lg(n - 2)$. Similarly, when keys $35, 34, 33$ are decreased, they too each take time at least $C \lg(n - 2)$. The total time to decrease these keys is at least $C(n - 2) \lg(n - 2)$.

of decreasing weight, where the weight of $(2, n)$ is less than the weight of $(1, 3)$. We continue in this way, stopping when m edges have been constructed. We assume that the weights assigned in this part of the construction are, in order, $n^2, n^2 - 1, n^2 - 2, \ldots$.

Suppose that G is input to Prim's algorithm and that the start vertex is 1. After 1 is deleted from the heap and the keys are decreased, we have

Vertex	*Minimum Weight to Tree*
2	1
3	$n^2 - (n - 3)$
4	$n^2 - (n - 4)$
$\vdots$	$\vdots$
$n - 2$	$n^2 - 2$
$n - 1$	$n^2 - 1$
n	n^2

Vertex 2 is deleted next. Assume that when the keys are decreased, the edges are examined in the order

$$(2, n), (2, n - 1), \ldots, (2, 4), (2, 3).$$

After vertex 2 is deleted from the heap, n's key is largest. Therefore, it is a terminal vertex in the heap. Since its new value is less than any of the current

keys, the time to decrease n's key is at least $C\lg(n-2)$ for some constant C. Now $(n-1)$'s key is largest. Therefore, it is a terminal vertex in the heap. Since its new value is less than any of the current keys, the time to decrease $(n-1)$'s key is also at least $C\lg(n-2)$. Similarly, the time to decrease each of the other keys is also at least $C\lg(n-2)$. The total time to decrease the keys (if all of these edges are present) is at least $C(n-2)\lg(n-2)$.

Vertex 3 is deleted next. Assume that when the keys are decreased, the edges are examined in the order

$$(3,n),(3,n-1),\ldots,(3,5),(3,4).$$

Arguing as in the preceding paragraph, we see that the time to decrease each of the keys is at least $C\lg(n-3)$, and the total time to decrease the keys (if all of these edges are present) is at least $C(n-3)\lg(n-3)$.

Let T be the time for all of the decrease operations for our graph G. Then

$$T \geq C[(n-2)\lg(n-2)+(n-3)\lg(n-3)+\cdots+(k+1)\lg(n-p)],$$

where the last edges constructed were k edges of the form (p,i). (The inequality could be strict since the right side may not account for all of the keys that eventually decrease to 1.) We show that

$$T \geq \frac{C}{2}m\lg\left(\frac{n-2}{2}\right).$$

First, suppose that $n-p \geq \lceil (n-2)/2 \rceil$. Then

$$\begin{aligned} T &\geq C[(n-2)\lg(n-2)+(n-3)\lg(n-3)+\cdots+(k+1)\lg(n-p)] \\ &\geq C\left[(n-2)\lg\left\lceil\frac{n-2}{2}\right\rceil+\cdots+(k+1)\lg\left\lceil\frac{n-2}{2}\right\rceil\right] \\ &\geq C[(n-2)+\cdots+(k+1)]\lg\left\lceil\frac{n-2}{2}\right\rceil \\ &\geq C[(n-3)+\cdots+k)]\lg\left(\frac{n-2}{2}\right). \end{aligned}$$

