Chapter 5

DYNAMIC PROGRAMMING

5.1 THE GENERAL METHOD

Dynamic programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decisions. In earlier chapters we saw many problems that can be viewed this way. Here are some examples:

Example 5.1 [Knapsack] The solution to the knapsack problem (Section 4.2) can be viewed as the result of a sequence of decisions. We have to decide the values of $x_i, 1 \leq i \leq n$. First we make a decision on x_1, then on x_2, then on x_3, and so on. An optimal sequence of decisions maximizes the objective function $\sum p_i x_i$. (It also satisfies the constraints $\sum w_i x_i \leq m$ and $0 \leq x_i \leq 1$.) □

Example 5.2 [Optimal merge patterns] This problem was discussed in Section 4.7. An optimal merge pattern tells us which pair of files should be merged at each step. As a decision sequence, the problem calls for us to decide which pair of files should be merged first, which pair second, which pair third, and so on. An optimal sequence of decisions is a least-cost sequence. □

Example 5.3 [Shortest path] One way to find a shortest path from vertex i to vertex j in a directed graph G is to decide which vertex should be the second vertex, which the third, which the fourth, and so on, until vertex j is reached. An optimal sequence of decisions is one that results in a path of least length. □

For some of the problems that may be viewed in this way, an optimal sequence of decisions can be found by making the decisions one at a time and never making an erroneous decision. This is true for all problems solvable by the greedy method. For many other problems, it is not possible to make stepwise decisions (based only on local information) in such a manner that the sequence of decisions made is optimal.

Example 5.4 [Shortest path] Suppose we wish to find a shortest path from vertex i to vertex j. Let A_i be the vertices adjacent from vertex i. Which of the vertices in A_i should be the second vertex on the path? There is no way to make a decision at this time and guarantee that future decisions leading to an optimal sequence can be made. If on the other hand we wish to find a shortest path from vertex i to all other vertices in G, then at each step, a correct decision can be made (see Section 4.8). □

One way to solve problems for which it is not possible to make a sequence of stepwise decisions leading to an optimal decision sequence is to try all possible decision sequences. We could enumerate all decision sequences and then pick out the best. But the time and space requirements may be prohibitive. Dynamic programming often drastically reduces the amount of enumeration by avoiding the enumeration of some decision sequences that cannot possibly be optimal. In dynamic programming an optimal sequence of decisions is obtained by making explicit appeal to the *principle of optimality.*

Definition 5.1 [Principle of optimality] The principle of optimality states that an optimal sequence of decisions has the property that whatever the initial state and decision are, the remaining decisions must constitute an optimal decision sequence with regard to the state resulting from the first decision. □

Thus, the essential difference between the greedy method and dynamic programming is that in the greedy method only one decision sequence is ever generated. In dynamic programming, many decision sequences may be generated. However, sequences containing suboptimal subsequences cannot be optimal (if the principle of optimality holds) and so will not (as far as possible) be generated.

Example 5.5 [Shortest path] Consider the shortest-path problem of Example 5.3. Assume that $i, i_1, i_2, \ldots, i_k, j$ is a shortest path from i to j. Starting with the initial vertex i, a decision has been made to go to vertex i_1. Following this decision, the problem state is defined by vertex i_1 and we need to find a path from i_1 to j. It is clear that the sequence $i_1, i_2, \ldots, i_k, j$ must constitute a shortest i_1 to j path. If not, let $i_1, r_1, r_2, \ldots, r_q, j$ be a shortest i_1 to j path. Then $i, i_1, r_1, \cdots, r_q, j$ is an i to j path that is shorter than the path $i, i_1, i_2, \ldots, i_k, j$. Therefore the principle of optimality applies for this problem. □

Example 5.6 [0/1 knapsack] The 0/1 knapsack problem is similar to the knapsack problem of Section 4.2 except that the x_i's are restricted to have a value of either 0 or 1. Using KNAP(l, j, y) to represent the problem

$$\begin{array}{c} \text{maximize } \sum_{l \le i \le j} p_i x_i \\ \text{subject to } \sum_{l \le i \le j} w_i x_i \le y \\ x_i = 0 \text{ or } 1,\ l \le i \le j \end{array} \tag{5.1}$$

the knapsack problem is KNAP$(1, n, m)$. Let $y_1, y_2, \ldots, y_n$ be an optimal sequence of 0/1 values for $x_1, x_2, \ldots, x_n$, respectively. If $y_1 = 0$, then $y_2, y_3, \ldots, y_n$ must constitute an optimal sequence for the problem KNAP(2, n, m). If it does not, then $y_1, y_2, \ldots, y_n$ is not an optimal sequence for KNAP$(1, n, m)$. If $y_1 = 1$, then $y_2, \ldots, y_n$ must be an optimal sequence for the problem KNAP$(2, n, m - w_1)$. If it isn't, then there is another 0/1 sequence $z_2, z_3, \ldots, z_n$ such that $\sum_{2 \le i \le n} w_i z_i \le m - w_1$ and $\sum_{2 \le i \le n} p_i z_i > \sum_{2 \le i \le n} p_i y_i$. Hence, the sequence $y_1, z_2, z_3, \ldots, z_n$ is a sequence for (5.1) with greater value. Again the principle of optimality applies. □

Let S_0 be the initial problem state. Assume that n decisions d_i, $1 \le i \le n$, have to be made. Let $D_1 = \{r_1, r_2, \ldots, r_j\}$ be the set of possible decision values for d_1. Let S_i be the problem state following the choice of decision r_i, $1 \le i \le j$. Let Γ_i be an optimal sequence of decisions with respect to the problem state S_i. Then, when the principle of optimality holds, an optimal sequence of decisions with respect to S_0 is the best of the decision sequences r_i, Γ_i, $1 \le i \le j$.

Example 5.7 [Shortest path] Let A_i be the set of vertices adjacent to vertex i. For each vertex $k \in A_i$, let Γ_k be a shortest path from k to j. Then, a shortest i to j path is the shortest of the paths $\{i, \Gamma_k | k \in A_i\}$. □

Example 5.8 [0/1 knapsack] Let $g_j(y)$ be the value of an optimal solution to KNAP$(j + 1, n, y)$. Clearly, $g_0(m)$ is the value of an optimal solution to KNAP$(1, n, m)$. The possible decisions for x_1 are 0 and 1 ($D_1 = \{0, 1\}$). From the principle of optimality it follows that

$$g_0(m) = \max \{g_1(m),\ g_1(m - w_1) + p_1\} \tag{5.2}$$

□

While the principle of optimality has been stated only with respect to the initial state and decision, it can be applied equally well to intermediate states and decisions. The next two examples show how this can be done.

Example 5.9 [Shortest path] Let k be an intermediate vertex on a shortest i to j path $i, i_1, i_2, \ldots, k, p_1, p_2, \ldots, j$. The paths $i, i_1, \ldots, k$ and $k, p_1, \ldots, j$ must, respectively, be shortest i to k and k to j paths. □

Example 5.10 [0/1 knapsack] Let $y_1, y_2, \ldots, y_n$ be an optimal solution to KNAP$(1, n, m)$. Then, for each j, $1 \leq j \leq n$, $y_1, \ldots, y_j$, and $y_{j+1}, \ldots, y_n$ must be optimal solutions to the problems KNAP$(1, j, \sum_{1 \leq i \leq j} w_i y_i)$ and KNAP$(j+1, n, m - \sum_{1 \leq i \leq j} w_i y_i)$ respectively. This observation allows us to generalize (5.2) to

$$g_i(y) = \max \{g_{i+1}(y),\ g_{i+1}(y - w_{i+1}) + p_{i+1}\} \tag{5.3}$$

□

The recursive application of the optimality principle results in a recurrence equation of type (5.3). Dynamic programming algorithms solve this recurrence to obtain a solution to the given problem instance. The recurrence (5.3) can be solved using the knowledge $g_n(y) = 0$ for all $y \geq 0$ and $g_n(y) = -\infty$ for $y < 0$. From $g_n(y)$, one can obtain $g_{n-1}(y)$ using (5.3) with $i = n - 1$. Then, using $g_{n-1}(y)$, one can obtain $g_{n-2}(y)$. Repeating in this way, one can determine $g_1(y)$ and finally $g_0(m)$ using (5.3) with $i = 0$.

Example 5.11 [0/1 knapsack] Consider the case in which $n = 3$, $w_1 = 2, w_2 = 3, w_3 = 4$, $p_1 = 1, p_2 = 2, p_3 = 5$, and $m = 6$. We have to compute $g_0(6)$. The value of $g_0(6) = \max \{g_1(6),\ g_1(4) + 1\}$.

In turn, $g_1(6) = \max \{g_2(6),\ g_2(3)+2\}$. But $g_2(6) = \max \{g_3(6),\ g_3(2)+5\} = \max \{0, 5\} = 5$. Also, $g_2(3) = \max \{g_3(3),\ g_3(3-4)+5\} = \max \{0, -\infty\} = 0$. Thus, $g_1(6) = \max \{5, 2\} = 5$.

Similarly, $g_1(4) = \max \{g_2(4),\ g_2(4-3)+2\}$. But $g_2(4) = \max \{g_3(4), g_3(4-4)+5\} = \max \{0, 5\} = 5$. The value of $g_2(1) = \max \{g_3(1),\ g_3(1-4)+5\} = \max \{0, -\infty\} = 0$. Thus, $g_1(4) = \max \{5, 0\} = 5$.

Therefore, $g_0(6) = \max \{5, 5+1\} = 6$. □

Example 5.12 [Shortest path] Let P_j be the set of vertices adjacent to vertex j (that is, $k \in P_j$ iff $\langle k, j \rangle \in E(G)$). For each $k \in P_j$, let Γ_k be a shortest i to k path. The principle of optimality holds and a shortest i to j path is the shortest of the paths $\{\Gamma_k, j | k \in P_j\}$.

To obtain this formulation, we started at vertex j and looked at the last decision made. The last decision was to use one of the edges $\langle k, j \rangle$, $k \in P_j$. In a sense, we are looking backward on the i to j path. □

Example 5.13 [0/1 knapsack] Looking backward on the sequence of decisions $x_1, x_2, \ldots, x_n$, we see that

$$f_j(y) = \max \{f_{j-1}(y),\ f_{j-1}(y - w_j) + p_j\} \tag{5.4}$$

where $f_j(y)$ is the value of an optimal solution to KNAP$(1, j, y)$.

The value of an optimal solution to KNAP$(1, n, m)$ is $f_n(m)$. Equation 5.4 can be solved by beginning with $f_0(y) = 0$ for all y, $y \geq 0$, and $f_0(y) = -\infty$, for all y, $y < 0$. From this, $f_1, f_2, \ldots, f_n$ can be successively obtained. □

The solution method outlined in Examples 5.12 and 5.13 may indicate that one has to look at all possible decision sequences to obtain an optimal decision sequence using dynamic programming. This is not the case. Because of the use of the principle of optimality, decision sequences containing subsequences that are suboptimal are *not* considered. Although the total number of different decision sequences is exponential in the number of decisions (if there are d choices for each of the n decisions to be made then there are d^n possible decision sequences), dynamic programming algorithms often have a polynomial complexity.

Another important feature of the dynamic programming approach is that optimal solutions to subproblems are retained so as to avoid recomputing their values. The use of these tabulated values makes it natural to recast the recursive equations into an iterative algorithm. Most of the dynamic programming algorithms in this chapter are expressed in this way.

The remaining sections of this chapter apply dynamic programming to a variety of problems. These examples should help you understand the method better and also realize the advantage of dynamic programming over explicitly enumerating all decision sequences.

EXERCISES

1. The principle of optimality does not hold for every problem whose solution can be viewed as the result of a sequence of decisions. Find two problems for which the principle does not hold. Explain why the principle does not hold for these problems.

2. For the graph of Figure 5.1, find the shortest path between the nodes 1 and 2. Use the recurrence relations derived in Examples 5.10 and 5.13.

5.2 MULTISTAGE GRAPHS

A multistage graph $G = (V, E)$ is a directed graph in which the vertices are partitioned into $k \geq 2$ disjoint sets V_i, $1 \leq i \leq k$. In addition, if $\langle u, v \rangle$ is an edge in E, then $u \in V_i$ and $v \in V_{i+1}$ for some $i, 1 \leq i < k$. The sets V_1 and V_k are such that $|V_1| = |V_k| = 1$. Let s and t, respectively, be the vertices in V_1 and V_k. The vertex s is the *source*, and t the *sink*. Let $c(i, j)$ be the cost of edge $\langle i, j \rangle$. The cost of a path from s to t is the sum of the costs of the edges on the path. The *multistage graph problem* is to find a minimum-cost

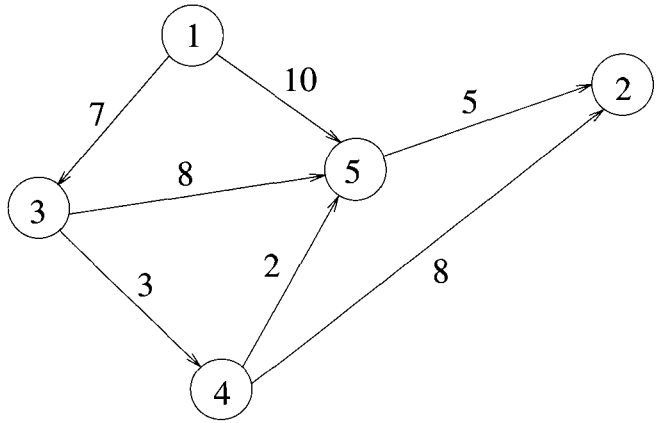


Figure 5.1 Graph for Exercise 2 (Section 5.1)

path from s to t. Each set V_i defines a stage in the graph. Because of the constraints on E, every path from s to t starts in stage 1, goes to stage 2, then to stage 3, then to stage 4, and so on, and eventually terminates in stage k. Figure 5.2 shows a five-stage graph. A minimum-cost s to t path is indicated by the broken edges.

Many problems can be formulated as multistage graph problems. We give only one example. Consider a resource allocation problem in which n units of resource are to be allocated to r projects. If j, $0 \leq j \leq n$, units of the resource are allocated to project i, then the resulting net profit is $N(i,j)$. The problem is to allocate the resource to the r projects in such a way as to maximize total net profit. This problem can be formulated as an $r+1$ stage graph problem as follows. Stage i, $1 \leq i \leq r$, represents project i. There are $n+1$ vertices $V(i,j)$, $0 \leq j \leq n$, associated with stage i, $2 \leq i \leq r$. Stages 1 and $r+1$ each have one vertex, $V(1,0) = s$ and $V(r+1,n) = t$, respectively. Vertex $V(i,j)$, $2 \leq i \leq r$, represents the state in which a total of j units of resource have been allocated to projects $1, 2, \ldots, i-1$. The edges in G are of the form $\langle V(i,j), V(i+1,l)\rangle$ for all $j \leq l$ and $1 \leq i < r$. The edge $\langle V(i,j), V(i+1,l)\rangle$, $j \leq l$, is assigned a weight or cost of $N(i, l-j)$ and corresponds to allocating $l-j$ units of resource to project i, $1 \leq i < r$. In addition, G has edges of the type $\langle V(r,j), V(r+1,n)\rangle$. Each such edge is assigned a weight of $\max_{0 \leq p \leq n-j}\{N(r,p)\}$. The resulting graph for a three-project problem with $n = 4$ is shown in Figure 5.3. It should be easy to see that an optimal allocation of resources is defined by a maximum cost s to t path. This is easily converted into a minimum-cost problem by changing the sign of all the edge costs.

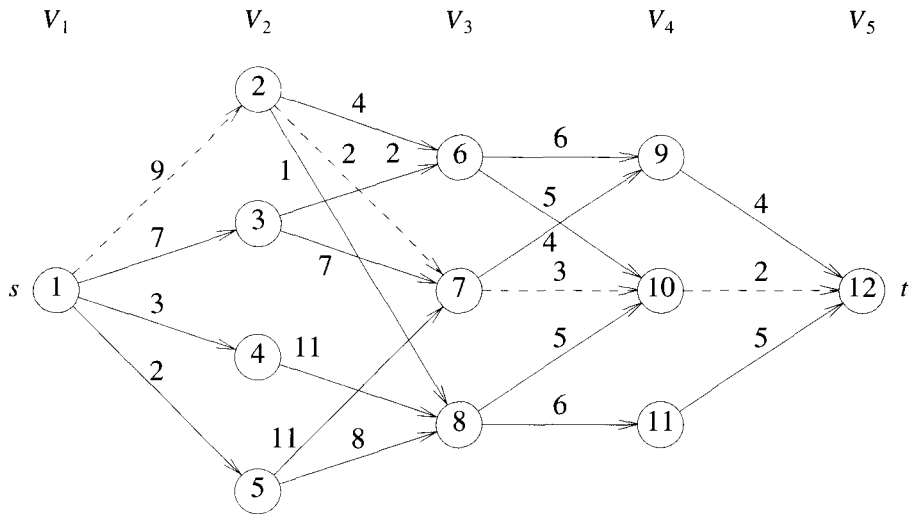


Figure 5.2 Five-stage graph

A dynamic programming formulation for a k-stage graph problem is obtained by first noticing that every s to t path is the result of a sequence of $k-2$ decisions. The ith decision involves determining which vertex in V_{i+1}, $1 \leq i \leq k-2$, is to be on the path. It is easy to see that the principle of optimality holds. Let $p(i,j)$ be a minimum-cost path from vertex j in V_i to vertex t. Let $cost(i,j)$ be the cost of this path. Then, using the forward approach, we obtain

$$cost(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j,l) + cost(i+1,l)\} \tag{5.5}$$

Since, $cost(k-1,j) = c(j,t)$ if $\langle j,t \rangle \in E$ and $cost(k-1,j) = \infty$ if $\langle j,t \rangle \notin E$, (5.5) may be solved for $cost(1,s)$ by first computing $cost(k-2,j)$ for all $j \in V_{k-2}$, then $cost(k-3,j)$ for all $j \in V_{k-3}$, and so on, and finally $cost(1,s)$. Trying this out on the graph of Figure 5.2, we obtain

$$\begin{aligned} cost(3,6) &= \min \{6 + cost(4,9), 5 + cost(4,10)\} \\ &= 7 \\ cost(3,7) &= \min \{4 + cost(4,9), 3 + cost(4,10)\} \\ &= 5 \end{aligned}$$

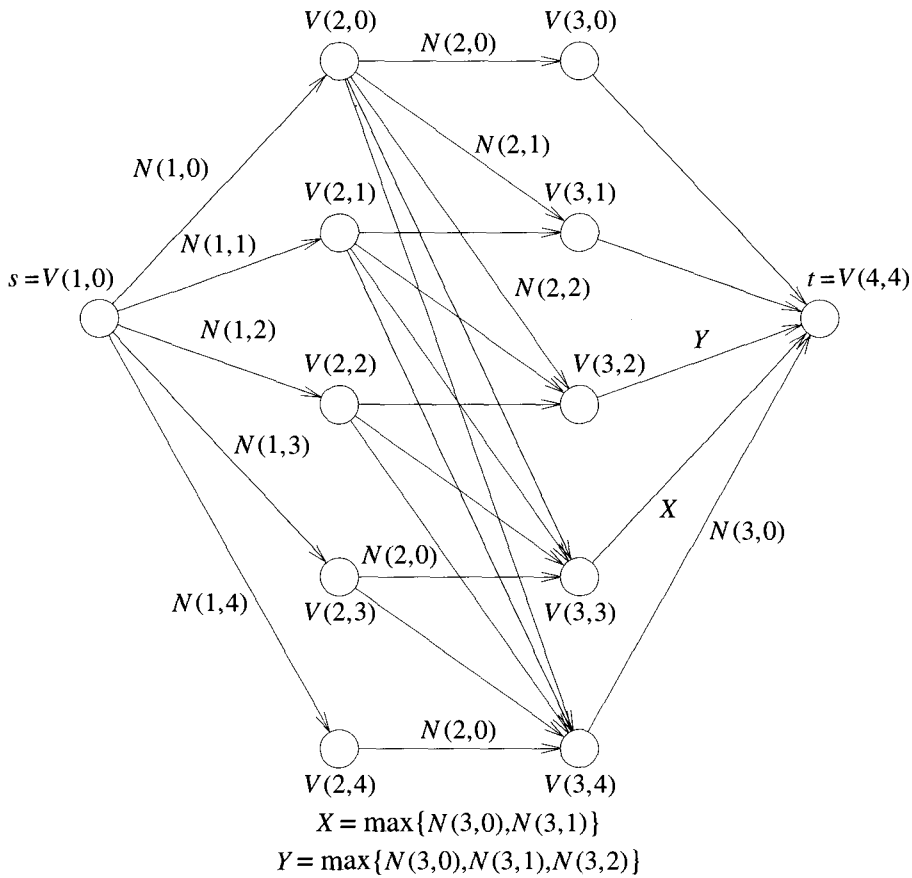


Figure 5.3 Four-stage graph corresponding to a three-project problem

$$
\begin{aligned}
cost(3,8) &= 7 \\
cost(2,2) &= \min\ \{4 + cost(3,6), 2 + cost(3,7), 1 + cost(3,8)\} \\
&= 7 \\
cost(2,3) &= 9 \\
cost(2,4) &= 18 \\
cost(2,5) &= 15 \\
cost(1,1) &= \min\ \{9 + cost(2,2), 7 + cost(2,3), 3 + cost(2,4), \\
&\qquad\qquad 2 + cost(2,5)\} \\
&= 16
\end{aligned}
$$

Note that in the calculation of $cost(2,2)$, we have reused the values of $cost(3,6), cost(3,7)$, and $cost(3,8)$ and so avoided their recomputation. A minimum cost s to t path has a cost of 16. This path can be determined easily if we record the decision made at each state (vertex). Let $d(i,j)$ be the value of l (where l is a node) that minimizes $c(j,l) + cost(i+1,l)$ (see Equation 5.5). For Figure 5.2 we obtain

$$
\begin{array}{llllllllll}
d(3,6) = 10; & d(3,7) = 10; & d(3,8) = 10; & & & & & & & \\
d(2,2) = 7; & d(2,3) = 6; & d(2,4) = 8; & d(2,5) = 8; & & & & & & \\
d(1,1) = 2 & & & & & & & & &
\end{array}
$$

Let the minimum-cost path be $s = 1, v_2, v_3, \ldots, v_{k-1}, t$. It is easy to see that $v_2 = d(1,1) = 2, v_3 = d(2, d(1,1)) = 7$, and $v_4 = d(3, d(2,\ d(1,1))) = d(3,7) = 10$.

Before writing an algorithm to solve (5.5) for a general k-stage graph, let us impose an ordering on the vertices in V. This ordering makes it easier to write the algorithm. We require that the n vertices in V are indexed 1 through n. Indices are assigned in order of stages. First, s is assigned index 1, then vertices in V_2 are assigned indices, then vertices from V_3, and so on. Vertex t has index n. Hence, indices assigned to vertices in V_{i+1} are bigger than those assigned to vertices in V_i (see Figure 5.2). As a result of this indexing scheme, *cost* and d can be computed in the order $n-1, n-2, \ldots, 1$. The first subscript in *cost*, p, and d only identifies the stage number and is omitted in the algorithm. The resulting algorithm, in pseudocode, is FGraph (Algorithm 5.1).

The complexity analysis of the function FGraph is fairly straightforward. If G is represented by its adjacency lists, then r in line 9 of Algorithm 5.1 can be found in time proportional to the degree of vertex j. Hence, if G has $|E|$ edges, then the time for the **for** loop of line 7 is $\Theta(|V| + |E|)$. The time for the **for** loop of line 16 is $\Theta(k)$. Hence, the total time is $\Theta(|V| + |E|)$. In addition to the space needed for the input, space is needed for *cost*[ ], d[ ], and p[ ].

```
1   Algorithm FGraph(G, k, n, p)
2   // The input is a k-stage graph G = (V, E) with n vertices
3   // indexed in order of stages. E is a set of edges and c[i, j]
4   // is the cost of ⟨i, j⟩. p[1 : k] is a minimum-cost path.
5   {
6       cost[n] := 0.0;
7       for j := n − 1 to 1 step −1 do
8       { // Compute cost[j].
9           Let r be a vertex such that ⟨j, r⟩ is an edge
10          of G and c[j, r] + cost[r] is minimum;
11          cost[j] := c[j, r] + cost[r];
12          d[j] := r;
13      }
14      // Find a minimum-cost path.
15      p[1] := 1; p[k] := n;
16      for j := 2 to k − 1 do p[j] := d[p[j − 1]];
17  }
```

Algorithm 5.1 Multistage graph pseudocode corresponding to the forward approach

The multistage graph problem can also be solved using the backward approach. Let $bp(i, j)$ be a minimum-cost path from vertex s to a vertex j in V_i. Let $bcost(i, j)$ be the cost of $bp(i, j)$. From the backward approach we obtain

$$bcost(i, j) = \min_{\substack{l \in V_{i-1} \\ \langle l, j \rangle \in E}} \{bcost(i-1, l) + c(l, j)\} \tag{5.6}$$

Since $bcost(2, j) = c(1, j)$ if $\langle 1, j \rangle \in E$ and $bcost(2, j) = \infty$ if $\langle 1, j \rangle \notin E$, $bcost(i, j)$ can be computed using (5.6) by first computing $bcost$ for $i = 3$, then for $i = 4$, and so on. For the graph of Figure 5.2, we obtain

$$\begin{aligned}
bcost(3,6) &= \min\ \{bcost(2,2) + c(2,6), bcost(2,3) + c(3,6)\} \\
&= \min\ \{9 + 4, 7 + 2\} \\
&= 9 \\
bcost(3,7) &= 11 \\
bcost(3,8) &= 10 \\
bcost(4,9) &= 15
\end{aligned}$$

$$
\begin{aligned}
bcost(4,10) &= 14 \\
bcost(4,11) &= 16 \\
bcost(5,12) &= 16
\end{aligned}
$$

The corresponding algorithm, in pseudocode, to obtain a minimum-cost $s-t$ path is BGraph (Algorithm 5.2). The first subscript on *bcost*, p, and d are omitted for the same reasons as before. This algorithm has the same complexity as FGraph provided G is now represented by its inverse adjacency lists (i.e., for each vertex v we have a list of vertices w such that $\langle w, v\rangle \in E$).

```
1   Algorithm BGraph(G, k, n, p)
2   // Same function as FGraph
3   {
4       bcost[1] := 0.0;
5       for j := 2 to n do
6       { // Compute bcost[j].
7           Let r be such that ⟨r, j⟩ is an edge of
8           G and bcost[r] + c[r, j] is minimum;
9           bcost[j] := bcost[r] + c[r, j];
10          d[j] := r;
11      }
12      // Find a minimum-cost path.
13      p[1] := 1; p[k] := n;
14      for j := k − 1 to 2 do p[j] := d[p[j + 1]];
15  }
```

Algorithm 5.2 Multistage graph pseudocode corresponding to backward approach

It should be easy to see that both FGraph and BGraph work correctly even on a more generalized version of multistage graphs. In this generalization, the graph is permitted to have edges $\langle u, v\rangle$ such that $u \in V_i, v \in V_j$, and $i < j$.

Note: In the pseudocodes FGraph and BGraph, $bcost(i, j)$ is set to ∞ for any $\langle i, j\rangle \notin E$. When programming these pseudocodes, one could use the maximum allowable floating point number for ∞. If the weight of any such edge is added to some other costs, a floating point overflow might occur. Care should be taken to avoid such overflows.

EXERCISES

1. Find a minimum-cost path from s to t in the multistage graph of Figure 5.4. Do this first using the forward approach and then using the backward approach.

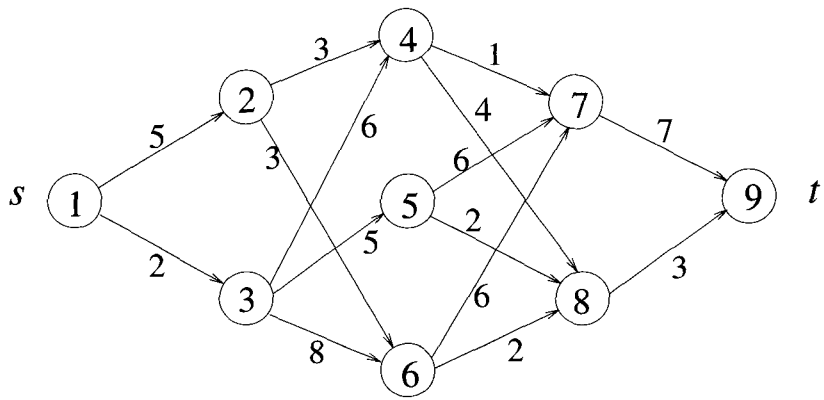


Figure 5.4 Multistage graph for Exercise 1

2. Refine Algorithm 5.1 into a program. Assume that G is represented by its adjacency lists. Test the correctness of your code using suitable graphs.

3. Program Algorithm 5.1. Assume that G is an array $G[1 : e, 1 : 3]$. Each edge $\langle i, j \rangle$, $i < j$, of G is stored in $G[q]$, for some q and $G[q, 1] = i$, $G[q, 2] = j$, and $G[q, 3] =$ cost of edge $\langle i, j \rangle$. Assume that $G[q, 1] \leq G[q + 1, 1]$ for $1 \leq q < e$, where e is the number of edges in the multistage graph. Test the correctness of your function using suitable multistage graphs. What is the time complexity of your function?

4. Program Algorithm 5.2 for the multistage graph problem using the backward approach. Assume that the graph is represented using inverse adjacency lists. Test its correctness. What is its complexity?

5. Do Exercise 4 using the graph representation of Exercise 3. This time, however, assume that $G[q, 2] \leq G[q + 1, 2]$ for $1 \leq q < e$.

6. Extend the discussion of this section to directed acyclic graphs (dags). Suppose the vertices of a dag are numbered so that all edges have the form $\langle i, j \rangle$, $i < j$. What changes, if any, need to be made to Algorithm 5.1 to find the length of the longest path from vertex 1 to vertex n?

7. [W. Miller] Show that BGraph1 computes shortest paths for directed acyclic graphs represented by adjacency lists (instead of inverse adjacency lists as in BGraph).

```
1   Algorithm BGraph1(G, n)
2   {
3       bcost[1] := 0.0;
4       for j := 2 to n do bcost[j] := ∞;
5       for j := 1 to n − 1 do
6           for each r such that ⟨j, r⟩ is an edge of G do
7               bcost[r] := min(bcost[r], bcost[j] + c[j, r]);
8   }
```

Note: There is a possibility of a floating point overflow in this function. In such cases the program should be suitably modified.

5.3 ALL-PAIRS SHORTEST PATHS

Let $G = (V, E)$ be a directed graph with n vertices. Let *cost* be a cost adjacency matrix for G such that $cost(i, i) = 0$, $1 \leq i \leq n$. Then $cost(i, j)$ is the length (or cost) of edge $\langle i, j \rangle$ if $\langle i, j \rangle \in E(G)$ and $cost(i, j) = \infty$ if $i \neq j$ and $\langle i, j \rangle \notin E(G)$. The *all-pairs shortest-path problem* is to determine a matrix A such that $A(i, j)$ is the length of a shortest path from i to j. The matrix A can be obtained by solving n single-source problems using the algorithm ShortestPaths of Section 4.8. Since each application of this procedure requires $O(n^2)$ time, the matrix A can be obtained in $O(n^3)$ time. We obtain an alternate $O(n^3)$ solution to this problem using the principle of optimality. Our alternate solution requires a weaker restriction on edge costs than required by ShortestPaths. Rather than require $cost(i, j) \geq 0$, for every edge $\langle i, j \rangle$, we only require that G have no cycles with negative length. Note that if we allow G to contain a cycle of negative length, then the shortest path between any two vertices on this cycle has length $-\infty$.

Let us examine a shortest i to j path in G, $i \neq j$. This path originates at vertex i and goes through some intermediate vertices (possibly none) and terminates at vertex j. We can assume that this path contains no cycles for if there is a cycle, then this can be deleted without increasing the path length (no cycle has negative length). If k is an intermediate vertex on this shortest path, then the subpaths from i to k and from k to j must be shortest paths from i to k and k to j, respectively. Otherwise, the i to j path is not of minimum length. So, the principle of optimality holds. This alerts us to the prospect of using dynamic programming. If k is the intermediate vertex with highest index, then the i to k path is a shortest i to k path in G going through no vertex with index greater than $k - 1$. Similarly the k to j path is a shortest k to j path in G going through no vertex of index greater than

$k-1$. We can regard the construction of a shortest i to j path as first requiring a decision as to which is the highest indexed intermediate vertex k. Once this decision has been made, we need to find two shortest paths, one from i to k and the other from k to j. Neither of these may go through a vertex with index greater than $k-1$. Using $A^k(i,j)$ to represent the length of a shortest path from i to j going through no vertex of index greater than k, we obtain

$$A(i,j) = \min \{ \min_{1 \le k \le n} \{A^{k-1}(i,k) + A^{k-1}(k,j)\}, cost(i,j)\} \tag{5.7}$$

Clearly, $A^0(i,j) = cost(i,j)$, $1 \le i \le n$, $1 \le j \le n$. We can obtain a recurrence for $A^k(i,j)$ using an argument similar to that used before. A shortest path from i to j going through no vertex higher than k either goes through vertex k or it does not. If it does, $A^k(i,j) = A^{k-1}(i,k) + A^{k-1}(k,j)$. If it does not, then no intermediate vertex has index greater than $k-1$. Hence $A^k(i,j) = A^{k-1}(i,j)$. Combining, we get

$$A^k(i,j) = \min \{A^{k-1}(i,j),\ A^{k-1}(i,k) + A^{k-1}(k,j)\},\quad k \ge 1 \tag{5.8}$$

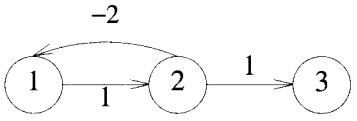
The following example shows that (5.8) is not true for graphs with cycles of negative length.

