1. Every free tree with $n \geq 1$ vertices contains exactly $n-1$ edges.
2. If we add any edge to a free tree, we get a cycle.

We can prove (1) by induction on $n$, or what is equivalent, by an argument concerning the "smallest counterexample." Suppose $G = (V, E)$ is a counterexample to (1) with the fewest vertices, say $n$ vertices. Now $n$ cannot be 1, because the only free tree on one vertex has zero edges, and (1) is satisfied. Therefore, $n$ must be greater than 1.

We now claim that in the free tree there must be some vertex with exactly one incident edge. In proof, no vertex can have zero incident edges, or $G$ would not be connected. Suppose every vertex has at least two edges incident. Then, start at some vertex $v_1$, and follow any edge from $v_1$. At each step, leave a vertex by a different edge from the one used to enter it, thereby forming a path $v_1, v_2, v_3, \ldots$.

Since there are only a finite number of vertices in $V$, all vertices on this path cannot be distinct; eventually, we find $v_i = v_j$ for some $i < j$. We cannot have $i = j-1$ because there are no loops from a vertex to itself, and we cannot have $i = j-2$ or else we entered and left vertex $v_{i+1}$ on the same edge. Thus, $i \leq j-3$, and we have a cycle $v_i, v_{i+1}, \ldots, v_j = v_i$. Thus, we have contradicted the hypothesis that $G$ had no vertex with only one edge incident, and therefore conclude that such a vertex $v$ with edge $(v, w)$ exists.

Now consider the graph $G'$ formed by deleting vertex $v$ and edge $(v, w)$ from $G$. $G'$ cannot contradict (1), because if it did, it would be a smaller counterexample than $G$. Therefore, $G'$ has $n-1$ vertices and $n-2$ edges. But $G$ has one more edge and one more vertex than $G'$, so $G$ has $n-1$ edges, proving that $G$ does indeed satisfy (1). Since there is no smallest counterexample to (1), we conclude there can be no counterexample at all, so (1) is true.

Now we can easily prove statement (2), that adding an edge to a free tree forms a cycle. If not, the result of adding the edge to a free tree of $n$ vertices would be a graph with $n$ vertices and $n$ edges. This graph would still be connected, and we supposed that adding the edge left the graph acyclic. Thus we would have a free tree whose vertex and edge count did not satisfy condition (1).

### Methods of Representation

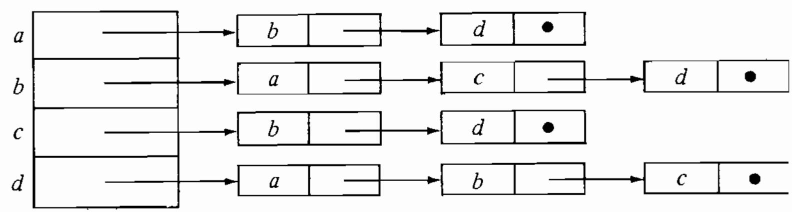
The methods of representing directed graphs can be used to represent undirected graphs. One simply represents an undirected edge between $v$ and $w$ by two directed edges, one from $v$ to $w$ and the other from $w$ to $v$.

**Example 7.3.** The adjacency matrix and adjacency list representations for the graph of Fig. 7.1(a) are shown in Fig. 7.3. □

Clearly, the adjacency matrix for a graph is symmetric. In the adjacency list representation if $(i, j)$ is an edge, then vertex $j$ is on the list for vertex $i$ and vertex $i$ is on the list for vertex $j$.

|   | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 0 | 1 |
| b | 1 | 0 | 1 | 1 |
| c | 0 | 1 | 0 | 1 |
| d | 1 | 1 | 1 | 0 |

(a) Adjacency matrix




(b) Adjacency list

**Fig. 7.3.** Representations.

## 7.2 Minimum-Cost Spanning Trees

Suppose $G = (V, E)$ is a connected graph in which each edge $(u, v)$ in $E$ has a cost $c(u, v)$ attached to it. A *spanning tree* for $G$ is a free tree that con[nects] all the vertices in $V$. The *cost* of a spanning tree is the sum of the costs o[f the] edges in the tree. In this section we shall show how to find a minimum spanning tree for $G$.

