9.2 Kruskal's Algorithm

In the previous section, we considered the greedy algorithm that "grows" a minimum spanning tree through a greedy inclusion of the nearest vertex to the vertices already in the tree. Remarkably, there is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named ***Kruskal's algorithm*** after Joseph Kruskal, who discovered this algorithm when he was a second-year graduate student [Kru56]. Kruskal's algorithm looks at a minimum spanning tree of a weighted connected graph $G = \langle V, E \rangle$ as an acyclic subgraph with $|V| - 1$ edges for which the sum of the edge weights is the smallest. (It is not difficult to prove that such a subgraph must be a tree.) Consequently, the algorithm constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list, adding the next edge on the list to the current subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

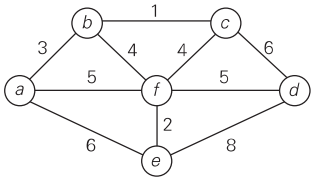
ALGORITHM *Kruskal(G)*

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = ⟨V, E⟩
//Output: E_T, the set of edges composing a minimum spanning tree of G
sort E in nondecreasing order of the edge weights w(e_{i_1}) ≤ ··· ≤ w(e_{i_|E|})
E_T ← ∅;  ecounter ← 0      //initialize the set of tree edges and its size
k ← 0                       //initialize the number of processed edges
while ecounter < |V| − 1 do
    k ← k + 1
    if E_T ∪ {e_{i_k}} is acyclic
        E_T ← E_T ∪ {e_{i_k}};  ecounter ← ecounter + 1
return E_T
```

The correctness of Kruskal's algorithm can be proved by repeating the essential steps of the proof of Prim's algorithm given in the previous section. The fact that E_T is actually a tree in Prim's algorithm but generally just an acyclic subgraph in Kruskal's algorithm turns out to be an obstacle that can be overcome.

Figure 9.5 demonstrates the application of Kruskal's algorithm to the same graph we used for illustrating Prim's algorithm in Section 9.1. As you trace the algorithm's operations, note the disconnectedness of some of the intermediate subgraphs.

Applying Prim's and Kruskal's algorithms to the same small graph by hand may create the impression that the latter is simpler than the former. This impression is wrong because, on each of its iterations, Kruskal's algorithm has to check whether the addition of the next edge to the edges already selected would create a



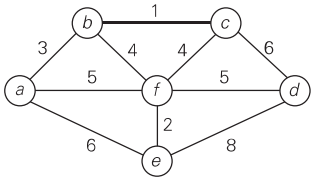
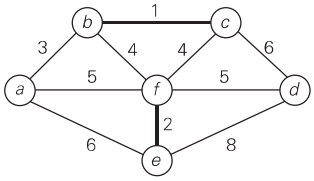
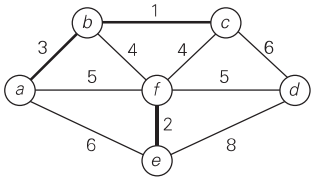
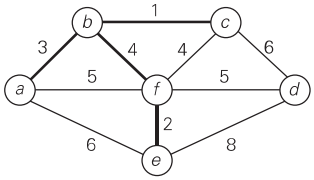
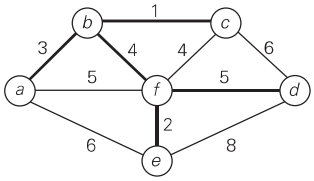
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de **1** 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 **2** 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 **3** 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 **4** 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 **5** 6 6 8	
df 5		

FIGURE 9.5 Application of Kruskal's algorithm. Selected edges are shown in bold.

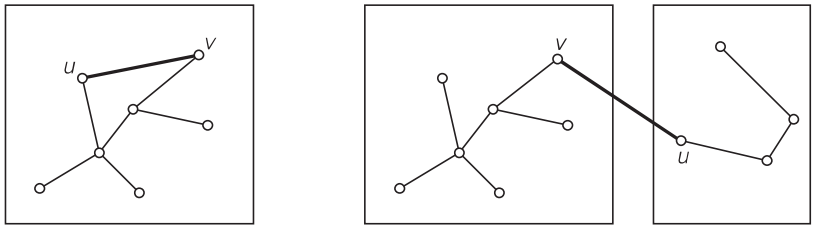


FIGURE 9.6 New edge connecting two vertices may (a) or may not (b) create a cycle.