Example 5.14 Figure 5.5 shows a digraph together with its matrix A^0. For this graph $A^2(1,3) \neq \min\{A^1(1,3),\ A^1(1,2) + A^1(2,3)\} = 2$. Instead we see that $A^2(1,3) = -\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \ldots, 1, 2, 3$$

can be made arbitrarily small. This is so because of the presence of the cycle 1 2 1 which has a length of -1. □

Recurrence (5.8) can be solved for A^n by first computing A^1, then A^2, then A^3, and so on. Since there is no vertex in G with index greater than n, $A(i,j) = A^n(i,j)$. Function AllPaths computes $A^n(i,j)$. The computation is done inplace so the superscript on A is not needed. The reason this computation can be carried out in-place is that $A^k(i,k) = A^{k-1}(i,k)$ and $A^k(k,j) = A^{k-1}(k,j)$. Hence, when A^k is formed, the kth column and row do not change. Consequently, when $A^k(i,j)$ is computed in line 11 of Algorithm 5.3, $A(i,k) = A^{k-1}(i,k) = A^k(i,k)$ and $A(k,j) = A^{k-1}(k,j) = A^k(k,j)$. So, the old values on which the new values are based do not change on this iteration.



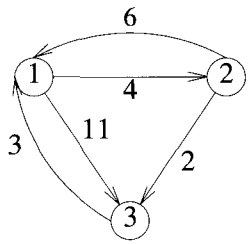
$$\begin{bmatrix} 0 & 1 & \infty \\ -2 & 0 & 1 \\ \infty & \infty & 0 \end{bmatrix}$$

Figure 5.5 Graph with negative cycle

```
0   Algorithm AllPaths(cost, A, n)
1   // cost[1 : n, 1 : n] is the cost adjacency matrix of a graph with
2   // n vertices; A[i, j] is the cost of a shortest path from vertex
3   // i to vertex j. cost[i, i] = 0.0, for 1 ≤ i ≤ n.
4   {
5       for i := 1 to n do
6           for j := 1 to n do
7               A[i, j] := cost[i, j]; // Copy cost into A.
8       for k := 1 to n do
9           for i := 1 to n do
10              for j := 1 to n do
11                  A[i, j] := min(A[i, j], A[i, k] + A[k, j]);
12  }
```

Algorithm 5.3 Function to compute lengths of shortest paths

Example 5.15 The graph of Figure 5.6(a) has the cost matrix of Figure 5.6(b). The initial A matrix, $A^{(0)}$, plus its values after 3 iterations $A^{(1)}$, $A^{(2)}$, and $A^{(3)}$ are given in Figure 5.6. □



(a) Example digraph

A^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

(b) A^0

A^1	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

(c) A^1

A^2	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

(d) A^2

A^3	1	2	3
1	0	4	6
2	5	0	2
3	3	7	0

(e) A^3

Figure 5.6 Directed graph and associated matrices

Let $M = \max\{cost(i,j) | \langle i,j \rangle \in E(G)\}$. It is easy to see that $A^n(ij) \leq (n-1)M$. From the working of AllPaths, it is clear that if $\langle i,j \rangle \notin E(G)$ and $i \neq j$, then we can initialize $cost(i,j)$ to any number greater than $(n-1)M$ (rather than the maximum allowable floating point number). If, at termination, $A(i,j) > (n-1)M$, then there is no directed path from i to j in G. Even for this choice of ∞, care should be taken to avoid any floating point overflows.

The time needed by AllPaths (Algorithm 5.3) is especially easy to determine because the looping is independent of the data in the matrix A. Line 11 is iterated n^3 times, and so the time for AllPaths is $\Theta(n^3)$. An exercise examines the extensions needed to obtain the i to j paths with these lengths. Some speedup can be obtained by noticing that the innermost **for** loop need be executed only when $A(i,k)$ and $A(k,j)$ are not equal to ∞.

EXERCISES

1. (a) Does the recurrence (5.8) hold for the graph of Figure 5.7? Why?

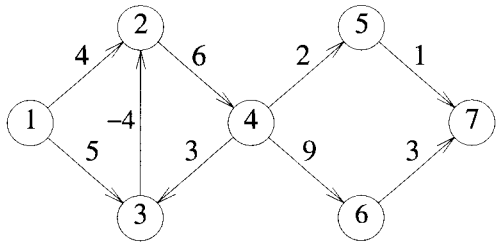


Figure 5.7 Graph for Exercise 1

   (b) Why does Equation 5.8 not hold for graphs with cycles of negative length?

2. Modify the function AllPaths so that a shortest path is output for each pair of vertices (i, j). What are the time and space complexities of the new algorithm?

3. Let A be the adjacency matrix of a directed graph G. Define the transitive closure A^+ of A to be a matrix with the property $A^+(i,j) = 1$ iff G has a directed path, containing at least one edge, from vertex i to vertex j. $A^+(i,j) = 0$ otherwise. The reflexive transitive closure A^* is a matrix with the property $A^*(i,j) = 1$ iff G has a path, containing zero or more edges, from i to j. $A^*(i,j) = 0$ otherwise.

   (a) Obtain A^+ and A^* for the directed graph of Figure 5.8.

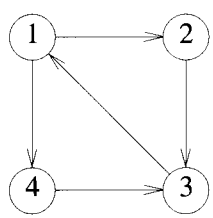


Figure 5.8 Graph for Exercise 3

   (b) Let $A^k(i,j) = 1$ iff there is a path with zero or more edges from i to j going through no vertex of index greater than k. Define A^0 in terms of the adjacency matrix A.

(c) Obtain a recurrence between A^k and A^{k-1} similar to (5.8). Use the logical operators **or** and **and** rather than **min** and $+$.

(d) Write an algorithm, using the recurrence of part (c), to find A^*. Your algorithm can use only $O(n^2)$ space. What is its time complexity?

(e) Show that $A^+ = A \times A^*$, where matrix multiplication is defined as $A^+(i,j) = \vee_{k=1}^{n}(A(i,k) \wedge A^*(k,j))$. The operation $\vee$ is the logical **or** operation, and $\wedge$ the logical **and** operation. Hence A^+ may be computed from A^*.

5.4 SINGLE-SOURCE SHORTEST PATHS: GENERAL WEIGHTS

We now consider the single-source shortest path problem discussed in Section 4.8 when some or all of the edges of the directed graph G may have negative length. ShortestPaths (Algorithm 4.14) does not necessarily give the correct results on such graphs. To see this, consider the graph of Figure 5.9. Let $v = 1$ be the source vertex. Referring back to Algorithm 4.14, since $n = 3$, the loop of lines 12 to 22 is iterated just once. Also $u = 3$ in lines 15 and 16, and so no changes are made to $dist[\ ]$. The algorithm terminates with $dist[2] = 7$ and $dist[3] = 5$. The shortest path from 1 to 3 is $1, 2, 3$. This path has length 2, which is less than the computed value of $dist[3]$.

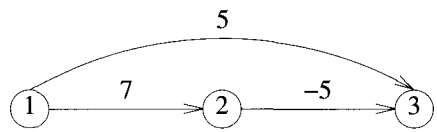


Figure 5.9 Directed graph with a negative-length edge

When negative edge lengths are permitted, we require that the graph have no cycles of negative length. This is necessary to ensure that shortest paths consist of a finite number of edges. For example, in the graph of Figure 5.5, the length of the shortest path from vertex 1 to vertex 3 is $-\infty$. The length of the path

$$1, 2, 1, 2, 1, 2, \cdots, 1, 2, 3$$

can be made arbitrarily small as was shown in Example 5.14.

When there are no cycles of negative length, there is a shortest path between any two vertices of an n-vertex graph that has at most $n - 1$ edges

on it. To see this, note that a path that has more than $n-1$ edges must repeat at least one vertex and hence must contain a cycle. Elimination of the cycles from the path results in another path with the same source and destination. This path is cycle-free and has a length that is no more than that of the original path, as the length of the eliminated cycles was at least zero. We can use this observation on the maximum number of edges on a cycle-free shortest path to obtain an algorithm to determine a shortest path from a source vertex to all remaining vertices in the graph. As in the case of ShortestPaths (Algorithm 4.14), we compute only the length, $dist[u]$, of the shortest path from the source vertex v to u. An exercise examines the extension needed to construct the shortest paths.

Let $dist^{\ell}[u]$ be the length of a shortest path from the source vertex v to vertex u under the constraint that the shortest path contains at most ℓ edges. Then, $dist^{1}[u] = cost[v,u]$, $1 \leq u \leq n$. As noted earlier, when there are no cycles of negative length, we can limit our search for shortest paths to paths with at most $n-1$ edges. Hence, $dist^{n-1}[u]$ is the length of an unrestricted shortest path from v to u.

Our goal then is to compute $dist^{n-1}[u]$ for all u. This can be done using the dynamic programming methodology. First, we make the following observations:

1. If the shortest path from v to u with at most k, $k > 1$, edges has no more than $k-1$ edges, then $dist^{k}[u] = dist^{k-1}[u]$.

2. If the shortest path from v to u with at most k, $k > 1$, edges has exactly k edges, then it is made up of a shortest path from v to some vertex j followed by the edge $\langle j,u \rangle$. The path from v to j has $k-1$ edges, and its length is $dist^{k-1}[j]$. All vertices i such that the edge $\langle i,u \rangle$ is in the graph are candidates for j. Since we are interested in a shortest path, the i that minimizes $dist^{k-1}[i] + cost[i,u]$ is the correct value for j.

These observations result in the following recurrence for $dist$:

$$dist^{k}[u] = \min \{dist^{k-1}[u], \min_{i} \{dist^{k-1}[i] + cost[i,u]\}\}$$

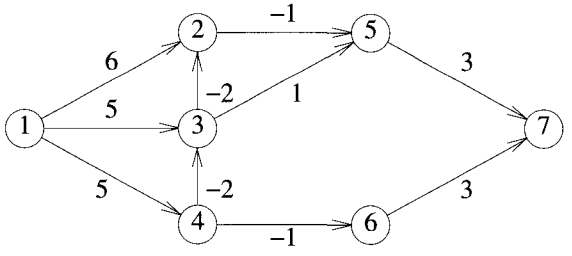
This recurrence can be used to compute $dist^{k}$ from $dist^{k-1}$, for $k = 2, 3, \ldots, n-1$.

Example 5.16 Figure 5.10 gives a seven-vertex graph, together with the arrays $dist^{k}$, $k = 1, \ldots, 6$. These arrays were computed using the equation just given. For instance, $dist^{k}[1] = 0$ for all k since 1 is the source node. Also, $dist^{1}[2] = 6, dist^{1}[3] = 5$, and $dist^{1}[4] = 5$, since there are edges from

1 to these nodes. The distance $dist^1[]$ is ∞ for the nodes 5, 6, and 7 since there are no edges to these from 1.

$$\begin{aligned} dist^2[2] &= \min\ \{dist^1[2], \min_i dist^1[i] + cost[i,2]\} \\ &= \min\ \{6, 0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty, \infty+\infty\} = 3 \end{aligned}$$

Here the terms $0+6, 5-2, 5+\infty, \infty+\infty, \infty+\infty$, and $\infty+\infty$ correspond to a choice of $i = 1, 3, 4, 5, 6$, and 7, respectively. The rest of the entries are computed in an analogous manner. □



(a) A directed graph

	$dist^k[1..7]$						
k	1	2	3	4	5	6	7
1	0	6	5	5	∞	∞	∞
2	0	3	3	5	5	4	∞
3	0	1	3	5	2	4	7
4	0	1	3	5	0	4	5
5	0	1	3	5	0	4	3
6	0	1	3	5	0	4	3

(b) $dist^k$

Figure 5.10 Shortest paths with negative edge lengths

An exercise shows that if we use the same memory location $dist[u]$ for $dist^k[u]$, $k = 1, \ldots, n-1$, then the final value of $dist[u]$ is still $dist^{n-1}[u]$. Using this fact and the recurrence for *dist* shown above, we arrive at the pseudocode of Algorithm 5.4 to compute the length of the shortest path from vertex v to each other vertex of the graph. This algorithm is referred to as the Bellman and Ford algorithm.

Each iteration of the **for** loop of lines 7 to 12 takes $O(n^2)$ time if adjacency matrices are used and $O(e)$ time if adjacency lists are used. Here e is the number of edges in the graph. The overall complexity is $O(n^3)$ when adjacency matrices are used and $O(ne)$ when adjacency lists are used. The observed complexity of the shortest-path algorithm can be reduced by noting that if none of the *dist* values change on one iteration of the **for** loop of lines 7 to 12, then none will change on successive iterations. So, this loop can be rewritten to terminate either after $n-1$ iterations or after the

```
1   Algorithm BellmanFord(v, cost, dist, n)
2   // Single-source/all-destinations shortest
3   // paths with negative edge costs
4   {
5       for i := 1 to n do // Initialize dist.
6           dist[i] := cost[v, i];
7       for k := 2 to n − 1 do
8           for each u such that u ≠ v and u has
9                   at least one incoming edge do
10              for each ⟨i, u⟩ in the graph do
11                  if dist[u] > dist[i] + cost[i, u] then
12                      dist[u] := dist[i] + cost[i, u];
13  }
```

Algorithm 5.4 Bellman and Ford algorithm to compute shortest paths

first iteration in which no *dist* values are changed, whichever occurs first. Another possibility is to maintain a queue of vertices i whose *dist* values changed on the previous iteration of the **for** loop. These are the only values for i that need to be considered in line 10 during the next iteration. When a queue of these values is maintained, we can rewrite the loop of lines 7 to 12 so that on each iteration, a vertex i is removed from the queue, and the *dist* values of all vertices adjacent from i are updated as in lines 11 and 12. Vertices whose *dist* values decrease as a result of this are added to the end of the queue unless they are already on it. The loop terminates when the queue becomes empty. These two strategies to improve the performance of BellmanFord are considered in the exercises. Other strategies for improving performance are discussed in References and Readings. □

EXERCISES

1. Find the shortest paths from node 1 to every other node in the graph of Figure 5.11 using the Bellman and Ford algorithm.

2. Prove the correctness of BellmanFord (Algorithm 5.4). Note that this algorithm does not faithfully implement the computation of the recurrence for $dist^k$. In fact, for $k < n-1$, the *dist* values following iteration k of the **for** loop of lines 7 to 12 may not be $dist^k$.

3. Transform BellmanFord into a program. Assume that graphs are represented using adjacency lists in which each node has an additional field

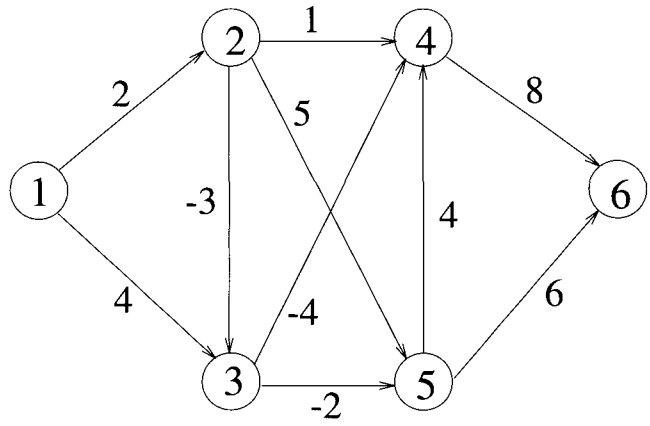


Figure 5.11 Graph for Exercise 1

called *cost* that gives the length of the edge represented by that node. As a result of this, there is no cost adjacency matrix. Generate some test graphs and test the correctness of your program.

4. Rewrite the algorithm BellmanFord so that the loop of lines 7 to 12 terminates either after $n - 1$ iterations or after the first iteration in which no *dist* values are changed, whichever occurs first.

5. Rewrite BellmanFord by replacing the loop of lines 7 to 12 with code that uses a queue of vertices that may potentially result in a reduction of other *dist* vertices. This queue initially contains all vertices that are adjacent from the source vertex v. On each successive iteration of the new loop, a vertex i is removed from the queue (unless the queue is empty), and the *dist* values to vertices adjacent from i are updated as in lines 11 and 12 of Algorithm 5.4. When the *dist* value of a vertex is reduced because of this, it is added to the queue unless it is already on the queue.

   (a) Prove that the new algorithm produces the same results as the original one.

   (b) Show that the complexity of the new algorithm is no more than that of the original one.

6. Compare the run-time performance of the Bellman and Ford algorithms of the preceding two exercises and that of Algorithm 5.4. For this, generate test graphs that will expose the relative performances of the three algorithms.

7. Modify algorithm BellmanFord so that it obtains the shortest paths, in addition to the lengths of these paths. What is the computing time of your algorithm?

5.5 OPTIMAL BINARY SEARCH TREES (*)

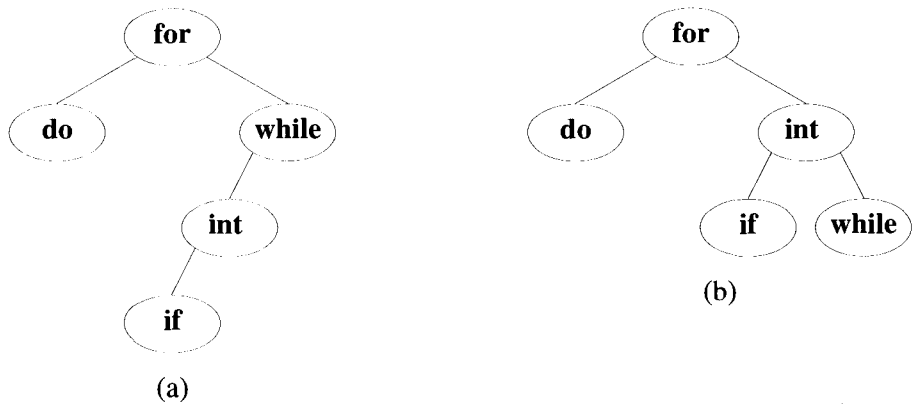


Figure 5.12 Two possible binary search trees

Given a fixed set of identifiers, we wish to create a binary search tree (see Section 2.3) organization. We may expect different binary search trees for the same identifier set to have different performance characteristics. The tree of Figure 5.12(a), in the worst case, requires four comparisons to find an identifier, whereas the tree of Figure 5.12(b) requires only three. On the average the two trees need 12/5 and 11/5 comparisons, respectively. For example, in the case of tree (a), it takes $1, 2, 2, 3$, and 4 comparisons, respectively, to find the identifiers **for, do, while, int,** and **if**. Thus the average number of comparisons is $\frac{1+2+2+3+4}{5} = \frac{12}{5}$. This calculation assumes that each identifier is searched for with equal probability and that no unsuccessful searches (i.e., searches for identifiers not in the tree) are made.

In a general situation, we can expect different identifiers to be searched for with different frequencies (or probabilities). In addition, we can expect unsuccessful searches also to be made. Let us assume that the given set of identifiers is $\{a_1, a_2, \ldots, a_n\}$ with $a_1 < a_2 < \cdots < a_n$. Let $p(i)$ be the probability with which we search for a_i. Let $q(i)$ be the probability that the identifier x being searched for is such that $a_i < x < a_{i+1}$, $0 \leq i \leq n$ (assume $a_0 = -\infty$ and $a_{n+1} = +\infty$). Then, $\sum_{0 \leq i \leq n} q(i)$ is the probability of

an unsuccessful search. Clearly, $\sum_{1 \le i \le n} p(i) + \sum_{0 \le i \le n} q(i) = 1$. Given this data, we wish to construct an optimal binary search tree for $\{a_1, a_2, \ldots, a_n\}$. First, of course, we must be precise about what we mean by an optimal binary search tree.

In obtaining a cost function for binary search trees, it is useful to add a fictitious node in place of every empty subtree in the search tree. Such nodes, called external nodes, are drawn square in Figure 5.13. All other nodes are internal nodes. If a binary search tree represents n identifiers, then there will be exactly n internal nodes and $n+1$ (fictitious) external nodes. Every internal node represents a point where a successful search may terminate. Every external node represents a point where an unsuccessful search may terminate.

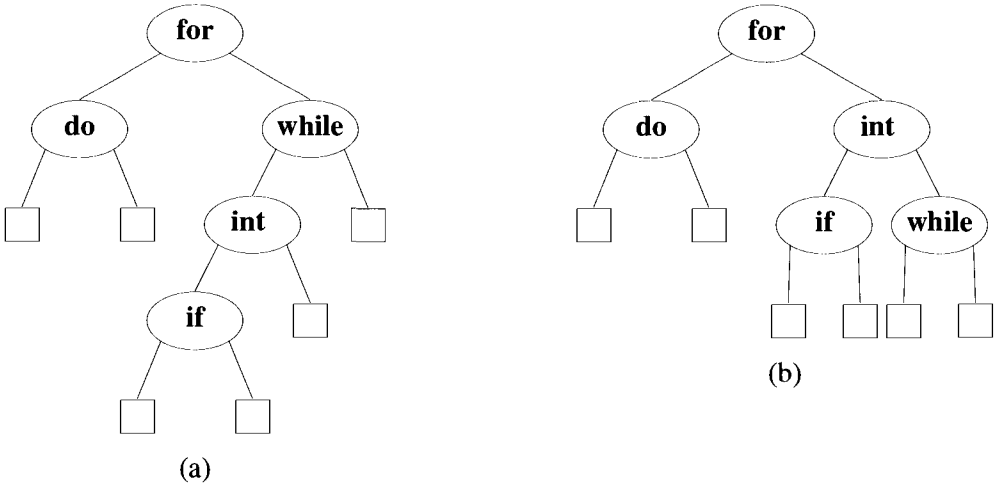


Figure 5.13 Binary search trees of Figure 5.12 with external nodes added

If a successful search terminates at an internal node at level l, then l iterations of the **while** loop of Algorithm 2.5 are needed. Hence, the expected cost contribution from the internal node for a_i is $p(i) * \text{level}(a_i)$.

Unsuccessful searches terminate with $t = 0$ (i.e., at an external node) in algorithm ISearch (Algorithm 2.5). The identifiers not in the binary search tree can be partitioned into $n+1$ equivalence classes $E_i, 0 \le i \le n$. The class E_0 contains all identifiers x such that $x < a_1$. The class E_i contains all identifiers x such that $a_i < x < a_{i+1}$, $1 \le i < n$. The class E_n contains all identifiers x, $x > a_n$. It is easy to see that for all identifiers in the same class E_i, the search terminates at the same external node. For identifiers in different E_i the search terminates at different external nodes. If the failure

node for E_i is at level l, then only $l-1$ iterations of the **while** loop are made. Hence, the cost contribution of this node is $q(i) * (level(E_i) - 1)$.

The preceding discussion leads to the following formula for the expected cost of a binary search tree:

$$\sum_{1 \le i \le n} p(i) * level(a_i) + \sum_{0 \le i \le n} q(i) * (level(E_i) - 1) \qquad (5.9)$$

We define an optimal binary search tree for the identifier set $\{a_1, a_2, \ldots, a_n\}$ to be a binary search tree for which (5.9) is minimum.

Example 5.17 The possible binary search trees for the identifier set (a_1, a_2, a_3) = (**do, if, while**) are given if Figure 5.14. With equal probabilities $p(i) = q(i) = 1/7$ for all i, we have

$$\begin{array}{llllll} cost(\text{tree a}) & = & 15/7 & cost(\text{tree b}) & = & 13/7 \\ cost(\text{tree c}) & = & 15/7 & cost(\text{tree d}) & = & 15/7 \\ cost(\text{tree e}) & = & 15/7 & & & \end{array}$$

As expected, tree b is optimal. With $p(1) = .5$, $p(2) = .1$, $p(3) = .05$, $q(0) = .15$, $q(1) = .1$, $q(2) = .05$ and $q(3) = .05$ we have

$$\begin{array}{llllll} cost(\text{tree a}) & = & 2.65 & cost(\text{tree b}) & = & 1.9 \\ cost(\text{tree c}) & = & 1.5 & cost(\text{tree d}) & = & 2.05 \\ cost(\text{tree e}) & = & 1.6 & & & \end{array}$$

For instance, $cost$(tree a) can be computed as follows. The contribution from successful searches is $3*0.5+2*0.1+0.05 = 1.75$ and the contribution from unsuccessful searches is $3 * 0.15 + 3 * 0.1 + 2 * 0.05 + 0.05 = 0.90$. All the other costs can also be calculated in a similar manner. Tree c is optimal with this assignment of p's and q's. □

To apply dynamic programming to the problem of obtaining an optimal binary search tree, we need to view the construction of such a tree as the result of a sequence of decisions and then observe that the principle of optimality holds when applied to the problem state resulting from a decision. A possible approach to this would be to make a decision as to which of the a_i's should be assigned to the root node of the tree. If we choose a_k, then it is clear that the internal nodes for $a_1, a_2, \ldots, a_{k-1}$ as well as the external nodes for the classes $E_0, E_1, \ldots, E_{k-1}$ will lie in the left subtree l of the root. The remaining nodes will be in the right subtree r. Define

$$cost(l) = \sum_{1 \le i < k} p(i) * \text{level}(a_i) + \sum_{0 \le i < k} q(i) * (\text{level}(E_i) - 1)$$

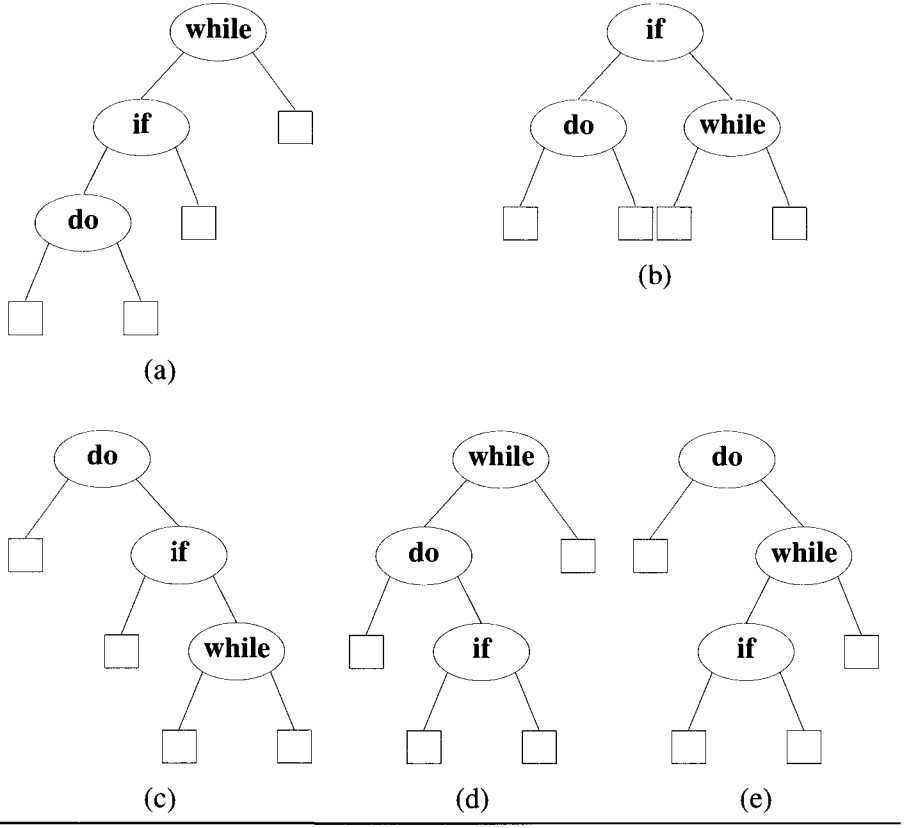


Figure 5.14 Possible binary search trees for the identifier set {**do, if, while**}

and

$$cost(r) = \sum_{k<i\leq n} p(i) * \text{level}(a_i) + \sum_{k<i\leq n} q(i) * (\text{level}(E_i) - 1)$$

In both cases the level is measured by regarding the root of the respective subtree to be at level 1.

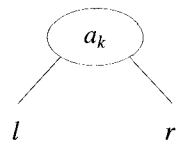


Figure 5.15 An optimal binary search tree with root a_k

Using $w(i,j)$ to represent the sum $q(i) + \sum_{l=i+1}^{j}(q(l) + p(l))$, we obtain the following as the expected cost of the search tree (Figure 5.15):

$$p(k) + cost(l) + cost(r) + w(0, k-1) + w(k, n) \tag{5.10}$$

If the tree is optimal, then (5.10) must be minimum. Hence, $cost(l)$ must be minimum over all binary search trees containing $a_1, a_2, \ldots, a_{k-1}$ and $E_0, E_1, \ldots, E_{k-1}$. Similarly $cost(r)$ must be minimum. If we use $c(i,j)$ to represent the cost of an optimal binary search tree t_{ij} containing $a_{i+1}, \ldots, a_j$ and $E_i, \ldots, E_j$, then for the tree to be optimal, we must have $cost(l) = c(0, k-1)$ and $cost(r) = c(k, n)$. In addition, k must be chosen such that

$$p(k) + c(0, k-1) + c(k, n) + w(0, k-1) + w(k, n)$$

is minimum. Hence, for $c(0,n)$ we obtain

$$c(0,n) = \min_{1\leq k\leq n} \{c(0, k-1) + c(k, n) + p(k) + w(0, k-1) + w(k, n)\} \tag{5.11}$$

We can generalize (5.11) to obtain for any $c(i,j)$

$$c(i,j) = \min_{i<k\leq j} \{c(i, k-1) + c(k, j) + p(k) + w(i, k-1) + w(k, j)\}$$

$$c(i,j) = \min_{i<k\leq j}\{c(i,k-1)+c(k,j)\}+w(i,j) \tag{5.12}$$

Equation 5.12 can be solved for $c(0,n)$ by first computing all $c(i,j)$ such that $j-i=1$ (note $c(i,i)=0$ and $w(i,i)=q(i)$, $0\leq i\leq n$). Next we can compute all $c(i,j)$ such that $j-i=2$, then all $c(i,j)$ with $j-i=3$, and so on. If during this computation we record the root $r(i,j)$ of each tree t_{ij}, then an optimal binary search tree can be constructed from these $r(i,j)$. Note that $r(i,j)$ is the value of k that minimizes (5.12).