**Example 7.4.** Figure 7.4 shows a weighted graph and its minimum-cost s[pan]ning tree. □

A typical application for minimum-cost spanning trees occurs in the d[esign] of communications networks. The vertices of a graph represent cities an[d the] edges possible communications links between the cities. The cost assoc[iated] with an edge represents the cost of selecting that link for the network[. A] minimum-cost spanning tree represents a communications network that [con]nects all the cities at minimal cost.

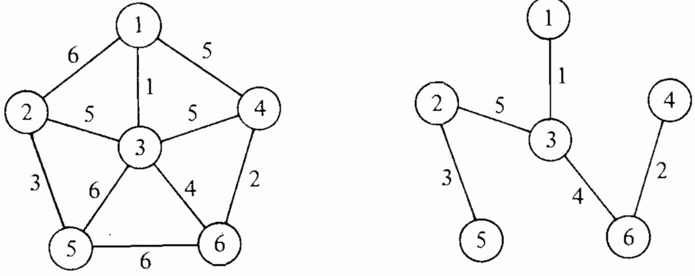



**Fig. 7.4.** A graph and spanning tree.

## The MST Property

There are several different ways to construct a minimum-cost spanning tree. Many of these methods use the following property of minimum-cost spanning trees, which we call the *MST property*. Let $G = (V, E)$ be a connected graph with a cost function defined on the edges. Let $U$ be some proper subset of the set of vertices $V$. If $(u, v)$ is an edge of lowest cost such that $u \in U$ and $v \in V-U$, then there is a minimum-cost spanning tree that includes $(u, v)$ as an edge.

The proof that every minimum-cost spanning tree satisfies the MST property is not hard. Suppose to the contrary that there is no minimum-cost spanning tree for $G$ that includes $(u, v)$. Let $T$ be any minimum-cost spanning tree for $G$. Adding $(u, v)$ to $T$ must introduce a cycle, since $T$ is a free tree and therefore satisfies property (2) for free trees. This cycle involves edge $(u, v)$. Thus, there must be another edge $(u', v')$ in $T$ such that $u' \in U$ and $v' \in V-U$, as illustrated in Fig. 7.5. If not, there would be no way for the cycle to get from $u$ to $v$ without following the edge $(u, v)$ a second time.

Deleting the edge $(u', v')$ breaks the cycle and yields a spanning tree $T'$ whose cost is certainly no higher than the cost of $T$ since by assumption $c(u, v) \leq c(u', v')$. Thus, $T'$ contradicts our assumption that there is no minimum-cost spanning tree that includes $(u, v)$.

## Prim's Algorithm

There are two popular techniques that exploit the MST property to construct a minimum-cost spanning tree from a weighted graph $G = (V, E)$. One such method is known as Prim's algorithm. Suppose $V = \{1, 2, \ldots, n\}$. Prim's algorithm begins with a set $U$ initialized to $\{1\}$. It then "grows" a spanning tree, one edge at a time. At each step, it finds a shortest edge $(u, v)$ that connects $U$ and $V-U$ and then adds $v$, the vertex in $V-U$, to $U$. It repeats this

