

# FUTURE VISION BIE

One Stop for All Study Materials  
& Lab Programs



*Future Vision*

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page



Or

Visit : <https://hemanthrajhemu.github.io>

Gain Access to All Study Materials according to VTU,  
CSE – Computer Science Engineering,  
ISE – Information Science Engineering,  
ECE - Electronics and Communication Engineering  
& MORE...

Join Telegram to get Instant Updates: [https://bit.ly/VTU\\_TELEGRAM](https://bit.ly/VTU_TELEGRAM)

Contact: MAIL: [futurevisionbie@gmail.com](mailto:futurevisionbie@gmail.com)

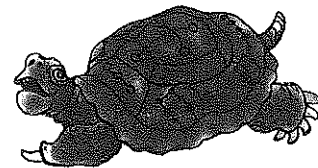
INSTAGRAM: [www.instagram.com/hemanthraj\\_hemu/](http://www.instagram.com/hemanthraj_hemu/)

INSTAGRAM: [www.instagram.com/futurevisionbie/](http://www.instagram.com/futurevisionbie/)

WHATSAPP SHARE: <https://bit.ly/FVBIESHARE>

Introduction to **The Design &  
Analysis of Algorithms**

**2ND EDITION**



**Anany Levitin**  
*Villanova University*



Boston San Francisco New York  
London Toronto Sydney Tokyo Singapore Madrid  
Mexico City Munich Paris Cape Town Hong Kong Montreal

<https://hemanthrajhemu.github.io>

<b>9</b>	<b>Greedy Technique</b>	<b>307</b>
9.1	Prim's Algorithm	308
	Exercises 9.1	313
9.2	Kruskal's Algorithm	315
	Disjoint Subsets and Union-Find Algorithms	317
	Exercises 9.2	321
9.3	Dijkstra's Algorithm	323
	Exercises 9.3	327
9.4	Huffman Trees	328
	Exercises 9.4	332
	Summary	333
<b>10</b>	<b>Iterative Improvement</b>	<b>335</b>
10.1	The Simplex Method	336
	Geometric Interpretation of Linear Programming	337
	An Outline of the Simplex Method	341
	Further Notes on the Simplex Method	347
	Exercises 10.1	349
10.2	The Maximum-Flow Problem	351
	Exercises 10.2	361
10.3	Maximum Matching in Bipartite Graphs	363
	Exercises 10.3	369
10.4	The Stable Marriage Problem	371
	Exercises 10.4	375
	Summary	376
<b>11</b>	<b>Limitations of Algorithm Power</b>	<b>379</b>
11.1	Lower-Bound Arguments	380
	Trivial Lower Bounds	381
	Information-Theoretic Arguments	382

# 9

## Greedy Technique

Greed, for lack of a better word, is good! Greed is right! Greed works!

—Michael Douglas, U.S. actor in the role of Gordon Gecko,  
in the film *Wall Street*, 1987

Let us start with the *change-making problem* faced by millions of cashiers all over the world (at least subconsciously): give change for a specific amount  $n$  with the least number of coins of the denominations  $d_1 > d_2 > \dots > d_m$  used in that locale. For example, the widely used coin denominations in the United States are  $d_1 = 25$  (quarter),  $d_2 = 10$  (dime),  $d_3 = 5$  (nickel), and  $d_4 = 1$  (penny). How would you give change with coins of these denominations of, say, 48 cents? If you came up with the answer 1 quarter, 2 dimes, and 3 pennies, you followed—consciously or not—a logical strategy of making a sequence of best choices among the currently available alternatives. Indeed, in the first step, you could have given one coin of any of the four denominations. “Greedy” thinking leads to giving one quarter because it reduces the remaining amount the most, namely, to 23 cents. In the second step, you had the same coins at your disposal, but you could not give a quarter because it would have violated the problem’s constraints. So your best selection in this step was one dime, reducing the remaining amount to 13 cents. Giving one more dime left you with 3 cents to be given with three pennies.

Is this solution to the instance of the change-making problem optimal? Yes, it is. In fact, it is possible to prove that the greedy algorithm yields an optimal solution for every positive integer amount with these coin denominations. At the same time, it is easy to give an example of “weird” coin denominations—e.g.,  $d_1 = 7$ ,  $d_2 = 5$ ,  $d_3 = 1$ —that may not yield an optimal solution for some amounts. (It is the reason you were asked to develop a dynamic programming algorithm for this problem in Exercises 8.4: that algorithm works for any set of coin denominations by either returning an optimal solution or reporting that no solution exists.)

The approach applied in the opening paragraph to the change-making problem is called *greedy*. Computer scientists consider it a general design technique despite the fact that it is applicable to optimization problems only. The greedy

approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step—and this is the central point of this technique—the choice made must be

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm

These requirements explain the technique's name: on each step, it suggests a “greedy” grab of the best alternative available in the hope that a sequence of locally optimal choices will yield a (globally) optimal solution to the entire problem. We refrain from a philosophical discussion of whether greed is good or bad. (If you have not seen the movie from which the chapter's epigraph is taken, its hero did not end up well.) From our algorithmic perspective, the question is whether a greedy strategy works or not. As we shall see, there are problems for which a sequence of locally optimal choices does yield an optimal solution for every instance of the problem in question. However, there are others for which this is not the case; for such problems, a greedy algorithm can still be of value if we are interested in or have to be satisfied with an approximate solution.

In the first two sections of the chapter, we discuss two classic algorithms for the minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. What is remarkable about these algorithms is the fact that they solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution. In Section 9.3, we introduce another classic algorithm—Dijkstra's algorithm for the shortest-path problem in a weighted graph. Section 9.4 is devoted to Huffman trees and their principal application, Huffman codes—an important data compression method that can be interpreted as an application of the greedy technique. Finally, a few examples of approximation algorithms based on the greedy approach are discussed in Section 12.3.

As a rule, greedy algorithms are both intuitively appealing and simple. Despite their apparent simplicity, there is a rather sophisticated theory behind the technique, which is based on the abstract combinatorial structure called “matroid.” We are not going to discuss it here; an interested reader can check such sources as [Cor01].