Example 5.18 Let $n=4$ and $(a_1,a_2,a_3,a_4)=$ (**do, if, int, while**). Let $p(1:4)=(3,3,1,1)$ and $q(0:4)=(2,3,1,1,1)$. The p's and q's have been multiplied by 16 for convenience. Initially, we have $w(i,i)=q(i), c(i,i)=0$ and $r(i,i)=0$, $0\leq i\leq 4$. Using Equation 5.12 and the observation $w(i,j)=p(j)+q(j)+w(i,j-1)$, we get

$$\begin{aligned}
w(0,1) &= p(1)+q(1)+w(0,0)=8\\
c(0,1) &= w(0,1)+\min\{c(0,0)+c(1,1)\} = 8\\
r(0,1) &= 1\\
w(1,2) &= p(2)+q(2)+w(1,1) = 7\\
c(1,2) &= w(1,2)+\min\{c(1,1)+c(2,2)\} = 7\\
r(0,2) &= 2\\
w(2,3) &= p(3)+q(3)+w(2,2) = 3\\
c(2,3) &= w(2,3)+\min\{c(2,2)+c(3,3)\} = 3\\
r(2,3) &= 3\\
w(3,4) &= p(4)+q(4)+w(3,3) = 3\\
c(3,4) &= w(3,4)+\min\{c(3,3)+c(4,4)\} = 3\\
r(3,4) &= 4
\end{aligned}$$

Knowing $w(i,i+1)$ and $c(i,i+1)$, $0\leq i<4$, we can again use Equation 5.12 to compute $w(i,i+2)$, $c(i,i+2)$, and $r(i,i+2)$, $0\leq i<3$. This process can be repeated until $w(0,4)$, $c(0,4)$, and $r(0,4)$ are obtained. The table of Figure 5.16 shows the results of this computation. The box in row i and column j shows the values of $w(j,j+i)$, $c(j,j+i)$ and $r(j,j+i)$ respectively. The computation is carried out by row from row 0 to row 4. From the table we see that $c(0,4)=32$ is the minimum cost of a binary search tree for (a_1,a_2,a_3,a_4). The root of tree t_{04} is a_2. Hence, the left subtree is t_{01} and the right subtree t_{24}. Tree t_{01} has root a_1 and subtrees t_{00} and t_{11}. Tree t_{24} has root a_3; its left subtree is t_{22} and its right subtree t_{34}. Thus, with the data in the table it is possible to reconstruct t_{04}. Figure 5.17 shows t_{04}. □

	0	1	2	3	4
0	$w_{00} = 2$ $c_{00} = 0$ $r_{00} = 0$	$w_{11} = 3$ $c_{11} = 0$ $r_{11} = 0$	$w_{22} = 1$ $c_{22} = 0$ $r_{22} = 0$	$w_{33} = 1$ $c_{33} = 0$ $r_{33} = 0$	$w_{44} = 1$ $c_{44} = 0$ $r_{44} = 0$
1	$w_{01} = 8$ $c_{01} = 8$ $r_{01} = 1$	$w_{12} = 7$ $c_{12} = 7$ $r_{12} = 2$	$w_{23} = 3$ $c_{23} = 3$ $r_{23} = 3$	$w_{34} = 3$ $c_{34} = 3$ $r_{34} = 4$	
2	$w_{02} = 12$ $c_{02} = 19$ $r_{02} = 1$	$w_{13} = 9$ $c_{13} = 12$ $r_{13} = 2$	$w_{24} = 5$ $c_{24} = 8$ $r_{24} = 3$		
3	$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 2$	$w_{14} = 11$ $c_{14} = 19$ $r_{14} = 2$			
4	$w_{04} = 16$ $c_{04} = 32$ $r_{04} = 2$				

Figure 5.16 Computation of $c(0,4)$, $w(0,4)$, and $r(0,4)$

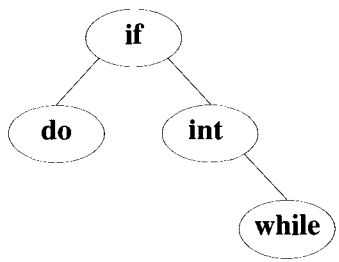


Figure 5.17 Optimal search tree for Example 5.18

The above example illustrates how Equation 5.12 can be used to determine the c's and r's and also how to reconstruct t_{0n} knowing the r's. Let us examine the complexity of this procedure to evaluate the c's and r's. The evaluation procedure described in the above example requires us to compute $c(i,j)$ for $(j-i) = 1, 2, \ldots, n$ in that order. When $j - i = m$, there are $n - m + 1$ $c(i,j)$'s to compute. The computation of each of these $c(i,j)$'s requires us to find the minimum of m quantities (see Equation 5.12). Hence, each such $c(i,j)$ can be computed in time $O(m)$. The total time for all $c(i,j)$'s with $j - i = m$ is therefore $O(nm - m^2)$. The total time to evaluate all the $c(i,j)$'s and $r(i,j)$'s is therefore

$$\sum_{1 \le m \le n} (nm - m^2) = O(n^3)$$

We can do better than this using a result due to D. E. Knuth which shows that the optimal k in Equation 5.12 can be found by limiting the search to the range $r(i, j-1) \le k \le r(i+1, j)$. In this case the computing time becomes $O(n^2)$ (see the exercises). The function OBST (Algorithm 5.5) uses this result to obtain the values of $w(i,j)$, $r(i,j)$, and $c(i,j)$, $0 \le i \le j \le n$, in $O(n^2)$ time. The tree t_{0n} can be constructed from the values of $r(i,j)$ in $O(n)$ time. The algorithm for this is left as an exercise.

EXERCISES

1. Use function OBST (Algorithm 5.5) to compute $w(i,j)$, $r(i,j)$, and $c(i,j)$, $0 \le i < j \le 4$, for the identifier set $(a_1, a_2, a_3, a_4) =$ (**cout, float, if, while**) with $p(1) = 1/20$, $p(2) = 1/5$, $p(3) = 1/10$, $p(4) = 1/20$, $q(0) = 1/5$, $q(1) = 1/10$, $q(2) = 1/5$, $q(3) = 1/20$, and $q(4) = 1/20$. Using the $r(i,j)$'s, construct the optimal binary search tree.

2. (a) Show that the computing time of function OBST (Algorithm 5.5) is $O(n^2)$.

   (b) Write an algorithm to construct the optimal binary search tree given the roots $r(i,j), 0 \le i < j \le n$. Show that this can be done in time $O(n)$.

3. Since often only the approximate values of the p's and q's are known, it is perhaps just as meaningful to find a binary search tree that is nearly optimal. That is, its cost, Equation 5.9, is almost minimal for the given p's and q's. This exercise explores an $O(n \log n)$ algorithm that results in nearly optimal binary search trees. The search tree heuristic we use is

```
1   Algorithm OBST(p, q, n)
2   // Given n distinct identifiers a_1 < a_2 < ··· < a_n and probabilities
3   // p[i], 1 ≤ i ≤ n, and q[i], 0 ≤ i ≤ n, this algorithm computes
4   // the cost c[i, j] of optimal binary search trees t_ij for identifiers
5   // a_{i+1}, ..., a_j. It also computes r[i, j], the root of t_ij.
6   // w[i, j] is the weight of t_ij.
7   {
8       for i := 0 to n − 1 do
9       {
10          // Initialize.
11          w[i, i] := q[i]; r[i, i] := 0; c[i, i] := 0.0;
12          // Optimal trees with one node
13          w[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
14          r[i, i + 1] := i + 1;
15          c[i, i + 1] := q[i] + q[i + 1] + p[i + 1];
16      }
17      w[n, n] := q[n]; r[n, n] := 0; c[n, n] := 0.0;
18      for m := 2 to n do  // Find optimal trees with m nodes.
19          for i := 0 to n − m do
20          {
21              j := i + m;
22              w[i, j] := w[i, j − 1] + p[j] + q[j];
23              // Solve 5.12 using Knuth's result.
24              k := Find(c, r, i, j);
25                  // A value of l in the range r[i, j − 1] ≤ l
26                  // ≤ r[i + 1, j] that minimizes c[i, l − 1] + c[l, j];
27              c[i, j] := w[i, j] + c[i, k − 1] + c[k, j];
28              r[i, j] := k;
29          }
30      write (c[0, n], w[0, n], r[0, n]);
31  }
```

```
1   Algorithm Find(c, r, i, j)
2   {
3       min := ∞;
4       for m := r[i, j − 1] to r[i + 1, j] do
5           if (c[i, m − 1] + c[m, j]) < min then
6           {
7               min := c[i, m − 1] + c[m, j]; l := m;
8           }
9       return l;
10  }
```

Algorithm 5.5 Finding a minimum-cost binary search tree

Choose the root k such that $|w(0, k-1) - w(k, n)|$ is as small as possible. Repeat this procedure to find the left and right subtrees of the root.

(a) Using this heuristic, obtain the resulting binary search tree for the data of Exercise 1. What is its cost?

(b) Write an algorithm implementing the above heuristic. Your algorithm should have time complexity $O(n \log n)$.

5.6 STRING EDITING

We are given two strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$, where x_i, $1 \leq i \leq n$, and y_j, $1 \leq j \leq m$, are members of a finite set of symbols known as the *alphabet*. We want to transform X into Y using a sequence of *edit operations* on X. The permissible edit operations are insert, delete, and change (a symbol of X into another), and there is a cost associated with performing each. The cost of a sequence of operations is the sum of the costs of the individual operations in the sequence. The problem of string editing is to identify a minimum-cost sequence of edit operations that will transform X into Y.

Let $D(x_i)$ be the cost of deleting the symbol x_i from X, $I(y_j)$ be the cost of inserting the symbol y_j into X, and $C(x_i, y_j)$ be the cost of changing the symbol x_i of X into y_j.

Example 5.19 Consider the sequences $X = x_1, x_2, x_3, x_4, x_5 = a, a, b, a, b$ and $Y = y_1, y_2, y_3, y_4 = b, a, b, b$. Let the cost associated with each insertion and deletion be 1 (for any symbol). Also let the cost of changing any symbol to any other symbol be 2. One possible way of transforming X into Y is delete each $x_i, 1 \leq i \leq 5$, and insert each $y_j, 1 \leq j \leq 4$. The total cost of this edit sequence is 9. Another possible edit sequence is delete x_1 and x_2 and insert y_4 at the end of string X. The total cost is only 3. □

A solution to the string editing problem consists of a sequence of decisions, one for each edit operation. Let $\mathcal{E}$ be a minimum-cost edit sequence for transforming X into Y. The first operation, O, in $\mathcal{E}$ is delete, insert, or change. If $\mathcal{E}' = \mathcal{E} - \{O\}$ and X' is the result of applying O on X, then $\mathcal{E}'$ should be a minimum-cost edit sequence that transforms X' into Y. Thus the principle of optimality holds for this problem. A dynamic programming solution for this problem can be obtained as follows. Define $cost(i, j)$ to be the minimum cost of any edit sequence for transforming $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $0 \leq i \leq n$ and $0 \leq j \leq m$). Compute $cost(i, j)$ for each i and j. Then $cost(n, m)$ is the cost of an optimal edit sequence.

For $i = j = 0$, $cost(i, j) = 0$, since the two sequences are identical (and empty). Also, if $j = 0$ and $i > 0$, we can transform X into Y by a sequence of

deletes. Thus, $cost(i,0) = cost(i-1,0)+D(x_i)$. Similarly, if $i = 0$ and $j > 0$, we get $cost(0,j) = cost(0,j-1) + I(y_j)$. If $i \neq 0$ and $j \neq 0$, $x_1, x_2, \ldots, x_i$ can be transformed into $y_1, y_2, \ldots, y_j$ in one of three ways:

1. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_j$ using a minimum-cost edit sequence and then delete x_i. The corresponding cost is $cost(i-1,j) + D(x_i)$.

2. Transform $x_1, x_2, \ldots, x_{i-1}$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then change the symbol x_i to y_j. The associated cost is $cost(i-1,j-1) + C(x_i, y_j)$.

3. Transform $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_{j-1}$ using a minimum-cost edit sequence and then insert y_j. This corresponds to a cost of $cost(i, j-1) + I(y_j)$.

The minimum cost of any edit sequence that transforms $x_1, x_2, \ldots, x_i$ into $y_1, y_2, \ldots, y_j$ (for $i > 0$ and $j > 0$) is the minimum of the above three costs, according to the principle of optimality. Therefore, we arrive at the following recurrence equation for $cost(i,j)$:

$$cost(i,j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1,0) + D(x_i) & j = 0,\ i > 0 \\ cost(0,j-1) + I(y_j) & i = 0,\ j > 0 \\ cost'(i,j) & i > 0,\ j > 0 \end{cases} \tag{5.13}$$

$$\text{where } cost'(i,j) = \min \{ \begin{array}{l} cost(i-1,j) + D(x_i), \\ cost(i-1,j-1) + C(x_i, y_j), \\ cost(i,j-1) + I(y_j) \ \} \end{array}$$

We have to compute $cost(i,j)$ for all possibles values of i and j ($0 \leq i \leq n$ and $0 \leq j \leq m$). There are $(n+1)(m+1)$ such values. These values can be computed in the form of a table, M, where each row of M corresponds to a particular value of i and each column of M corresponds to a specific value of j. $M(i,j)$ stores the value $cost(i,j)$. The zeroth row can be computed first since it corresponds to performing a series of insertions. Likewise the zeroth column can also be computed. After this, one could compute the entries of M in row-major order, starting from the first row. Rows should be processed in the order $1, 2, \ldots, n$. Entries in any row are computed in increasing order of column number.

The entries of M can also be computed in column-major order, starting from the first column. Looking at Equation 5.13, we see that each entry of M takes only $O(1)$ time to compute. Therefore the whole algorithm takes $O(mn)$ time. The value $cost(n,m)$ is the final answer we are interested in. Having computed all the entries of M, a minimum edit sequence can be

obtained by a simple backward trace from $cost(n, m)$. This backward trace is enabled by recording which of the three options for $i > 0, j > 0$ yielded the minimum cost for each i and j.

Example 5.20 Consider the string editing problem of Example 5.19. $X = a, a, b, a, b$ and $Y = b, a, b, b$. Each insertion and deletion has a unit cost and a change costs 2 units. For the cases $i = 0, j > 1$, and $j = 0, i > 1$, $cost(i, j)$ can be computed first (Figure 5.18). Let us compute the rest of the entries in row-major order. The next entry to be computed is $cost(1, 1)$.

$$\begin{aligned} cost(1,1) &= \min \{cost(0,1) + D(x_1), cost(0,0) + C(x_1, y_1), cost(1,0) + I(y_1)\} \\ &= \min \{2, 2, 2\} = 2 \end{aligned}$$

Next is computed $cost(1, 2)$.

$$\begin{aligned} cost(1,2) &= \min \{cost(0,2) + D(x_1), cost(0,1) + C(x_1, y_2), cost(1,1) + I(y_2)\} \\ &= \min \{3, 1, 3\} = 1 \end{aligned}$$

The rest of the entries are computed similarly. Figure 5.18 displays the whole table. The value $cost(5, 4) = 3$. One possible minimum-cost edit sequence is delete x_1, delete x_2, and insert y_4. Another possible minimum cost edit sequence is change x_1 to y_2 and delete x_4. □

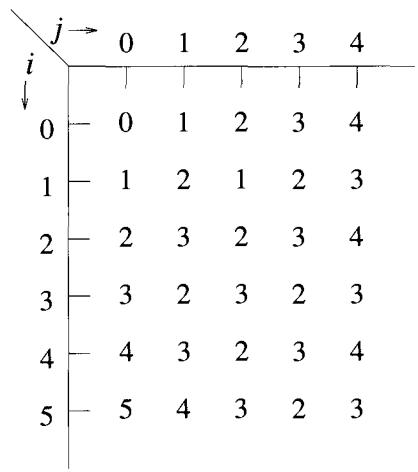


Figure 5.18 Cost table for Example 5.20

EXERCISES

1. Let $X = a, a, b, a, a, b, a, b, a, a$ and $Y = b, a, b, a, a, b, a, b$. Find a minimum-cost edit sequence that transforms X into Y.
2. Present a pseudocode algorithm that implements the string editing algorithm discussed in this section. Program it and test its correctness using suitable data.
3. Modify the above program not only to compute $cost(n, m)$ but also to output a minimum-cost edit sequence. What is the time complexity of your program?
4. Given a sequence X of symbols, a subsequence of X is defined to be any contiguous portion of X. For example, if $X = x_1, x_2, x_3, x_4, x_5, x_2, x_3$ and x_1, x_2, x_3 are subsequences of X. Given two sequences X and Y, present an algorithm that will identify the longest subsequence that is common to both X and Y. This problem is known as *the longest common subsequence problem.* What is the time complexity of your algorithm?

5.7 0/1 KNAPSACK

The terminology and notation used in this section is the same as that in Section 5.1. A solution to the knapsack problem can be obtained by making a sequence of decisions on the variables $x_1, x_2, \ldots, x_n$. A decision on variable x_i involves determining which of the values 0 or 1 is to be assigned to it. Let us assume that decisions on the x_i are made in the order $x_n, x_{n-1}, \ldots, x_1$. Following a decision on x_n, we may be in one of two possible states: the capacity remaining in the knapsack is m and no profit has accrued or the capacity remaining is $m - w_n$ and a profit of p_n has accrued. It is clear that the remaining decisions $x_{n-1}, \ldots, x_1$ must be optimal with respect to the problem state resulting from the decision on x_n. Otherwise, $x_n, \ldots, x_1$ will not be optimal. Hence, the principle of optimality holds.

Let $f_j(y)$ be the value of an optimal solution to KNAP$(1, j, y)$. Since the principle of optimality holds, we obtain

$$f_n(m) = \max \{f_{n-1}(m), f_{n-1}(m - w_n) + p_n\} \tag{5.14}$$

For arbitrary $f_i(y)$, $i > 0$, Equation 5.14 generalizes to

$$f_i(y) = \max \{f_{i-1}(y), f_{i-1}(y - w_i) + p_i\} \tag{5.15}$$

Equation 5.15 can be solved for $f_n(m)$ by beginning with the knowledge $f_0(y)$ $= 0$ for all y and $f_i(y) = -\infty, y < 0$. Then $f_1, f_2, \ldots, f_n$ can be successively computed using (5.15).

When the w_i's are integer, we need to compute $f_i(y)$ for integer y, $0 \leq y \leq m$. Since $f_i(y) = -\infty$ for $y < 0$, these function values need not be computed explicitly. Since each f_i can be computed from f_{i-1} in $\Theta(m)$ time, it takes $\Theta(mn)$ time to compute f_n. When the w_i's are real numbers, $f_i(y)$ is needed for real numbers y such that $0 \leq y \leq m$. So, f_i cannot be explicitly computed for all y in this range. Even when the w_i's are integer, the explicit $\Theta(mn)$ computation of f_n may not be the most efficient computation. So, we explore an alternative method for both cases.

Notice that $f_i(y)$ is an ascending step function; i.e., there are a finite number of y's, $0 = y_1 < y_2 < \cdots < y_k$, such that $f_i(y_1) < f_i(y_2) < \cdots < f_i(y_k)$; $f_i(y) = -\infty$, $y < y_1$; $f_i(y) = f(y_k)$, $y \geq y_k$; and $f_i(y) = f_i(y_j)$, $y_j \leq y < y_{j+1}$. So, we need to compute only $f_i(y_j)$, $1 \leq j \leq k$. We use the ordered set $S^i = \{(f(y_j), y_j) | 1 \leq j \leq k\}$ to represent $f_i(y)$. Each member of S^i is a pair (P, W), where $P = f_i(y_j)$ and $W = y_j$. Notice that $S^0 = \{(0,0)\}$. We can compute S^{i+1} from S^i by first computing

$$S_1^i = \{(P, W) | (P - p_i, W - w_i) \in S^i\} \tag{5.16}$$

Now, S^{i+1} can be computed by merging the pairs in S^i and S_1^i together. Note that if S^{i+1} contains two pairs (P_j, W_j) and (P_k, W_k) with the property that $P_j \leq P_k$ and $W_j \geq W_k$, then the pair (P_j, W_j) can be discarded because of (5.15). Discarding or purging rules such as this one are also known as *dominance rules.* Dominated tuples get purged. In the above, (P_k, W_k) dominates (P_j, W_j).

Interestingly, the strategy we have come up with can also be derived by attempting to solve the knapsack problem via a systematic examination of the up to 2^n possibilities for $x_1, x_2, \ldots, x_n$. Let S^i represent the possible states resulting from the 2^i decision sequences for $x_1, \ldots, x_i$. A state refers to a pair (P_j, W_j), W_j being the total weight of objects included in the knapsack and P_j being the corresponding profit. To obtain S^{i+1}, we note that the possibilities for x_{i+1} are $x_{i+1} = 0$ or $x_{i+1} = 1$. When $x_{i+1} = 0$, the resulting states are the same as for S^i. When $x_{i+1} = 1$, the resulting states are obtained by adding (p_{i+1}, w_{i+1}) to each state in S^i. Call the set of these additional states S_1^i. The S_1^i is the same as in Equation 5.16. Now, S^{i+1} can be computed by merging the states in S^i and S_1^i together.

Example 5.21 Consider the knapsack instance $n = 3$, $(w_1, w_2, w_3) = (2, 3, 4)$, $(p_1, p_2, p_3) = (1, 2, 5)$, and $m = 6$. For these data we have

$$\begin{aligned}
S^0 &= \{(0,0)\}; S_1^0 = \{(1,2)\} \\
S^1 &= \{(0,0), (1,2)\}; S_1^1 = \{(2,3), (3,5)\} \\
S^2 &= \{(0,0), (1,2), (2,3), (3,5)\}; S_1^2 = \{(5,4), (6,6), (7,7), (8,9)\} \\
S^3 &= \{(0,0), (1,2), (2,3), (5,4), (6,6), (7,7), (8,9)\}
\end{aligned}$$

Note that the pair (3, 5) has been eliminated from S^3 as a result of the purging rule stated above. □

When generating the S^i's, we can also purge all pairs (P, W) with $W > m$ as these pairs determine the value of $f_n(x)$ only for $x > m$. Since the knapsack capacity is m, we are not interested in the behavior of f_n for $x > m$. When all pairs (P_j, W_j) with $W_j > m$ are purged from the S^i's, $f_n(m)$ is given by the P value of the last pair in S^n (note that the S^i's are ordered sets). Note also that by computing S^n, we can find the solutions to all the knapsack problems KNAP$(1, n, x)$, $0 \leq x \leq m$, and not just KNAP$(1, n, m)$. Since, we want only a solution to KNAP$(1, n, m)$, we can dispense with the computation of S^n. The last pair in S^n is either the last one in S^{n-1} or it is $(P_j + p_n, W_j + w_n)$, where $(P_j, W_j) \in S^{n-1}$ such that $W_j + w_n \leq m$ and W_j is maximum.

If $(P1, W1)$ is the last tuple in S^n, a set of 0/1 values for the x_i's such that $\sum p_i x_i = P1$ and $\sum w_i x_i = W1$ can be determined by carrying out a search through the S^is. We can set $x_n = 0$ if $(P1, W1) \in S^{n-1}$. If $(P1, W1) \notin S^{n-1}$, then $(P1 - p_n, W1 - w_n) \in S^{n-1}$ and we can set $x_n = 1$. This leaves us to determine how either $(P1, W1)$ or $(P1 - p_n, W1 - w_n)$ was obtained in S^{n-1}. This can be done recursively.

Example 5.22 With $m = 6$, the value of $f_3(6)$ is given by the tuple (6, 6) in S^3 (Example 5.21). The tuple (6, 6) $\notin S^2$, and so we must set $x_3 = 1$. The pair (6, 6) came from the pair $(6 - p_3, 6 - w_3) = (1, 2)$. Hence (1, 2) $\in S^2$. Since $(1, 2) \in S^1$, we can set $x_2 = 0$. Since (1, 2) $\notin S^0$, we obtain $x_1 = 1$. Hence an optimal solution is $(x_1, x_2, x_3) = (1, 0, 1)$. □

We can sum up all we have said so far in the form of an informal algorithm DKP (Algorithm 5.6). To evaluate the complexity of the algorithm, we need to specify how the sets S^i and S_1^i are to be represented; provide an algorithm to merge S^i and S_1^i; and specify an algorithm that will trace through $S^{n-1}, \ldots, S^1$ and determine a set of 0/1 values for $x_n, \ldots, x_1$.

We can use an array $pair[\,]$ to represent all the pairs (P, W). The P values are stored in $pair[\,].p$ and the W values in $pair[\,].w$. Sets $S^0, S^1, \ldots, S^{n-1}$ can be stored adjacent to each other. This requires the use of pointers $b[i]$, $0 \leq i \leq n$, where $b[i]$ is the location of the first element in S^i, $0 \leq i < n$, and $b[n]$ is one more than the location of the last element in S^{n-1}.

Example 5.23 Using the representation above, the sets S^0, S^1, and S^2 of Example 5.21 appear as

```
1  Algorithm DKP(p, w, n, m)
2  {
3      S^0 := {(0,0)};
4      for i := 1 to n − 1 do
5      {
6          S_1^{i−1} := {(P,W)|(P − p_i, W − w_i) ∈ S^{i−1} and W ≤ m};
7          S^i := MergePurge(S^{i−1}, S_1^{i−1});
8      }
9      (PX, WX) :=last pair in S^{n−1};
10     (PY, WY) := (P' + p_n, W' + w_n) where W' is the largest W in
11         any pair in S^{n−1} such that W + w_n ≤ m;
12     // Trace back for x_n, x_{n−1}, ..., x_1.
13     if (PX > PY) then x_n := 0;
14     else x_n := 1;
15     TraceBackFor(x_{n−1}, ..., x_1);
16 }
```

Algorithm 5.6 Informal knapsack algorithm

	1	2	3	4	5	6	7
$pair[\,].p$	0	0	1	0	1	2	3
$pair[\,].w$	0	0	2	0	2	3	5
	↑ $b[0]$	↑ $b[1]$		↑ $b[2]$			↑ $b[3]$

□

The merging and purging of S^{i-1} and S_1^{i-1} can be carried out at the same time that S_1^{i-1} is generated. Since the pairs in S^{i-1} are in increasing order of P and W, the pairs for S^i are generated in this order. If the next pair generated for S_1^{i-1} is (PQ, WQ), then we can merge into S^i all pairs from S^{i-1} with W value $\leq WQ$. The purging rule can be used to decide whether any pairs get purged. Hence, no additional space is needed in which to store S_1^{i-1}.

DKnap (Algorithm 5.7) generates S^i from S^{i-1} in this way. The S^i's are generated in the **for** loop of lines 7 to 42 of Algorithm 5.7. At the start of each iteration $t = b[i-1]$ and h is the index of the last pair in S^{i-1}. The variable k points to the next tuple in S^{i-1} that has to be merged into S^i. In line 10, the function Largest determines the largest q, $t \leq q \leq h$,

for which $pair[q].w + w[i] \leq m$. This can be done by performing a binary search. The code for this function is left as an exercise. Since u is set such that for all $W_j, h \geq j > u$, $W_j + w_i > m$, the pairs for S_1^{i-1} are $(P(j) + p_i, W(j) + w_i)$, $1 \leq j \leq u$. The **for** loop of lines 11 to 33 generates these pairs. Each time a pair (pp, ww) is generated, all pairs (P, W) in S^{i-1} with $W < ww$ not yet purged or merged into S^i are merged into S^i. Note that none of these may be purged. Lines 21 to 25 handle the case when the next pair in S^{i-1} has a W value equal to ww. In this case the pair with lesser P value gets purged. In case $pp > P(next-1)$, then the pair (pp, ww) gets purged. Otherwise, (pp, ww) is added to S^i. The **while** loop of lines 31 and 32 purges all unmerged pairs in S^{i-1} that can be purged at this time. Finally, following the merging of S_1^{i-1} into S^i, there may be pairs remaining in S^{i-1} to be merged into S^i. This is taken care of in the **while** loop of lines 35 to 39. Note that because of lines 31 and 32, none of these pairs can be purged. Function TraceBack (line 43) implements the **if** statement and trace-back step of the function DKP (Algorithm 5.6). This is left as an exercise.

If $|S^i|$ is the number of pairs in S^i, then the array $pair$ should have a minimum dimension of $d = \sum_{0 \leq i \leq n-1} |S^i|$. Since it is not possible to predict the exact space needed, it is necessary to test for $next > d$ each time $next$ is incremented. Since each S^i, $i > 0$, is obtained by merging S^{i-1} and S_1^{i-1} and $|S_1^{i-1}| \leq |S^{i-1}|$, it follows that $|S^i| \leq 2|S^{i-1}|$. In the worst case no pairs will get purged and

$$\sum_{0 \leq i \leq n-1} |S^i| = \sum_{0 \leq i \leq n-1} 2^i = 2^n - 1$$

The time needed to generate S^i from S^{i-1} is $\Theta(|S^{i-1}|)$. Hence, the time needed to compute all the S^i's, $0 \leq i < n$, is $\Theta(\sum |S^{i-1}|)$. Since $|S^i| \leq 2^i$, the time needed to compute all the S^i's is $O(2^n)$. If the p_j's are integers, then each pair (P, W) in S^i has an integer P and $P \leq \sum_{1 \leq j \leq i} p_j$. Similarly, if the w_j's are integers, each W is an integer and $W \leq m$. In any S^i the pairs have distinct W values and also distinct P values. Hence,

$$|S^i| \leq 1 + \sum_{1 \leq j \leq i} p_j$$

when the p_j's are integers and

$$|S^i| \leq 1 + \min \{ \sum_{1 \leq j \leq i} w_j, m \}$$

```
PW = record {float p; float w; }

1   Algorithm DKnap(p, w, x, n, m)
2   {
3       // pair[ ] is an array of PW's.
4       b[0] := 1; pair[1].p := pair[1].w := 0.0; // S^0
5       t := 1; h := 1; // Start and end of S^0
6       b[1] := next := 2; // Next free spot in pair[ ]
7       for i := 1 to n - 1 do
8       { // Generate S^i.
9           k := t;
10          u := Largest(pair, w, t, h, i, m);
11          for j := t to u do
12          { // Generate S_1^{i-1} and merge.
13              pp := pair[j].p + p[i]; ww := pair[j].w + w[i];
14                  // (pp, ww) is the next element in S_1^{i-1}.
15              while ((k ≤ h) and (pair[k].w ≤ ww)) do
16              {
17                  pair[next].p := pair[k].p;
18                  pair[next].w := pair[k].w;
19                  next := next + 1; k := k + 1;
20              }
21              if ((k ≤ h) and (pair[k].w = ww)) then
22              {
23                  if pp < pair[k].p then pp := pair[k].p;
24                  k := k + 1;
25              }
26              if pp > pair[next - 1].p then
27              {
28                  pair[next].p := pp; pair[next].w := ww;
29                  next := next + 1 ;
30              }
31              while ((k ≤ h) and (pair[k].p ≤ pair[next - 1].p))
32                  do k := k + 1;
33          }
34          // Merge in remaining terms from S^{i-1}.
35          while (k ≤ h) do
36          {
37              pair[next].p := pair[k].p; pair[next].w := pair[k].w;
38              next := next + 1; k := k + 1;
39          }
40          // Initialize for S^{i+1}.
41          t := h + 1; h := next - 1; b[i + 1] := next;
42      }
43      TraceBack(p, w, pair, x, m, n);
44  }
```

Algorithm 5.7 Algorithm for 0/1 knapsack problem

when the w_j's are integers. When both the p_j's and w_j's are integers, the time and space complexity of DKnap (excluding the time for TraceBack) is $O(\min\{2^n, n\sum_{1\le i\le n} p_i, nm\})$. In this bound $\sum_{1\le i\le n} p_i$ can be replaced by $\sum_{1\le i\le n} p_i/\gcd(p_1,\ldots,p_n)$ and m by $\gcd(w_1, w_2, \ldots, w_n, m)$ (see the exercises). The exercises indicate how TraceBack may be implemented so as to have a space complexity $O(1)$ and a time complexity $O(n^2)$.

Although the above analysis may seem to indicate that DKnap requires too much computational resource to be practical for large n, in practice many instances of this problem can be solved in a reasonable amount of time. This happens because usually, all the p's and w's are integers and m is much smaller than 2^n. The purging rule is effective in purging most of the pairs that would otherwise remain in the S^i's.

Algorithm DKnap can be speeded up by the use of heuristics. Let L be an estimate on the value of an optimal solution such that $f_n(m) \ge L$. Let PLEFT$(i) = \sum_{i<j\le n} p_j$. If S^i contains a tuple (P, W) such that $P +$ PLEFT$(i) < L$, then (P, W) can be purged from S^i. To see this, observe that (P, W) can contribute at best the pair $(P + \sum_{i<j\le n} p_j, W + \sum_{i<j\le n} w)$ to S_1^{n-1}. Since $P + \sum_{i<j\le n} p_j = P +$ PLEFT$(i) < L$, it follows that this pair cannot lead to a pair with value at least L and so cannot determine an optimal solution. A simple way to estimate L such that $L \le f_n(m)$ is to consider the last pair (P, W) in S^i. Then, $P \le f_n(m)$. A better estimate is obtained by adding some of the remaining objects to (P, W). Example 5.24 illustrates this. Heuristics for the knapsack problem are discussed in greater detail in the chapter on branch-and-bound. The exercises explore a divide-and-conquer approach to speed up DKnap so that the worst case time is $O(2^{n/2})$.

Example 5.24 Consider the following instance of the knapsack problem: $n = 6, (p_1, p_2, p_3, p_4, p_5, p_6) = (w_1, w_2, w_3, w_4, w_5, w_6) =$ (100, 50, 20, 10, 7, 3), and $m = 165$. Attempting to fill the knapsack using objects in the order 1, 2, 3, 4, 5, and 6, we see that objects 1, 2, 4, and 6 fit in and yield a profit of 163 and a capacity utilization of 163. We can thus begin with $L = 163$ as a value with the property $L \le f_n(m)$. Since $p_i = w_i$, every pair $(P, W) \in S^i$, $0 \le i \le 6$ has $P = W$. Hence, each pair can be replaced by the singleton P or W. PLEFT(0) = 190, PLEFT(1) = 90, PLEFT(2) = 40, PLEFT(3) = 20, PLEFT(4) = 10, PLEFT(5) = 3, and PLEFT(6) = 0. Eliminating from each S^i any singleton P such that $P+$ PLEFT$(i) < L$, we obtain

$$S^0 = \{0\};\quad S_1^0 = \{100\}$$
$$S^1 = \{100\};\quad S_1^1 = \{150\}$$
$$S^2 = \{150\};\quad S_1^2 = \phi$$

$$S^3 = \{150\}; \quad S_1^3 = \{160\}$$
$$S^4 = \{160\}; \quad S_1^4 = \phi$$
$$S^5 = \{160\}$$

The singleton 0 is deleted from S^1 as 0 + PLEFT(1)< 163. The set S_1^2 does not contain the singleton 150 + 20 = 170 as $m < 170$. S^3 does not contain the 100 or the 120 as each is less than L − PLEFT(3). And so on. The value $f_6(165)$ can be determined from S^5. In this example, the value of L did not change. In general, L will change if a better estimate is obtained as a result of the computation of some S^i. If the heuristic wasn't used, then the computation would have proceeded as

$$\begin{aligned}
S^0 &= \{0\} \\
S^1 &= \{0, 100\} \\
S^2 &= \{0, 50, 100, 150\} \\
S^3 &= \{0, 20, 50, 70, 100, 120, 150\} \\
S^4 &= \{0, 10, 20, 30, 50, 60, 70, 80, 100, 110, 120, 130, 150, 160\} \\
S^5 &= \{0, 7, 10, 17, 20, 27, 30, 37, 50, 57, 60, 67, 70, 77, 80, 87, 100, \\
&\quad 107, 110, 117, 120, 127, 130, 137, 150, 157, 160\}
\end{aligned}$$

The value $f_6(165)$ can now be determined from S^5, using the knowledge $(p_6, w_6) = (3, 3)$. □

EXERCISES

1. Generate the sets S^i, $0 \leq i \leq 4$ (Equation 5.16), when $(w_1, w_2, w_3, w_4) = (10, 15, 6, 9)$ and $(p_1, p_2, p_3, p_4) = (2, 5, 8, 1)$.