cycle. It is not difficult to see that a new cycle is created if and only if the new edge connects two vertices already connected by a path, i.e., if and only if the two vertices belong to the same connected component (Figure 9.6). Note also that each connected component of a subgraph generated by Kruskal's algorithm is a tree because it has no cycles.

In view of these observations, it is convenient to use a slightly different interpretation of Kruskal's algorithm. We can consider the algorithm's operations as a progression through a series of forests containing *all* the vertices of a given graph and *some* of its edges. The initial forest consists of $|V|$ trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called ***union-find*** algorithms. We discuss them in the following subsection. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm will be in $O(|E| \log |E|)$.

Disjoint Subsets and Union-Find Algorithms

Kruskal's algorithm is one of a number of applications that require a dynamic partition of some n element set S into a collection of disjoint subsets $S_1, S_2, \ldots, S_k$. After being initialized as a collection of n one-element subsets, each containing a different element of S, the collection is subjected to a sequence of intermixed union and find operations. (Note that the number of union operations in any such sequence must be bounded above by $n - 1$ because each union increases a subset's size at least by 1 and there are only n elements in the entire set S.) Thus, we are

dealing here with an abstract data type of a collection of disjoint subsets of a finite set with the following operations:

makeset(x) creates a one-element set $\{x\}$. It is assumed that this operation can be applied to each of the elements of set S only once.

find(x) returns a subset containing x.

union(x, y) constructs the union of the disjoint subsets S_x and S_y containing x and y, respectively, and adds it to the collection to replace S_x and S_y, which are deleted from it.

For example, let $S = \{1, 2, 3, 4, 5, 6\}$. Then *makeset*(i) creates the set $\{i\}$ and applying this operation six times initializes the structure to the collection of six singleton sets:

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

Performing *union*(1, 4) and *union*(5, 2) yields

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

and, if followed by *union*(4, 5) and then by *union*(3, 6), we end up with the disjoint subsets

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

Most implementations of this abstract data type use one element from each of the disjoint subsets in a collection as that subset's ***representative***. Some implementations do not impose any specific constraints on such a representative; others do so by requiring, say, the smallest element of each subset to be used as the subset's representative. Also, it is usually assumed that set elements are (or can be mapped into) integers.

There are two principal alternatives for implementing this data structure. The first one, called the ***quick find***, optimizes the time efficiency of the find operation; the second one, called the ***quick union***, optimizes the union operation.

The quick find uses an array indexed by the elements of the underlying set S; the array's values indicate the representatives of the subsets containing those elements. Each subset is implemented as a linked list whose header contains the pointers to the first and last elements of the list along with the number of elements in the list (see Figure 9.7 for an example).

Under this scheme, the implementation of *makeset*(x) requires assigning the corresponding element in the representative array to x and initializing the corresponding linked list to a single node with the x value. The time efficiency of this operation is obviously in $\Theta(1)$, and hence the initialization of n singleton subsets is in $\Theta(n)$. The efficiency of *find*(x) is also in $\Theta(1)$: all we need to do is to retrieve the x's representative in the representative array. Executing *union*(x, y) takes longer. A straightforward solution would simply append the y's list to the end of the x's list, update the information about their representative for all the elements in the

subset representatives	
element index	representative
1	1
2	1
3	3
4	1
5	1
6	3

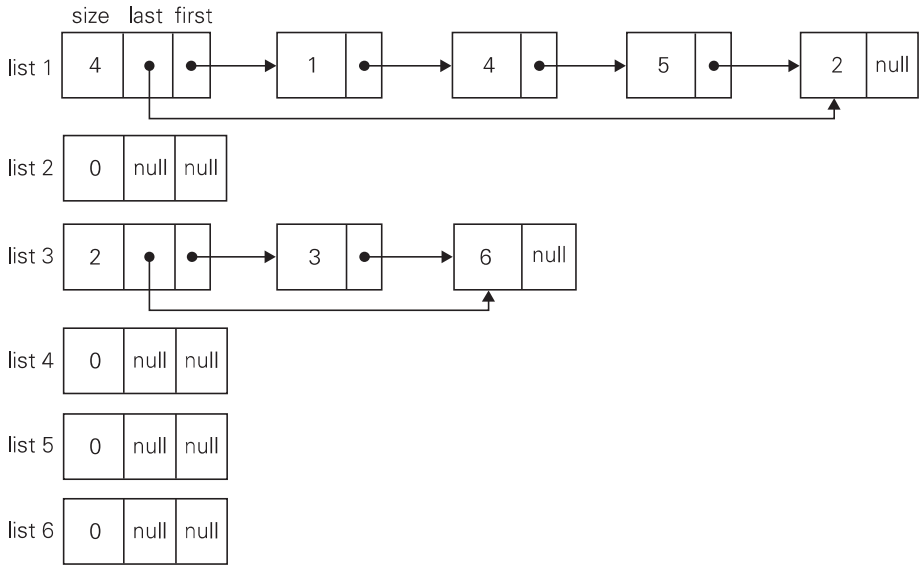


FIGURE 9.7 Linked-list representation of subsets {1, 4, 5, 2} and {3, 6} obtained by quick find after performing *union*(1, 4), *union*(5, 2), *union*(4, 5), and *union*(3, 6). The lists of size 0 are considered deleted from the collection.

