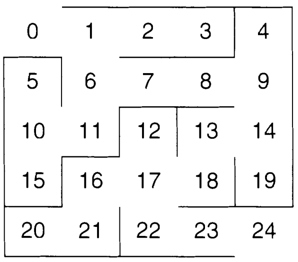


{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24}

Figure 24.5 Eventually, 24 walls have been knocked down, and all the elements are in the same set.

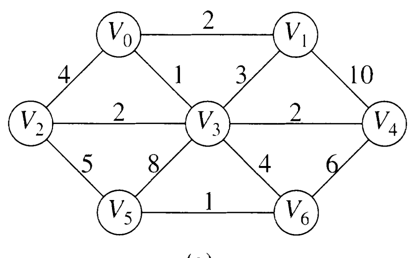
The running time of the algorithm is dominated by the union/find costs. The size of the union/find universe is the number of cells. The number of `find` operations is proportional to the number of cells because the number of removed walls is 1 less than the number of cells. If we look carefully, however, we can see that there are only about twice as many walls as cells in the first place. Thus, if N is the number of cells and as there are two `find`s per randomly targeted wall, we get an estimate of between (roughly) $2N$ and $4N$ `find` operations throughout the algorithm. Therefore the algorithm's running time depends on the cost of $O(N)$ `union` and $O(N)$ `find` operations.

24.2.2 Application: Minimum Spanning Trees

The *minimum spanning tree* is a connected subgraph of *G* that spans all vertices at minimum total cost.

A **spanning tree** of an undirected graph is a tree formed by graph edges that connect all the vertices of the graph. Unlike the graphs in Chapter 15, an edge (u, v) in a graph G is identical to an edge (v, u). The cost of a spanning tree is the sum of the costs of the edges in the tree. The **minimum spanning tree** is a connected subgraph of G that spans all vertices at minimum cost. A minimum spanning tree exists only if the subgraph of G is connected. As we show shortly, testing a graph's connectivity can be done as part of the minimum spanning tree computation.

In Figure 24.6(b), the graph is a minimum spanning tree of the graph in Figure 24.6(a) (it happens to be unique, which is unusual if the graph has many edges of equal cost). Note that the number of edges in the minimum spanning tree is $|V| - 1$. The minimum spanning tree is a *tree* because it is acyclic, it is *spanning* because it covers every vertex, and it is *minimum* for the obvious reason. Suppose that we need to connect several towns with roads, minimizing the total construction cost, with the provision that we can



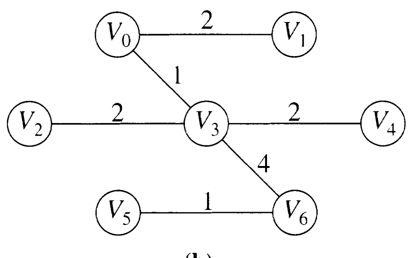


Figure 24.6 (a) A graph *G* and (b) its minimum spanning tree.

transfer to another road only at a town (in other words, no extra junctions are allowed). Then we need to solve a minimum spanning tree problem, where each vertex is a town, and each edge is the cost of building a road between the two cities it connects.

A related problem is the *minimum Steiner tree problem,* which is like the minimum spanning tree problem, except that junctions can be created as part of the solution. The minimum Steiner tree problem is much more difficult to solve. However, it can be shown that if the cost of a connection is proportional to the Euclidean distance, the minimum spanning tree is at most 15 percent more expensive than the minimum Steiner tree. Thus a minimum spanning tree, which is easy to compute, provides a good approximation for the minimum Steiner tree, which is hard to compute.

Kruskal's algorithm is used to select edges in order of increasing cost and adds an edge to the tree if it does not create a cycle.

A simple algorithm, commonly called **Kruskal's algorithm,** is used to select edges continually in order of smallest weight and to add an edge to the tree if it does not cause a cycle. Formally, Kruskal's algorithm maintains a forest—a collection of trees. Initially, there are $|V|$ single-node trees. Adding an edge merges two trees into one. When the algorithm terminates, there is only one tree, which is the minimum spanning tree.[2] By counting the number of accepted edges, we can determine when the algorithm should terminate.

Figure 24.7 shows the action of Kruskal's algorithm on the graph shown in Figure 24.6. The first five edges are all accepted because they do not create cycles. The next two edges, (v_1, v_3) (of cost 3) and then (v_0, v_2) (of cost 4), are rejected because each would create a cycle in the tree. The next edge considered is accepted, and because it is the sixth edge in a seven-vertex graph, we can terminate the algorithm.

2. If the graph is not connected, the algorithm will terminate with more than one tree. Each tree then represents a minimum spanning tree for each connected component of the graph.