Since the sum

$$(n-3)+\cdots+k$$

counts all of the edges except those of weight 1 and those incident on vertex 1,

$$(n-3)+\cdots+k = m-(n-1)-(n-2).$$

Because $m \geq 4n$,

$$m-(n-1)-(n-2) \geq \frac{m}{2}$$

(see Exercise 12). It follows that

$$T \geq \frac{C}{2}m\lg\left(\frac{n-2}{2}\right).$$

Now suppose that $n - p < \lceil (n-2)/2 \rceil$. In this case,

$$\begin{aligned} T &\geq C[(n-2)\lg(n-2) + (n-3)\lg(n-3) + \cdots + (k+1)\lg(n-p)] \\ &\geq C\left[(n-2)\lg(n-2) + \cdots + \left\lceil \frac{n-2}{2} \right\rceil \lg\left\lceil \frac{n-2}{2} \right\rceil\right] \\ &\geq C\left[(n-2)\lg\left\lceil \frac{n-2}{2} \right\rceil + \cdots + \left\lceil \frac{n-2}{2} \right\rceil \lg\left\lceil \frac{n-2}{2} \right\rceil\right] \\ &\geq C\left[(n-2) + \cdots + \left\lceil \frac{n-2}{2} \right\rceil\right] \lg\left(\frac{n-2}{2}\right) \\ &\geq \frac{C}{2} m \lg\left(\frac{n-2}{2}\right). \end{aligned}$$

The last inequality follows from the inequality

$$(n-2) + \cdots + \left\lceil \frac{n-2}{2} \right\rceil \geq \frac{n(n-1)}{4},$$

which holds for $n \geq 4$ (see Exercise 13), and the fact that the maximum number of edges in the graph is $n(n-1)/2$.

In either case, we have

$$T \geq \frac{C}{2} m \lg\left(\frac{n-2}{2}\right) = \Omega(m \lg n).$$

We showed earlier that the worst-case time of Prim's algorithm using a binary heap is $O(m \lg n)$. It follows that the worst-case time of Prim's algorithm using a binary heap is $\Theta(m \lg n)$.

Exercises

1S. Trace Prim's algorithm for the graph of Exercise 1, Section 7.2. Assume that the start vertex is 1.

2. Trace Prim's algorithm for the graph of Exercise 2, Section 7.2. Assume that the start vertex is 8.

3. Trace Prim's algorithm for the graph of Exercise 3, Section 7.2. Assume that the start vertex is 11.

4S. Write an algorithm whose input is the *parent* array constructed by Prim's algorithm and whose output is a list of the edges in the minimal spanning tree constructed by Prim's algorithm.

5. Explain why we can't eliminate the *parent* array in Algorithm 7.3.4 and replace the statement

$$parent[w] = v$$

with

$println(v + \text{“ ”} + w)$

6. What is the worst-case time (in terms of n) of Prim's algorithm when the input is the complete graph on n vertices? Assume that h is implemented using a binary minheap.

7S. What is the worst-case time of Prim's algorithm if h is implemented using an array that is always sorted from largest to smallest weight?

8. What is the worst-case time of Prim's algorithm if h is implemented using an unsorted array?

9. Are there graphs for which Prim's algorithm is faster than Kruskal's algorithm?

10S. Are there graphs for which Prim's algorithm is slower than Kruskal's algorithm?

11. Provide an implementation of Prim's algorithm that uses an adjacency matrix instead of adjacency lists. What is the worst-case time of your algorithm? Assume that h is implemented using a binary minheap.

12. Show that if $m \geq 4n$,

$$m - (n-1) - (n-2) \geq \frac{m}{2}.$$

13S. Show that if $n \geq 4$,

$$(n-2) + (n-3) + \cdots + \left\lceil \frac{n-2}{2} \right\rceil \geq \frac{n(n-1)}{4}.$$

14. Show that any implementation of Prim's algorithm must examine each edge's weight at least once, and thus has time $\Omega(m)$.

15. Show that any implementation of Prim's algorithm that uses comparisons of weights can sort an array of size $\Theta(n)$ and, so, has worst-case time $\Omega(n \lg n)$.

7.4 Dijkstra's Algorithm

The map in Figure 7.4.1(a) shows six cities and the time in minutes to drive between cities that are directly connected by a road. A computer technician based in Riverview, who is desperately needed in Wolf, must find the quickest route from Riverview to Wolf.

WWW The map in Figure 7.4.1(a) can be considered a graph [see Figure 7.4.1(b)] in which the cities become vertices, the roads become edges, and the times