2. Write a function Largest$(pair, w, t, h, i, m)$ that uses binary search to determine the largest q, $t \leq q \leq h$, such that $pair[q].w + w[i] \leq m$.

3. Write a function TraceBack to determine an optimal solution $x_1, x_2, \ldots, x_n$ to the knapsack problem. Assume that $S^i, 0 \leq i < n$, have already been computed as in function DKnap. Knowing $b(i)$ and $b(i+1)$, you can use a binary search to determine whether $(P', W') \in S^i$. Hence, the time complexity of your algorithm should be no more than $O(n \max_i\{\log |S^i|\}) = O(n^2)$.

4. Give an example of a set of knapsack instances for which $|S^i| = 2^i$, $0 \leq i \leq n$. Your set should include one instance for each n.

5. (a) Show that if the p_j's are integers, then the size of each S^i, $|S^i|$, in the knapsack problem is no more than $1+\sum_{1\leq i\leq j} p_j/gcd(p_1,p_2,\ldots,p_n)$, where $gcd(p_1,p_2,\ldots,p_n)$ is the greatest common divisor of the p_i's.

   (b) Show that when the w_j's are integer, then $|S^i| \leq 1+\min\{\sum_{1\leq j\leq i} w_j, m\}/gcd(w_1,w_2,\ldots,w_n,m)$.

6. (a) Using a divide-and-conquer approach coupled with the set generation approach of the text, show how to obtain an $O(2^{n/2})$ algorithm for the 0/1 knapsack problem.

   (b) Develop an algorithm that uses this approach to solve the 0/1 knapsack problem.

   (c) Compare the run time and storage requirements of this approach with those of Algorithm 5.7. Use suitable test data.

7. Consider the integer knapsack problem obtained by replacing the 0/1 constraint in (5.2) by $x_i \geq 0$ and integer. Generalize $f_i(x)$ to this problem in the obvious way.

   (a) Obtain the dynamic programming recurrence relation corresponding to (5.15).

   (b) Show how to transform this problem into a 0/1 knapsack problem. (*Hint*: Introduce new 0/1 variables for each x_i. If $0 \leq x_i < 2^j$, then introduce j variables, one for each bit in the binary representation of x_i.)

5.8 RELIABILITY DESIGN

In this section we look at an example of how to use dynamic programming to solve a problem with a multiplicative optimization function. The problem is to design a system that is composed of several devices connected in series (Figure 5.19). Let r_i be the reliability of device D_i (that is, r_i is the probability that device i will function properly). Then, the reliability of the entire system is Πr_i. Even if the individual devices are very reliable (the r_i's are very close to one), the reliability of the system may not be very good. For example, if $n = 10$ and $r_i = .99$, $1 \leq i \leq 10$, then $\Pi r_i = .904$. Hence, it is desirable to duplicate devices. Multiple copies of the same device type are connected in parallel (Figure 5.20) through the use of switching circuits. The switching circuits determine which devices in any given group are functioning properly. They then make use of one such device at each stage.

If stage i contains m_i copies of device D_i, then the probability that all m_i have a malfunction is $(1-r_i)^{m_i}$. Hence the reliability of stage i becomes

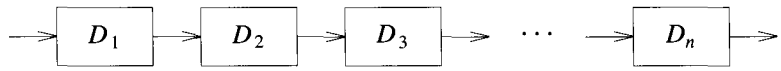


Figure 5.19 n devices D_i, $1 \leq i \leq n$, connected in series

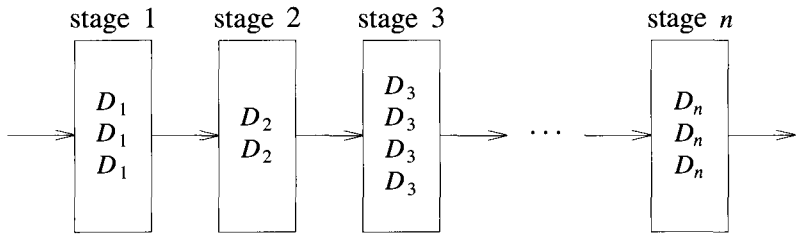


Figure 5.20 Multiple devices connected in parallel in each stage

$1-(1-r_i)^{m_i}$. Thus, if $r_i = .99$ and $m_i = 2$, the stage reliability becomes .9999. In any practical situation, the stage reliability is a little less than $1-(1-r_i)^{m_i}$ because the switching circuits themselves are not fully reliable. Also, failures of copies of the same device may not be fully independent (e.g., if failure is due to design defect). Let us assume that the reliability of stage i is given by a function $\phi_i(m_i)$, $1 \leq n$. (It is quite conceivable that $\phi_i(m_i)$ may decrease after a certain value of m_i.) The reliability of the system of stages is $\Pi_{1 \leq i \leq n} \phi_i(m_i)$.

Our problem is to use device duplication to maximize reliability. This maximization is to be carried out under a cost constraint. Let c_i be the cost of each unit of device i and let c be the maximum allowable cost of the system being designed. We wish to solve the following maximization problem:

$$\text{maximize } \Pi_{1 \leq i \leq n} \ \phi_i(m_i)$$

$$\text{subject to } \sum_{1 \leq i \leq n} c_i m_i \leq c \tag{5.17}$$

$$m_i \geq 1 \text{ and integer, } 1 \leq i \leq n$$

A dynamic programming solution can be obtained in a manner similar to that used for the knapsack problem. Since, we can assume each $c_i > 0$, each m_i must be in the range $1 \leq m_i \leq u_i$, where

$$u_i = \left\lfloor (c + c_i - \sum_{1}^{n} c_j)/c_i \right\rfloor$$

The upper bound u_i follows from the observation that $m_j \geq 1$. An optimal solution $m_1, m_2, \ldots, m_n$ is the result of a sequence of decisions, one decision for each m_i. Let $f_i(x)$ represent the maximum value of $\Pi_{1 \leq j \leq i} \phi(m_j)$ subject to the constraints $\sum_{1 \leq j \leq i} c_j m_j \leq x$ and $1 \leq m_j \leq u_j$, $1 \leq j \leq i$. Then, the value of an optimal solution is $f_n(c)$. The last decision made requires one to choose m_n from $\{1, 2, 3, \ldots, u_n\}$. Once a value for m_n has been chosen, the remaining decisions must be such as to use the remaining funds $c - c_n m_n$ in an optimal way. The principal of optimality holds and

$$f_n(c) = \max_{1 \leq m_n \leq u_n} \{\phi_n(m_n) f_{n-1}(c - c_n m_n)\} \tag{5.18}$$

For any $f_i(x)$, $i \geq 1$, this equation generalizes to

$$f_i(x) = \max_{1 \leq m_i \leq u_i} \{\phi_i(m_i) f_{i-1}(x - c_i m_i)\} \tag{5.19}$$

Clearly, $f_0(x) = 1$ for all x, $0 \leq x \leq c$. Hence, (5.19) can be solved using an approach similar to that used for the knapsack problem. Let S^i consist of tuples of the form (f, x), where $f = f_i(x)$. There is at most one tuple for each different x that results from a sequence of decisions on $m_1, m_2, \ldots, m_n$. The dominance rule (f_1, x_1) dominates (f_2, x_2) iff $f_1 \geq f_2$ and $x_1 \leq x_2$ holds for this problem too. Hence, dominated tuples can be discarded from S^i.

Example 5.25 We are to design a three stage system with device types D_1, D_2, and D_3. The costs are \$30, \$15, and \$20 respectively. The cost of the system is to be no more than \$105. The reliability of each device type is .9, .8 and .5 respectively. We assume that if stage i has m_i devices of type i in parallel, then $\phi_i(m_i) = 1 - (1 - r_i)^{m_i}$. In terms of the notation used earlier, $c_1 = 30$, $c_2 = 15$, $c_3 = 20$, $c = 105$, $r_1 = .9$, $r_2 = .8$, $r_3 = .5, u_1 = 2, u_2 = 3$, and $u_3 = 3$.

We use S^i to represent the set of all undominated tuples (f, x) that may result from the various decision sequences for $m_1, m_2, \ldots, m_i$. Hence, $f(x) = f_i(x)$. Beginning with $S^0 = \{(1, 0)\}$, we can obtain each S^i from S^{i-1} by trying out all possible values for m_i and combining the resulting tuples together. Using S_j^i to represent all tuples obtainable from S^{i-1} by choosing $m_i = j$, we obtain $S_1^1 = \{(.9,\ 30)\}$ and $S_2^1 = \{(.9,\ 30), (.99, 60)\}$. The set

$S_1^2 = \{(.72, 45), (.792, 75)\}$; $S_2^2 = \{(.864, 60)\}$. Note that the tuple (.9504, 90) which comes from (.99, 60) has been eliminated from S_2^2 as this leaves only \$10. This is not enough to allow $m_3 = 1$. The set $S_3^2 = \{(.8928, 75)\}$. Combining, we get $S^2 = \{(.72, 45), (.864, 60), (.8928, 75)\}$ as the tuple (.792, 75) is dominated by (.864, 60). The set $S_1^3 = \{(.36, 65), (.432, 80), (.4464, 95)\}$, $S_2^3 = \{(.54, 85), (.648, 100)\}$, and $S_3^3 = \{(.63, 105)\}$. Combining, we get $S^3 = \{(.36, 65), (.432, 80), (.54, 85), (.648, 100)\}$.

The best design has a reliability of .648 and a cost of 100. Tracing back through the S^i's, we determine that $m_1 = 1, m_2 = 2$, and $m_3 = 2$. □

As in the case of the knapsack problem, a complete dynamic programming algorithm for the reliability problem will use heuristics to reduce the size of the S^i's. There is no need to retain any tuple (f, x) in S^i with x value greater that $c - \sum_{i \leq j \leq n} c_j$ as such a tuple will not leave adequate funds to complete the system. In addition, we can devise a simple heuristic to determine the best reliability obtainable by completing a tuple (f, x) in S^i. If this is less than a heuristically determined lower bound on the optimal system reliability, then (f, x) can be eliminated from S^i.

EXERCISE

1. (a) Present an algorithm similar to DKnap to solve the recurrence (5.19).
   (b) What are the time and space requirements of your algorithm?
   (c) Test the correctness of your algorithm using suitable test data.

5.9 THE TRAVELING SALESPERSON PROBLEM

We have seen how to apply dynamic programming to a subset selection problem (0/1 knapsack). Now we turn our attention to a permutation problem. Note that permutation problems usually are much harder to solve than subset problems as there are $n!$ different permutations of n objects whereas there are only 2^n different subsets of n objects ($n! > 2^n$). Let $G = (V, E)$ be a directed graph with edge costs c_{ij}. The variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$. Let $|V| = n$ and assume $n > 1$. A *tour* of G is a directed simple cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The *traveling salesperson problem* is to find a tour of minimum cost.

The traveling salesperson problem finds application in a variety of situations. Suppose we have to route a postal van to pick up mail from mail

boxes located at n different sites. An $n+1$ vertex graph can be used to represent the situation. One vertex represents the post office from which the postal van starts and to which it must return. Edge $\langle i, j \rangle$ is assigned a cost equal to the distance from site i to site j. The route taken by the postal van is a tour, and we are interested in finding a tour of minimum length.

As a second example, suppose we wish to use a robot arm to tighten the nuts on some piece of machinery on an assembly line. The arm will start from its initial position (which is over the first nut to be tightened), successively move to each of the remaining nuts, and return to the initial position. The path of the arm is clearly a tour on a graph in which vertices represent the nuts. A minimum-cost tour will minimize the time needed for the arm to complete its task (note that only the total arm movement time is variable; the nut tightening time is independent of the tour).

Our final example is from a production environment in which several commodities are manufactured on the same set of machines. The manufacture proceeds in cycles. In each production cycle, n different commodities are produced. When the machines are changed from production of commodity i to commodity j, a change over cost c_{ij} is incurred. It is desired to find a sequence in which to manufacture these commodities. This sequence should minimize the sum of change over costs (the remaining production costs are sequence independent). Since the manufacture proceeds cyclically, it is necessary to include the cost of starting the next cycle. This is just the change over cost from the last to the first commodity. Hence, this problem can be regarded as a traveling salesperson problem on an n vertex graph with edge cost c_{ij}'s being the changeover cost from commodity i to commodity j.

In the following discussion we shall, without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge $\langle 1, k \rangle$ for some $k \in V - \{1\}$ and a path from vertex k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in $V - \{1, k\}$ exactly once. It is easy to see that if the tour is optimal, then the path from k to 1 must be a shortest k to 1 path going through all vertices in $V - \{1, k\}$. Hence, the principle of optimality holds. Let $g(i, S)$ be the length of a shortest path starting at vertex i, going through all vertices in S, and terminating at vertex 1. The function $g(1, V - \{1\})$ is the length of an optimal salesperson tour. From the principal of optimality it follows that

$$g(1, V - \{1\}) = \min_{2 \le k \le n} \{c_{1k} + g(k, V - \{1, k\})\} \tag{5.20}$$

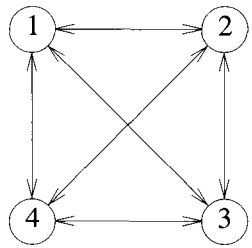
Generalizing (5.20), we obtain (for $i \notin S$)

$$g(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \tag{5.21}$$

Equation 5.20 can be solved for $g(1, V - \{1\})$ if we know $g(k, V - \{1, k\})$ for all choices of k. The g values can be obtained by using (5.21). Clearly,

$g(i, \phi) = c_{i1}$, $1 \leq i \leq n$. Hence, we can use (5.21) to obtain $g(i, S)$ for all S of size 1. Then we can obtain $g(i, S)$ for S with $|S| = 2$, and so on. When $|S| < n - 1$, the values of i and S for which $g(i, S)$ is needed are such that $i \neq 1$, $1 \notin S$, and $i \notin S$.

Example 5.26 Consider the directed graph of Figure 5.21(a). The edge lengths are given by matrix c of Figure 5.21(b).



(a)

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

(b)

Figure 5.21 Directed graph and edge length matrix c

Thus $g(2, \phi) = c_{21} = 5, g(3, \phi) = c_{31} = 6$, and $g(4, \phi) = c_{41} = 8$. Using (5.21), we obtain

$$\begin{array}{lllll} g(2,\{3\}) & = & c_{23} + g(3,\phi) & = & 15 \\ g(3,\{2\}) & = & 18 & & \\ g(4,\{2\}) & = & 13 & & \end{array} \qquad \begin{array}{lll} g(2,\{4\}) & = & 18 \\ g(3,\{4\}) & = & 20 \\ g(4,\{3\}) & = & 15 \end{array}$$

Next, we compute $g(i, S)$ with $|S| = 2$, $i \neq 1$, $1 \notin S$ and $i \notin S$.

$$\begin{array}{lll} g(2,\{3,4\}) & = & \min\{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} = 25 \\ g(3,\{2,4\}) & = & \min\{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = 25 \\ g(4,\{2,3\}) & = & \min\{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = 23 \end{array}$$

Finally, from (5.20) we obtain

$$\begin{array}{lll} g(1,\{2,3,4\}) & = & \min\{c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\})\} \\ & = & \min\{35, 40, 43\} \\ & = & 35 \end{array}$$

An optimal tour of the graph of Figure 5.21(a) has length 35. A tour of this length can be constructed if we retain with each $g(i, S)$ the value of j that minimizes the right-hand side of (5.21). Let $J(i, S)$ be this value. Then, $J(1, \{2, 3, 4\}) = 2$. Thus the tour starts from 1 and goes to 2. The remaining tour can be obtained from $g(2, \{3, 4\})$. So $J(2, \{3, 4\}) = 4$. Thus the next edge is $\langle 2, 4\rangle$. The remaining tour is for $g(4, \{3\})$. So $J(4, \{3\}) = 3$. The optimal tour is 1, 2, 4, 3, 1. □

Let N be the number of $g(i, S)$'s that have to be computed before (5.20) can be used to compute $g(1, V - \{1\})$. For each value of $|S|$ there are $n - 1$ choices for i. The number of distinct sets S of size k not including 1 and i is $\binom{n-2}{k}$. Hence

$$N = \sum_{k=0}^{n-2} (n-1)\binom{n-2}{k} = (n-1)2^{n-2}$$

An algorithm that proceeds to find an optimal tour by using (5.20) and (5.21) will require $\Theta(n^2 2^n)$ time as the computation of $g(i, S)$ with $|S| = k$ requires $k - 1$ comparisons when solving (5.21). This is better than enumerating all $n!$ different tours to find the best one. The most serious drawback of this dynamic programming solution is the space needed, $O(n2^n)$. This is too large even for modest values of n.

EXERCISE

1. (a) Obtain a data representation for the values $g(i, S)$ of the traveling salesperson problem. Your representation should allow for easy access to the value of $g(i, S)$, given i and S. (i) How much space does your representation need for an n vertex graph? (ii) How much time is needed to retrieve or update the value of $g(i, S)$?

   (b) Using the representation of (a), develop an algorithm corresponding to the dynamic programming solution of the traveling salesperson problem.

   (c) Test the correctness of your algorithm using suitable test data.

5.10 FLOW SHOP SCHEDULING

Often the processing of a job requires the performance of several distinct tasks. Computer programs run in a multiprogramming environment are input and then executed. Following the execution, the job is queued for output

and the output eventually printed. In a general flow shop we may have n jobs each requiring m tasks $T_{1i}, T_{2i}, \ldots, T_{mi}$, $1 \leq i \leq n$, to be performed. Task T_{ji} is to be performed on processor P_j, $1 \leq j \leq m$. The time required to complete task T_{ji} is t_{ji}. A schedule for the n jobs is an assignment of tasks to time intervals on the processors. Task T_{ji} must be assigned to processor P_j. No processor may have more than one task assigned to it in any time interval. Additionally, for any job i the processing of task T_{ji}, $j > 1$, cannot be started until task $T_{j-1,i}$ has been completed.

Example 5.27 Two jobs have to be scheduled on three processors. The task times are given by the matrix $\mathcal{J}$

$$\mathcal{J} = \begin{bmatrix} 2 & 0 \\ 3 & 3 \\ 5 & 2 \end{bmatrix}$$

Two possible schedules for the jobs are shown in Figure 5.22. □

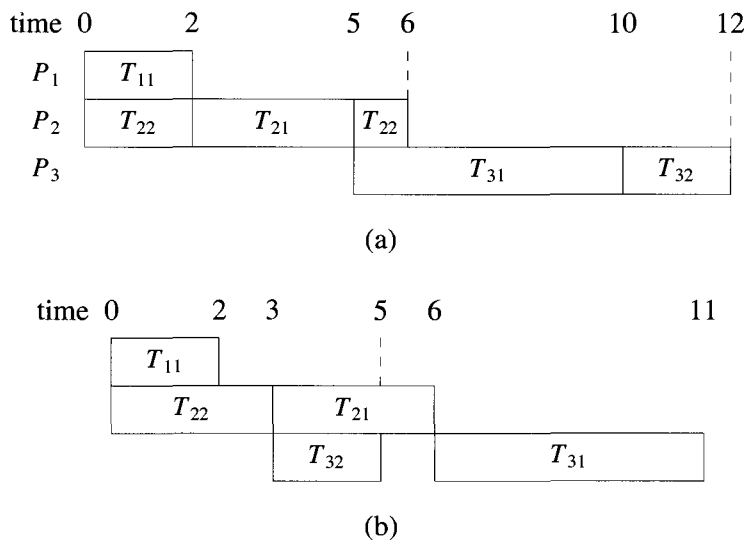


Figure 5.22 Two possible schedules for Example 5.27

A *nonpreemptive* schedule is a schedule in which the processing of a task on any processor is not terminated until the task is complete. A schedule for which this need not be true is called *preemptive*. The schedule of Figure 5.22(a) is a preemptive schedule. Figure 5.22(b) shows a nonpreemptive schedule. The *finish time* $f_i(S)$ of job i is the time at which all tasks of job i have been completed in schedule S. In Figure 5.22(a), $f_1(S) = 10$ and $f_2(S) = 12$. In Figure 5.22(b), $f_1(S) = 11$ and $f_2(S) = 5$. The finish time $F(S)$ of a schedule S is given by

$$F(S) = \max_{1 \le i \le n} \{f_i(S)\} \tag{5.22}$$

The *mean flow time* MFT(S) is defined to be

$$\text{MFT}(S) = \frac{1}{n} \sum_{1 \le i \le n} f_i(S) \tag{5.23}$$

An optimal finish time (OFT) schedule for a given set of jobs is a nonpreemptive schedule S for which $F(S)$ is minimum over all nonpreemptive schedules S. A preemptive optimal finish time (POFT) schedule, optimal mean finish time schedule (OMFT), and preemptive optimal mean finish (POMFT) schedule are defined in the obvious way.

Although the general problem of obtaining OFT and POFT schedules for $m > 2$ and of obtaining OMFT schedules is computationally difficult (see Chapter 11), dynamic programming leads to an efficient algorithm to obtain OFT schedules for the case $m = 2$. In this section we consider this special case.

For convenience, we shall use a_i to represent t_{1i}, and b_i to represent t_{2i}. For the two-processor case, one can readily verify that nothing is to be gained by using different processing orders on the two processors (this is not true for $m > 2$). Hence, a schedule is completely specified by providing a permutation of the jobs. Jobs will be executed on each processor in this order. Each task will be started at the earliest possible time. The schedule of Figure 5.23 is completely specified by the permutation (5, 1, 3, 2, 4). We make the simplifying assumption that $a_i \neq 0$, $1 \le i \le n$. Note that if jobs with $a_i = 0$ are allowed, then an optimal schedule can be constructed by first finding an optimal permutation for all jobs with $a_i \neq 0$ and then adding all jobs with $a_i = 0$ (in any order) in front of this permutation (see the exercises).

It is easy to see that an optimal permutation (schedule) has the property that given the first job in the permutation, the remaining permutation is optimal with respect to the state the two processors are in following the completion of the first job. Let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be a permutation prefix defining a schedule for jobs $T_1, T_2, \ldots, T_k$. For this schedule let f_1 and f_2 be the times at which the processing of jobs $T_1, T_2, \ldots, T_k$ is completed on processors P_1

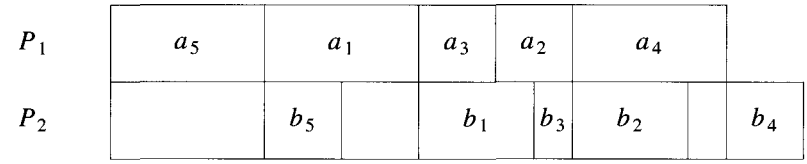


Figure 5.23 A schedule

and P_2 respectively. Let $t = f_2 - f_1$. The state of the processors following the sequence of decisions $T_1, T_2, \ldots, T_k$ is completely characterized by t. Let $g(S,t)$ be the length of an optimal schedule for the subset of jobs S under the assumption that processor 2 is not available until time t. The length of an optimal schedule for the job set $\{1,2,\ldots,n\}$ is $g(\{1,2,\ldots,n\},0)$.

Since the principle of optimality holds, we obtain

$$g(\{1,2,\cdots,n\},0) = \min_{1\le i\le n}\{a_i + g(\{1,2,\cdots,n\} - \{i\}, b_i)\} \tag{5.24}$$

Equation 5.24 generalizes to (5.25) for arbitrary S and t. This generalization requires that $g(\phi,t) = \max\{t,0\}$ and that $a_i \neq 0,\ 1 \le i \le n$.

$$g(S,t) = \min_{i\in S}\ \{a_i + g(S-\{i\}, b_i + \max\{t-a_i,0\})\} \tag{5.25}$$

The term $\max\ \{t-a_i,0\}$ comes into (5.25) as task T_{2i} cannot start until $\max\{a_i,t\}$ (P_2 is not available until time t). Hence $f_2 - f_1 = b_i + \max\{a_i,t\} - a_i = b_i + \max\{t-a_i,0\}$. We can solve for $g(S,t)$ using an approach similar to that used to solve (5.21). However, it turns out that (5.25) can be solved algebraically and a very simple rule to generate an optimal schedule obtained.

Consider any schedule R for a subset of jobs S. Assume that P_2 is not available until time t. Let i and j be the first two jobs in this schedule. Then, from (5.25) we obtain

$$\begin{aligned} g(S,t) &= a_i + g(S-\{i\}, b_i + \max\ \{t-a_i,0\}) \\ g(S,t) &= a_i + a_j + g(S-\{i,j\}, b_j + \max\ \{b_i + \max\ \{t-a_i,0\} - a_j, 0\}) \end{aligned} \tag{5.26}$$

Equation 5.26 can be simplified using the following result:

$$\begin{aligned} t_{ij} &= b_j + \max\ \{b_i + \max\ \{t - a_i, 0\} - a_j, 0\} \\ &= b_j + b_i - a_j + \max\ \{\max\ \{t - a_i, 0\}, a_j - b_i\} \\ &= b_j + b_i - a_j + \max\ \{t - a_i, a_j - b_i, 0\} \\ t_{ij} &= b_j + b_i - a_j - a_i + \max\ \{t, a_i + a_j - b_i, a_i\} \end{aligned} \tag{5.27}$$

If jobs i and j are interchanged in R, then the finish time $g'(S,t)$ is

$$g'(S,t) = a_i + a_j + g(S - \{i,j\}, t_{ji})$$

$$\text{where } t_{ji} = b_j + b_i - a_j - a_i + \max\ \{t, a_i + a_j - b_j, a_j\}$$

Comparing $g(S,t)$ and $g'(S,t)$, we see that if (5.28) below holds, then $g(S,t) \leq g'(S,t)$.

$$\max\ \{t, a_i + a_j - b_i, a_i\} \leq \max\ \{t, a_i + a_j - b_j, a_j\} \tag{5.28}$$

In order for (5.28) to hold for all values of t, we need

$$\max\ \{a_i + a_j - b_i, a_i\} \leq \max\ \{a_i + a_j - b_j, a_j\}$$

$$\text{or } a_i + a_j + \max\ \{-b_i, -a_j\} \leq a_i + a_j + \max\ \{-b_j, -a_i\}$$

$$\text{or } \min\ \{b_i, a_j\} \geq \min\ \{b_j, a_i\} \tag{5.29}$$

From (5.29) we can conclude that there exists an optimal schedule in which for every pair (i,j) of adjacent jobs, $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$. Exercise 4 shows that all schedules with this property have the same length. Hence, it suffices to generate any schedule for which (5.29) holds for every pair of adjacent jobs. We can obtain a schedule with this property by making the following observations from (5.29). If $\min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ is a_i, then job i should be the first job in an optimal schedule. If $\min\{a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ is b_j, then job j should be the last job in an optimal schedule. This enables us to make a decision as to the positioning of one of the n jobs. Equation 5.29 can now be used on the remaining $n-1$ jobs to correctly position another job, and so on. The scheduling rule resulting from (5.29) is therefore:

1. Sort all the a_i's and b_j's into nondecreasing order.

2. Consider this sequence in this order. If the next number in the sequence is a_j and job j hasn't yet been scheduled, schedule job j at the leftmost available spot. If the next number is b_j and job j hasn't yet been scheduled, schedule job j at the rightmost available spot. If j has already been scheduled, go to the next number in the sequence.

Note that the above rule also correctly positions jobs with $a_i = 0$. Hence, these jobs need not be considered separately.

Example 5.28 Let $n = 4, (a_1, a_2, a_3, a_4) = (3, 4, 8, 10)$, and $(b_1, b_2, b_3, b_4) = (6, 2, 9, 15)$. The sorted sequence of a's and b's is $(b_2, a_1, a_2, b_1, a_3, b_3, a_4, b_4) = (2, 3, 4, 6, 8, 9, 10, 15)$. Let $\sigma_1, \sigma_2, \sigma_3$, and σ_4 be the optimal schedule. Since the smallest number is b_2, we set $\sigma_4 = 2$. The next number is a_1 and we set $\sigma_1 = a_1$. The next smallest number is a_2. Job 2 has already been scheduled. The next number is b_1. Job 1 has already been scheduled. The next is a_3 and we set σ_3. This leaves σ_3 free and job 4 unscheduled. Thus, $\sigma_3 = 4$. □

The scheduling rule above can be implemented to run in time $O(n \log n)$ (see exercises). Solving (5.24) and (5.25) directly for $g(1, 2, \ldots, n, 0)$ for the optimal schedule will take $\Omega(2^n)$ time as there are these many different S's for which $g(S, t)$ will be computed.

EXERCISES

1. N jobs are to be processed. Two machines A and B are available. If job i is processed on machine A, then a_i units of processing time are needed. If it is processed on machine B, then b_i units of processing time are needed. Because of the peculiarities of the jobs and the machines, it is quite possible that $a_i \geq b_i$ for some i while $a_j < b_j$ for some j, $j \neq i$. Obtain a dynamic programming formulation to determine the minimum time needed to process all the jobs. Note that jobs cannot be split between machines. Indicate how you would go about solving the recurrence relation obtained. Do this on an example of your choice. Also indicate how you would determine an optimal assignment of jobs to machines.

2. N jobs have to be scheduled for processing on one machine. Associated with job i is a 3-tuple (p_i, t_i, d_i). The variable t_i is the processing time needed to complete job i. If job i is completed by its deadline d_i, then a profit p_i is earned. If not, then nothing is earned. From Section 4.4 we know that J is a subset of jobs that can all be completed by their

deadlines iff the jobs in J can be processed in nondecreasing order of deadlines without violating any deadline. Assume $d_i \leq d_{i+1}$, $1 \leq i < n$. Let $f_i(x)$ be the maximum profit that can be earned from a subset J of jobs when $n = i$. Here $f_n(d_n)$ is the value of an optimal selection of jobs J. Let $f_0(x) = 0$. Show that for $x \leq t_i$,

$$f_i(x) = \max \{f_{i-1}(x),\ f_{i-1}(x - t_i) + p_i\}$$

3. Let I be any instance of the two-processor flow shop problem.
   (a) Show that the length of every POFT schedule for I is the same as the length of every OFT schedule for I. Hence, the algorithm of Section 5.10 also generates a POFT schedule.
   (b) Show that there exists an OFT schedule for I in which jobs are processed in the same order on both processors.
   (c) Show that there exists an OFT schedule for I defined by some permutation σ of the jobs (see part (b)) such that all jobs with $a_i = 0$ are at the front of this permutation. Further, show that the order in which these jobs appear at the front of the permutation is not important.

4. Let I be any instance of the two-processor flow shop problem. Let $\sigma = \sigma_1\sigma_2 \cdots \sigma_n$ be a permutation defining an OFT schedule for I.
   (a) Use (5.29) to argue that there exists an OFT σ such that $\min \{b_i, a_j) \geq \min \{b_j, a_i\}$ for every i and j such that $i = \sigma_k$ and $j = \sigma_{k+1}$ (that is, i and j are adjacent).
   (b) For a σ satisfying the conditions of part (a), show that $\min\{b_i, a_j\} \geq \min\{b_j, a_i\}$ for every i and j such that $i = \sigma_k$ and $j = \sigma_r, k < r$.
   (c) Show that all schedules corresponding to σ's satisfying the conditions of part (a) have the same finish time. (*Hint*: use part (b) to transform one of two different schedules satisfying (a) into the other without increasing the finish time.)

5.11 REFERENCES AND READINGS

Two classic references on dynamic programming are:

Introduction to Dynamic Programming, by G. Nemhauser, John Wiley and Sons, 1966.

Applied Dynamic Programming by R. E. Bellman and S. E. Dreyfus, Princeton University Press, 1962.

See also *Dynamic Programming,* by E. V. Denardo, Prentice-Hall, 1982.

The dynamic programming formulation for the shortest-paths problem was given by R. Floyd.

Bellman and Ford's algorithm for the single-source shortest-path problem (with general edge weights) can be found in *Dynamic Programming* by R. E. Bellman, Princeton University Press, 1957.