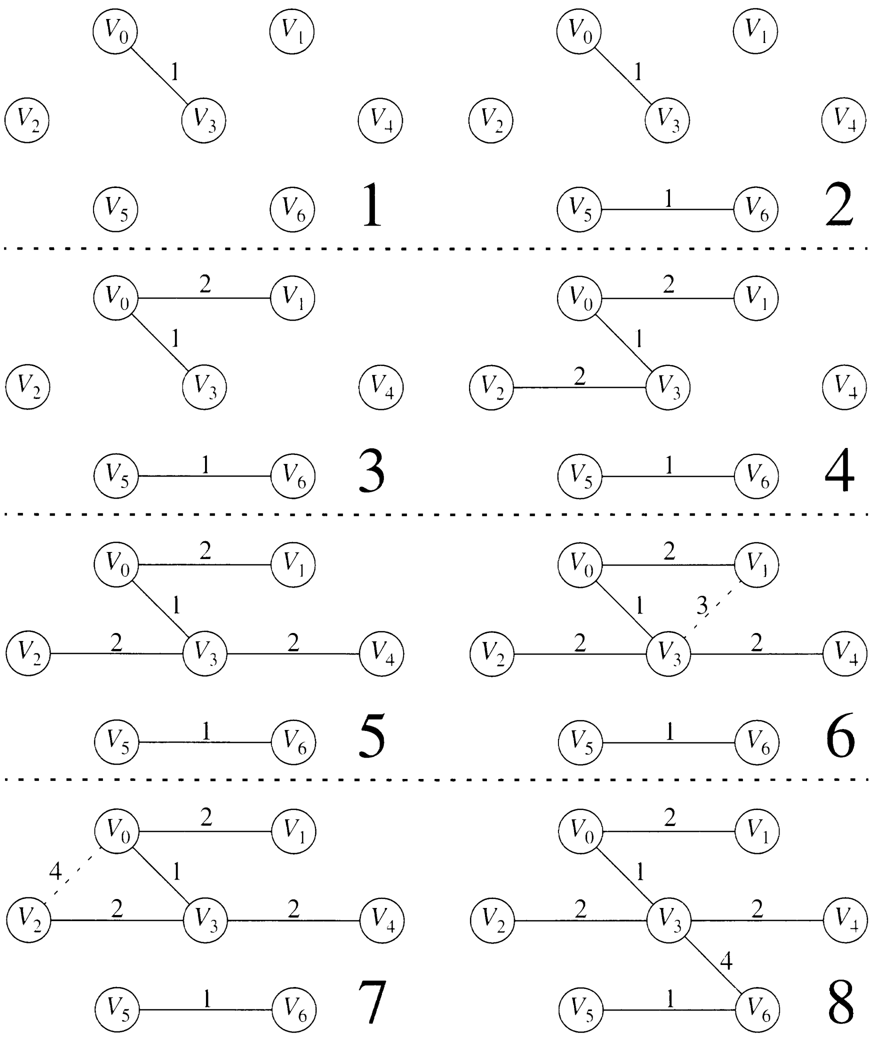


Figure 24.7 Kruskal's algorithm after each edge has been considered. The stages proceed left-to-right, top-to-bottom, as numbered.

The edges can be sorted, or a priority queue can be used.

Ordering the edges for testing is simple enough to do. We can sort them at a cost of $|E|\log|E|$ and then step through the ordered array of edges. Alternatively, we can construct a priority queue of $|E|$ edges and repeatedly obtain edges by calling `deleteMin`. Although the worst-case bound is unchanged, using a priority queue is sometimes better because Kruskal's algorithm tends to test only a small fraction of the edges on random graphs. Of course, in the worst case, all the edges may have to be tried. For instance, if there were an extra vertex v_8 and edge (v_5, v_8) of cost 100, all the edges would have to be examined. In this case, a quicksort at the start would be faster. In effect, the choice between a priority queue and an initial sort is a gamble on how many edges are likely to have to be examined.

More interesting is the issue of how we decide whether an edge (u, v) should be accepted or rejected. Clearly, adding the edge (u, v) causes a cycle if (and only if) u and v are already connected in the current spanning **forest,** which is a collection of trees. Thus we merely maintain each connected component in the spanning forest as a disjoint set. Initially, each vertex is in its own disjoint set. If u and v are in the same disjoint set, as determined by two `find` operations, the edge is rejected because u and v are already connected. Otherwise, the edge is accepted and a `union` operation is performed on the two disjoint sets containing u and v, in effect, combining the connected components. This result is what we want because once edge (u, v) has been added to the spanning forest, if w was connected to u and x was connected to v, x and w must be connected and thus belong in the same set.

The test for cycles is done by using a union/find data structure.

24.2.3 Application: The Nearest Common Ancestor Problem

Another illustration of the union/find data structure is the offline **nearest common ancestor** (NCA) **problem.**

OFFLINE NEAREST COMMON ANCESTOR PROBLEM

GIVEN A TREE AND A LIST OF PAIRS OF NODES IN THE TREE, FIND THE NEAREST COMMON ANCESTOR FOR EACH PAIR OF NODES.

As an example, Figure 24.8 shows a tree with a pair list containing five requests. For the pair of nodes u and z, node C is the nearest ancestor of both. (A and B are also ancestors, but they are not the closest.) The problem is offline because we can see the entire request sequence prior to providing the first answer. Solution of this problem is important in graph theory applications and computational biology (where the tree represents evolution) applications.

Solutions of the NCA is important in graph algorithm and computational biology applications.

The algorithm works by performing a postorder tree traversal. When we are about to return from processing a node, we examine the pair list to determine whether any ancestor calculations are to be performed. If u is the current node, (u, v) is in the pair list and we have already finished the recursive call to v, we have enough information to determine NCA(u, v).

A postorder traversal can be used to solve the problem.

Figure 24.9 helps in understanding how this algorithm works. Here, we are about to finish the recursive call to D. All shaded nodes have been visited by a recursive call, and except for the nodes on the path to D, all the recursive calls have already finished. We mark a node after its recursive call has been completed. If v is marked, then NCA(D, v) is some node on the path to D. The **anchor** of a visited (but not necessarily marked) node v is the node

The *anchor* of a visited (but not necessarily marked) node *v* is the node on the current access path that is closest to *v*.