## 9.1 Prim's Algorithm

The following problem arises naturally in several practical situations: given  $n$  points, connect them in the cheapest possible way so that there will be a path between every pair of points. We can represent the points by vertices of a graph,

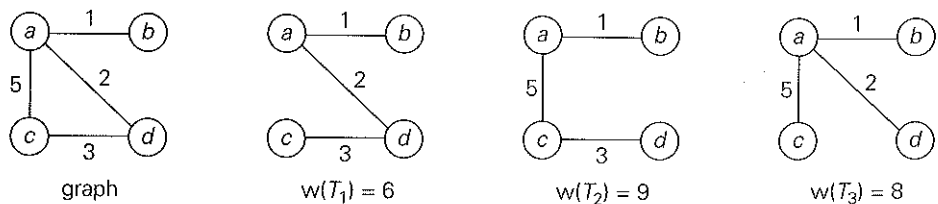
possible connections by the graph's edges, and the connection costs by the edge weights. Then the question can be posed as the minimum spanning tree problem, defined formally as follows.

**DEFINITION** A *spanning tree* of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. A *minimum spanning tree* of a weighted connected graph is its spanning tree of the smallest weight, where the *weight* of a tree is defined as the sum of the weights on all its edges. The *minimum spanning tree problem* is the problem of finding a minimum spanning tree for a given weighted connected graph.

Figure 9.1 presents a simple example illustrating these notions.

If we were to try an exhaustive-search approach to constructing a minimum spanning tree, we would face two serious obstacles. First, the number of spanning trees grows exponentially with the graph size (at least for dense graphs). Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a *minimum* spanning tree for a weighted graph by using one of several efficient algorithms available for this problem. In this section, we outline *Prim's algorithm*, which goes back to at least 1957 [Pri57].

Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set  $V$  of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. (By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily.) The algorithm stops after all the graph's vertices have been included in the tree being constructed. Since the algorithm expands a tree by exactly one vertex on each of its iterations, the total number of such iterations is  $n - 1$ , where  $n$  is the number of vertices in the graph. The tree generated by the algorithm is obtained as the set of edges used for the tree expansions.



**FIGURE 9.1** Graph and its spanning trees;  $T_1$  is the minimum spanning tree

Here is a pseudocode of this algorithm.

**ALGORITHM** *Prim(G)*

```
//Prim'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  $G$ 
 $V_T \leftarrow \{v_0\}$  //the set of tree vertices can be initialized with any vertex
 $E_T \leftarrow \emptyset$ 
for  $i \leftarrow 1$  to  $|V| - 1$  do
    find a minimum-weight edge  $e^* = (v^*, u^*)$  among all the edges  $(v, u)$ 
    such that  $v$  is in  $V_T$  and  $u$  is in  $V - V_T$ 
     $V_T \leftarrow V_T \cup \{u^*\}$ 
     $E_T \leftarrow E_T \cup \{e^*\}$ 
return  $E_T$ 
```

The nature of Prim's algorithm makes it necessary to provide each vertex not in the current tree with the information about the shortest edge connecting the vertex to a tree vertex. We can provide such information by attaching two labels to a vertex: the name of the nearest tree vertex and the length (the weight) of the corresponding edge. Vertices that are not adjacent to any of the tree vertices can be given the  $\infty$  label indicating their "infinite" distance to the tree vertices and a null label for the name of the nearest tree vertex. (Alternatively, we can split the vertices that are not in the tree into two sets, the "fringe" and the "unseen." The fringe contains only the vertices that are not in the tree but are adjacent to at least one tree vertex. These are the candidates from which the next tree vertex is selected. The unseen vertices are all the other vertices of the graph, called "unseen" because they are yet to be affected by the algorithm.) With such labels, finding the next vertex to be added to the current tree  $T = (V_T, E_T)$  becomes a simple task of finding a vertex with the smallest distance label in the set  $V - V_T$ . Ties can be broken arbitrarily.

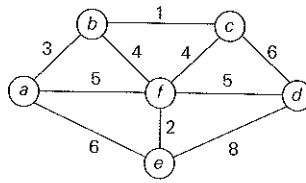
After we have identified a vertex  $u^*$  to be added to the tree, we need to perform two operations:

- Move  $u^*$  from the set  $V - V_T$  to the set of tree vertices  $V_T$ .
- For each remaining vertex  $u$  in  $V - V_T$  that is connected to  $u^*$  by a shorter edge than the  $u$ 's current distance label, update its labels by  $u^*$  and the weight of the edge between  $u^*$  and  $u$ , respectively.<sup>1</sup>

Figure 9.2 demonstrates the application of Prim's algorithm to a specific graph.

Does Prim's algorithm always yield a minimum spanning tree? The answer to this question is yes. Let us prove by induction that each of the subtrees  $T_i$ ,  $i = 0, \dots, n - 1$ , generated by Prim's algorithm is a part (i.e., a subgraph) of some

1. If the implementation with the fringe-unseen split is pursued, all the unseen vertices adjacent to  $u^*$  must also be moved to the fringe.



Tree vertices	Remaining vertices	Illustration
$a(-, -)$	<b>b(a, 3)</b> c(-, $\infty$ ) d(-, $\infty$ ) e(a, 6) f(a, 5)	
$b(a, 3)$	<b>c(b, 1)</b> d(-, $\infty$ ) e(a, 6) f(b, 4)	
$c(b, 1)$	d(c, 6) e(a, 6) <b>f(b, 4)</b>	
$f(b, 4)$	d(f, 5) <b>e(f, 2)</b>	
$e(f, 2)$	<b>d(f, 5)</b>	
$d(f, 5)$		

**FIGURE 9.2** Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.



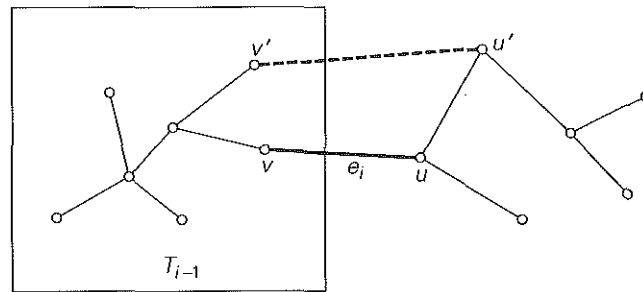


FIGURE 9.3 Correctness proof of Prim's algorithm

minimum spanning tree. (This immediately implies, of course, that the last tree in the sequence,  $T_{n-1}$ , is a minimum spanning tree itself because it contains all  $n$  vertices of the graph.) The basis of the induction is trivial, since  $T_0$  consists of a single vertex and hence must be a part of any minimum spanning tree. For the inductive step, let us assume that  $T_{i-1}$  is part of some minimum spanning tree  $T$ . We need to prove that  $T_i$ , generated from  $T_{i-1}$  by Prim's algorithm, is also a part of a minimum spanning tree. We prove this by contradiction by assuming that no minimum spanning tree of the graph can contain  $T_i$ . Let  $e_i = (v, u)$  be the minimum weight edge from a vertex in  $T_{i-1}$  to a vertex not in  $T_{i-1}$  used by Prim's algorithm to expand  $T_{i-1}$  to  $T_i$ . By our assumption,  $e_i$  cannot belong to the minimum spanning tree  $T$ . Therefore, if we add  $e_i$  to  $T$ , a cycle must be formed (Figure 9.3).