The construction of optimal binary search trees using dynamic programming is described in *The Art of Programming: Sorting and Searching*, Vol. 3, by D. E. Knuth, Addison Wesley, 1973.

The string editing algorithm discussed in this chapter is in "The string-to-string correction problem," by R. A. Wagner and M. J. Fischer, *Journal of the ACM* 21, no. 1 (1974): 168–173.

The set generation approach to solving the 0/1 knapsack problem was formulated by G. Nemhauser and Z. Ullman, and E. Horowitz and S. Sahni.

Exercise 6 in Section 5.7 is due to E. Horowitz and S. Sahni.

The dynamic programming formulation for the traveling salesperson problem was given by M. Held and R. Karp.

The dynamic programming solution to the matrix product chain problem (Exercises 1 and 2 in Additional Exercises) is due to S. Godbole.

5.12 ADDITIONAL EXERCISES

1. [Matrix product chains] Let A, B, and C be three matrices such that $C = A \times B$. Let the dimensions of A, B, and C respectively be $m \times n, n \times p$, and $m \times p$. From the definition of matrix multiplication,

$$C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$$

   (a) Write an algorithm to compute C directly using the above formula. Show that the number of multiplications needed by your algorithm is mnp.

   (b) Let $M_1 \times M_2 \times \cdots \times M_r$ be a chain of matrix products. This chain may be evaluated in several different ways. Two possibilities are $(\cdots((M_1 \times M_2) \times M_3) \times M_4) \times \cdots) \times M_r$ and $(M_1 \times (M_2 \times (\cdots \times (M_{r-1} \times M_r)\cdots)$. The cost of any computation of $M_1 \times$

$M_2 \times \cdots \times M_r$ is the number of multiplications used. Consider the case $r = 4$ and matrices M_1 through M_4 with dimensions $100 \times 1, 1 \times 100, 100 \times 1$, and 1×100 respectively. What is the cost of each of the five ways to compute $M_1 \times M_2 \times M_3 \times M_4$? Show that the optimal way has a cost of 10,200 and the worst way has a cost of 1,020,000. Assume that all matrix products are computed using the algorithm of part (a).

(c) Let M_{ij} denote the matrix product $M_i \times M_{i+1} \times \cdots \times M_j$. Thus, $M_{ii} = M_i$, $1 \leq i \leq r$. $S = P_1, P_2, \ldots, P_{r-1}$ is a *product sequence* computing M_{1r} iff each product P_k is of the form $M_{ij} \times M_{j+1,q}$, where M_{ij} and $M_{j+1,q}$ have been computed either by an earlier product $P_l, l < k$, or represent an input matrix M_{tt}. Note that $M_{ij} \times M_{j+1,q} = M_{iq}$. Also note that every valid computation of M_{1r} using only pairwise matrix products at each step is defined by a product sequence. Two product sequences $S_1 = P_1, P_2, \ldots, P_{r-1}$ and $S_2 = U_1, U_2, \ldots, U_{r-1}$ are *different* if $P_i \neq U_i$ for some i. Show that the number of different product sequences if $(r-1)!$

(d) Although there are $(r-1)!$ different product sequences, many of these are essentially the same in the sense that the same pairs of matrices are multiplied. For example, the sequences $S_1 = (M_1 \times M_2), (M_3 \times M_4), (M_{12} \times M_{34})$ and $S_2 = (M_3 \times M_4), (M_1 \times M_2), (M_{12} \times M_{34})$ are different under the definition of part (c). However, the same pairs of matrices are multiplied in both S_1 and S_2. Show that if we consider only those product sequences that differ from each other in at least one matrix product, then the number of different sequences is equal to the number of different binary trees having exactly $r-1$ nodes.

(e) Show that the number of different binary trees with n nodes is

$$\frac{1}{n+1}\binom{2n}{n}$$

2. [Matrix product chains] In the preceding exercise it was established that the number of different ways to evaluate a matrix product chain is very large even when r is relatively small (say 10 or 20). In this exercise we shall develop an $O(r^3)$ algorithm to find an optimal product sequence (that is, one of minimum cost). Let $D(i), 0 \leq i \leq r$, represent the dimensions of the matrices; that is, M_i has $D(i-1)$ rows and $D(i)$ columns. Let $C(i,j)$ be the cost of computing M_{ij} using an optimal product sequence for M_{ij}. Observe that $C(i,i) = 0, 1 \leq i \leq r$, and that $C(i, i+1) = D(i-1)D(i)D(i+1), 1 \leq i \leq r$.

(a) Obtain a recurrence relation for $C(i,j), j > i$. This recurrence relation will be similar to Equation 5.14.

(b) Write an algorithm to solve the recurrence relation of part (a) for $C(1,r)$. Your algorithm should be of complexity $O(r^3)$.

(c) What changes are needed in the algorithm of part (b) to determine an optimal product sequence. Write an algorithm to determine such a sequence. Show that the overall complexity of your algorithm remains $O(r^3)$.

(d) Work through your algorithm (by hand) for the product chain of part (b) of the previous exercise. What are the values of $C(i,j), 1 \leq i \leq r$ and $j \geq i$? What is an optimal way to compute M_{14}?

3. There are two warehouses W_1 and W_2 from which supplies are to be shipped to destinations $D_i, 1 \leq i \leq n$. Let d_i be the demand at D_i and let r_i be the inventory at W_i. Assume $r_1 + r_2 = \sum d_i$. Let $c_{ij}(x_{ij})$ be the cost of shipping x_{ij} units from warehouse W_i to destination D_j. The warehouse problem is to find nonnegative integers $x_{ij}, 1 \leq i \leq 2$ and $1 \leq j \leq n$, such that $x_{1j} + x_{2j} = d_j$, $1 \leq j \leq n$, and $\sum_{i,j} c_{ij}(x_{ij})$ is minimized. Let $g_i(x)$ be the cost incurred when W_1 has an inventory of x and supplies are sent to $D_j, 1 \leq j \leq i$, in an optimal manner (the inventory at W_2 is $\sum_{1 \leq j \leq i} d_j - x$). The cost of an optimal solution to the warehouse problem is $g_n(r_1)$.

   (a) Use the optimality principle to obtain a recurrence relation for $g_i(x)$.

   (b) Write an algorithm to solve this recurrence and obtain an optimal sequence of values for $x_{ij}, 1 \leq i \leq 2, 1 \leq j \leq n$.

4. Given a warehouse with a storage capacity of B units and an initial stock of v units, let y_i be the quantity sold in each month, $i, 1 \leq i \leq n$. Let P_i be the per-unit selling price in month i, and x_i the quantity purchased in month i. The buying price is c_i per unit. At the end of each month, the stock in hand must be no more than B. That is,

$$v + \sum_{1 \leq i \leq j} (x_i - y_i) \leq B, \quad 1 \leq j \leq n$$

   The amount sold in each month cannot be more than the stock at the end of the previous month (new stock arrives only at the end of a month). That is,

$$y_i \leq v + \sum_{1 \leq j < i} (x_j - y_j), \quad 1 \leq i \leq n$$

Also, we require x_i and y_i to be nonnegative integers. The total profit derived is

$$P_n = \sum_{j=1}^{n} (p_j y_j - c_j x_j)$$

The problem is to determine x_j and y_j such that P_n is maximized. Let $f_i(v_i)$ represent the maximum profit that can be earned in months $i+1, i+2, \ldots, n$, starting with v_i units of stock at the end of month i. Then $f_0(v)$ is the maximum value of P_n.

(a) Obtain the dynamic programming recurrence for $f_i(v_i)$ in terms of $f_{i+1}(v_i)$.

(b) What is $f_n(v_i)$?

(c) Solve part (a) analytically to obtain the formula

$$f_i(v_i) = a_i x_i + b_i v_i$$

for some constants a_i and b_i.

(d) Show that an optimal P_n is obtained by using the following strategy:

i. $p_i \geq c_i$

A. If $b_{i+1} \geq c_i$, then $y_i = v_i$ and $x_i = B$.

B. If $b_{i+1} \leq c_i$, then $y_i = v_i$ and $x_i = 0$.

ii. $c_i \geq p_i$

A. If $b_{i+1} \geq c_i$, then $y_i = 0$ and $x_i = B - v_i$.

B. If $b_{i+1} \leq p_i$, then $y_i = v_i$ and $x_i = 0$.

C. If $p_i \leq b_{i+1} \leq c_i$, then $y_i = 0$ and $x_i = 0$.

(e) Use the p_i and c_i in Figure 5.24 and obtain an optimal decision sequence from part (d).

i	1	2	3	4	5	6	7	8
p_i	8	8	2	3	4	3	2	5
c_i	3	6	7	1	4	5	1	3

Figure 5.24 p_i and c_i for Exercise 4

Assume the warehouse capacity to be 100 and the initial stock to be 60.

(f) From part (d) conclude that an optimal set of values for x_i and y_i will always lead to the following policy: Do no buying or selling for the first k months (k may be zero) and then oscillate between a full and an empty warehouse for the remaining months.

5. Assume that n programs are to be stored on two tapes. Let l_i be the length of tape needed to store the ith program. Assume that $\sum l_i \leq L$, where L is the length of each tape. A program can be stored on either of the two tapes. If S_1 is the set of programs on tape 1, then the worst-case access time for a program is proportional to $\max\{\sum_{i\in S_1} l_i, \sum_{i\notin S_1} l_i\}$. An optimal assignment of programs to tapes minimizes the worst-case access times. Formulate a dynamic programming approach to determine the worst-case access time of an optimal assignment. Write an algorithm to determine this time. What is the complexity of your algorithm?

6. Redo Exercise 5 making the assumption that programs will be stored on tape 2 using a different tape density than that used on tape 1. If l_i is the tape length needed by program i when stored on tape 1, then al_i is the tape length needed on tape 2.

7. Let L be an array of n distinct integers. Give an efficient algorithm to find the length of a longest increasing subsequence of entries in L. For example, if the entries are $11, 17, 5, 8, 6, 4, 7, 12, 3$, a longest increasing subsequence is $5, 6, 7, 12$. What is the run time of your algorithm?

Chapter 7

BACKTRACKING

7.1 THE GENERAL METHOD

In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation. The name backtrack was first coined by D. H. Lehmer in the 1950s. Early workers who studied the process were R. J. Walker, who gave an algorithmic account of it in 1960, and S. Golomb and L. Baumert who presented a very general description of it as well as a variety of applications.

In many applications of the backtrack method, the desired solution is expressible as an n-tuple $(x_1, \ldots, x_n)$, where the x_i are chosen from some finite set S_i. Often the problem to be solved calls for finding one vector that maximizes (or minimizes or satisfies) a *criterion function* $P(x_1, \ldots, x_n)$. Sometimes it seeks all vectors that satisfy P. For example, sorting the array of integers in $a[1 : n]$ is a problem whose solution is expressible by an n-tuple, where x_i is the index in a of the ith smallest element. The criterion function P is the inequality $a[x_i] \leq a[x_{i+1}]$ for $1 \leq i < n$. The set S_i is finite and includes the integers 1 through n. Though sorting is not usually one of the problems solved by backtracking, it is one example of a familiar problem whose solution can be formulated as an n-tuple. In this chapter we study a collection of problems whose solutions are best done using backtracking.

Suppose m_i is the size of set S_i. Then there are $m = m_1 m_2 \cdots m_n$ n-tuples that are possible candidates for satisfying the function P. The *brute force approach* would be to form all these n-tuples, evaluate each one with P, and save those which yield the optimum. The backtrack algorithm has as its virtue the ability to yield the same answer with far fewer than m trials. Its basic idea is to build up the solution vector one component at a time and to use modified criterion functions $P_i(x_1, \ldots, x_i)$ (sometimes called

bounding functions) to test whether the vector being formed has any chance of success. The major advantage of this method is this: if it is realized that the partial vector $(x_1, x_2, \ldots, x_i)$ can in no way lead to an optimal solution, then $m_{i+1} \cdots m_n$ possible test vectors can be ignored entirely.

Many of the problems we solve using backtracking require that all the solutions satisfy a complex set of constraints. For any problem these constraints can be divided into two categories: *explicit* and *implicit*.

Definition 7.1 Explicit constraints are rules that restrict each x_i to take on values only from a given set. □

Common examples of explicit constraints are

$$\begin{array}{rcrcl} x_i \geq 0 & \text{or} & S_i & = & \{\text{all nonnegative real numbers}\} \\ x_i = 0 \text{ or } 1 & \text{or} & S_i & = & \{0,1\} \\ l_i \leq x_i \leq u_i & \text{or} & S_i & = & \{a : l_i \leq a \leq u_i\} \end{array}$$

The explicit constraints depend on the particular instance I of the problem being solved. All tuples that satisfy the explicit constraints define a possible *solution space* for I.

Definition 7.2 The implicit constraints are rules that determine which of the tuples in the solution space of I satisfy the criterion function. Thus implicit constraints describe the way in which the x_i must relate to each other. □

Example 7.1 [8-queens] A classic combinatorial problem is to place eight queens on an 8×8 chessboard so that no two "attack," that is, so that no two of them are on the same row, column, or diagonal. Let us number the rows and columns of the chessboard 1 through 8 (Figure 7.1). The queens can also be numbered 1 through 8. Since each queen must be on a different row, we can without loss of generality assume queen i is to be placed on row i. All solutions to the 8-queens problem can therefore be represented as 8-tuples $(x_1, \ldots, x_8)$, where x_i is the column on which queen i is placed. The explicit constraints using this formulation are $S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \leq i \leq 8$. Therefore the solution space consists of 8^8 8-tuples. The implicit constraints for this problem are that no two x_i's can be the same (i.e., all queens must be on different columns) and no two queens can be on the same diagonal. The first of these two constraints implies that all solutions are permutations of the 8-tuple (1, 2, 3, 4, 5, 6, 7, 8). This realization reduces the size of the solution space from 8^8 tuples to 8! tuples. We see later how to formulate the second constraint in terms of the x_i. Expressed as an 8-tuple, the solution in Figure 7.1 is (4, 6, 8, 2, 7, 1, 3, 5). □

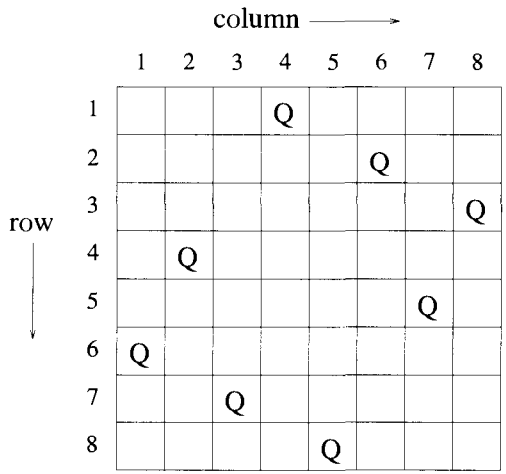


Figure 7.1 One solution to the 8-queens problem

Example 7.2 [Sum of subsets] Given positive numbers w_i, $1 \leq i \leq n$, and m, this problem calls for finding all subsets of the w_i whose sums are m. For example, if $n = 4$, $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$, and $m = 31$, then the desired subsets are (11, 13, 7) and (24, 7). Rather than represent the solution vector by the w_i which sum to m, we could represent the solution vector by giving the indices of these w_i. Now the two solutions are described by the vectors (1, 2, 4) and (3, 4). In general, all solutions are k-tuples $(x_1, x_2, \ldots, x_k)$, $1 \leq k \leq n$, and different solutions may have different-sized tuples. The explicit constraints require $x_i \in \{j \mid j \text{ is an integer and } 1 \leq j \leq n\}$. The implicit constraints require that no two be the same and that the sum of the corresponding w_i's be m. Since we wish to avoid generating multiple instances of the same subset (e.g., (1, 2, 4) and (1, 4, 2) represent the same subset), another implicit constraint that is imposed is that $x_i < x_{i+1}$, $1 \leq i < k$.

In another formulation of the sum of subsets problem, each solution subset is represented by an n-tuple $(x_1, x_2, \ldots, x_n)$ such that $x_i \in \{0, 1\}$, $1 \leq i \leq n$. Then $x_i = 0$ if w_i is not chosen and $x_i = 1$ if w_i is chosen. The solutions to the above instance are (1, 1, 0, 1) and (0, 0, 1, 1). This formulation expresses all solutions using a fixed-sized tuple. Thus we conclude that there may be several ways to formulate a problem so that all solutions are tuples that satisfy some constraints. One can verify that for both of the above formulations, the solution space consists of 2^n distinct tuples. □

Backtracking algorithms determine problem solutions by systematically searching the solution space for the given problem instance. This search is facilitated by using a *tree organization* for the solution space. For a given solution space many tree organizations may be possible. The next two examples examine some of the ways to organize a solution into a tree.

Example 7.3 [n-queens] The n-queens problem is a generalization of the 8-queens problem of Example 7.1. Now n queens are to be placed on an $n \times n$ chessboard so that no two attack; that is, no two queens are on the same row, column, or diagonal. Generalizing our earlier discussion, the solution space consists of all $n!$ permutations of the n-tuple $(1, 2, \ldots, n)$. Figure 7.2 shows a possible tree organization for the case $n = 4$. A tree such as this is called a *permutation tree*. The edges are labeled by possible values of x_i. Edges from level 1 to level 2 nodes specify the values for x_1. Thus, the leftmost subtree contains all solutions with $x_1 = 1$; its leftmost subtree contains all solutions with $x_1 = 1$ and $x_2 = 2$, and so on. Edges from level i to level $i+1$ are labeled with the values of x_i. The solution space is defined by all paths from the root node to a leaf node. There are $4! = 24$ leaf nodes in the tree of Figure 7.2. □

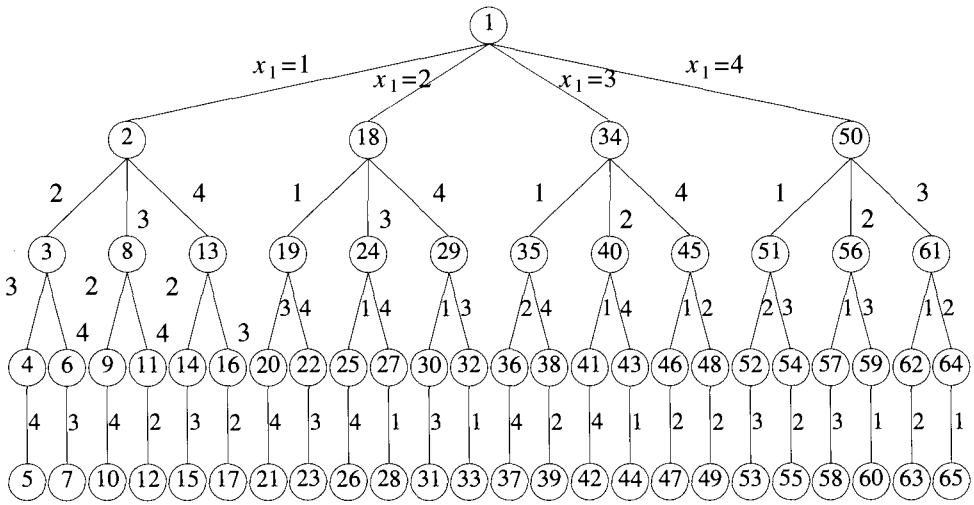


Figure 7.2 Tree organization of the 4-queens solution space. Nodes are numbered as in depth first search.

Example 7.4 [Sum of subsets] In Example 7.2 we gave two possible formulations of the solution space for the sum of subsets problem. Figures 7.3 and 7.4 show a possible tree organization for each of these formulations for the case $n = 4$. The tree of Figure 7.3 corresponds to the variable tuple size formulation. The edges are labeled such that an edge from a level i node to a level $i+1$ node represents a value for x_i. At each node, the solution space is partitioned into subsolution spaces. The solution space is defined by all paths from the root node to any node in the tree, since any such path corresponds to a subset satisfying the explicit constraints. The possible paths are () (this corresponds to the empty path from the root to itself), (1), (1,2), (1,2,3), (1,2,3,4), (1,2,4), (1,3,4), (2), (2,3), and so on. Thus, the leftmost subtree defines all subsets containing w_1, the next subtree defines all subsets containing w_2 but not w_1, and so on.

The tree of Figure 7.4 corresponds to the fixed tuple size formulation. Edges from level i nodes to level $i+1$ nodes are labeled with the value of x_i, which is either zero or one. All paths from the root to a leaf node define the solution space. The left subtree of the root defines all subsets containing w_1, the right subtree defines all subsets not containing w_1, and so on. Now there are 2^4 leaf nodes which represent 16 possible tuples. □

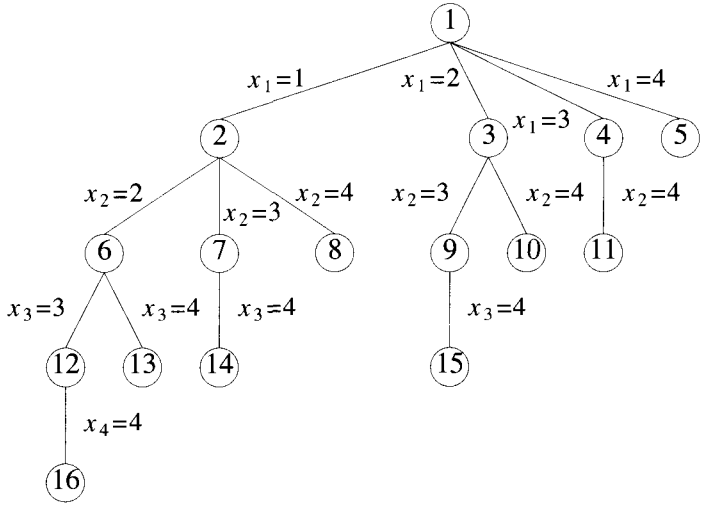


Figure 7.3 A possible solution space organization for the sum of subsets problem. Nodes are numbered as in breadth-first search.

At this point it is useful to develop some terminology regarding tree organizations of solution spaces. Each node in this tree defines a *problem*

state. All paths from the root to other nodes define the *state space* of the problem. *Solution states* are those problem states s for which the path from the root to s defines a tuple in the solution space. In the tree of Figure 7.3 all nodes are solution states whereas in the tree of Figure 7.4 only leaf nodes are solution states. *Answer states* are those solution states s for which the path from the root to s defines a tuple that is a member of the set of solutions (i.e., it satisfies the implicit constraints) of the problem. The tree organization of the solution space is referred to as the *state space tree*.

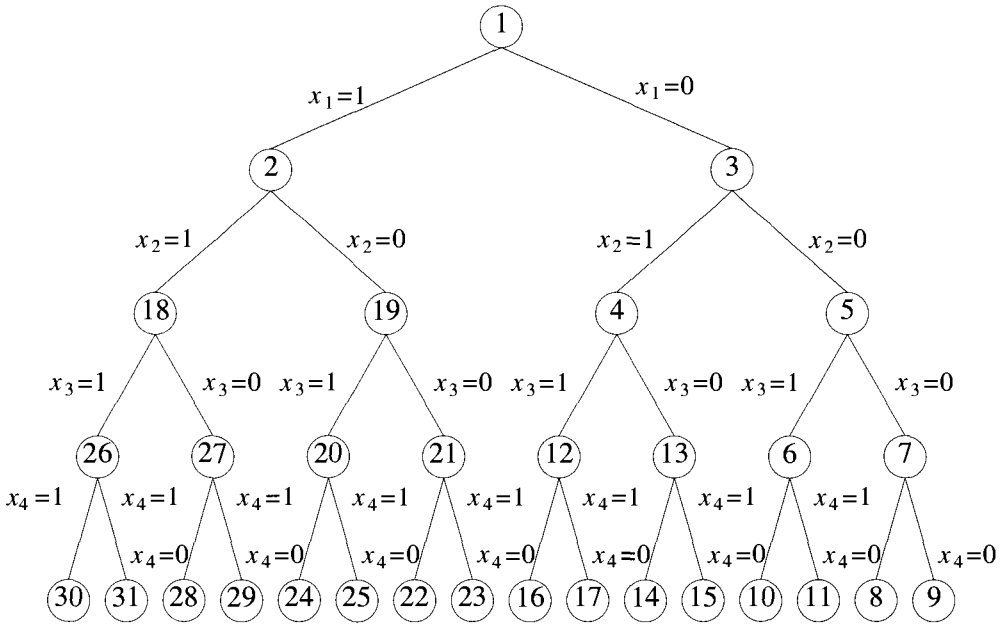


Figure 7.4 Another possible organization for the sum of subsets problems. Nodes are numbered as in D-search.

At each internal node in the space tree of Examples 7.3 and 7.4 the solution space is partitioned into disjoint sub-solution spaces. For example, at node 1 of Figure 7.2 the solution space is partitioned into four disjoint sets. Subtrees 2, 18, 34, and 50 respectively represent all elements of the solution space with $x_1 = 1$, 2, 3, and 4. At node 2 the sub-solution space with $x_1 = 1$ is further partitioned into three disjoint sets. Subtree 3 represents all solution space elements with $x_1 = 1$ and $x_2 = 2$. For all the state space trees we study in this chapter, the solution space is partitioned into disjoint sub-solution spaces at each internal node. It should be noted that this is

not a requirement on a state space tree. The only requirement is that every element of the solution space be represented by at least one node in the state space tree.

The state space tree organizations described in Example 7.4 are called *static trees*. This terminology follows from the observation that the tree organizations are independent of the problem instance being solved. For some problems it is advantageous to use different tree organizations for different problem instances. In this case the tree organization is determined dynamically as the solution space is being searched. Tree organizations that are problem instance dependent are called *dynamic trees*. As an example, consider the fixed tuple size formulation for the sum of subsets problem (Example 7.4). Using a dynamic tree organization, one problem instance with $n = 4$ can be solved by means of the organization given in Figure 7.4. Another problem instance with $n = 4$ can be solved by means of a tree in which at level 1 the partitioning corresponds to $x_2 = 1$ and $x_2 = 0$. At level 2 the partitioning could correspond to $x_1 = 1$ and $x_1 = 0$, at level 3 it could correspond to $x_3 = 1$ and $x_3 = 0$, and so on. We see more of dynamic trees in Sections 7.6 and 8.3.

Once a state space tree has been conceived of for any problem, this problem can be solved by systematically generating the problem states, determining which of these are solution states, and finally determining which solution states are answer states. There are two fundamentally different ways to generate the problem states. Both of these begin with the root node and generate other nodes. A node which has been generated and all of whose children have not yet been generated is called a *live node*. The live node whose children are currently being generated is called the E-node (node being expanded). A *dead node* is a generated node which is not to be expanded further or all of whose children have been generated. In both methods of generating problem states, we have a list of live nodes. In the first of these two methods as soon as a new child C of the current E-node R is generated, this child will become the new E-node. Then R will become the E-node again when the subtree C has been fully explored. This corresponds to a depth first generation of the problem states. In the second state generation method, the E-node remains the E-node until it is dead. In both methods, *bounding functions* are used to kill live nodes without generating all their children. This is done carefully enough that at the conclusion of the process at least one answer node is always generated or all answer nodes are generated if the problem requires us to find all solutions. Depth first node generation with bounding functions is called *backtracking*. State generation methods in which the E-node remains the E-node until it is dead lead to *branch-and-bound* methods. The branch-and-bound technique is discussed in Chapter 8.

The nodes of Figure 7.2 have been numbered in the order they would be generated in a depth first generation process. The nodes in Figures 7.3 and

7.4 have been numbered according to two generation methods in which the E-node remains the E-node until it is dead. In Figure 7.3 each new node is placed into a queue. When all the children of the current E-node have been generated, the next node at the front of the queue becomes the new E-node. In Figure 7.4 new nodes are placed into a stack instead of a queue. Current terminology is not uniform in referring to these two alternatives. Typically the queue method is called breadth first generation and the stack method is called D-search (depth search).

Example 7.5 [4-queens] Let us see how backtracking works on the 4-queens problem of Example 7.3. As a bounding function, we use the obvious criteria that if $(x_1, x_2, \ldots, x_i)$ is the path to the current E-node, then all children nodes with parent-child labelings x_{i+1} are such that $(x_1, \ldots, x_{i+1})$ represents a chessboard configuration in which no two queens are attacking. We start with the root node as the only live node. This becomes the E-node and the path is (). We generate one child. Let us assume that the children are generated in ascending order. Thus, node number 2 of Figure 7.2 is generated and the path is now (1). This corresponds to placing queen 1 on column 1. Node 2 becomes the E-node. Node 3 is generated and immediately killed. The next node generated is node 8 and the path becomes (1, 3). Node 8 becomes the E-node. However, it gets killed as all its children represent board configurations that cannot lead to an answer node. We backtrack to node 2 and generate another child, node 13. The path is now (1, 4). Figure 7.5 shows the board configurations as backtracking proceeds. Figure 7.5 shows graphically the steps that the backtracking algorithm goes through as it tries to find a solution. The dots indicate placements of a queen which were tried and rejected because another queen was attacking. In Figure 7.5(b) the second queen is placed on columns 1 and 2 and finally settles on column 3. In Figure 7.5(c) the algorithm tries all four columns and is unable to place the next queen on a square. Backtracking now takes place. In Figure 7.5(d) the second queen is moved to the next possible column, column 4 and the third queen is placed on column 2. The boards in Figure 7.5 (e), (f), (g), and (h) show the remaining steps that the algorithm goes through until a solution is found.

Figure 7.6 shows the part of the tree of Figure 7.2 that is generated. Nodes are numbered in the order in which they are generated. A node that gets killed as a result of the bounding function has a B under it. Contrast this tree with Figure 7.2 which contains 31 nodes. □

With this example completed, we are now ready to present a precise formulation of the backtracking process. We continue to treat backtracking in a general way. We assume that all answer nodes are to be found and not just one. Let $(x_1, x_2, \ldots, x_i)$ be a path from the root to a node in a state space tree. Let $T(x_1, x_2, \ldots, x_i)$ be the set of all possible values for x_{i+1} such that $(x_1, x_2, \ldots, x_{i+1})$ is also a path to a problem state. $T(x_1, x_2, \ldots, x_n) = \emptyset$.

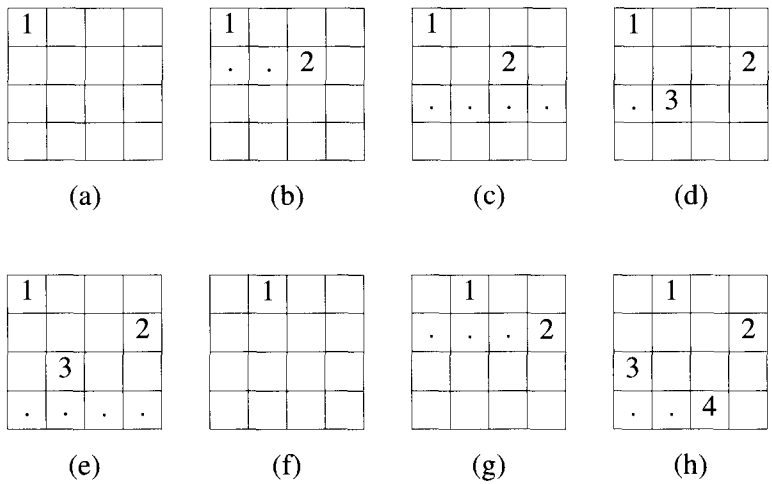


Figure 7.5 Example of a backtrack solution to the 4-queens problem

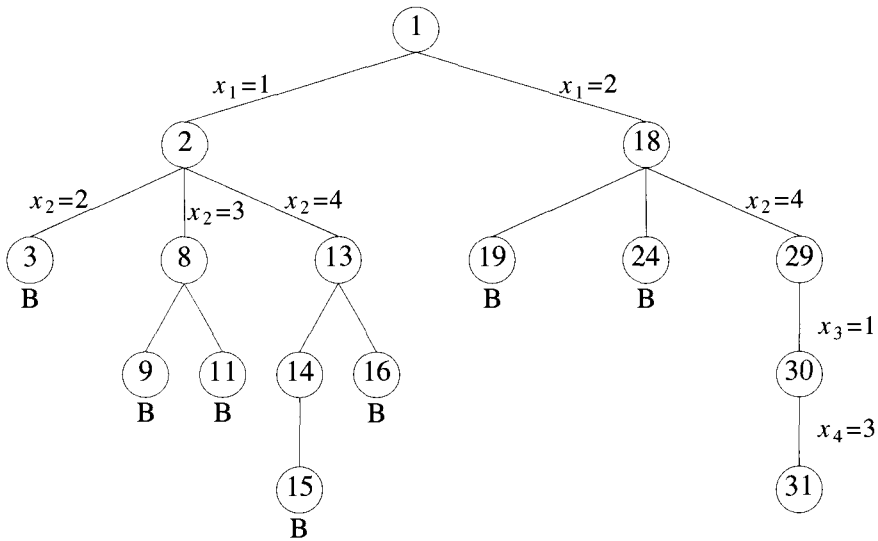


Figure 7.6 Portion of the tree of Figure 7.2 that is generated during backtracking

We assume the existence of bounding function B_{i+1} (expressed as predicates) such that if $B_{i+1}(x_1, x_2, \ldots, x_{i+1})$ is false for a path $(x_1, x_2, \ldots, x_{i+1})$ from the root node to a problem state, then the path cannot be extended to reach an answer node. Thus the candidates for position $i+1$ of the solution vector $(x_1, \ldots, x_n)$ are those values which are generated by T and satisfy B_{i+1}. Algorithm 7.1 presents a recursive formulation of the backtracking technique. It is natural to describe backtracking in this way since it is essentially a postorder traversal of a tree (see Section 6.1). This recursive version is initially invoked by

```
Backtrack(1);
```

```
1   Algorithm Backtrack(k)
2   // This schema describes the backtracking process using
3   // recursion. On entering, the first k − 1 values
4   // x[1], x[2], ..., x[k − 1] of the solution vector
5   // x[1 : n] have been assigned. x[ ] and n are global.
6   {
7       for (each x[k] ∈ T(x[1], ..., x[k − 1])) do
8       {
9           if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
10          {
11              if (x[1], x[2], ..., x[k] is a path to an answer node)
12                  then write (x[1 : k]);
13              if (k < n) then Backtrack(k + 1);
14          }
15      }
16  }
```

Algorithm 7.1 Recursive backtracking algorithm

The solution vector $(x_1, \ldots, x_n)$, is treated as a global array $x[1:n]$. All the possible elements for the kth position of the tuple that satisfy B_k are generated, one by one, and adjoined to the current vector $(x_1, \ldots, x_{k-1})$. Each time x_k is attached, a check is made to determine whether a solution has been found. Then the algorithm is recursively invoked. When the **for** loop of line 7 is exited, no more values for x_k exist and the current copy of Backtrack ends. The last unresolved call now resumes, namely, the one that continues to examine the remaining elements assuming only $k-2$ values have been set.

