Program 18.2  Depth-first search (adjacency-lists)

This implementation of dfsR is DFS for graphs represented with adjacency lists. The algorithm is the same as for the adjacency-matrix representation (Program 18.1): to visit a vertex, we mark it and then scan through its incident edges, making a recursive call whenever we encounter an edge to an unmarked vertex.

```c
void dfsR(Graph G, Edge e)
{
    link t; int w = e.w;
    pre[w] = cnt++;
    for (t = G->adj[w]; t != NULL; t = t->next)
        if (pre[t->v] == -1)
            dfsR(G, EDGE(w, t->v));
}
```

The DFS in Program 18.2 ignores self-loops and duplicate edges if they are present, so that we do not necessarily have to take the trouble to remove them from the adjacency lists.

The order in which DFS discovers the edges and vertices in the graph depends entirely on the order in which the edges appear in the adjacency lists in the graph representation. We might also consider examining the entries in each row in the adjacency matrix in some different order, which would lead to another different search dynamic (see Exercise 18.5).

Despite all of these possibilities, the critical fact remains that DFS visits all the edges and all the vertices connected to the start vertex, regardless of in what order it examines the edges incident on each vertex. This fact is a direct consequence of Property 18.1, since the proof of that property does not depend on the order in which the doors are opened at any given intersection. All the DFS-based algorithms that we examine have this same essential property. Although the dynamics of their operation might vary substantially depending on the graph representation and details of the implementation of the search, the recursive structure gives us a way to make relevant inferences about

Figure 18.7  DFS trace (adjacency lists)

This trace shows the order in which DFS checks the edges and vertices for the adjacency-lists representation of the same graph as in Figure 18.5.
Figure 18.13
Depth-first search
This figure illustrates the progress of DFS in a random Euclidean near-neighbor graph (left). The figures show the DFS tree vertices and edges in the graph as the search progresses through 1/4, 1/2, 3/4, and all of the vertices (top to bottom). The DFS tree (tree edges only) is shown at the right. As is evident from this example, the search tree for DFS tends to be quite tall and thin for this type of graph (as it is for many other types of graphs commonly encountered in practice). We normally find a vertex nearby that we have not seen before.
Table 20.2 Empirical study of MST algorithms

This table shows relative timings for various algorithms for finding the MST, for random weighted graphs of various density. For low densities, Kruskal’s algorithm is best because it amounts to a fast sort. For high densities, the classical implementation of Prim’s algorithm is best because it does not incur list-processing overhead. For intermediate densities, the PFS implementation of Prim’s algorithm runs within a small factor of the time that it takes to examine each graph edge.

<table>
<thead>
<tr>
<th>Density</th>
<th>E</th>
<th>V</th>
<th>C</th>
<th>H</th>
<th>J</th>
<th>P</th>
<th>K</th>
<th>K*</th>
<th>e/E</th>
<th>B</th>
<th>e/E</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>20000</td>
<td>10000</td>
<td>2</td>
<td>22</td>
<td>27</td>
<td>9</td>
<td>11</td>
<td>1.00</td>
<td>14</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>25000</td>
<td>8</td>
<td>69</td>
<td>84</td>
<td>24</td>
<td>31</td>
<td>1.00</td>
<td>38</td>
<td>3.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>50000</td>
<td>15</td>
<td>169</td>
<td>203</td>
<td>49</td>
<td>66</td>
<td>1.00</td>
<td>89</td>
<td>3.8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200000</td>
<td>100000</td>
<td>30</td>
<td>389</td>
<td>478</td>
<td>108</td>
<td>142</td>
<td>1.00</td>
<td>189</td>
<td>3.6</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>20000</td>
<td>1000</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>20</td>
<td>6</td>
<td>5</td>
<td>.20</td>
<td>9</td>
<td>4.2</td>
</tr>
<tr>
<td></td>
<td>50000</td>
<td>2500</td>
<td>12</td>
<td>12</td>
<td>13</td>
<td>130</td>
<td>16</td>
<td>15</td>
<td>.28</td>
<td>25</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>100000</td>
<td>5000</td>
<td>14</td>
<td>27</td>
<td>28</td>
<td>34</td>
<td>31</td>
<td>.30</td>
<td>55</td>
<td>4.6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>200000</td>
<td>10000</td>
<td>29</td>
<td>61</td>
<td>61</td>
<td>73</td>
<td>68</td>
<td>.35</td>
<td>123</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>10000</td>
<td>1000</td>
<td>14</td>
<td>17</td>
<td>17</td>
<td>24</td>
<td>30</td>
<td>19</td>
<td>.06</td>
<td>51</td>
<td>4.6</td>
</tr>
<tr>
<td></td>
<td>250000</td>
<td>2500</td>
<td>36</td>
<td>44</td>
<td>44</td>
<td>130</td>
<td>81</td>
<td>53</td>
<td>.05</td>
<td>143</td>
<td>5.2</td>
</tr>
<tr>
<td></td>
<td>500000</td>
<td>5000</td>
<td>73</td>
<td>93</td>
<td>93</td>
<td>181</td>
<td>113</td>
<td>.06</td>
<td>312</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000000</td>
<td>10000</td>
<td>151</td>
<td>204</td>
<td>198</td>
<td>377</td>
<td>218</td>
<td>.06</td>
<td>658</td>
<td>5.6</td>
<td></td>
</tr>
<tr>
<td>V/2.5</td>
<td>400000</td>
<td>1000</td>
<td>61</td>
<td>60</td>
<td>59</td>
<td>20</td>
<td>137</td>
<td>78</td>
<td>.02</td>
<td>188</td>
<td>4.5</td>
</tr>
<tr>
<td></td>
<td>2500000</td>
<td>2500</td>
<td>597</td>
<td>409</td>
<td>400</td>
<td>128</td>
<td>1056</td>
<td>687</td>
<td>.01</td>
<td>1472</td>
<td>5.5</td>
</tr>
</tbody>
</table>