In addition to edge  $e_i = (v, u)$ , this cycle must contain another edge  $(v', u')$  connecting a vertex  $v' \in T_{i-1}$  to a vertex  $u'$  that is not in  $T_{i-1}$ . (It is possible that  $v'$  coincides with  $v$  or  $u'$  coincides with  $u$  but not both.) If we now delete the edge  $(v', u')$  from this cycle, we obtain another spanning tree of the entire graph whose weight is less than or equal to the weight of  $T$  since the weight of  $e_i$  is less than or equal to the weight of  $(v', u')$ . Hence, this spanning tree is a minimum spanning tree, which contradicts the assumption that no minimum spanning tree contains  $T_i$ . This completes the correctness proof of Prim's algorithm.

How efficient is Prim's algorithm? The answer depends on the data structures chosen for the graph itself and for the priority queue of the set  $V - V_T$  whose vertex priorities are the distances to the nearest tree vertices. (You may want to take another look at the example in Figure 9.2 to see that the set  $V - V_T$  indeed operates as a priority queue.) In particular, if a graph is represented by its weight matrix and the priority queue is implemented as an unordered array, the algorithm's running time will be in  $\Theta(|V|^2)$ . Indeed, on each of the  $|V| - 1$  iterations, the array implementing the priority queue is traversed to find and delete the minimum and then to update, if necessary, the priorities of the remaining vertices.

We can also implement the priority queue as a *min-heap*. A min-heap is a mirror image of the heap structure discussed in Section 6.4. (In fact, it can be implemented by constructing a heap after negating all the key values given.) Namely, a min-heap is a complete binary tree in which every element is less than or equal

to its children. All the principal properties of heaps remain valid for min-heaps, with some obvious modifications. For example, the root of a min-heap contains the smallest rather than the largest element. Deletion of the smallest element from and insertion of a new element into a min-heap of size  $n$  are  $O(\log n)$  operations, and so is the operation of changing an element's priority (see Problem 10).

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is in  $O(|E| \log |V|)$ . This is because the algorithm performs  $|V| - 1$  deletions of the smallest element and makes  $|E|$  verifications and, possibly, changes of an element's priority in a min-heap of size not greater than  $|V|$ . Each of these operations, as noted earlier, is a  $O(\log |V|)$  operation. Hence, the running time of this implementation of Prim's algorithm is in

$$(|V| - 1 + |E|)O(\log |V|) = O(|E| \log |V|)$$

because, in a connected graph,  $|V| - 1 \leq |E|$ .

In the next section, you will find another greedy algorithm for the minimum spanning tree problem, which is "greedy" in a manner different from that of Prim's algorithm.

---

## Exercises 9.1

---

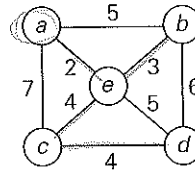
1. Give an instance of the change-making problem for which the greedy algorithm does not yield an optimal solution.
2. Write a pseudocode of the greedy algorithm for the change-making problem with an amount  $n$  and coin denominations  $d_1 > d_2 > \dots > d_m$  as its input. What is the time efficiency class of your algorithm?
3. Consider the problem of scheduling  $n$  jobs of known durations  $t_1, t_2, \dots, t_n$  for execution by a single processor. The jobs can be executed in any order, one job at a time. You want to find a schedule that minimizes the total time spent by all the jobs in the system. (The time spent by one job in the system is the sum of the time spent by this job in waiting plus the time spent on its execution.) Design a greedy algorithm for this problem. Does the greedy algorithm always yield an optimal solution?
4. Design a greedy algorithm for the assignment problem (see Section 3.4). Does your greedy algorithm always yield an optimal solution?
5. *Bridge crossing revisited* Consider the generalization of the bridge crossing puzzle (Problem 2 in Exercises 1.2) in which we have  $n > 1$  people whose bridge crossing times are  $t_1, t_2, \dots, t_n$ . All the other conditions of the problem remain the same: only two people at the time can cross the bridge (and they move with the speed of the slower of the two) and they must carry with them the only flashlight the group has.

Design a greedy algorithm for this problem and find how long it will take to cross the bridge by using this algorithm. Does your algorithm yield a

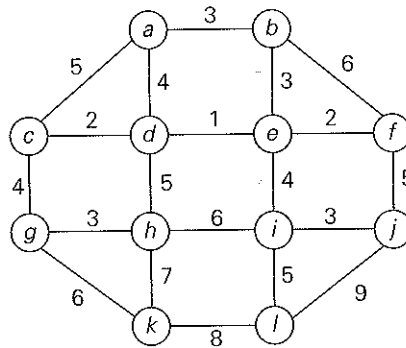
minimum crossing time for every instance of the problem? If it does—prove it; if it does not—find an instance with the smallest number of people for which this happens.



6. *Bachet-Fibonacci weighing problem* Find an optimal set of  $n$  weights  $\{w_1, w_2, \dots, w_n\}$  so that it would be possible to weigh on a balance scale any integer load in the largest possible range from 1 to  $W$ , provided
- weights can be put only on the free cup of the scale.
  - weights can be put on both cups of the scale.
7. a. Apply Prim's algorithm to the following graph. Include in the priority queue all the vertices not already in the tree.



- b. Apply Prim's algorithm to the following graph. Include in the priority queue only the fringe vertices (the vertices not in the current tree which are adjacent to at least one tree vertex).



- The notion of a minimum spanning tree is applicable to a connected weighted graph. Do we have to check a graph's connectivity before applying Prim's algorithm or can the algorithm do it by itself?
- a. How can we use Prim's algorithm to find a spanning tree of a connected graph with no weights on its edges?  
b. Is it a good algorithm for this problem?
- Prove that any weighted connected graph with distinct weights has exactly one minimum spanning tree.
- Outline an efficient algorithm for changing an element's value in a min-heap. What is the time efficiency of your algorithm?