Note that this algorithm causes *all* solutions to be printed and assumes that tuples of various sizes may make up a solution. If only a single solution is desired, then a flag can be added as a parameter to indicate the first occurrence of success.

```
1   Algorithm IBacktrack(n)
2   // This schema describes the backtracking process.
3   // All solutions are generated in x[1 : n] and printed
4   // as soon as they are determined.
5   {
6       k := 1;
7       while (k ≠ 0) do
8       {
9           if (there remains an untried x[k] ∈ T(x[1], x[2], ...,
10              x[k − 1])  and B_k(x[1], ..., x[k]) is true) then
11          {
12                  if (x[1], ..., x[k] is a path to an answer node)
13                      then write (x[1 : k]);
14                  k := k + 1; // Consider the next set.
15          }
16          else k := k − 1; // Backtrack to the previous set.
17      }
18  }
```

Algorithm 7.2 General iterative backtracking method

An iterative version of Algorithm 7.1 appears in Algorithm 7.2. Note that $T()$ will yield the set of all possible values that can be placed as the first component x_1 of the solution vector. The component x_1 will take on those values for which the bounding function $B_1(x_1)$ is true. Also note how the elements are generated in a depth first manner. The variable k is continually incremented and a solution vector is grown until either a solution is found or no untried value of x_k remains. When k is decremented, the algorithm must resume the generation of possible elements for the kth position that have not yet been tried. Therefore one must develop a procedure that generates these values in some order. If only one solution is desired, replacing **write** $(x[1 : k])$; with **{write** $(x[1 : k])$; **return;}** suffices.

The efficiency of both the backtracking algorithms we've just seen depends very much on four factors: (1) the time to generate the next x_k, (2) the number of x_k satisfying the explicit constraints, (3) the time for the bounding functions B_k, and (4) the number of x_k satisfying the B_k. Bound-

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9                or (Abs(x[j] − i) = Abs(j − k)))
10                   // or in the same diagonal
11               then return false;
12      return true;
13  }
```

Algorithm 7.4 Can a new queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 7.5 All solutions to the n-queens problem

At this point we might wonder how effective function NQueens is over the brute force approach. For an 8×8 chessboard there are $\binom{64}{8}$ possible ways to place 8 pieces, or approximately 4.4 billion 8-tuples to examine. However, by allowing only placements of queens on distinct rows and columns, we require the examination of at most 8!, or only 40,320 8-tuples.

We can use Estimate to estimate the number of nodes that will be generated by NQueens. Note that the assumptions that are needed for Estimate do hold for NQueens. The bounding function is static. No change is made to the function as the search proceeds. In addition, all nodes on the same level of the state space tree have the same degree. In Figure 7.8 we see five 8×8 chessboards that were created using Estimate.

As required, the placement of each queen on the chessboard was chosen randomly. With each choice we kept track of the number of columns a queen could legitimately be placed on. These numbers are listed in the vector beneath each chessboard. The number following the vector represents the value that function Estimate would produce from these sizes. The average of these five trials is 1625. The total number of nodes in the 8-queens state space tree is

$$1 + \sum_{j=0}^{7} \left[\Pi_{i=0}^{j} (8 - i) \right] = 69,281$$

So the estimated number of unbounded nodes is only about 2.34% of the total number of nodes in the 8-queens state space tree. (See the exercises for more ideas about the efficiency of NQueens.)

EXERCISES

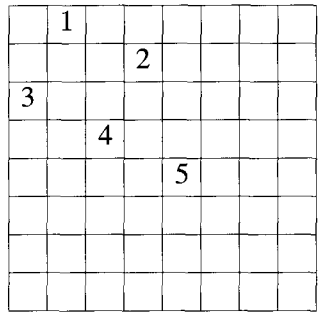
1. Algorithm NQueens can be made more efficient by redefining the function Place(k, i) so that it either returns the next legitimate column on which to place the kth queen or an illegal value. Rewrite both functions (Algorithms 7.4 and 7.5) so they implement this alternate strategy.

2. For the n-queens problem we observe that some solutions are simply reflections or rotations of others. For example, when $n = 4$, the two solutions given in Figure 7.9 are equivalent under reflection.

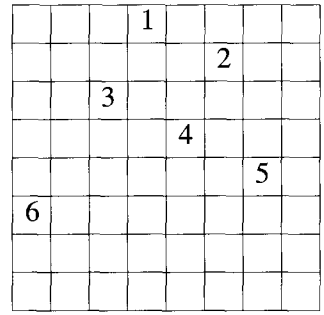
   Observe that for finding inequivalent solutions the algorithm need only set $x[1] = 2, 3, \ldots, \lceil n/2 \rceil$.

   (a) Modify NQueens so that only inequivalent solutions are computed.

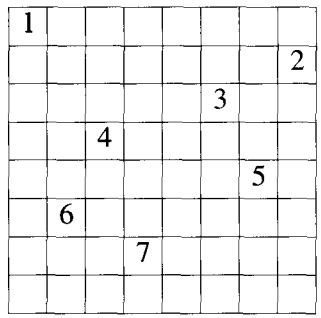
   (b) Run the n-queens program devised above for $n = 8$, 9, and 10. Tabulate the number of solutions your program finds for each value of n.



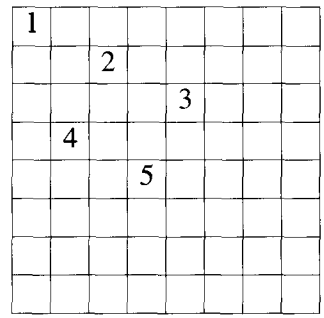
(8,5,4,3,2) = 1649



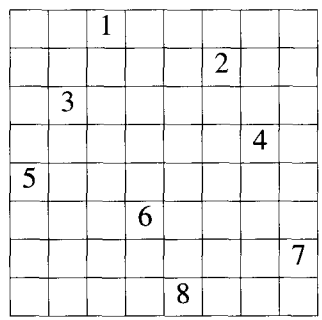
(8,5,3,1,2,1) = 769



(8,6,4,2,1,1,1) = 1401



(8,6,4,3,2) = 1977



(8,5,3,2,2,1,1,1) = 2329

Figure 7.8 Five walks through the 8-queens problem plus estimates of the tree size

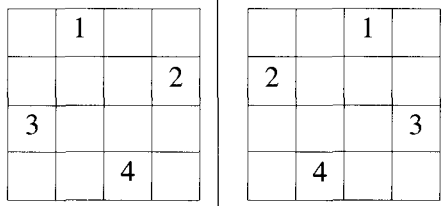


Figure 7.9 Equivalent solutions to the 4-queens problem

3. Given an $n \times n$ chessboard, a knight is placed on an arbitrary square with coordinates (x, y). The problem is to determine $n^2 - 1$ knight moves such that every square of the board is visited once if such a sequence of moves exists. Present an algorithm to solve this problem.

7.3 SUM OF SUBSETS

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem. Examples 7.2 and 7.4 showed how we could formulate this problem using either fixed- or variable-sized tuples. We consider a backtracking solution using the fixed tuple size strategy. In this case the element x_i of the solution vector is either one or zero depending on whether the weight w_i is included or not.

The children of any node in Figure 7.4 are easily generated. For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.

A simple choice for the bounding functions is $B_k(x_1, \ldots, x_k)$ = true iff

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

Clearly $x_1, \ldots, x_k$ cannot lead to an answer node if this condition is not satisfied. The bounding functions can be strengthened if we assume the w_i's are initially in nondecreasing order. In this case $x_1, \ldots, x_k$ cannot lead to an answer node if

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} > m$$

The bounding functions we use are therefore

$$B_k(x_1, \ldots, x_k) = true \text{ iff } \sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$$

$$\text{and } \sum_{i=1}^{k} w_i x_i + w_{k+1} \leq m \tag{7.1}$$

Since our algorithm will not make use of B_n, we need not be concerned by the appearance of w_{n+1} in this function. Although we have now specified all that is needed to directly use either of the backtracking schemas, a simpler algorithm results if we tailor either of these schemas to the problem at hand. This simplification results from the realization that if $x_k = 1$, then

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i > m$$

For simplicity we refine the recursive schema. The resulting algorithm is SumOfSub (Algorithm 7.6).

Algorithm SumOfSub avoids computing $\sum_{i=1}^{k} w_i x_i$ and $\sum_{i=k+1}^{n} w_i$ each time by keeping these values in variables s and r respectively. *The algorithm assumes* $w_1 \leq m$ *and* $\sum_{i=1}^{n} w_i \geq m$. The initial call is SumOfSub$(0, 1, \sum_{i=1}^{n} w_i)$. It is interesting to note that the algorithm does not explicitly use the test $k > n$ to terminate the recursion. This test is not needed as on entry to the algorithm, $s \neq m$ and $s + r \geq m$. Hence, $r \neq 0$ and so k can be no greater than n. Also note that in the **else if** statement (line 11), since $s + w_k < m$ and $s + r \geq m$, it follows that $r \neq w_k$ and hence $k + 1 \leq n$. Observe also that if $s + w_k = m$ (line 9), then $x_{k+1}, \ldots, x_n$ must be zero. These zeros are omitted from the output of line 9. In line 11 we do not test for $\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geq m$, as we already know $s + r \geq m$ and $x_k = 1$.

Example 7.6 Figure 7.10 shows the portion of the state space tree generated by function SumOfSub while working on the instance $n = 6$, $m = 30$, and $w[1 : 6] = \{5, 10, 12, 13, 15, 18\}$. The rectangular nodes list the values of s, k, and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out. At nodes A, B, and C the output is respectively (1, 1, 0, 0, 1), (1, 0, 1, 1), and (0, 0, 1, 0, 0, 1). Note that the tree of Figure 7.10 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made (this count excludes the 64 leaf nodes as no call need be made from a leaf). □

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

Algorithm 7.6 Recursive backtracking algorithm for sum of subsets problem

EXERCISES

1. Prove that the size of the set of all subsets of n elements is 2^n.

2. Let $w = \{5, 7, 10, 12, 15, 18, 20\}$ and $m = 35$. Find all possible subsets of w that sum to m. Do this using SumOfSub. Draw the portion of the state space tree that is generated.

3. With $m = 35$, run SumOfSub on the data (a) $w = \{5, 7, 10, 12, 15, 18, 20\}$, (b) $w = \{20, 18, 15, 12, 10, 7, 5\}$, and (c) $w = \{15, 7, 20, 5, 18, 10, 12\}$. Are there any discernible differences in the computing times?

4. Write a backtracking algorithm for the sum of subsets problem using the state space tree corresponding to the variable tuple size formulation.

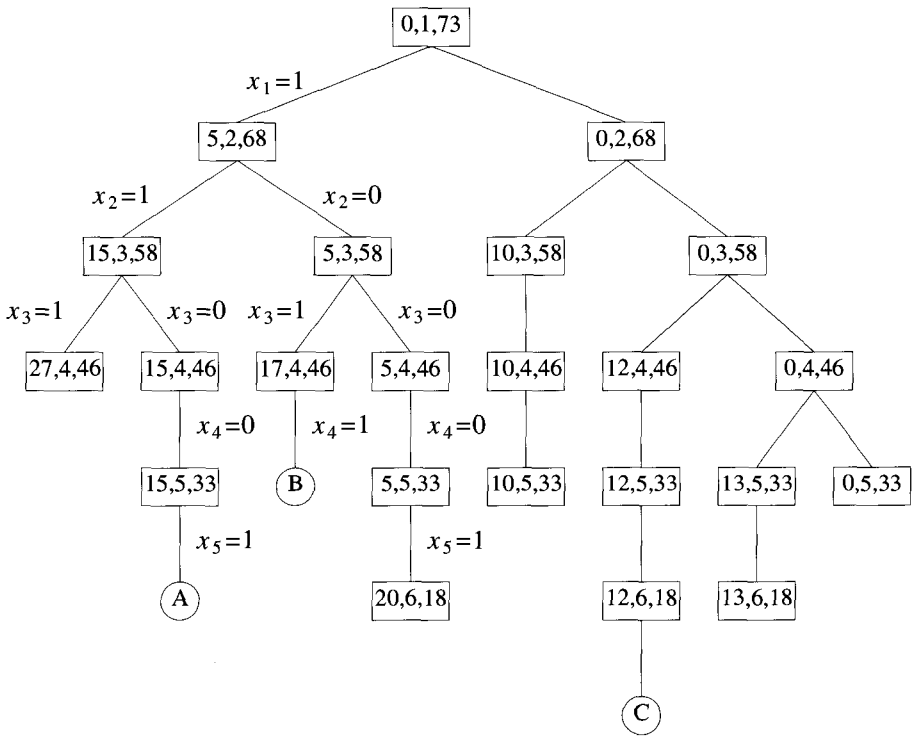


Figure 7.10 Portion of state space tree generated by SumOfSub

7.4 GRAPH COLORING

Let G be a graph and m be a given positive integer. We want to discover whether the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed the *m-colorability decision* problem and it is discussed again in Chapter 11. Note that if d is the degree of the given graph, then it can be colored with $d + 1$ colors. The *m-colorability optimization* problem asks for the smallest integer m for which the graph G can be colored. This integer is referred to as the *chromatic number* of the graph. For example, the graph of Figure 7.11 can be colored with three colors 1, 2, and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

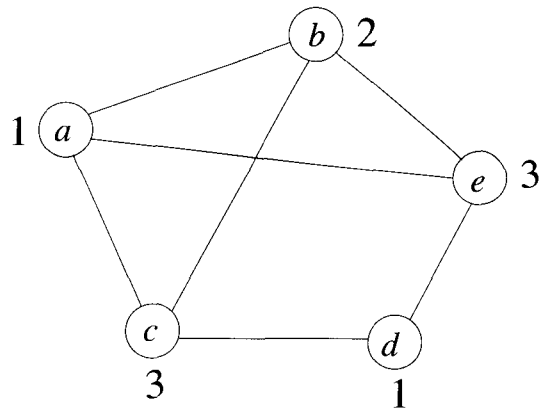


Figure 7.11 An example graph and its coloring

A graph is said to be *planar* iff it can be drawn in a plane in such a way that no two edges cross each other. A famous special case of the m-colorability decision problem is the 4-color problem for planar graphs. This problem asks the following question: given any map, can the regions be colored in such a way that no two adjacent regions have the same color yet only four colors are needed? This turns out to be a problem for which graphs are very useful, because a map can easily be transformed into a graph. Each region of the map becomes a node, and if two regions are adjacent, then the corresponding nodes are joined by an edge. Figure 7.12 shows a map with five regions and its corresponding graph. This map requires four colors. For many years it was known that five colors were sufficient to color any map, but no map that required more than four colors had ever been found. After several hundred years, this problem was solved by a group of mathematicians with the help of a computer. They showed that in fact four colors are sufficient. In this section we consider not only graphs that are produced from maps but all graphs. We are interested in determining all the different ways in which a given graph can be colored using at most m colors.

Suppose we represent a graph by its adjacency matrix $G[1 : n, 1 : n]$, where $G[i, j] = 1$ if (i, j) is an edge of G, and $G[i, j] = 0$ otherwise. The colors are represented by the integers $1, 2, \ldots, m$ and the solutions are given by the n-tuple $(x_1, \ldots, x_n)$, where x_i is the color of node i. Using the recursive backtracking formulation as given in Algorithm 7.1, the resulting algorithm is mColoring (Algorithm 7.7). The underlying state space tree used is a tree of degree m and height $n + 1$. Each node at level i has m children corresponding to the m possible assignments to x_i, $1 \leq i \leq n$. Nodes at

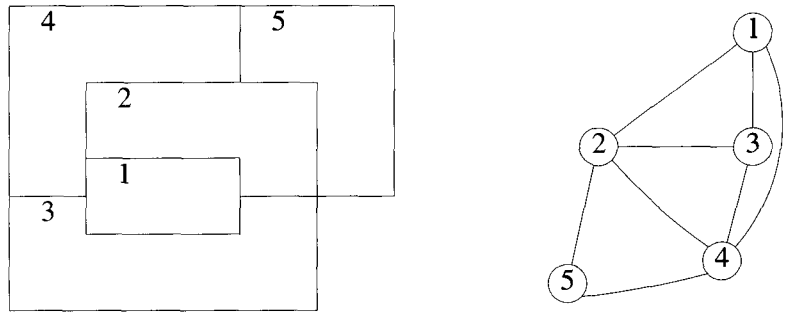


Figure 7.12 A map and its planar graph representation

level $n + 1$ are leaf nodes. Figure 7.13 shows the state space tree when $n = 3$ and $m = 3$.

Function mColoring is begun by first assigning the graph to its adjacency matrix, *setting the array* $x[\ ]$ *to zero*, and then invoking the statement mColoring(1);.

Notice the similarity between this algorithm and the general form of the recursive backtracking schema of Algorithm 7.1. Function NextValue (Algorithm 7.8) produces the possible colors for x_k after x_1 through x_{k-1} have been defined. The main loop of mColoring repeatedly picks an element from the set of possibilities, assigns it to x_k, and then calls mColoring recursively. For instance, Figure 7.14 shows a simple graph containing four nodes. Below that is the tree that is generated by mColoring. Each path to a leaf represents a coloring using at most three colors. Note that only 12 solutions exist with *exactly* three colors. In this tree, after choosing $x_1 = 2$ and $x_2 = 1$, the possible choices for x_3 are 2 and 3. After choosing $x_1 = 2$, $x_2 = 1$, and $x_3 = 2$, possible values for x_4 are 1 and 3. And so on.

An upper bound on the computing time of mColoring can be arrived at by noticing that the number of internal nodes in the state space tree is $\sum_{i=0}^{n-1} m^i$. At each internal node, $O(mn)$ time is spent by NextValue to determine the children corresponding to legal colorings. Hence the total time is bounded by $\sum_{i=0}^{n-1} m^{i+1}n = \sum_{i=1}^{n} m^i n = n(m^{n+1} - 2)/(m - 1) = O(nm^n)$.

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

Algorithm 7.7 Finding all m-colorings of a graph

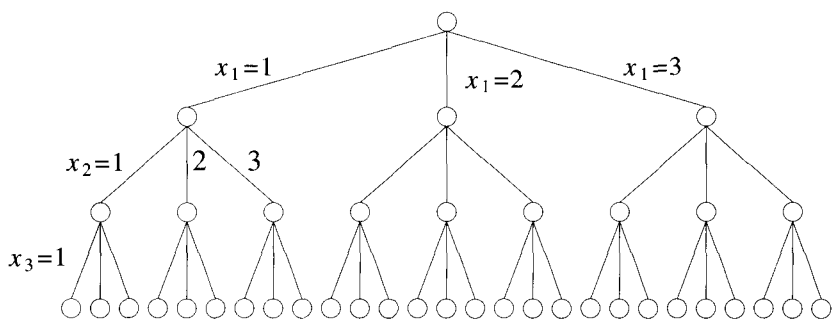


Figure 7.13 State space tree for mColoring when n =3 and m = 3

```
1   Algorithm NextValue(k)
2   // x[1], ..., x[k − 1] have been assigned integer values in
3   // the range [1, m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0, m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17              // If (k, j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

Algorithm 7.8 Generating a next color

EXERCISE

1. Program and run mColoring (Algorithm 7.7) using as data the complete graphs of size $n = 2$, 3, 4, 5, 6, and 7. Let the desired number of colors be $k = n$ and $k = n/2$. Tabulate the computing times for each value of n and k.

7.5 HAMILTONIAN CYCLES

Let $G = (V, E)$ be a connected graph with n vertices. A Hamiltonian cycle (suggested by Sir William Hamilton) is a round-trip path along n edges of G that visits every vertex once and returns to its starting position. In other words if a Hamiltonian cycle begins at some vertex $v_1 \in G$ and the vertices

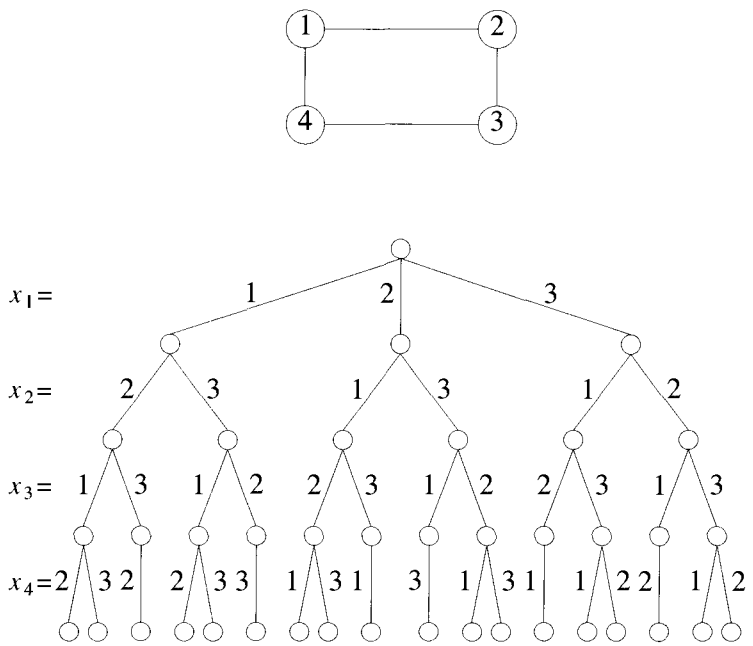


Figure 7.14 A 4-node graph and all possible 3-colorings

of G are visited in the order $v_1, v_2, \ldots, v_{n+1}$, then the edges (v_i, v_{i+1}) are in E, $1 \leq i \leq n$, and the v_i are distinct except for v_1 and v_{n+1}, which are equal.

The graph $G1$ of Figure 7.15 contains the Hamiltonian cycle 1, 2, 8, 7, 6, 5, 4, 3, 1. The graph $G2$ of Figure 7.15 contains no Hamiltonian cycle. There is no known easy way to determine whether a given graph contains a Hamiltonian cycle. We now look at a backtracking algorithm that finds all the Hamiltonian cycles in a graph. The graph may be directed or undirected. Only distinct cycles are output.

The backtracking solution vector $(x_1, \ldots, x_n)$ is defined so that x_i represents the ith visited vertex of the proposed cycle. Now all we need do is determine how to compute the set of possible vertices for x_k if $x_1, \ldots, x_{k-1}$ have already been chosen. If $k = 1$, then x_1 can be any of the n vertices. To avoid printing the same cycle n times, we require that $x_1 = 1$. If $1 < k < n$, then x_k can be any vertex v that is distinct from $x_1, x_2, \ldots, x_{k-1}$ and v is connected by an edge to x_{k-1}. The vertex x_n can only be the one remaining vertex and it must be connected to both x_{n-1} and x_1. We begin by presenting function NextValue(k) (Algorithm 7.9), which determines a possible next

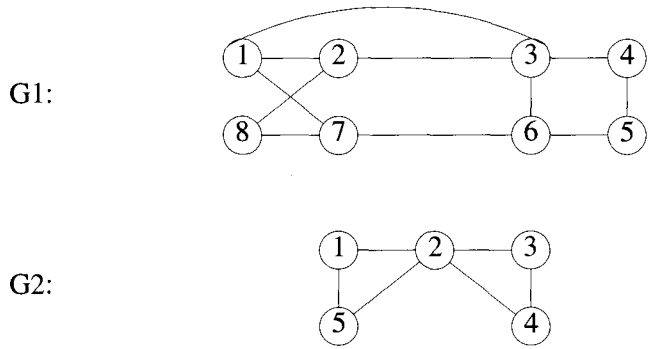


Figure 7.15 Two graphs, one containing a Hamiltonian cycle

vertex for the proposed cycle.

Using NextValue we can particularize the recursive backtracking schema to find all Hamiltonian cycles (Algorithm 7.10). This algorithm is started by first initializing the adjacency matrix $G[1:n, 1:n]$, then setting $x[2:n]$ to zero and $x[1]$ to 1, and then executing Hamiltonian(2).

Recall from Section 5.9 the traveling salesperson problem which asked for a tour that has minimum cost. This tour is a Hamiltonian cycle. For the simple case of a graph all of whose edge costs are identical, Hamiltonian will find a minimum-cost tour if a tour exists. If the common edge cost is c, the cost of a tour is cn since there are n edges in a Hamiltonian cycle.

EXERCISES

1. Determine the order of magnitude of the worst-case computing time for the backtracking procedure that finds all Hamiltonian cycles.

2. Draw the portion of the state space tree generated by Algorithm 7.10 for the graph $G1$ of Figure 7.15.

3. Generalize Hamiltonian so that it processes a graph whose edges have costs associated with them and finds a Hamiltonian cycle with minimum cost. You can assume that all edge costs are positive.

```
1   Algorithm NextValue(k)
2   // x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
3   // no vertex has as yet been assigned to x[k]. After execution,
4   // x[k] is assigned to the next highest numbered vertex which
5   // does not already appear in x[1 : k − 1] and is connected by
6   // an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
7   // in addition x[k] is connected to x[1].
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
12          if (x[k] = 0) then return;
13          if (G[x[k − 1], x[k]] ≠ 0) then
14          { // Is there an edge?
15              for j := 1 to k − 1 do if (x[j] = x[k]) then break;
16                                  // Check for distinctness.
17              if (j = k) then // If true, then the vertex is distinct.
18                  if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
19                      then return;
20          }
21      } until (false);
22  }
```

Algorithm 7.9 Generating a next vertex

```
1   Algorithm Hamiltonian(k)
2   // This algorithm uses the recursive formulation of
3   // backtracking to find all the Hamiltonian cycles
4   // of a graph. The graph is stored as an adjacency
5   // matrix G[1 : n, 1 : n]. All cycles begin at node 1.
6   {
7       repeat
8       { // Generate values for x[k].
9           NextValue(k); // Assign a legal next value to x[k].
10          if (x[k] = 0) then return;
11          if (k = n) then write (x[1 : n]);
12          else Hamiltonian(k + 1);
13      } until (false);
14  }
```

Algorithm 7.10 Finding all Hamiltonian cycles

7.6 KNAPSACK PROBLEM

In this section we reconsider a problem that was defined and solved by a dynamic programming algorithm in Chapter 5, the 0/1 knapsack optimization problem. Given n positive weights w_i, n positive profits p_i, and a positive number m that is the knapsack capacity, this problem calls for choosing a subset of the weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \quad \text{and} \quad \sum_{1 \le i \le n} p_i x_i \text{ is maximized} \tag{7.2}$$

The x_i's constitute a zero-one-valued vector.

The solution space for this problem consists of the 2^n distinct ways to assign zero or one values to the x_i's. Thus the solution space is the same as that for the sum of subsets problem. Two possible tree organizations are possible. One corresponds to the fixed tuple size formulation (Figure 7.4) and the other to the variable tuple size formulation (Figure 7.3). Backtracking algorithms for the knapsack problem can be arrived at using either of these two state space trees. Regardless of which is used, bounding functions are needed to help kill some live nodes without expanding them. A good bounding function for this problem is obtained by using an upper bound on the value of the best feasible solution obtainable by expanding the given live node and any of its descendants. If this upper bound is not higher than

the value of the best solution determined so far, then that live node can be killed.

We continue the discussion using the fixed tuple size formulation. If at node Z the values of x_i, $1 \leq i \leq k$, have already been determined, then an upper bound for Z can be obtained by relaxing the requirement $x_i = 0$ or 1 to $0 \leq x_i \leq 1$ for $k+1 \leq i \leq n$ and using the greedy algorithm of Section 4.2 to solve the relaxed problem. Function Bound(cp, cw, k) (Algorithm 7.11) determines an upper bound on the best solution obtainable by expanding any node Z at level $k+1$ of the state space tree. The object weights and profits are $w[i]$ and $p[i]$. It is assumed that $p[i]/w[i] \geq p[i+1]/w[i+1]$, $1 \leq i < n$.

```
1   Algorithm Bound(cp, cw, k)
2   // cp is the current profit total, cw is the current
3   // weight total; k is the index of the last removed
4   // item; and m is the knapsack size.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           c := c + w[i];
9           if (c < m) then b := b + p[i];
10          else return b + (1 − (c − m)/w[i]) ∗ p[i];
11      }
12      return b;
13  }
```

Algorithm 7.11 A bounding function

From Bound it follows that the bound for a feasible left child of a node Z is the same as that for Z. Hence, the bounding function need not be used whenever the backtracking algorithm makes a move to the left child of a node. The resulting algorithm is BKnap (Algorithm 7.12). It was obtained from the recursive backtracking schema. Initially set $fp := -1$;. This algorithm is invoked as

```
BKnap(1, 0, 0);
```

When $fp \neq -1$, $x[i]$, $1 \leq i \leq n$, is such that $\sum_{i=1}^{n} p[i]x[i] = fp$. In lines 8 to 18 left children are generated. In line 20, Bound is used to test whether a

```
1   Algorithm BKnap(k, cp, cw)
2   // m is the size of the knapsack; n is the number of weights
3   // and profits. w[ ] and p[ ] are the weights and profits.
4   // p[i]/w[i] ≥ p[i + 1]/w[i + 1]. fw is the final weight of
5   // knapsack; fp is the final maximum profit. x[k] = 0 if w[k]
6   // is not in the knapsack; else x[k] = 1.
7   {
8       // Generate left child.
9       if (cw + w[k] ≤ m) then
10      {
11          y[k] := 1;
12          if (k < n) then BKnap(k + 1, cp + p[k], cw + w[k]);
13          if ((cp + p[k] > fp) and (k = n)) then
14          {
15              fp := cp + p[k]; fw := cw + w[k];
16              for j := 1 to k do x[j] := y[j];
17          }
18      }
19      // Generate right child.
20      if (Bound(cp, cw, k) ≥ fp) then
21      {
22          y[k] := 0; if (k < n) then BKnap(k + 1, cp, cw);
23          if ((cp > fp) and (k = n)) then
24          {
25              fp := cp; fw := cw;
26              for j := 1 to k do x[j] := y[j];
27          }
28      }
29  }
```

Algorithm 7.12 Backtracking solution to the 0/1 knapsack problem

right child should be generated. The path $y[i]$, $1 \leq i \leq k$, is the path to the current node. The current weight $cw = \sum_{i=1}^{k-1} w[i]y[i]$ and $cp = \sum_{i=1}^{k-1} p[i]y[i]$. In lines 13 to 17 and 23 to 27 the solution vector is updated if need be.

So far, all our backtracking algorithms have worked on a static state space tree. We now see how a dynamic state space tree can be used for the knapsack problem. One method for dynamically partitioning the solution space is based on trying to obtain an optimal solution using the greedy algorithm of Section 4.2. We first replace the integer constraint $x_i = 0$ or 1 by the constraint $0 \leq x_i \leq 1$. This yields the relaxed problem

$$\max \sum_{1 \leq i \leq n} p_i x_i \quad \text{subject to} \quad \sum_{1 \leq i \leq n} w_i x_i \leq m \tag{7.3}$$

$$0 \leq x_i \leq 1, \qquad 1 \leq i \leq n$$

If the solution generated by the greedy method has all x_i's equal to zero or one, then it is also an optimal solution to the original 0/1 knapsack problem. If this is not the case, then exactly one x_i will be such that $0 < x_i < 1$. We partition the solution space of (7.2) into two subspaces. In one $x_i = 0$ and in the other $x_i = 1$. Thus the left subtree of the state space tree will correspond to $x_i = 0$ and the right to $x_i = 1$. In general, at each node Z of the state space tree the greedy algorithm is used to solve (7.3) under the added restrictions corresponding to the assignments already made along the path from the root to this node. In case the solution is all integer, then an optimal solution for this node has been found. If not, then there is exactly one x_i such that $0 < x_i < 1$. The left child of Z corresponds to $x_i = 0$, and the right to $x_i = 1$.