Key:
- **C**: extract edges only
- **H**: Prim’s algorithm (adjacency lists/indexed heap)
- **J**: Johnson’s version of Prim’s algorithm (d-heap priority queue)
- **P**: Prim’s algorithm (adjacency-matrix representation)
- **K**: Kruskal’s algorithm
- **K***: Partial-sort version of Kruskal’s algorithm
- **B**: Boruvka’s algorithm
- **e**: edges examined (union operations)
SHORTEST PATHS

21.7 Negative Weights

We now turn to the challenge of coping with negative weights in shortest-paths problems. Perhaps negative edge weights seem unlikely given our focus through most of this chapter on intuitive examples, where weights represent distances or costs; however, we also saw in Section 21.6 that negative edge weights arise in a natural way when we reduce other problems to shortest-paths problems. Negative weights

Problem: Given a digraph, a vertex-indexed array of positive weights, and a start vertex v, find the paths from v to each other vertex such that the sum of the weights of the vertices on the path is minimized.

21.100 Program 21.8 does not check whether the job-scheduling problem that it takes as input is feasible (has a cycle). Characterize the schedules that it prints out for infeasible problems.

21.101 Design an ADT interface that gives clients the ability to pose and solve job-scheduling problems. Provide ADT function implementations for your interface, basing your solution to the job-scheduling problem on a reduction to the shortest-paths problem in acyclic networks, as in Program 21.8.

21.102 Add a function to your ADT interface (and provide an implementation) that prints out a longest path in the schedule. (Such a path is known as a critical path.)

21.103 Provide an ADT function implementation for your interface from Exercise 21.101 that outputs a PostScript program that draws the schedule in the style of Figure 21.24 (see Section 4.3).


21.105 Provide ADT function implementations for your interface from Exercise 21.101, basing your solution to the job-scheduling problem on a reduction to the difference-constraints problem.

21.106 A PERT (performance-evaluation-review-technique) chart is a network that represents a job-scheduling problem, with edges representing jobs, as described in Figure 21.25. Develop an implementation of your job-scheduling interface of Exercise 21.101 that is based on PERT charts.

21.107 How many vertices are there in a PERT chart for a job-scheduling problem with V jobs and E constraints?

21.108 Write programs to convert between the edge-based job-scheduling representation (PERT chart) discussed in Exercise 21.106 and the vertex-based representation used in the text (see Figure 21.22).

Figure 21.25
A PERT chart

A PERT chart is a network representation for job-scheduling problems where we represent jobs by edges. The network at the top is a representation of the job-scheduling problem depicted in Figure 21.22, where jobs 0 through 9 in Figure 21.22 are represented by edges 0-1, 1-2, 2-3, 4-3, 5-3, 0-3, 5-4, 1-4, 4-2, and 1-5, respectively, here. The critical path in the schedule is the longest path in the network.
be used for practical applications when space is not at premium and a guaranteed worst-case running time is desirable. Both algorithms are of interest as prototypes of the general divide-and-conquer and combine-and-conquer algorithm design paradigms.

Exercises
8.24 Show the merges that bottom-up mergesort (Program 8.5) does for the keys EASYQUESTION.
8.25 Implement a bottom-up mergesort that starts by sorting blocks of $M$ elements with insertion sort. Determine empirically the value of $M$ for which your program runs fastest to sort random files of $N$ elements, for $N = 10^3$, $10^4$, $10^5$, and $10^6$.
8.26 Draw trees that summarize the merges that Program 8.5 performs, for $N = 16, 24, 31, 32, 33$, and 39.
8.27 Write a recursive mergesort that performs the same merges that bottom-up mergesort does.
8.28 Write a bottom-up mergesort that performs the same merges that top-down mergesort does. (This exercise is much more difficult than is Exercise 8.27.)
8.29 Suppose that the file size is a power of 2. Remove the recursion from top-down mergesort to get a nonrecursive mergesort that performs the same sequence of merges.
8.30 Prove that the number of passes taken by top-down mergesort is also the number of bits in the binary representation of $N$ (see Property 8.6).

8.6 Performance Characteristics of Mergesort

Table 8.1 shows the relative effectiveness of the various improvements that we have examined. As is often the case, these studies indicate that we can cut the running time by half or more when we focus on improving the inner loop of the algorithm.

In addition to netting the improvements discussed in Section 8.2, a good Java VM implementation might avoid unnecessary array accesses to reduce the inner loop of mergesort to a comparison (with conditional branch), two index increments ($k$ and either $i$ or $j$), and a test with conditional branch for loop completion. The total number of instructions in such an inner loop is slightly higher than that for quicksort, but the instructions are executed only $N \lg N$ times, where quicksort’s are executed 39 percent more often (or 29 percent with the
Figure 16.9
Growth of a large B tree, page occupancy exposed

This version of Figure 16.8 shows how pages fill during the B tree growth process. Again, most insertions land in a page that is not full and just increase its occupancy by 1. When an insertion does land in a full page, the page splits into two half-empty pages.