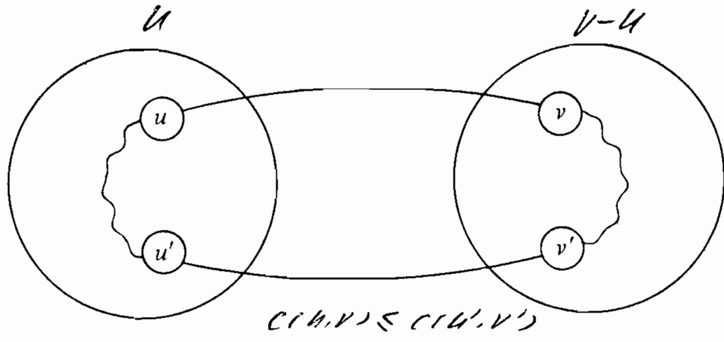
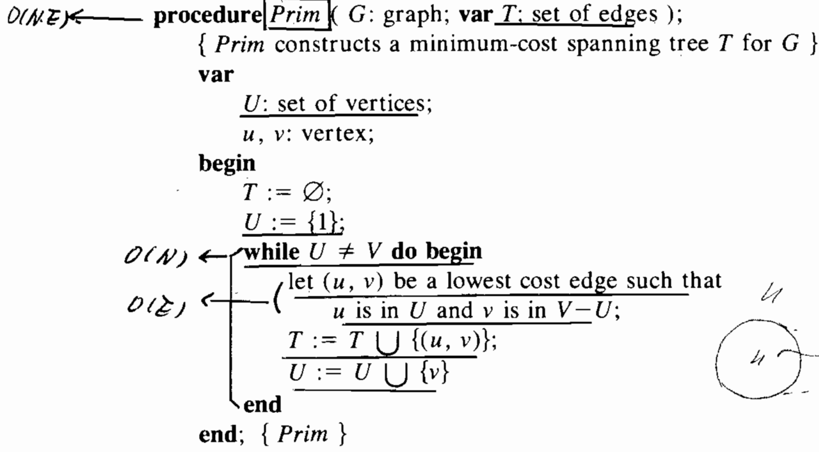



**Fig. 7.5.** Resulting cycle.

step until $U = V$. The algorithm is summarized in Fig. 7.6 and the sequence of edges added to $T$ for the graph of Fig. 7.4(a) is shown in Fig. 7.7.



```
procedure Prim ( G: graph; var T: set of edges );
    { Prim constructs a minimum-cost spanning tree T for G }
    var
        U: set of vertices;
        u, v: vertex;
    begin
        T := ∅;
        U := {1};
        while U ≠ V do begin
            let (u, v) be a lowest cost edge such that
                u is in U and v is in V−U;
            T := T ∪ {(u, v)};
            U := U ∪ {v}
        end
    end; { Prim }
```

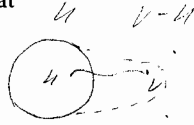


**Fig. 7.6.** Sketch of Prim's algorithm.

One simple way to find the lowest-cost edge between $U$ and $V-U$ at each step is to maintain two arrays. One array *CLOSEST*[$i$] gives the vertex in $U$ that is currently closest to vertex $i$ in $V-U$. The other array *LOWCOST*[$i$] gives the cost of the edge ($i$, *CLOSEST*[$i$]).

At each step we can scan *LOWCOST* to find the vertex, say $k$, in $V-U$ that is closest to $U$. We print the edge ($k$, *CLOSEST*[$k$]). We then update the *LOWCOST* and *CLOSEST* arrays, taking into account the fact that $k$ has been

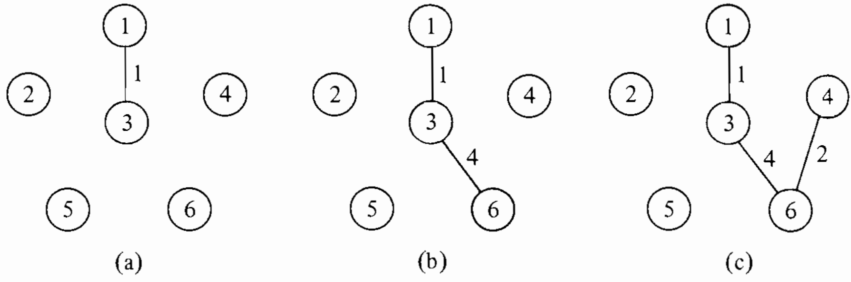


**Fig. 7.7.** Sequences of edges added by Prim's algorithm.

added to $U$. A Pascal version of this algorithm is given in Fig. 7.8. We assume $C$ is an $n \times n$ array such that $C[i, j]$ is the cost of edge $(i, j)$. If edge $(i, j)$ does not exist, we assume $C[i, j]$ is some appropriate large value.