y list, and then delete the y's list from the collection. It is easy to verify, however, that with this algorithm the sequence of union operations

$$union(2, 1), union(3, 2), \ldots, union(i + 1, i), \ldots, union(n, n - 1)$$

runs in $\Theta(n^2)$ time, which is slow compared with several known alternatives.

A simple way to improve the overall efficiency of a sequence of union operations is to always append the shorter of the two lists to the longer one, with ties broken arbitrarily. Of course, the size of each list is assumed to be available by, say, storing the number of elements in the list's header. This modification is called the

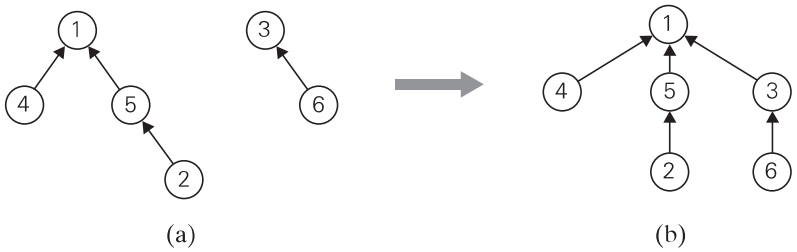


FIGURE 9.8 (a) Forest representation of subsets {1, 4, 5, 2} and {3, 6} used by quick union. (b) Result of *union*(5, 6).

union by size. Though it does not improve the worst-case efficiency of a single application of the union operation (it is still in $\Theta(n)$), the worst-case running time of any legitimate sequence of union-by-size operations turns out to be in $O(n \log n)$.[3]

Here is a proof of this assertion. Let a_i be an element of set S whose disjoint subsets we manipulate, and let A_i be the number of times a_i's representative is updated in a sequence of union-by-size operations. How large can A_i get if set S has n elements? Each time a_i's representative is updated, a_i must be in a smaller subset involved in computing the union whose size will be at least twice as large as the size of the subset containing a_i. Hence, when a_i's representative is updated for the first time, the resulting set will have at least two elements; when it is updated for the second time, the resulting set will have at least four elements; and, in general, if it is updated A_i times, the resulting set will have at least 2^{A_i} elements. Since the entire set S has n elements, $2^{A_i} \leq n$ and hence $A_i \leq \log_2 n$. Therefore, the total number of possible updates of the representatives for all n elements in S will not exceed $n \log_2 n$.

Thus, for union by size, the time efficiency of a sequence of at most $n - 1$ unions and m finds is in $O(n \log n + m)$.

The ***quick union***—the second principal alternative for implementing disjoint subsets—represents each subset by a rooted tree. The nodes of the tree contain the subset's elements (one per node), with the root's element considered the subset's representative; the tree's edges are directed from children to their parents (Figure 9.8). In addition, a mapping of the set elements to their tree nodes—implemented, say, as an array of pointers—is maintained. This mapping is not shown in Figure 9.8 for the sake of simplicity.