## 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* [Kru56], after Joseph Kruskal, who discovered the algorithm when he was a second-year graduate student. Kruskal's algorithm looks at a minimum spanning tree for 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, which 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.

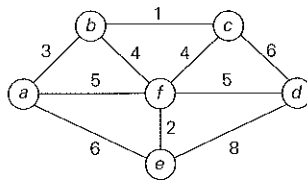
### ALGORITHM *Kruskal*( $G$ )

```
//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph  $G = \langle V, E \rangle$ 
//Output:  $E_T$ , the set of edges composing a minimum spanning tree of  $G$ 
sort  $E$  in nondecreasing order of the edge weights  $w(e_{i_1}) \leq \dots \leq w(e_{i_{|E|}})$ 
 $E_T \leftarrow \emptyset$ ;  $ecounter \leftarrow 0$  //initialize the set of tree edges and its size
 $k \leftarrow 0$  //initialize the number of processed edges
while  $ecounter < |V| - 1$  do
     $k \leftarrow k + 1$ 
    if  $E_T \cup \{e_{i_k}\}$  is acyclic
         $E_T \leftarrow E_T \cup \{e_{i_k}\}$ ;  $ecounter \leftarrow ecounter + 1$ 
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 9.4 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 graphs.

Applying Prim's and Kruskal's algorithms to the same small graph by hand may create an 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



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

FIGURE 9.4 Application of Kruskal's algorithm. Selected edges are shown in bold.

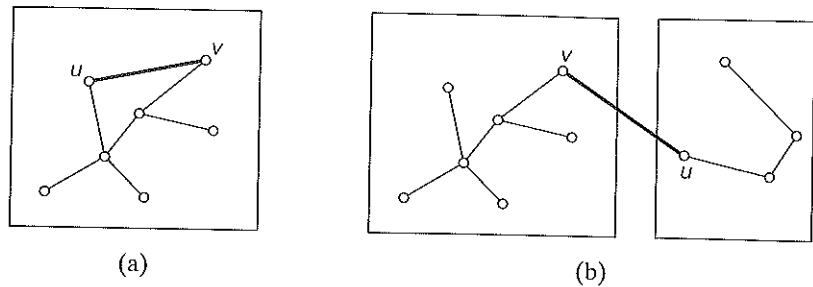


FIGURE 9.5 New edge connecting two vertices may (a) or may not (b) create a cycle

a 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 9.5). 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 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  $S_1, S_2, \dots, S_k$ . 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:

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

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

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

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

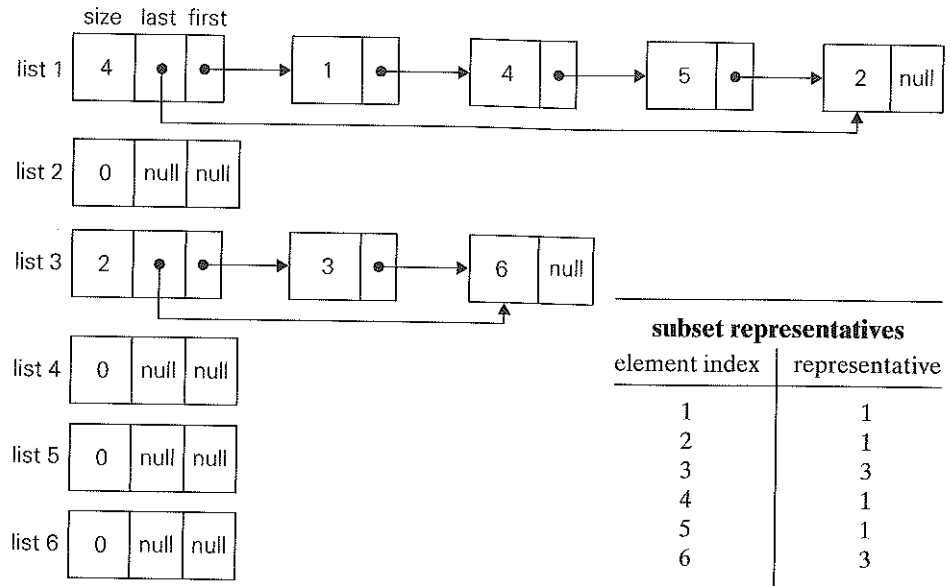
$$\{1, 4, 5, 2\}, \{3, 6\}.$$

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 9.6 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



**FIGURE 9.6** Linked-list representation of subsets  $\{1, 4, 5, 2\}$  and  $\{3, 6\}$  obtained by quick find after performing  $\text{union}(1, 4)$ ,  $\text{union}(5, 2)$ ,  $\text{union}(4, 5)$ , and  $\text{union}(3, 6)$ . 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

$$\text{union}(2, 1), \text{union}(3, 2), \dots, \text{union}(i + 1, i), \dots, \text{union}(n, n - 1)$$

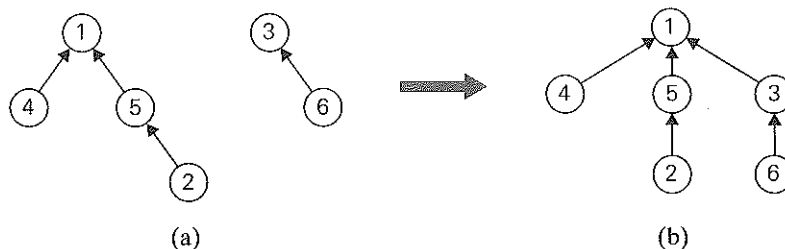
runs in  $\Theta(n^2)$  time, which is slow compared with several known alternatives.

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 **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  $O(n \log n)$ .<sup>2</sup>

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

2. This is a specific example of the usefulness of the *amortized efficiency* we mentioned back in Chapter 2. The time efficiency of any sequence of  $n$  union-by-size operations is more efficient than the worst-case efficiency of its single application times  $n$ .





**FIGURE 9.7** (a) Forest representation of subsets  $\{1, 4, 5, 2\}$  and  $\{3, 6\}$  used by quick union. (b) Result of  $union(5, 6)$ .

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 9.7). (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 9.7 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 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

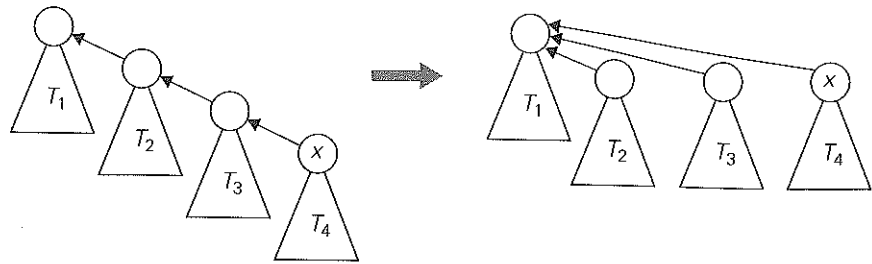


FIGURE 9.8 Path compression

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  $O(n + m \log n)$ .

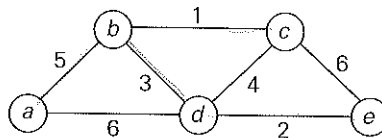
In fact, an even better efficiency can be obtained by combining either variety of quick union with **path compression**. This modification makes every node encountered during the execution of a find operation point to the tree's root (Figure 9.8).

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.

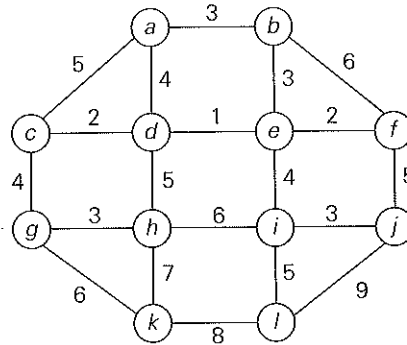
## Exercises 9.2

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. Will either Kruskal's or Prim'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 the 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.



11. *Steiner tree* Four villages are located at the vertices of a unit square in the Euclidean plane. You are asked to connect them by the shortest network of roads so that there is a path between every pair of the villages along those roads. Find such a network.

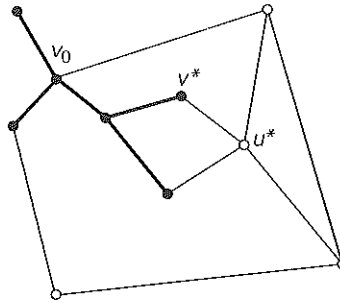
### 9.3 Dijkstra's Algorithm

In this section, we consider the *single-source shortest-paths problem*: for a given vertex called the *source* in a weighted connected graph, find shortest paths to all its other vertices. It is important to stress that we are not interested here in a single shortest path that starts at the source and visits all the other vertices. This would have been a much more difficult problem (actually, a version of the traveling salesman problem mentioned in Section 3.4 and discussed again later in the book). The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have edges in common.