Whenever we find another vertex $k$ for the spanning tree, we make $LOWCOST[k]$ be *infinity*, a very large value, so this vertex will no longer be considered in subsequent passes for inclusion in $U$. The value *infinity* is greater than the cost of any edge or the cost associated with a missing edge.

The time complexity of Prim's algorithm is $O(n^2)$, since we make $n-1$ iterations of the loop of lines (4)–(16) and each iteration of the loop takes $O(n)$ time, due to the inner loops of lines (7)–(10) and (13)–(16). As $n$ gets large the performance of this algorithm may become unsatisfactory. We now give another algorithm due to Kruskal for finding minimum-cost spanning trees whose performance is at most $O(e \log e)$, where $e$ is the number of edges in the given graph. If $e$ is much less than $n^2$, Kruskal's algorithm is superior, although if $e$ is about $n^2$, we would prefer Prim's algorithm.

```
             procedure Prim ( C: array[1..n, 1..n] of real );
                 { Prim prints the edges of a minimum-cost spanning tree for
                     with vertices {1, 2, . . . ,n} and cost matrix C on edges }
                 var
                     LOWCOST: array[1..n] of real;
                     CLOSEST: array[1..n] of integer;
                     i, j, k, min: integer;
                     { i and j are indices. During a scan of the LOWCOST ar
                         k is the index of the closest vertex found so far, and
                         min = LOWCOST[k] }
                 begin
(1)                  for i := 2 to n do begin
                         { initialize with only vertex 1 in the set U }
(2)                      LOWCOST[i] := C[1, i];
(3)                      CLOSEST[i] := 1
                     end;
(4)                  for i := 2 to n do begin
                         { find the closest vertex k outside of U to
                             some vertex in U }
(5)                      min := LOWCOST[2];
(6)                      k := 2;
(7)                      for j := 3 to n do
(8)                          if LOWCOST[j] < min then begin
(9)                              min := LOWCOST[j];
(10)                             k := j
                             end;
(11)                     writeln(k, CLOSEST[k]); { print edge }
(12)                     LOWCOST[k] := infinity; { k is added to U }
(13)                     for j := 2 to n do { adjust costs to U }
(14)                         if (C[k, j] < LOWCOST[j]) and
                                 (LOWCOST[j] < infinity) then begin
(15)                             LOWCOST[j] := C[k, j];
(16)                             CLOSEST[j] := k
                             end
                     end
                 end; { Prim }
```

**Fig. 7.8.** Prim's algorithm.

## II. Kruskal's Algorithm

Suppose again we are given a connected graph $G = (V, E)$, $V = \{1, 2, \ldots, n\}$ and a cost function $c$ defined on the edges of $E$. An way to construct a minimum-cost spanning tree for $G$ is to start with a

$T = (V, \emptyset)$ consisting only of the $n$ vertices of $G$ and having no edges. Each vertex is therefore in a connected component by itself. As the algorithm proceeds, we shall always have a collection of connected components, and for each component we shall have selected edges that form a spanning tree.

To build progressively larger components, we examine edges from $E$, in order of increasing cost. If the edge connects two vertices in two different connected components, then we add the edge to $T$. If the edge connects two vertices in the same component, then we discard the edge, since it would cause a cycle if we added it to the spanning tree for that connected component. When all vertices are in one component, $T$ is a minimum-cost spanning tree for $G$.