The justification for this partitioning scheme is that the noninteger x_i is what prevents the greedy solution from being a feasible solution to the 0/1 knapsack problem. So, we would expect to reach a feasible greedy solution quickly by forcing this x_i to be integer. Choosing left branches to correspond to $x_i = 0$ rather than $x_i = 1$ is also justifiable. Since the greedy algorithm requires $p_j/w_j \geq p_{j+1}/w_{j+1}$, we would expect most objects with low index (i.e., small j and hence high density) to be in an optimal filling of the knapsack. When x_i is set to zero, we are not preventing the greedy algorithm from using any of the objects with $j < i$ (unless x_j has already been set to zero). On the other hand, when x_i is set to one, some of the x_j's with $j < i$ will not be able to get into the knapsack. Therefore we expect to arrive at an optimal solution with $x_i = 0$. So we wish the backtracking algorithm to try this alternative first. Hence the left subtree corresponds to $x_i = 0$.

Example 7.7 Let us try out a backtracking algorithm and the above dynamic partitioning scheme on the following data: $p = \{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. The greedy

solution corresponding to the root node (i.e., Equation (7.3)) is $x = \{1, 1, 1, 1, 1, 21/45, 0, 0\}$. Its value is 164.88. The two subtrees of the root correspond to $x_6 = 0$ and $x_6 = 1$, respectively (Figure 7.16). The greedy solution at node 2 is $x = \{1, 1, 1, 1, 1, 0, 21/45, 0\}$. Its value is 164.66. The solution space at node 2 is partitioned using $x_7 = 0$ and $x_7 = 1$. The next E-node is node 3. The solution here has $x_8 = 21/55$. The partitioning now is with $x_8 = 0$ and $x_8 = 1$. The solution at node 4 is all integer so there is no need to expand this node further. The best solution found so far has value 139 and $x = \{1, 1, 1, 1, 1, 0, 0, 0\}$. Node 5 is the next E-node. The greedy solution for this node is $x = \{1, 1, 1, 22/23, 0, 0, 0, 1\}$. Its value is 159.56. The partitioning is now with $x_4 = 0$ and $x_4 = 1$. The greedy solution at node 6 has value 156.66 and $x_5 = 2/3$. Next, node 7 becomes the E-node. The solution here is $\{1, 1, 1, 0, 0, 0, 0, 1\}$. Its value is 128. Node 7 is not expanded as the greedy solution here is all integer. At node 8 the greedy solution has value 157.71 and $x_3 = 4/7$. The solution at node 9 is all integer and has value 140. The greedy solution at node 10 is $\{1, 0, 1, 0, 1, 0, 0, 1\}$. Its value is 150. The next E-node is 11. Its value is 159.52 and $x_3 = 20/21$. The partitioning is now on $x_3 = 0$ and $x_3 = 1$. The remainder of the backtracking process on this knapsack instance is left as an exercise. □

Experimental work due to E. Horowitz and S. Sahni, cited in the references, indicates that backtracking algorithms for the knapsack problem generally work in less time when using a static tree than when using a dynamic tree. The dynamic partitioning scheme is, however, useful in the solution of integer linear programs. The general integer linear program is mathematically stated in (7.4).

$$\begin{aligned} &\text{minimize} \quad \textstyle\sum_{1 \le j \le n} c_j \, x_j \\ &\text{subject to} \quad \textstyle\sum_{1 \le j \le n} a_{ij} \, x_j \le b_i, \quad 1 \le i \le m \\ &\qquad x_j's \text{ are nonnegative integers} \end{aligned} \tag{7.4}$$

If the integer constraints on the x_i's in (7.4) are replaced by the constraint $x_i \ge 0$, then we obtain a linear program whose optimal solution has a value at least as large as the value of an optimal solution to (7.4). Linear programs can be solved using the simplex methods (see the references). If the solution is not all integer, then a noninteger x_i is chosen to partition the solution space. Let us assume that the value of x_i in the optimal solution to the linear program corresponding to any node Z in the state space is v and v is not an integer. The left child of Z corresponds to $x_i \le \lfloor v \rfloor$ whereas the right child of Z correspond to $x_i \ge \lceil v \rceil$. Since the resulting state space tree has a potentially infinite depth (note that on the path from the root to a node Z

the solution space can be partitioned on one x_i many times as each x_i can have as value any nonnegative integer), it is almost always searched using a branch-and-bound method (see Chapter 8).

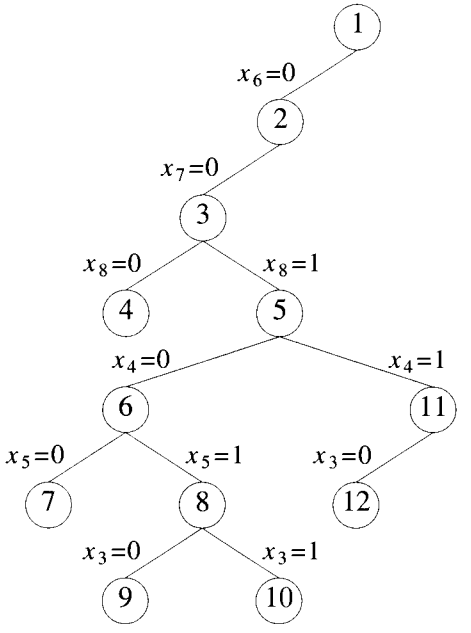


Figure 7.16 Part of the dynamic state space tree generated in Example 7.7

EXERCISES

1. (a) Present a backtracking algorithm for solving the knapsack optimization problem using the variable tuple size formulation.
   (b) Draw the portion of the state space tree your algorithm will generate when solving the knapsack instance of Example 7.7.
2. Complete the state space tree of Figure 7.16.
3. Give a backtracking algorithm for the knapsack problem using the dynamic state space tree discussed in this section.
4. [Programming project] (a) Program the algorithms of Exercises 1 and 3. Run these two programs and BKnap using the following data: $p =$

$\{11, 21, 31, 33, 43, 53, 55, 65\}$, $w = \{1, 11, 21, 23, 33, 43, 45, 55\}$, $m = 110$, and $n = 8$. Which algorithm do you expect to perform best?

(b) Now program the dynamic programming algorithm of Section 5.7 for the knapsack problem. Use the heuristics suggested at the end of Section 5.7. Obtain computing times and compare this program with the backtracking programs.

5. (a) Obtain a knapsack instance for which more nodes are generated by the backtracking algorithm using a dynamic tree than using a static tree.

   (b) Obtain a knapsack instance for which more nodes are generated by the backtracking algorithm using a static tree than using a dynamic tree.

   (c) Strengthen the backtracking algorithms with the following heuristic: Build an array $minw[\ ]$ with the property that $minw[i]$ is the index of the object that has least weight among objects $i, i+1, \ldots, n$. Now any E-node at which decisions for $x_1, \ldots, x_{i-1}$ have been made and at which the unutilized knapsack capacity is less than $w[minw[i]]$ can be terminated provided the profit earned up to this node is no more than the maximum determined so far. Incorporate this into your programs of Exercise 4(a). Rerun the new programs on the same data sets and see what (if any) improvements result.

7.7 REFERENCES AND READINGS

An early modern account of backtracking was given by R. J. Walker. The technique for estimating the efficiency of a backtrack program was first proposed by M. Hall and D. E. Knuth and the dynamic partitioning scheme for the 0/1 knapsack problem was proposed by H. Greenberg and R. Hegerich. Experimental results showing static trees to be superior for this problem can be found in "Computing partitions with applications to the knapsack problem," by E. Horowitz and S. Sahni, *Journal of the ACM* 21, no. 2 (1974): 277–292.

Data presented in the above paper shows that the divide-and-conquer dynamic programming algorithm for the knapsack problem is superior to BKnap.

For a proof of the four-color theorem see *Every Planar Map is Four Colorable*, by K. I. Appel, American Mathematical Society, Providence, RI, 1989.

A discussion of the simplex method for solving linear programs may be found in:

Linear Programming: An Introduction with Applications, by A. Sultan, Academic Press, 1993.

Linear Optimization and Extensions, by M. Padberg, Springer-Verlag, 1995.

7.8 ADDITIONAL EXERCISES

1. Suppose you are given n men and n women and two $n \times n$ arrays P and Q such that $P(i,j)$ is the preference of man i for woman j and $Q(i,j)$ is the preference of woman i for man j. Given an algorithm that finds a pairing of men and women such that the sum of the product of the preferences is maximized.

2. Let $A(1:n, 1:n)$ be an $n \times n$ matrix. The *determinant* of A is the number

$$\det(A) = \sum_s \text{sgn}(s) a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}$$

where the sum is taken over all permutations $s(1), \ldots, s(n)$ of $\{1, 2, \ldots, n\}$ and $\text{sgn}(s)$ is $+1$ or -1 according to whether s is an even or odd permutation. The *permanent* of A is defined as

$$\text{per}(A) = \sum_s a_{1,s(1)} a_{2,s(2)} \cdots a_{n,s(n)}$$

The determinant can be computed as a by-product of Gaussian elimination requiring $O(n^3)$ operations, but no polynomial time algorithm is known for computing permanents. Write an algorithm that computes the permanent of a matrix by generating the elements of s using backtracking. Analyze the time of your algorithm.

3. Let MAZE$(1:n, 1:n)$ be a zero- or one-valued, two-dimensional array that represents a maze. A one means a blocked path whereas a zero stands for an open position. You are to develop an algorithm that begins at MAZE(1, 1) and tries to find a path to position MAZE(n, n). Once again backtracking is necessary here. See if you can analyze the time complexity of your algorithm.

4. The *assignment problem* is usually stated this way: There are n people to be assigned to n jobs. The cost of assigning the ith person to the jth job is $cost(i,j)$. You are to develop an algorithm that assigns every job to a person and at the same time minimizes the total cost of the assignment.

5. This problem is called the postage stamp problem. Envision a country that issues n different denominations of stamps but allows no more than m stamps on a single letter. For given values of m and n, write an algorithm that computes the greatest consecutive range of postage values, from one on up, and all possible sets of denominations that realize that range. For example, for $n = 4$ and $m = 5$, the stamps with values (1, 4, 12, 21) allow the postage values 1 through 71. Are there any other sets of four denominations that have the same range?

6. Here is a game called Hi-Q. Thirty-two pieces are arranged on a board as shown in Figure 7.17. Only the center position is unoccupied. A piece is only allowed to move by jumping over one of its neighbors into an empty space. Diagonal jumps are not permitted. When a piece is jumped, it is removed from the board. Write an algorithm that determines a series of jumps so that all the pieces except one are eventually removed and that final piece ends up at the center position.

7. Imagine a set of 12 plane figures each composed of five equal-size squares. Each figure differs in shape from the others, but together they can be arranged to make different-sized rectangles. In Figure 7.18 there is a picture of 12 pentominoes that are joined to create a 6×10 rectangle. Write an algorithm that finds all possible ways to place the pentominoes so that a 6×10 rectangle is formed.

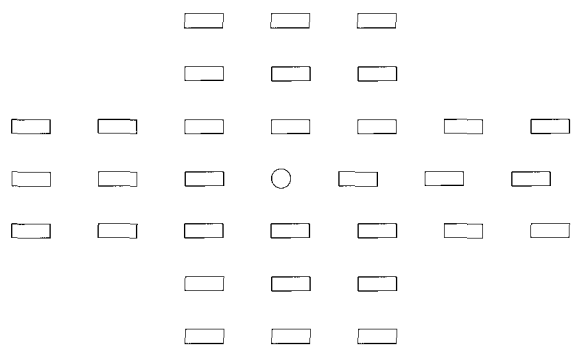


Figure 7.17 A Hi-Q board in its initial state

8. Suppose a set of electric components such as transistors are to be placed on a circuit board. We are given a connection matrix CONN, where CONN(i, j) equals the number of connections between component i and component j, and a matrix DIST, where DIST (r, s) is

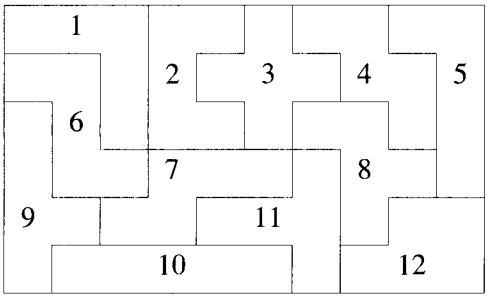


Figure 7.18 A pentomino configuration

the distance between position r and position s on the circuit board. The wiring of the board consists of placing each of n components at some location. The cost of a wiring is the sum of the products of $CONN(i,j) * DIST(r,s)$, where component i is placed at location r and component j is placed at location s. Compose an algorithm that finds an assignment of components to locations that minimizes the total cost of the wiring.

9. Suppose there are n jobs to be executed but only k processors that can work in parallel. The time required by job i is t_i. Write an algorithm that determines which jobs are to be run on which processors and the order in which they should be run so that the finish time of the last job is minimized.

10. Two graphs $G(V,E)$ and $H(A,B)$ are called *isomorphic* if there is a one-to-one onto correspondence of the vertices that preserves the adjacency relationships. More formally if f is a function from V to A and (v,w) is an edge in E, then $(f(v),f(w))$ is an edge in H. Figure 7.19 shows two directed graphs that are isomorphic under the mapping that $1,2,3,4$, and 5 and go to a,b,c,d, and e. A brute force algorithm to test two graphs for isomorphism would try out all $n!$ possible correspondences and then test to see whether adjacency was preserved. A backtracking algorithm can do better than this by applying some obvious pruning to the resultant state space tree. First of all we know that for a correspondence to exist between two vertices, they must have the same degree. So we can select at an early stage vertices of degree k for which the second graph has the fewest number of vertices of degree k. This exercise calls for devising an isomorphism algorithm that is based on backtracking and makes use of these ideas.

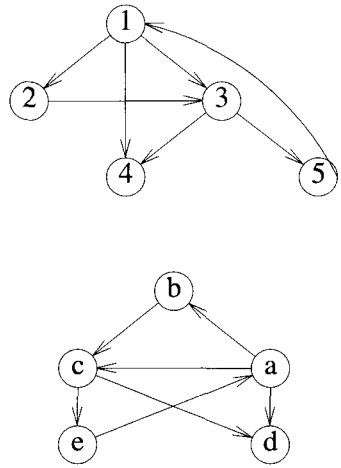


Figure 7.19 Two isomorphic graphs (Exercise 10)

11. A graph is called *complete* if all its vertices are connected to all the other vertices in the graph. A maximal complete subgraph of a graph is called a *clique*. By "maximal" we mean that this subgraph is contained within no other subgraph that is also complete. A clique of size k has $\binom{k}{i}$ subcliques of size i, $1 \leq i \leq k$. This implies that any algorithm that looks for a maximal clique must be careful to generate each subclique the fewest number of times possible. One way to generate the clique is to extend a clique of size m to size $m + 1$ and to continue this process by trying out all possible vertices. But this strategy generates the same clique many times; this can be avoided as follows. Given a clique X, suppose node v is the first node that is added to produce a clique of size one greater. After the backtracking process examines all possible cliques that are produced from X and v, then no vertex adjacent to v need be added to X and examined. Let X and Y be cliques and let X be properly contained in Y. If all cliques containing X and vertex v have been generated, then all cliques with Y and v can be ignored. Write a backtracking algorithm that generates the maximal cliques of an undirected graph and makes use of these last rules for pruning the state space tree.

Chapter 8

BRANCH-AND-BOUND

8.1 THE METHOD

This chapter makes extensive use of terminology defined in Section 7.1. The reader is urged to review this section before proceeding.

The term branch-and-bound refers to all state space search methods in which all children of the E-node are generated before any other live node can become the E-node. We have already seen (in Section 7.1) two graph search strategies, BFS and D-search, in which the exploration of a new node cannot begin until the node currently being explored is fully explored. Both of these generalize to branch-and-bound strategies. In branch-and-bound terminology, a BFS-like state space search will be called FIFO (**F**irst **I**n **F**irst **O**ut) search as the list of live nodes is a first-in-first-out list (or queue). A D-search-like state space search will be called LIFO (**L**ast **I**n **F**irst **O**ut) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

Example 8.1 [4-queens] Let us see how a FIFO branch-and-bound algorithm would search the state space tree (Figure 7.2) for the 4-queens problem. Initially, there is only one live node, node 1. This represents the case in which no queen has been placed on the chessboard. This node becomes the E-node. It is expanded and its children, nodes 2, 18, 34, and 50, are generated. These nodes represent a chessboard with queen 1 in row 1 and columns 1, 2, 3, and 4 respectively. The only live nodes now are nodes 2, 18, 34, and 50. If the nodes are generated in this order, then the next E-node is node 2. It is expanded and nodes 3, 8, and 13 are generated. Node 3 is immediately killed using the bounding function of Example 7.5. Nodes 8 and 13 are added to the queue of live nodes. Node 18 becomes the next E-node. Nodes 19, 24, and 29 are generated. Nodes 19 and 24 are killed as a result of the bounding functions. Node 29 is added to the queue of live

nodes. The E-node is node 34. Figure 8.1 shows the portion of the tree of Figure 7.2 that is generated by a FIFO branch-and-bound search. Nodes that are killed as a result of the bounding functions have a "B" under them. Numbers inside the nodes correspond to the numbers in Figure 7.2. Numbers outside the nodes give the order in which the nodes are generated by FIFO branch-and-bound. At the time the answer node, node 31, is reached, the only live nodes remaining are nodes 38 and 54. A comparison of Figures 7.6 and 8.1 indicates that backtracking is a superior search method for this problem. □

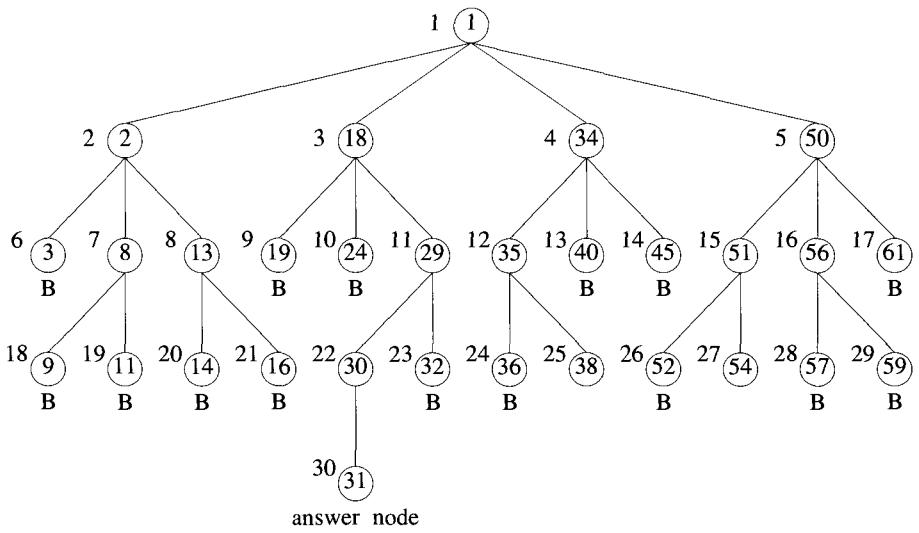


Figure 8.1 Portion of 4-queens state space tree generated by FIFO branch-and-bound

8.1.1 Least Cost (LC) Search

In both LIFO and FIFO branch-and-bound the selection rule for the next E-node is rather rigid and in a sense blind. The selection rule for the next E-node does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. Thus, in Example 8.1, when node 30 is generated, it should have become obvious to the search algorithm that this node will lead to an answer node in one move. However, the rigid FIFO rule first requires the expansion of all live nodes generated before node 30 was expanded.

The search for an answer node can often be speeded by using an "intelligent" ranking function $\hat{c}(\cdot)$ for live nodes. The next E-node is selected on the basis of this ranking function. If in the 4-queens example we use a ranking function that assigns node 30 a better rank than all other live nodes, then node 30 will become the E-node following node 29. The remaining live nodes will never become E-nodes as the expansion of node 30 results in the generation of an answer node (node 31).

The ideal way to assign ranks would be on the basis of the additional computational effort (or cost) needed to reach an answer node from the live node. For any node x, this cost could be (1) the number of nodes in the subtree x that need to be generated before an answer node is generated or, more simply, (2) the number of levels the nearest answer node (in the subtree x) is from x. Using cost measure 2, the cost of the root of the tree of Figure 8.1 is 4 (node 31 is four levels from node 1). The costs of nodes 18 and 34, 29 and 35, and 30 and 38 are respectively 3, 2, and 1. The costs of all remaining nodes on levels 2, 3, and 4 are respectively greater than 3, 2, and 1. Using these costs as a basis to select the next E-node, the E-nodes are nodes 1, 18, 29, and 30 (in that order). The only other nodes to get generated are nodes 2, 34, 50, 19, 24, 32, and 31. It should be easy to see that if cost measure 1 is used, then the search would always generate the minimum number of nodes every branch-and-bound type algorithm must generate. If cost measure 2 is used, then the only nodes to become E-nodes are the nodes on the path from the root to the nearest answer node. The difficulty with using either of these ideal cost functions is that computing the cost of a node usually involves a search of the subtree x for an answer node. Hence, by the time the cost of a node is determined, that subtree has been searched and there is no need to explore x again. For this reason, search algorithms usually rank nodes only on the basis of an estimate $\hat{g}(\cdot)$ of their cost.

Let $\hat{g}(x)$ be an estimate of the additional effort needed to reach an answer node from x. Node x is assigned a rank using a function $\hat{c}(\cdot)$ such that $\hat{c}(x) = f(h(x)) + \hat{g}(x)$, where $h(x)$ is the cost of reaching x from the root and $f(\cdot)$ is any nondecreasing function. At first, we may doubt the usefulness of using an $f(\cdot)$ other than $f(h(x)) = 0$ for all $h(x)$. We can justify such an $f(\cdot)$ on the grounds that the effort already expended in reaching the live nodes cannot be reduced and all we are concerned with now is minimizing the additional effort we spend to find an answer node. Hence, the effort already expended need not be considered.

Using $f(\cdot) \equiv 0$ usually biases the search algorithm to make deep probes into the search tree. To see this, note that we would normally expect $\hat{g}(y) \leq \hat{g}(x)$ for y, a child of x. Hence, following x, y will become the E-node, then one of y's children will become the E-node, next one of y's grandchildren will become the E-node, and so on. Nodes in subtrees other than the subtree x will not get generated until the subtree x is fully searched. This would not

be a cause for concern if $\hat{g}(x)$ were the true cost of x. Then, we would not wish to explore the remaining subtrees in any case (as x is guaranteed to get us to an answer node quicker than any other existing live node). However, $\hat{g}(x)$ is only an estimate of the true cost. So, it is quite possible that for two nodes w and z, $\hat{g}(w) < \hat{g}(z)$ and z is much closer to an answer node than w. It is therefore desirable not to overbias the search algorithm in favor of deep probes. By using $f(\cdot) \not\equiv 0$, we can force the search algorithm to favor a node z close to the root over a node w which is many levels below z. This would reduce the possibility of deep and fruitless searches into the tree.

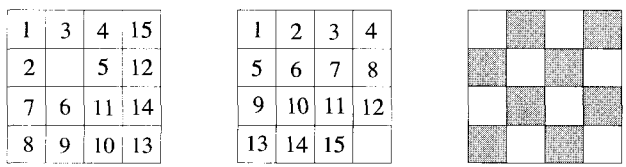
A search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next E-node would always choose for its next E-node a live node with least $\hat{c}(\cdot)$. Hence, such a search strategy is called an LC-search (**L**east **C**ost search). It is interesting to note that BFS and D-search are special cases of LC-search. If we use $\hat{g}(x) \equiv 0$ and $f(h(x)) =$ level of node x, then a LC-search generates nodes by levels. This is essentially the same as a BFS. If $f(h(x)) \equiv 0$ and $\hat{g}(x) \geq \hat{g}(y)$ whenever y is a child of x, then the search is essentially a D-search. An LC-search coupled with bounding functions is called an LC branch-and-bound search.

In discussing LC-searches, we sometimes make reference to a cost function $c(\cdot)$ defined as follows: if x is an answer node, then $c(x)$ is the cost (level, computational difficulty, etc.) of reaching x from the root of the state space tree. If x is not an answer node, then $c(x) = \infty$ providing the subtree x contains no answer node; otherwise $c(x)$ equals the cost of a minimum-cost answer node in the subtree x. It should be easy to see that $\hat{c}(\cdot)$ with $f(h(x)) = h(x)$ is an approximation to $c(\cdot)$. From now on $c(x)$ is referred to as the cost of x.

8.1.2 The 15-puzzle: An Example

The 15-puzzle (invented by Sam Loyd in 1878) consists of 15 numbered tiles on a square frame with a capacity of 16 tiles (Figure 8.2). We are given an initial arrangement of the tiles, and the objective is to transform this arrangement into the goal arrangement of Figure 8.2(b) through a series of legal moves. The only legal moves are ones in which a tile adjacent to the empty spot (ES) is moved to ES. Thus from the initial arrangement of Figure 8.2(a), four moves are possible. We can move any one of the tiles numbered 2, 3, 5, or 6 to the empty spot. Following this move, other moves can be made. Each move creates a new arrangement of the tiles. These arrangements are called the *states* of the puzzle. The initial and goal arrangements are called the initial and goal states. A state is reachable from the initial state iff there is a sequence of legal moves from the initial state to this state. The state space of an initial state consists of all states that can be reached from the initial state. The most straightforward way to solve the puzzle would be to search the state space for the goal state and use the

path from the initial state to the goal state as the answer. It is easy to see that there are 16! ($16! \approx 20.9 \times 10^{12}$) different arrangements of the tiles on the frame. Of these only one-half are reachable from any given initial state. Indeed, the state space for the problem is very large. Before attempting to search this state space for the goal state, it would be worthwhile to determine whether the goal state is reachable from the initial state. There is a very simple way to do this. Let us number the frame positions 1 to 16. Position i is the frame position containing tile numbered i in the goal arrangement of Figure 8.2(b). Position 16 is the empty spot. Let $position(i)$ be the position number in the initial state of the tile numbered i. Then $position(16)$ will denote the position of the empty spot.



(a) An arrangement (b) Goal arrangement (c)

Figure 8.2 15-puzzle arrangements

For any state let $less(i)$ be the number of tiles j such that $j < i$ and $position(j) > position(i)$. For the state of Figure 8.2(a) we have, for example, $less(1) = 0$, $less(4) = 1$, and $less(12) = 6$. Let x =1 if in the initial state the empty spot is at one of the shaded positions of Figure 8.2(c) and $x = 0$ if it is at one of the remaining positions. Then, we have the following theorem:

Theorem 8.1 The goal state of Figure 8.2(b) is reachable from the initial state iff $\sum_{i=1}^{16} less(i) + x$ is even.

Proof: Left as an exercise. □

Theorem 8.1 can be used to determine whether the goal state is in the state space of the initial state. If it is, then we can proceed to determine a sequence of moves leading to the goal state. To carry out this search, the state space can be organized into a tree. The children of each node x in this tree represent the states reachable from state x by one legal move. It is convenient to think of a move as involving a move of the empty space rather than a move of a tile. The empty space, on each move, moves either up, right, down, or left. Figure 8.3 shows the first three levels of the state

space tree of the 15-puzzle beginning with the initial state shown in the root. Parts of levels 4 and 5 of the tree are also shown. The tree has been pruned a little. *No node p has a child state that is the same as p's parent.* The subtree eliminated in this way is already present in the tree and has root *parent(p)*. As can be seen, there is an answer node at level 4.

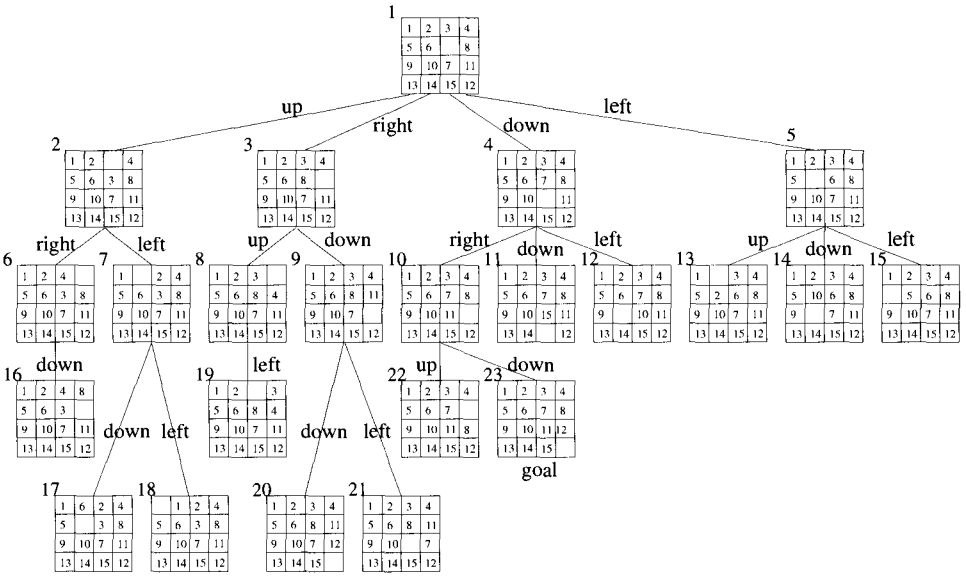


Figure 8.3 Part of the state space tree for the 15-puzzle

A depth first state space tree generation will result in the subtree of Figure 8.4 when the next moves are attempted in the order: move the empty space up, right, down, and left. Successive board configurations reveal that each move gets us farther from the goal rather than closer. The search of the state space tree is blind. It will take the leftmost path from the root regardless of the starting configuration. As a result, an answer node may never be found (unless the leftmost path ends in such a node). In a FIFO search of the tree of Figure 8.3, the nodes will be generated in the order numbered. A breadth first search will always find a goal node nearest to the root. However, such a search is also blind in the sense that no matter what the initial configuration, the algorithm attempts to make the same sequence of moves. A FIFO search always generates the state space tree by levels.

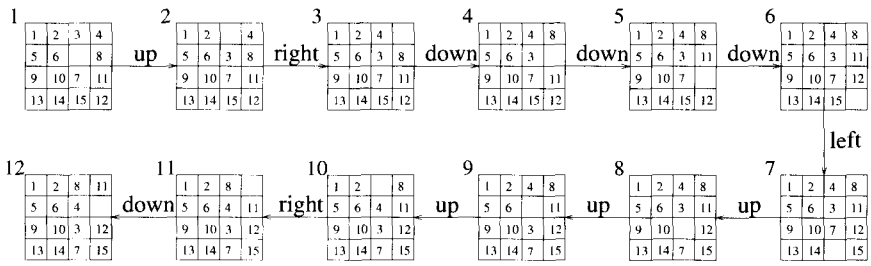


Figure 8.4 First ten steps in a depth first search

What we would like, is a more "intelligent" search method, one that seeks out an answer node and adapts the path it takes through the state space tree to the specific problem instance being solved. We can associate a cost $c(x)$ with each node x in the state space tree. The cost $c(x)$ is the length of a path from the root to a nearest goal node (if any) in the subtree with root x. Thus, in Figure 8.3, $c(1) = c(4) = c(10) = c(23) = 3$. When such a cost function is available, a very efficient search can be carried out. We begin with the root as the E-node and generate a child node with $c()$-value the same as the root. Thus children nodes 2, 3, and 5 are eliminated and only node 4 becomes a live node. This becomes the next E-node. Its first child, node 10, has $c(10) = c(4) = 3$. The remaining children are not generated. Node 4 dies and node 10 becomes the E-node. In generating node 10's children, node 22 is killed immediately as $c(22) > 3$. Node 23 is generated next. It is a goal node and the search terminates. In this search strategy, the only nodes to become E-nodes are nodes on the path from the root to a nearest goal node. Unfortunately, this is an impractical strategy as it is not possible to easily compute the function $c(\cdot)$ specified above.

We can arrive at an easy to compute estimate $\hat{c}(x)$ of $c(x)$. We can write $\hat{c}(x) = f(x) + \hat{g}(x)$, where $f(x)$ is the length of the path from the root to node x and $\hat{g}(x)$ is an estimate of the length of a shortest path from x to a goal node in the subtree with root x. One possible choice for $\hat{g}(x)$ is

$$\hat{g}(x) = \text{number of nonblank tiles not in their goal position}$$

Clearly, at least $\hat{g}(x)$ moves have to be made to transform state x to a goal state. More than $\hat{g}(x)$ moves may be needed to achieve this. To see this, examine the problem state of Figure 8.5. There $\hat{g}(x) = 1$ as only tile 7 is not in its final spot (the count for $\hat{g}(x)$ excludes the blank tile). However, the number of moves needed to reach the goal state is many more than $\hat{g}(x)$. So $\hat{c}(x)$ is a *lower bound* on the value of $c(x)$.