A variety of practical applications of the shortest-paths problem have made the problem a very popular object of study. There are several well-known algorithms for solving it, including Floyd's algorithm for the more general all-pairs shortest-paths problem discussed in Chapter 8. Here, we consider the best-known algorithm for the single-source shortest-paths problem, called *Dijkstra's algorithm*.<sup>3</sup> This algorithm is applicable to graphs with nonnegative weights only. Since in most applications this condition is satisfied, the limitation has not impaired the popularity of Dijkstra's algorithm.

Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on. In general, before its  $i$ th iteration commences, the algorithm has already identified the shortest paths to  $i - 1$  other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree  $T_i$  of the given graph (Figure 9.9). Since all the edge weights are nonnegative, the next vertex nearest to the source can be found among the vertices adjacent to the vertices of  $T_i$ . The set of vertices adjacent to the vertices in  $T_i$  can be referred to as "fringe vertices"; they are the candidates from which Dijkstra's algorithm selects the next vertex nearest to the source. (Actually, all the other vertices can be treated as fringe vertices connected to tree vertices by edges of infinitely large weights.) To identify the  $i$ th nearest vertex, the algorithm computes, for every fringe vertex

3. Edsger W. Dijkstra (1930–2002), a noted Dutch pioneer of the science and industry of computing, discovered this algorithm in the mid-1950s. Dijkstra said of his algorithm: "This was the first graph problem I ever posed myself and solved. The amazing thing was that I didn't publish it. It was not amazing at the time. At the time, algorithms were hardly considered a scientific topic."



**FIGURE 9.9** Idea of Dijkstra's algorithm. The subtree of the shortest paths already found is shown in bold. The next nearest to the source  $v_0$  vertex,  $u^*$ , is selected by comparing the lengths of the subtree's paths increased by the distances to vertices adjacent to the subtree's vertices.

$u$ , the sum of the distance to the nearest tree vertex  $v$  (given by the weight of the edge  $(v, u)$ ) and the length  $d_v$  of the shortest path from the source to  $v$  (previously determined by the algorithm) and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

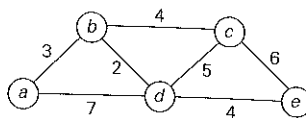
To facilitate the algorithm's operations, we label each vertex with two labels. The numeric label  $d$  indicates the length of the shortest path from the source to this vertex found by the algorithm so far; when a vertex is added to the tree,  $d$  indicates the length of the shortest path from the source to that vertex. The other label indicates the name of the next-to-last vertex on such a path, i.e., the parent of the vertex in the tree being constructed. (It can be left unspecified for the source  $s$  and vertices that are adjacent to none of the current tree vertices.) With such labeling, finding the next nearest vertex  $u^*$  becomes a simple task of finding a fringe vertex with the smallest  $d$  value. Ties can be broken arbitrarily.

After we have identified a vertex  $u^*$  to be added to the tree, we need to perform two operations:

- Move  $u^*$  from the fringe to the set of tree vertices.
- For each remaining fringe vertex  $u$  that is connected to  $u^*$  by an edge of weight  $w(u^*, u)$  such that  $d_{u^*} + w(u^*, u) < d_u$ , update the labels of  $u$  by  $u^*$  and  $d_{u^*} + w(u^*, u)$ , respectively.

Figure 9.10 demonstrates the application of Dijkstra's algorithm to a specific graph.

The labeling and mechanics of Dijkstra's algorithm are quite similar to those used by Prim's algorithm (see Section 9.1). Both of them construct an expanding subtree of vertices by selecting the next vertex from the priority queue of the remaining vertices. It is important not to mix them up, however. They solve



Tree vertices	Remaining vertices	Illustration
a(-, 0)	<b>b(a, 3)</b> c(-, ∞) d(a, 7) e(-, ∞)	
b(a, 3)	c(b, 3 + 4) <b>d(b, 3 + 2)</b> e(-, ∞)	
d(b, 5)	<b>c(b, 7)</b> e(d, 5 + 4)	
c(b, 7)	<b>e(d, 9)</b>	
e(d, 9)		

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are

- from a to b: a - b of length 3
- from a to d: a - b - d of length 5
- from a to c: a - b - c of length 7
- from a to e: a - b - d - e of length 9

FIGURE 9.10 Application of Dijkstra's algorithm. The next closest vertex is shown in bold.

different problems and therefore operate with priorities computed in a different manner: Dijkstra's algorithm compares path lengths and therefore must add edge weights, while Prim's algorithm compares the edge weights as given.

Now we can give a pseudocode of Dijkstra's algorithm. It is spelled out—in more detail than Prim's algorithm was in Section 9.1—in terms of explicit operations on two sets of labeled vertices: the set  $V_T$  of vertices for which a shortest path has already been found and the priority queue  $Q$  of the fringe vertices. (Note that in the following pseudocode,  $V_T$  contains a given source vertex and the fringe contains the vertices adjacent to it *after* iteration 0 is completed.)

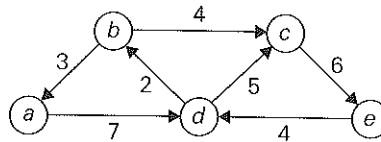
**ALGORITHM** *Dijkstra*( $G, s$ )