**Example 7.5.** Consider the weighted graph of Fig. 7.4(a). The sequence of edges added to $T$ is shown in Fig. 7.9. The edges of cost 1, 2, 3, and 4 are considered first, and all are accepted, since none of them causes a cycle. The edges (1, 4) and (3, 4) of cost 5 cannot be accepted, because they connect vertices in the same component in Fig. 7.9(d), and therefore would complete a cycle. However, the remaining edge of cost 5, namely (2, 3), does not create a cycle. Once it is accepted, we are done. □

We can implement this algorithm using sets and set operations discussed in Chapters 4 and 5. First, we need a set consisting of the edges in $E$. We then apply the DELETEMIN operator repeatedly to this set to select edges in order of increasing cost. The set of edges therefore forms a priority queue, and a partially ordered tree is an appropriate data structure to use here. We also need to maintain a set of connected components $C$. The operations we apply to it are:

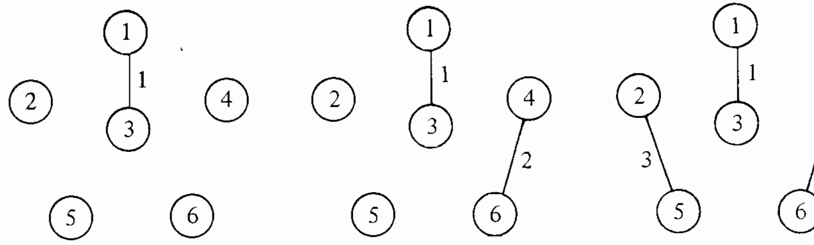
1. MERGE($A$, $B$, $C$) to merge the components $A$ and $B$ in $C$ and to call the result either $A$ or $B$ arbitrarily.†
2. FIND($v$, $C$) to return the name of the component of $C$ of which vertex $v$ is a member. This operation will be used to determine whether the two vertices of an edge are in the same or different components.
3. INITIAL($A$, $v$, $C$) to make $A$ the name of a component in $C$ containing only vertex $v$ initially.

These are the operations of the MERGE-FIND ADT called MFSET, which we encountered in Section 5.5. A sketch of a program called *Kruskal* to find a minimum-cost spanning tree using these operations is shown in Fig. 7.10.

We can use the techniques of Section 5.5 to implement the operations used in this program. The running time of this program is dependent on two factors. If there are $e$ edges, it takes $O(e\log e)$ time to insert the edges into the priority queue.‡ In each iteration of the while-loop, finding the least cost

† Note that MERGE and FIND are defined slightly differently from Section 5.5, since $C$ is a parameter telling where $A$ and $B$ can be found.

‡ We can initialize a partially ordered tree of $e$ elements in $O(e)$ time if we do it all at once. We discuss this technique in Section 8.4, and we should probably use it here, since



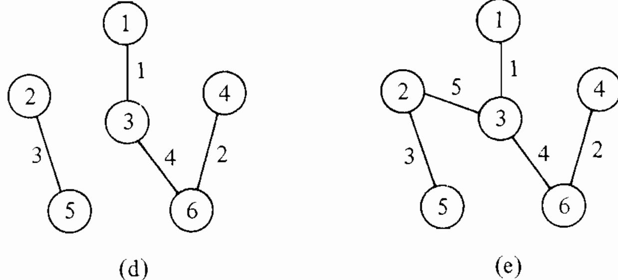




**Fig. 7.9.** Sequence of edges added by Kruskal's algorithm.

edge in *edges* takes $O(\log e)$ time. Thus, the priority queue operations $O(e \log e)$ time in the worst case. The total time required to perform MERGE and FIND operations depends on the method used to implement MFSET. As shown in Section 5.5, there are $O(e \log e)$ and $O(e\ \alpha$ methods. In either case, Kruskal's algorithm can be implemented to ru $O(e \log e)$ time.

### 7.3 Traversals

In a number of graph problems, we need to visit the vertices of a graph tematically. Depth-first search and breadth-first search, the subjects of section, are two important techniques for doing this. Both techniques ca used to determine efficiently all vertices that are connected to a given vert

if many fewer than $e$ edges are examined before the minimum-cost spanning tree is found, we may save significant time.