For this implementation, *makeset*(x) requires the creation of a single-node tree, which is a $\Theta(1)$ operation; hence, the initialization of n singleton subsets is in $\Theta(n)$. A *union*(x, y) is implemented by attaching the root of the y's tree to the root of the x's tree (and deleting the y's tree from the collection by making the pointer to its root null). The time efficiency of this operation is clearly $\Theta(1)$. A *find*(x) is

3. This is a specific example of the usefulness of the *amortized efficiency* we mentioned back in Chapter 2.

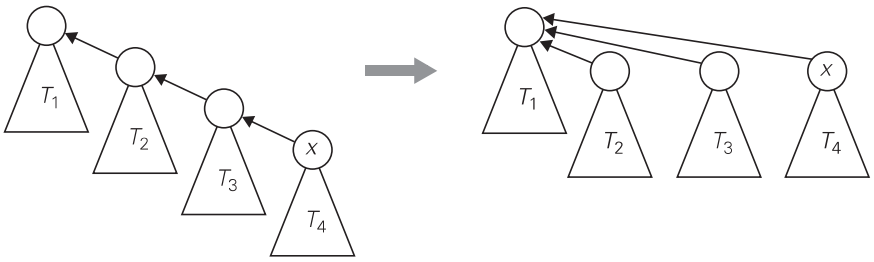


FIGURE 9.9 Path compression.

performed by following the pointer chain from the node containing x to the tree's root whose element is returned as the subset's representative. Accordingly, the time efficiency of a single find operation is in $O(n)$ because a tree representing a subset can degenerate into a linked list with n nodes.

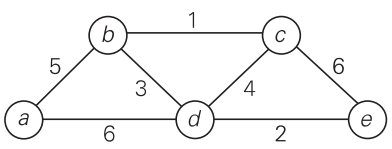
This time bound can be improved. The straightforward way for doing so is to always perform a union operation by attaching a smaller tree to the root of a larger one, with ties broken arbitrarily. The size of a tree can be measured either by the number of nodes (this version is called ***union by size***) or by its height (this version is called ***union by rank***). Of course, these options require storing, for each node of the tree, either the number of node descendants or the height of the subtree rooted at that node, respectively. One can easily prove that in either case the height of the tree will be logarithmic, making it possible to execute each find in $O(\log n)$ time. Thus, for quick union, the time efficiency of a sequence of at most $n - 1$ unions and m finds is in $O(n + m \log n)$.

In fact, an even better efficiency can be obtained by combining either variety of quick union with ***path compression***. This modification makes every node encountered during the execution of a find operation point to the tree's root (Figure 9.9). According to a quite sophisticated analysis that goes beyond the level of this book (see [Tar84]), this and similar techniques improve the efficiency of a sequence of at most $n - 1$ unions and m finds to only slightly worse than linear.

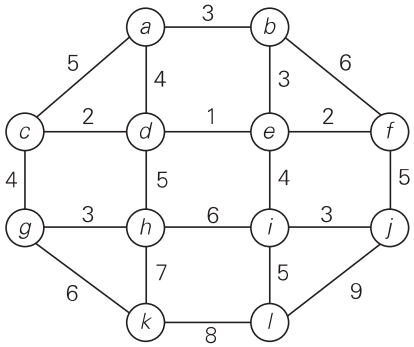
Exercises 9.2

1. Apply Kruskal's algorithm to find a minimum spanning tree of the following graphs.

a.



b.



2. Indicate whether the following statements are true or false:

a. If e is a minimum-weight edge in a connected weighted graph, it must be among edges of at least one minimum spanning tree of the graph.

b. If e is a minimum-weight edge in a connected weighted graph, it must be among edges of each minimum spanning tree of the graph.

c. If edge weights of a connected weighted graph are all distinct, the graph must have exactly one minimum spanning tree.

d. If edge weights of a connected weighted graph are not all distinct, the graph must have more than one minimum spanning tree.

3. What changes, if any, need to be made in algorithm *Kruskal* to make it find a ***minimum spanning forest*** for an arbitrary graph? (A minimum spanning forest is a forest whose trees are minimum spanning trees of the graph's connected components.)

4. Does Kruskal's algorithm work correctly on graphs that have negative edge weights?

5. Design an algorithm for finding a ***maximum spanning tree***—a spanning tree with the largest possible edge weight—of a weighted connected graph.

6. Rewrite pseudocode of Kruskal's algorithm in terms of the operations of the disjoint subsets' ADT.

7. Prove the correctness of Kruskal's algorithm.

8. Prove that the time efficiency of $find(x)$ is in $O(\log n)$ for the union-by-size version of quick union.

9. Find at least two Web sites with animations of Kruskal's and Prim's algorithms. Discuss their merits and demerits.

10. Design and conduct an experiment to empirically compare the efficiencies of Prim's and Kruskal's algorithms on random graphs of different sizes and densities.