```
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph  $G = \langle V, E \rangle$  with nonnegative weights
//      and its vertex  $s$ 
//Output: The length  $d_v$  of a shortest path from  $s$  to  $v$ 
//      and its penultimate vertex  $p_v$  for every vertex  $v$  in  $V$ 
Initialize( $Q$ ) //initialize vertex priority queue to empty
for every vertex  $v$  in  $V$  do
     $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ 
    Insert( $Q, v, d_v$ ) //initialize vertex priority in the priority queue
 $d_s \leftarrow 0$ ; Decrease( $Q, s, d_s$ ) //update priority of  $s$  with  $d_s$ 
 $V_T \leftarrow \emptyset$ 
for  $i \leftarrow 0$  to  $|V| - 1$  do
     $u^* \leftarrow \text{DeleteMin}(Q)$  //delete the minimum priority element
     $V_T \leftarrow V_T \cup \{u^*\}$ 
    for every vertex  $u$  in  $V - V_T$  that is adjacent to  $u^*$  do
        if  $d_{u^*} + w(u^*, u) < d_u$ 
             $d_u \leftarrow d_{u^*} + w(u^*, u)$ ;  $p_u \leftarrow u^*$ 
            Decrease( $Q, u, d_u$ )
```

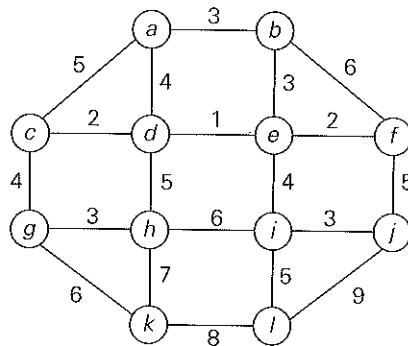
The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. For the reasons explained in the analysis of Prim's algorithm in Section 9.1, it is in  $\Theta(|V|^2)$  for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in  $O(|E| \log |V|)$ . A still better upper bound can be achieved for both Prim's and Dijkstra's algorithms if the priority queue is implemented using a sophisticated data structure called the *Fibonacci heap* (e.g., [Wei98]). However, its complexity and a considerable overhead make such an improvement primarily of theoretical value.

## Exercises 9.3

1. Explain what adjustments, if any, need to be made in Dijkstra's algorithm and/or in an underlying graph to solve the following problems.
  - a. Solve the single-source shortest-paths problem for directed weighted graphs.
  - b. Find a shortest path between two given vertices of a weighted graph or digraph. (This variation is called the *single-pair shortest-path problem*.)
  - c. Find the shortest paths to a given vertex from each other vertex of a weighted graph or digraph. (This variation is called the *single-destination shortest-paths problem*.)
  - d. Solve the single-source shortest-paths problem in a graph with nonnegative numbers assigned to its vertices (and the length of a path defined as the sum of the vertex numbers on the path).
2. Solve the following instances of the single-source shortest-paths problem with vertex  $a$  as the source:
  - a.




b.



3. Give a counterexample that shows that Dijkstra's algorithm may not work for a weighted connected graph with negative weights.
4. Let  $T$  be a tree constructed by Dijkstra's algorithm in the process of solving the single-source shortest-paths problem for a weighted connected graph  $G$ .
  - a. True or false:  $T$  is a spanning tree of  $G$ ?
  - b. True or false:  $T$  is a minimum spanning tree of  $G$ ?



5. Write a pseudocode of a simpler version of Dijkstra's algorithm that finds only the distances (i.e., the lengths of shortest paths but not shortest paths themselves) from a given vertex to all other vertices of a graph represented by its weight matrix.
  6. Prove the correctness of Dijkstra's algorithm for graphs with positive weights.
  7. Design a linear-time algorithm for solving the single-source shortest-paths problem for dags (directed acyclic graphs) represented by their adjacency lists.
  8. Design an efficient algorithm for finding the length of a longest path in a dag. (This problem is important because it determines a lower bound on the total time needed for completing a project composed of precedence-constrained tasks.)
-  9. *Shortest-path modeling* Assume that you have a model of a weighted connected graph made of balls (representing the vertices) connected by strings of appropriate lengths (representing the edges).
- a. Describe how you can solve the single-pair shortest-path problem with this model.
  - b. Describe how you can solve the single-source shortest-paths problem with this model.
10. Revisit Problem 6 in Exercises 1.3 about determining the best route for a subway passenger to take from one designated station to another in a well-developed subway system like those in Washington, DC, and London, UK. Write a program for this task.

## 9.4 Huffman Trees

Suppose we have to encode a text that comprises characters from some  $n$ -character alphabet by assigning to each of the text's characters some sequence of bits called the *codeword*. For example, we can use a *fixed-length encoding* that assigns to each character a bit string of the same length  $m$  ( $m \geq \log_2 n$ ). This is exactly what the standard ASCII code does. One way of getting a coding scheme that yields a shorter bit string on the average is based on the old idea of assigning shorter codewords to more frequent characters and longer codewords to less frequent characters. (This idea was used, in particular, in the telegraph code invented in the mid-19th century by Samuel Morse. In that code, frequent letters such as  $e$  ( $\cdot$ ) and  $a$  ( $\cdot -$ ) are assigned short sequences of dots and dashes while infrequent letters such as  $q$  ( $- - -$ ) and  $z$  ( $- - \cdot$ ) have longer ones.)

*Variable-length encoding*, which assigns codewords of different lengths to different characters, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the  $i$ th) character? To avoid this complication, we can limit ourselves to *prefix-free* (or simply *prefix*) codes. In a prefix

code, no codeword is a prefix of a codeword of another character. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some character, replace these bits by this character, and repeat this operation until the bit string's end is reached.

If we want to create a binary prefix code for some alphabet, it is natural to associate the alphabet's characters with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1 (or vice versa). The codeword of a character can then be obtained by recording the labels on the simple path from the root to the character's leaf. Since there is no simple path to a leaf that continues to another leaf, no codeword can be a prefix of another codeword; hence, any such tree yields a prefix code.

Among the many trees that can be constructed in this manner for a given alphabet with known frequencies of the character occurrences, how can we construct a tree that would assign shorter bit strings to high-frequency characters and longer ones to low-frequency characters? It can be done by the following greedy algorithm, invented by David Huffman while he was a graduate student at MIT [Huf52].

### Huffman's Algorithm

**Step 1** Initialize  $n$  one-node trees and label them with the characters of the alphabet. Record the frequency of each character in its tree's root to indicate the tree's *weight*. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

**Step 2** Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in the exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a *Huffman tree*. It defines—in the manner described—a *Huffman code*.

**EXAMPLE** Consider the five-character alphabet {A, B, C, D, \_} with the following occurrence probabilities:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 9.11. The resulting codewords are as follows:

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

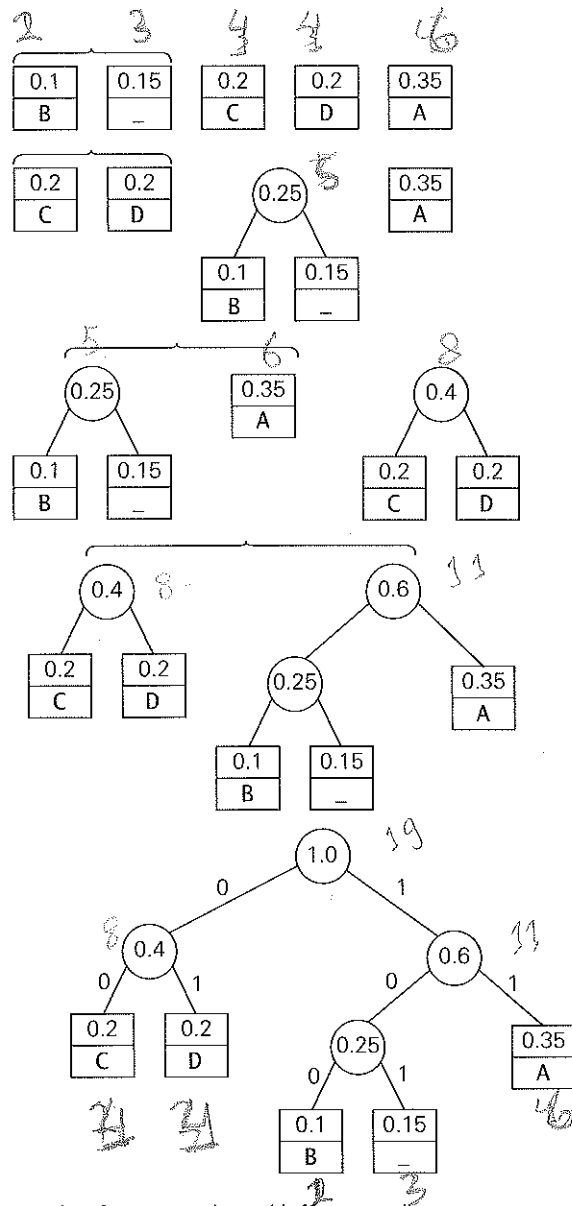


FIGURE 9.11 Example of constructing a Huffman coding tree

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD\_AD.

With the occurrence probabilities given and the codeword lengths obtained, the expected number of bits per character in this code is

$$2 \cdot 0.35 + 3 \cdot 0.1 + 2 \cdot 0.2 + 2 \cdot 0.2 + 3 \cdot 0.15 = 2.25.$$

Had we used a fixed-length encoding for the same alphabet, we would have to use at least three bits per each character. Thus, for this toy example, Huffman's code achieves the *compression ratio*—a standard measure of a compression algorithm's effectiveness—of  $(3 - 2.25)/3 \cdot 100\% = 25\%$ . In other words, we should expect that Huffman's encoding of a text will use 25% less memory than its fixed-length encoding. (Extensive experiments with Huffman codes have shown that the compression ratio for this scheme typically falls between 20% and 80%, depending on the characteristics of the text being compressed.) ■

Huffman's encoding is one of the most important file compression methods. In addition to its simplicity and versatility, it yields an optimal, i.e., minimal-length, encoding (provided the probabilities of character occurrences are independent and known in advance). The simplest version of Huffman compression calls, in fact, for a preliminary scanning of a given text to count the frequencies of character occurrences in it. Then these frequencies are used to construct a Huffman coding tree and encode the text as described above. This scheme makes it necessary, however, to include the information about the coding tree into the encoded text to make its decoding possible. This drawback can be overcome by using *dynamic Huffman encoding*, in which the coding tree is updated each time a new character is read from the source text (see, e.g., [Say00]).

It is important to note that applications of Huffman's algorithm are not limited to data compression. Suppose we have  $n$  positive numbers  $w_1, w_2, \dots, w_n$  that have to be assigned to  $n$  leaves of a binary tree, one per node. If we define the *weighted path length* as the sum  $\sum_{i=1}^n l_i w_i$ , where  $l_i$  is the length of the simple path from the root to the  $i$ th leaf, how can we construct a binary tree with minimum weighted path length? It is this more general problem that Huffman's algorithm actually solves. (For the coding application,  $l_i$  and  $w_i$  are the length of the codeword and the frequency of the  $i$ th character, respectively.) This problem arises in many situations involving decision making. Consider, for example, the game of guessing a chosen object from  $n$  possibilities (say, an integer between 1 and  $n$ ) by asking questions answerable by yes or no. Different strategies for playing this game can be modeled by *decision trees*<sup>4</sup> such as those depicted in Figure 9.12 for  $n = 4$ .

The length of the simple path from the root to a leaf in such a tree is equal to the number of questions needed to get to the chosen number represented by the leaf. If number  $i$  is chosen with probability  $p_i$ , the sum  $\sum_{i=1}^n l_i p_i$ , where  $l_i$  is the length of the simple path from the root to the  $i$ th leaf, indicates the average number of questions needed to "guess" the chosen number with a game strategy represented by its decision tree. If each of the numbers is chosen with the same probability of  $1/n$ , the best strategy is to successively eliminate half (or almost half) the candidates as binary search does. This may not be the case for arbitrary  $p_i$ 's,

4. Decision trees are discussed in more detail in Section 11.2.

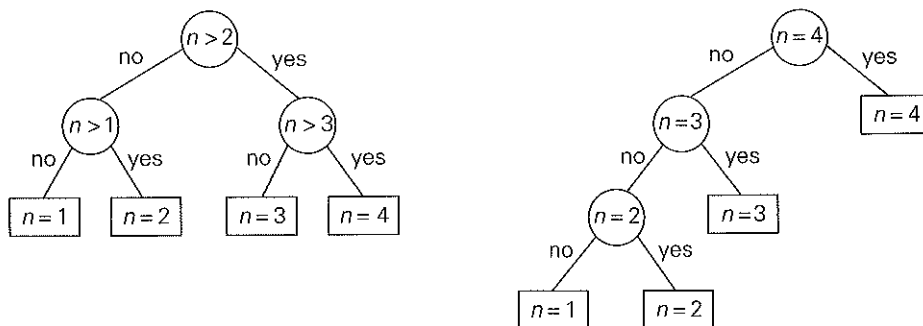


FIGURE 9.12 Two decision trees for guessing an integer between 1 and 4

however. (For example, if  $n = 4$  and  $p_1 = 0.1$ ,  $p_2 = 0.2$ ,  $p_3 = 0.3$ , and  $p_4 = 0.4$ , the minimum weighted path tree is the rightmost one in Figure 9.12.) Thus, we need Huffman's algorithm to solve this problem in its general case.

In conclusion, it is worthwhile to remember that this is the second time we are encountering the problem of constructing an optimal binary tree. In Section 8.3, we discussed the problem of constructing an optimal binary search tree with positive numbers (the search probabilities) assigned to every node of the tree. In this section, given numbers are assigned just to leaves. The latter problem turns out to be easier: it can be solved by the greedy algorithm whereas the former is solved by the more complicated dynamic programming algorithm.


## Exercises 9.4

1. a. Construct a Huffman code for the following data:

character	A	B	C	D	_
probability	0.4	0.1	0.2	0.15	0.15

- b. Encode the text ABACABAD using the code of question (a).  
 c. Decode the text whose encoding is 100010111001010 in the code of question (a).
2. For data transmission purposes, it is often desirable to have a code with a minimum variance of the codeword lengths (among codes of the same average length). Compute the average and variance of the codeword length in two Huffman codes that result from a different tie breaking during a Huffman code construction for the following data:

character	A	B	C	D	E
probability	0.1	0.1	0.2	0.2	0.4

- del
3. Indicate whether each of the following properties are true for every Huffman code.
    - a. The codewords of the two least frequent characters have the same length.
    - b. The codeword's length of a more frequent character is always smaller than or equal to the codeword's length of a less frequent one.
  4. What is the maximal length of a codeword possible in a Huffman encoding of an alphabet of  $n$  characters?
  5.
    - a. Write a pseudocode for the Huffman-tree construction algorithm.
    - b. What is the time efficiency class of the algorithm for constructing a Huffman tree as a function of the alphabet's size?
  6. Show that a Huffman tree can be constructed in linear time if the alphabet's characters are given in a sorted order of their frequencies.
  7. Given a Huffman coding tree, which algorithm would you use to get the codewords for all the characters? What is its time-efficiency class as a function of the alphabet's size?  $O(n)$   $n$  alphabet  $2(n-1)$  nodes
  8. Explain how one can generate a Huffman code without an explicit generation of a Huffman coding tree.
  9.
    - a. Write a program that constructs a Huffman code for a given English text and encode it.
    - b. Write a program for decoding an English text that has been encoded with a Huffman code.
    - c. Experiment with your encoding program to find a range of typical compression ratios for Huffman's encoding of English texts of, say, 1000 words.
    - d. Experiment with your encoding program to find out how sensitive the compression ratios are to using standard estimates of frequencies instead of actual frequencies of character occurrences in English texts.
  10.  *Card guessing* Design a strategy that minimizes the expected number of questions asked in the following game [Gar94], #52. You have a deck of cards that consists of one ace of spades, two deuces of spades, three threes, and on up to nine nines, making 45 cards in all. Someone draws a card from the shuffled deck, which you have to identify by asking questions answerable with yes or no.

## SUMMARY

- The *greedy technique* suggests constructing a solution to an optimization problem through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.

On each step, the choice made must be *feasible*, *locally optimal*, and *irrevocable*.

- *Prim's algorithm* is a greedy algorithm for constructing a minimum spanning tree of a weighted connected graph. It works by attaching to a previously constructed subtree a vertex closest to the vertices already in the tree.
- *Kruskal's algorithm* is another greedy algorithm for the minimum spanning tree problem. It constructs a minimum spanning tree by selecting edges in increasing order of their weights provided that the inclusion does not create a cycle. Checking the latter condition efficiently requires an application of one of the so-called *union-find algorithms*.
- *Dijkstra's algorithm* solves the single-source shortest-paths problem of finding shortest paths from a given vertex (the source) to all the other vertices of a weighted graph or digraph. It works as Prim's algorithm but compares path lengths rather than edge lengths. Dijkstra's algorithm always yields a correct solution for a graph with nonnegative weights.
- A *Huffman tree* is a binary tree that minimizes the weighted path length from the root to the leaves of predefined weights. The most important application of Huffman trees are Huffman codes.
- A *Huffman code* is an optimal prefix-free variable-length encoding scheme that assigns bit strings to characters based on their frequencies in a given text. This is accomplished by a greedy construction of a binary tree whose leaves represent the alphabet characters and whose edges are labeled with 0's and 1's.