An LC-search of Figure 8.3 using $\hat{c}(x)$ will begin by using node 1 as the E-node. All its children are generated. Node 1 dies and leaves behind the live nodes 2, 3, 4, and 5. The next node to become the E-node is a live node with least $\hat{c}(x)$. Then $\hat{c}(2) = 1+4$, $\hat{c}(3) = 1+4$, $\hat{c}(4) = 1+2$, and $\hat{c}(5) = 1+4$. Node 4 becomes the E-node. Its children are generated. The live nodes at this time are 2, 3, 5, 10, 11, and 12. So $\hat{c}(10) = 2+1$, $\hat{c}(11) = 2+3$, and $\hat{c}(12) = 2+3$. The live node with least $\hat{c}$ is node 10. This becomes the next E-node. Nodes 22 and 23 are generated next. Node 23 is determined to be a goal node and the search terminates. In this case LC-search was almost as efficient as using the exact function $c()$. It should be noted that with a suitable choice for $\hat{c}()$, an LC-search will be far more selective than any of the other search methods we have discussed.

1	2	3	4
5	6		8
9	10	11	12
13	14	15	7

Figure 8.5 Problem state

8.1.3 Control Abstractions for LC-Search

Let t be a state space tree and $c()$ a cost function for the nodes in t. If x is a node in t, then $c(x)$ is the minimum cost of any answer node in the subtree with root x. Thus, $c(t)$ is the cost of a minimum-cost answer node in t. As remarked earlier, it is usually not possible to find an easily computable function $c()$ as defined above. Instead, a heuristic $\hat{c}$ that estimates $c()$ is used. This heuristic should be easy to compute and generally has the property that if x is either an answer node or a leaf node, then $c(x) = \hat{c}(x)$. LCSearch (Algorithm 8.1) uses $\hat{c}$ to find an answer node. The algorithm uses two functions Least() and Add(x) to delete and add a live node from or to the list of live nodes, respectively. Least() finds a live node with least $\hat{c}()$. This node is deleted from the list of live nodes and returned. Add(x) adds the new live node x to the list of live nodes. The list of live nodes will usually be implemented as a min-heap (Section 2.4). Algorithm LCSearch outputs the path from the answer node it finds to the root node t. This is easy to do if with each node x that becomes live, we associate a field *parent* which gives the parent of node x. When an answer node g is found, the path from

g to t can be determined by following a sequence of *parent* values starting from the current E-node (which is the parent of g) and ending at node t.

```
    listnode = record {
        listnode *next, *parent; float cost;
    }

1   Algorithm LCSearch(t)
2   // Search t for an answer node.
3   {
4       if *t is an answer node then output *t and return;
5       E := t; // E-node.
6       Initialize the list of live nodes to be empty;
7       repeat
8       {
9           for each child x of E do
10          {
11              if x is an answer node then output the path
12                  from x to t and return;
13              Add(x); // x is a new live node.
14              (x → parent) := E; // Pointer for path to root.
15          }
16          if there are no more live nodes then
17          {
18              write ("No answer node"); return;
19          }
20          E := Least();
21      } until (false);
22  }
```

Algorithm 8.1 LC-search

The correctness of algorithm LCSearch is easy to establish. Variable E always points to the current E-node. By the definition of LC-search, the root node is the first E-node (line 5). Line 6 initializes the list of live nodes. At any time during the execution of LCSearch, this list contains all live nodes except the E-node. Thus, initially this list should be empty (line 6). The **for** loop of line 9 examines all the children of the E-node. If one of the children is an answer node, then the algorithm outputs the path from x to t and terminates. If a child of E is not an answer node, then it becomes a live node. It is added to the list of live nodes (line 13) and its *parent* field set to

E (line 14). When all the children of E have been generated, E becomes a dead node and line 16 is reached. This happens only if none of E's children is an answer node. So, the search must continue further. If there are no live nodes left, then the entire state space tree has been searched and no answer nodes found. The algorithm terminates in line 18. Otherwise, Least(), by definition, correctly chooses the next E-node and the search continues from here.

From the preceding discussion, it is clear that LCSearch terminates only when either an answer node is found or the entire state space tree has been generated and searched. Thus, termination is guaranteed only for finite state space trees. Termination can also be guaranteed for infinite state space trees that have at least one answer node provided a "proper" choice for the cost function $\hat{c}()$ is made. This is the case, for example, when $\hat{c}(x) > \hat{c}(y)$ for every pair of nodes x and y such that the level number of x is "sufficiently" higher than that of y. For infinite state space trees with no answer nodes, LCSearch will not terminate. Thus, it is advisable to restrict the search to find answer nodes with a cost no more than a given bound C.

One should note the similarity between algorithm LCSearch and algorithms for a breadth first search and D-search of a state space tree. If the list of live nodes is implemented as a queue with Least() and Add(x) being algorithms to delete an element from and add an element to the queue, then LCSearch will be transformed to a FIFO search schema. If the list of live nodes is implemented as a stack with Least() and Add(x) being algorithms to delete and add elements to the stack, then LCSearch will carry out a LIFO search of the state space tree. Thus, the algorithms for LC, FIFO, and LIFO search are essentially the same. The only difference is in the implementation of the list of live nodes. This is to be expected as the three search methods differ only in the selection rule used to obtain the next E-node.

8.1.4 Bounding

A branch-and-bound method searches a state space tree using any search mechanism in which all the children of the E-node are generated before another node becomes the E-node. We assume that each answer node x has a cost $c(x)$ associated with it and that a minimum-cost answer node is to be found. Three common search strategies are FIFO, LIFO, and LC. (Another method, heuristic search, is discussed in the exercises.) A cost function $\hat{c}(\cdot)$ such that $\hat{c}(x) \leq c(x)$ is used to provide lower bounds on solutions obtainable from any node x. If *upper* is an upper bound on the cost of a minimum-cost solution, then all live nodes x with $\hat{c}(x) > upper$ may be killed as all answer nodes reachable from x have cost $c(x) \geq \hat{c}(x) > upper$. The starting value for *upper* can be obtained by some heuristic or can be set to ∞. Clearly, so long as the initial value for *upper* is no less than the cost of a minimum-cost answer node, the above rules to kill live nodes will not result in the killing of

a live node that can reach a minimum-cost answer node. Each time a new answer node is found, the value of *upper* can be updated.

Let us see how these ideas can be used to arrive at branch-and-bound algorithms for optimization problems. In this section we deal directly only with minimization problems. A maximization problem is easily converted to a minimization problem by changing the sign of the objective function. We need to be able to formulate the search for an optimal solution as a search for a least-cost answer node in a state space tree. To do this, it is necessary to define the cost function $c(\cdot)$ such that $c(x)$ is minimum for all nodes representing an optimal solution. The easiest way to do this is to use the objective function itself for $c(\cdot)$. For nodes representing feasible solutions, $c(x)$ is the value of the objective function for that feasible solution. For nodes representing infeasible solutions, $c(x) = \infty$. For nodes representing partial solutions, $c(x)$ is the cost of the minimum-cost node in the subtree with root x. Since $c(x)$ is in general as hard to compute as the original optimization problem is to solve, the branch-and-bound algorithm will use an estimate $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x. In general then, the $\hat{c}(\cdot)$ function used in a branch-and-bound solution to optimization functions will estimate the objective function value and not the computational difficulty of reaching an answer node. In addition, to be consistent with the terminology used in connection with the 15-puzzle, any node representing a feasible solution (a solution node) will be an answer node. However, only minimum-cost answer nodes will correspond to an optimal solution. Thus, answer nodes and solution nodes are indistinguishable.

As an example optimization problem, consider the job sequencing with deadlines problem introduced in Section 4.4. We generalize this problem to allow jobs with different processing times. We are given n jobs and one processor. Each job i has associated with it a three tuple (p_i, d_i, t_i). Job i requires t_i units of processing time. If its processing is not completed by the deadline d_i, then a penalty p_i is incurred. The objective is to select a subset J of the n jobs such that all jobs in J can be completed by their deadlines. Hence, a penalty can be incurred only on those jobs not in J. The subset J should be such that the penalty incurred is minimum among all possible subsets J. Such a J is optimal.

Consider the following instance: $n = 4$, $(p_1, d_1, t_1) = (5, 1, 1)$, $(p_2, d_2, t_2) = (10, 3, 2)$, $(p_3, d_3, t_3) = (6,\ 2,\ 1)$, and $(p_4, d_4, t_4) = (3,\ 1,\ 1)$. The solution space for this instance consists of all possible subsets of the job index set $\{1, 2, 3, 4\}$. The solution space can be organized into a tree by means of either of the two formulations used for the sum of subsets problem (Example 7.2). Figure 8.6 corresponds to the variable tuple size formulation while Figure 8.7 corresponds to the fixed tuple size formulation. In both figures square nodes represent infeasible subsets. In Figure 8.6 all nonsquare nodes are answer nodes. Node 9 represents an optimal solution and is the only minimum-cost answer node. For this node $J = \{2, 3\}$ and the penalty (cost)

is 8. In Figure 8.7 only nonsquare leaf nodes are answer nodes. Node 25 represents the optimal solution and is also a minimum-cost answer node. This node corresponds to $J = \{2, 3\}$ and a penalty of 8. The costs of the answer nodes of Figure 8.7 are given below the nodes.

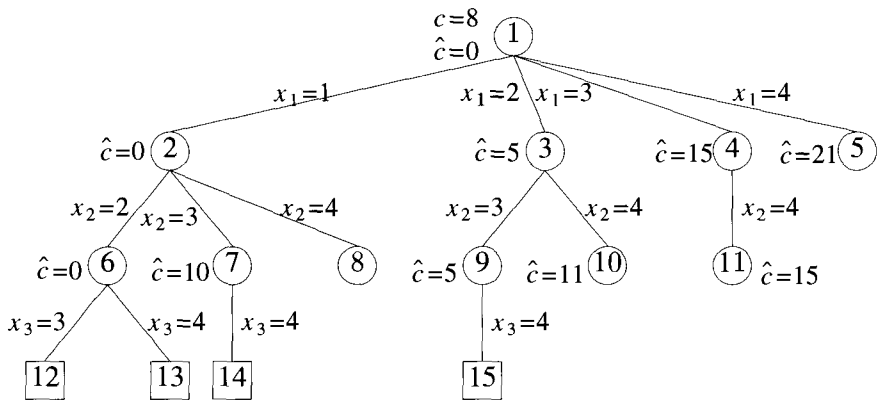


Figure 8.6 State space tree corresponding to variable tuple size formulation

We can define a cost function $c()$ for the state space formulations of Figures 8.6 and 8.7. For any circular node x, $c(x)$ is the minimum penalty corresponding to any node in the subtree with root x. The value of $c(x) = \infty$ for a square node. In the tree of Figure 8.6, $c(3) = 8$, $c(2) = 9$, and $c(1) = 8$. In the tree of Figure 8.7, $c(1) = 8$, $c(2) = 9$, $c(5) = 13$, and $c(6) = 8$. Clearly, $c(1)$ is the penalty corresponding to an optimal selection J.

A bound $\hat{c}(x)$ such that $\hat{c}(x) \leq c(x)$ for all x is easy to obtain. Let S_x be the subset of jobs selected for J at node x. If $m = \max\{i | i \in S_x\}$, then $\hat{c}(x) = \sum_{\substack{i<m \\ i \notin S_x}} p_i$ is an estimate for $c(x)$ with the property $\hat{c}(x) \leq c(x)$. For each circular node x in Figures 8.6 and 8.7, the value of $\hat{c}(x)$ is the number outside node x. For a square node, $\hat{c}(x) = \infty$. For example, in Figure 8.6 for node 6, $S_6 = \{1, 2\}$ and hence $m = 2$. Also, $\sum_{\substack{i<2 \\ i \notin S_2}} p_i = 0$. For node 7, $S_7 = \{1, 3\}$ and $m = 3$. Therefore, $\sum_{\substack{i<2 \\ i \notin S_2}} p_i = p_2 = 10$. And so on. In Figure 8.7, node 12 corresponds to the omission of job 1 and hence a penalty of 5; node 13 corresponds to the omission of jobs 1 and 3 and hence a penalty of 11; and so on.

A simple upper bound $u(x)$ on the cost of a minimum-cost answer node in the subtree x is $u(x) = \sum_{i \notin S_x} p_i$. Note that $u(x)$ is the cost of the solution S_x corresponding to node x.

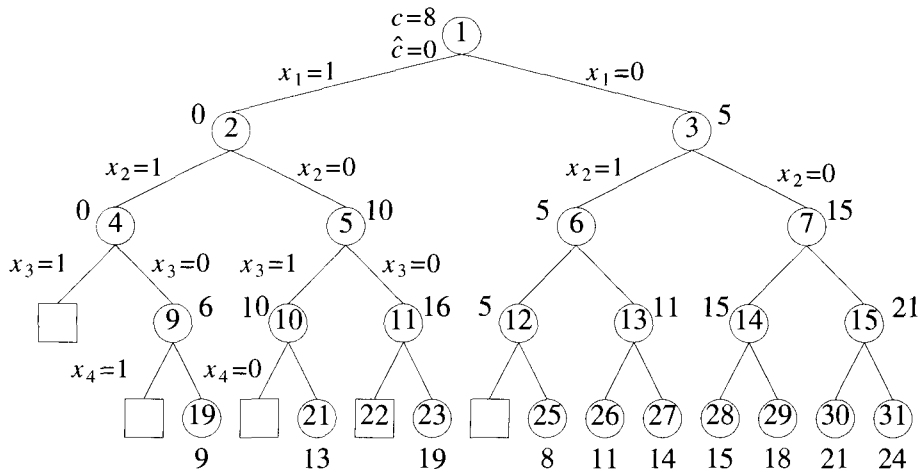


Figure 8.7 State space tree corresponding to fixed tuple size formulation

8.1.5 FIFO Branch-and-Bound

A FIFO branch-and-bound algorithm for the job sequencing problem can begin with $upper = \infty$ (or $upper = \sum_{1 \le i \le n} p_i$) as an upper bound on the cost of a minimum-cost answer node. Starting with node 1 as the E-node and using the variable tuple size formulation of Figure 8.6, nodes 2, 3, 4, and 5 are generated (in that order). Then $u(2) = 19$, $u(3) = 14$, $u(4) = 18$, and $u(5) = 21$. For example, node 2 corresponds to the inclusion of job 1. Thus $u(2)$ is obtained by summing the penalties of all the other jobs. The variable *upper* is updated to 14 when node 3 is generated. Since $\hat{c}(4)$ and $\hat{c}(5)$ are greater than *upper*, nodes 4 and 5 get killed (or bounded). Only nodes 2 and 3 remain alive. Node 2 becomes the next E-node. Its children, nodes 6, 7, and 8 are generated. Then $u(6) = 9$ and so *upper* is updated to 9. The cost $\hat{c}(7) = 10 > upper$ and node 7 gets killed. Node 8 is infeasible and so it is killed. Next, node 3 becomes the E-node. Nodes 9 and 10 are now generated. Then $u(9) = 8$ and so *upper* becomes 8. The cost $\hat{c}(10) = 11 > upper$, and this node is killed. The next E-node is node 6. Both its children are infeasible. Node 9's only child is also infeasible. The minimum-cost answer node is node 9. It has a cost of 8.

When implementing a FIFO branch-and-bound algorithm, it is not economical to kill live nodes with $\hat{c}(x) > upper$ each time *upper* is updated. This is so because live nodes are in the queue in the order in which they were generated. Hence, nodes with $\hat{c}(x) > upper$ are distributed in some

random way in the queue. Instead, live nodes with $\hat{c}(x) > upper$ can be killed when they are about to become E-nodes.

From here on we shall refer to the FIFO-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ as FIFOBB.

8.1.6 LC Branch-and-Bound

An LC branch-and-bound search of the tree of Figure 8.6 will begin with $upper = \infty$ and node 1 as the first E-node. When node 1 is expanded, nodes 2, 3, 4, and 5 are generated in that order. As in the case of FIFOBB, *upper* is updated to 14 when node 3 is generated and nodes 4 and 5 are killed as $\hat{c}(4) > upper$ and $\hat{c}(5) > upper$. Node 2 is the next E-node as $\hat{c}(2) = 0$ and $\hat{c}(3) = 5$. Nodes 6, 7, and 8 are generated and *upper* is updated to 9 when node 6 is generated. So, node 7 is killed as $\hat{c}(7) = 10 > upper$. Node 8 is infeasible and so killed. The only live nodes now are nodes 3 and 6. Node 6 is the next E-node as $\hat{c}(6) = 0 < \hat{c}(3)$. Both its children are infeasible. Node 3 becomes the next E-node. When node 9 is generated, *upper* is updated to 8 as $u(9) = 8$. So, node 10 with $\hat{c}(10) = 11$ is killed on generation. Node 9 becomes the next E-node. Its only child is infeasible. No live nodes remain. The search terminates with node 9 representing the minimum-cost answer node.

From here on we refer to the LC(LIFO)-based branch-and-bound algorithm with an appropriate $\hat{c}(.)$ and $u(.)$ as LCBB (LIFOBB).

EXERCISES

1. Prove Theorem 8.1.

2. Present an algorithm schema FifoBB for a FIFO branch-and-bound search for a least-cost answer node.

3. Give an algorithm schema LcBB for a LC branch-and-bound search for a least-cost answer node.

4. Write an algorithm schema LifoBB, for a LIFO branch-and-bound search for a least-cost answer node.

5. Draw the portion of the state space tree generated by FIFOBB, LCBB, and LIFOBB for the job sequencing with deadlines instance $n = 5$, $(p_1, p_2, \ldots, p_5) = (6, 3, 4, 8, 5)$, $(t_1, t_2, \ldots, t_5) = (2, 1, 2, 1, 1)$, and $(d_1, d_2, \ldots, d_5) = (3, 1, 4, 2, 4)$. What is the penalty corresponding to an optimal solution? Use a variable tuple size formulation and $\hat{c}(\cdot)$ and $u(\cdot)$ as in Section 8.1.

6. Write a branch-and-bound algorithm for the job sequencing with deadlines problem. Use the fixed tuple size formulation.

7. (a) Write a branch-and-bound algorithm for the job sequencing with deadlines problem using a dominance rule (see Section 5.7). Your algorithm should work with a fixed tuple size formulation and should generate nodes by levels. Nodes on each level should be kept in an order permitting easy use of your dominance rule.

   (b) Convert your algorithm into a program and, using randomly generated problem instances, determine the worth of the dominance rule as well as the bounding functions. To do this, you will have to run four versions of your program: ProgA$\cdots$ bounding functions and dominance rules are removed, ProgB$\cdots$ dominance rule is removed, ProgC$\cdots$ bounding function is removed, and ProgD$\cdots$ bounding functions and dominance rules are included. Determine computing time figures as well as the number of nodes generated.

8.2 0/1 KNAPSACK PROBLEM

To use the branch-and-bound technique to solve any problem, it is first necessary to conceive of a state space tree for the problem. We have already seen two possible state space tree organizations for the knapsack problem (Section 7.6). Still, we cannot directly apply the techniques of Section 8.1 since these were discussed with respect to minimization problems whereas the knapsack problem is a maximization problem. This difficulty is easily overcome by replacing the objective function $\sum p_i x_i$ by the function $-\sum p_i x_i$. Clearly, $\sum p_i x_i$ is maximized iff $-\sum p_i x_i$ is minimized. This modified knapsack problem is stated as (8.1).

$$\text{minimize } -\sum_{i=1}^{n} p_i x_i$$

$$\text{subject to } \sum_{i=1}^{n} w_i x_i \leq m \tag{8.1}$$

$$x_i = 0 \text{ or } 1, \quad 1 \leq i \leq n$$

We continue the discussion assuming a fixed tuple size formulation for the solution space. The discussion is easily extended to the variable tuple size formulation. Every leaf node in the state space tree representing an assignment for which $\sum_{1 \leq i \leq n} w_i x_i \leq m$ is an answer (or solution) node. All other leaf nodes are infeasible. For a minimum-cost answer node to correspond to any optimal solution, we need to define $c(x) = -\sum_{1 \leq i \leq n} p_i x_i$ for every

answer node x. The cost $c(x) = \infty$ for infeasible leaf nodes. For nonleaf nodes, $c(x)$ is recursively defined to be min $\{c(lchild(x)), c(rchild(x))\}$.

We now need two functions $\hat{c}(x)$ and $u(x)$ such that $\hat{c}(x) \leq c(x) \leq u(x)$ for every node x. The cost $\hat{c}(\cdot)$ and $u(\cdot)$ satisfying this requirement may be obtained as follows. Let x be a node at level j, $1 \leq j \leq n+1$. At node x assignments have already been made to x_i, $1 \leq i < j$. The cost of these assignments is $-\sum_{1 \leq i < j} p_i x_i$. So, $c(x) \leq -\sum_{1 \leq i < j} p_i x_i$ and we may use $u(x) = -\sum_{1 \leq i < j} p_i x_i$. If $q = -\sum_{1 \leq i < j} p_i x_i$, then an improved upper bound function $u(x)$ is $u(x) = \mathsf{UBound}(q, \sum_{1 \leq i < j} w_i x_i, j-1, m)$, where UBound is defined in Algorithm 8.2. As for $c(x)$, it is clear that $\mathsf{Bound}(-q, \sum_{1 \leq i < j} w_i x_i, j-1) \leq c(x)$, where Bound is as given in Algorithm 7.11.

```
1   Algorithm UBound(cp, cw, k, m)
2   // cp, cw, k, and m have the same meanings as in
3   // Algorithm 7.11. w[i] and p[i] are respectively
4   // the weight and profit of the ith object.
5   {
6       b := cp; c := cw;
7       for i := k + 1 to n do
8       {
9           if (c + w[i] ≤ m) then
10          {
11              c := c + w[i]; b := b − p[i];
12          }
13      }
14      return b;
15  }
```

Algorithm 8.2 Function $u(\cdot)$ for knapsack problem

8.2.1 LC Branch-and-Bound Solution

Example 8.2 [LCBB] Consider the knapsack instance $n = 4$, $(p_1, p_2, p_3, p_4) = (10, 10, 12, 18)$, $(w_1, w_2, w_3, w_4) = (2, 4, 6, 9)$, and $m = 15$. Let us trace the working of an LC branch-and-bound search using $\hat{c}(\cdot)$ and $u(\cdot)$ as defined previously. We continue to use the fixed tuple size formulation. The search begins with the root as the E-node. For this node, node 1 of Figure 8.8, we have $\hat{c}(1) = -38$ and $u(1) = -32$.

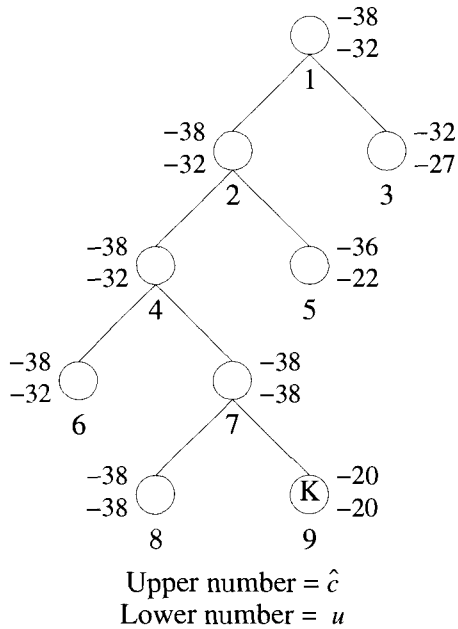


Figure 8.8 LC branch-and-bound tree for Example 8.2

The computation of $u(1)$ and $\hat{c}(1)$ is done as follows. The bound $u(1)$ has a value UBound$(0,0,0,15)$. UBound scans through the objects from left to right starting from j; it adds these objects into the knapsack until the first object that doesn't fit is encountered. At this time, the negation of the total profit of all the objects in the knapsack plus cw is returned. In Function UBound, c and b start with a value of zero. For $i = 1, 2$, and 3, c gets incremented by $2, 4$, and 6, respectively. The variable b also gets decremented by $10, 10$, and 12, respectively. When $i = 4$, the test $(c + w[i] \leq m)$ fails and hence the value returned is -32. Function Bound is similar to UBound, except that it also considers a fraction of the first object that doesn't fit the knapsack. For example, in computing $\hat{c}(1)$, the first object that doesn't fit is 4 whose weight is 9. The total weight of the objects 1, 2, and 3 is 12. So, Bound considers a fraction $\frac{3}{9}$ of the object 4 and hence returns $-32 - \frac{3}{9} * 18 = -38$.

Since node 1 is not a solution node, LCBB sets $ans = 0$ and $upper = -32$ (ans being a variable to store intermediate answer nodes). The E-node is expanded and its two children, nodes 2 and 3, generated. The cost $\hat{c}(2) = -38$, $\hat{c}(3) = -32$, $u(2) = -32$, and $u(3) = -27$. Both nodes are put onto the list of live nodes. Node 2 is the next E-node. It is expanded and nodes 4 and 5 generated. Both nodes get added to the list of live nodes. Node 4 is the live node with least $\hat{c}$ value and becomes the next E-node. Nodes 6 and 7 are generated. Assuming node 6 is generated first, it is added to the list of live nodes. Next, node 7 joins this list and $upper$ is updated to -38. The next E-node will be one of nodes 6 and 7. Let us assume it is node 7. Its two children are nodes 8 and 9. Node 8 is a solution node. Then $upper$ is updated to -38 and node 8 is put onto the live nodes list. Node 9 has $\hat{c}(9) > upper$ and is killed immediately. Nodes 6 and 8 are two live nodes with least $\hat{c}$. Regardless of which becomes the next E-node, $\hat{c}(E) \geq upper$ and the search terminates with node 8 the answer node. At this time, the value -38 together with the path 8, 7, 4, 2, 1 is printed out and the algorithm terminates. From the path one cannot figure out the assignment of values to the x_i's such that $\sum p_i x_i = upper$. Hence, a proper implementation of LCBB has to keep additional information from which the values of the x_i's can be extracted. One way is to associate with each node a one bit field, tag. The sequence of tag bits from the answer node to the root give the x_i values. Thus, we have $tag(2) = tag(4) = tag(6) = tag(8) = 1$ and $tag(3) = tag(5) = tag(7) = tag(9) = 0$. The tag sequence for the path 8, 7, 4, 2, 1 is 1 0 1 1 and so $x_4 = 1, x_3 = 0, x_2 = 1$, and $x_1 = 1$. □

To use LCBB to solve the knapsack problem, we need to specify (1) the structure of nodes in the state space tree being searched, (2) how to generate the children of a given node, (3) how to recognize a solution node, and (4) a representation of the list of live nodes and a mechanism for adding a node into the list as well as identifying the least-cost node. The node structure needed depends on which of the two formulations for the state space tree is being used. Let us continue with a fixed size tuple formulation. Each node

x that is generated and put onto the list of live nodes must have a *parent* field. In addition, as noted in Example 8.2, each node should have a one bit *tag* field. This field is needed to output the x_i values corresponding to an optimal solution. To generate x's children, we need to know the level of node x in the state space tree. For this we shall use a field *level*. The left child of x is chosen by setting $x_{level(x)} = 1$ and the right child by setting $x_{level(x)} = 0$. To determine the feasibility of the left child, we need to know the amount of knapsack space available at node x. This can be determined either by following the path from node x to the root or by explicitly retaining this value in the node. Say we choose to retain this value in a field *cu* (capacity unused). The evaluation of $\hat{c}(x)$ and $u(x)$ requires knowledge of the profit $\sum_{1 \le i < level(x)} p_i x_i$ earned by the filling corresponding to node x. This can be computed by following the path from x to the root. Alternatively, this value can be explicitly retained in a field *pe*. Finally, in order to determine the live node with least $\hat{c}$ value or to insert nodes properly into the list of live nodes, we need to know $\hat{c}(x)$. Again, we have a choice. The value $\hat{c}(x)$ may be stored explicitly in a field *ub* or may be computed when needed. Assuming all information is kept explicitly, we need nodes with six fields each: *parent*, *level*, *tag*, *cu*, *pe*, and *ub*.

Using this six-field node structure, the children of any live node x can be easily determined. The left child y is feasible iff $cu(x) \ge w_{level(x)}$. In this case, $parent(y) = x$, $level(y) = level(x) + 1$, $cu(y) = cu(x) - w_{level(x)}$, $pe(y) = pe(x) + p_{level(x)}$, $tag(y) = 1$, and $ub(y) = ub(x)$. The right child can be generated similarly. Solution nodes are easily recognized too. Node x is a solution node iff $level(x) = n + 1$.

We are now left with the task of specifying the representation of the list of live nodes. The functions we wish to perform on this list are (1) test if the list is empty, (2) add nodes, and (3) delete a node with least *ub*. We have seen a data structure that allows us to perform these three functions efficiently: a min-heap. If there are m live nodes, then function (1) can be carried out in $\Theta(1)$ time, whereas functions (2) and (3) require only $O(\log m)$ time.

8.2.2 FIFO Branch-and-Bound Solution

Example 8.3 Now, let us trace through the FIFOBB algorithm using the same knapsack instance as in Example 8.2. Initially the root node, node 1 of Figure 8.9, is the E-node and the queue of live nodes is empty. Since this is not a solution node, *upper* is initialized to $u(1) = -32$.

We assume the children of a node are generated left to right. Nodes 2 and 3 are generated and added to the queue (in that order). The value of *upper* remains unchanged. Node 2 becomes the next E-node. Its children, nodes 4 and 5, are generated and added to the queue. Node 3, the next

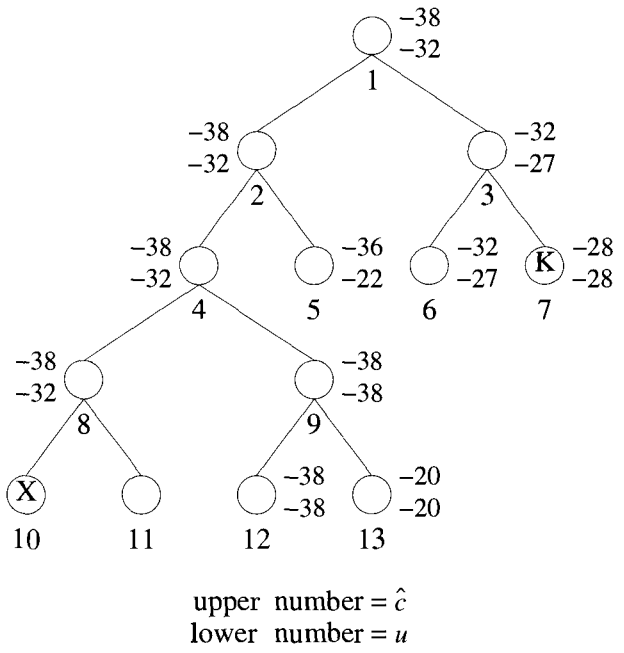


Figure 8.9 FIFO branch-and-bound tree for Example 8.3

E-node, is expanded. Its children nodes are generated. Node 6 gets added to the queue. Node 7 is immediately killed as $\hat{c}(7) > upper$. Node 4 is expanded next. Nodes 8 and 9 are generated and added to the queue. Then $upper$ is updated to $u(9) = -38$. Nodes 5 and 6 are the next two nodes to become E-nodes. Neither is expanded as for each, $\hat{c}() > upper$. Node 8 is the next E-node. Nodes 10 and 11 are generated. Node 10 is infeasible and so killed. Node 11 has $\hat{c}(11) > upper$ and so is also killed. Node 9 is expanded next. When node 12 is generated, $upper$ and ans are updated to -38 and 12 respectively. Node 12 joins the queue of live nodes. Node 13 is killed before it can get onto the queue of live nodes as $\hat{c}(13) > upper$. The only remaining live node is node 12. It has no children and the search terminates. The value of $upper$ and the path from node 12 to the root is output. As in the case of Example 8.2, additional information is needed to determine the x_i values on this path. □

At first we may be tempted to discard FIFOBB in favor of LCBB in solving knapsack. Our intuition leads us to believe that LCBB will examine fewer nodes in its quest for an optimal solution. However, we should keep in mind that insertions into and deletions form a heap are far more expensive (proportional to the logarithm of the heap size) than the corresponding operations on a queue ($\Theta(1)$). Consequently, the work done for each E-node is more in LCBB than in FIFOBB. Unless LCBB uses far fewer E-nodes than FIFOBB, FIFOBB will outperform (in terms of real computation time) LCBB.

We have now seen four different approaches to solving the knapsack problem: dynamic programming, backtracking, LCBB, and FIFOBB. If we compare the dynamic programming algorithm DKnap (Algorithm 5.7) and FIFOBB, we see that there is a correspondence between generating the $S^{(i)}$'s and generating nodes by levels. $S^{(i)}$ contains all pairs (P, W) corresponding to nodes on level $i+1$, $0 \leq i \leq n$. Hence, both algorithms generate the state space tree by levels. The dynamic programming algorithm, however, keeps the nodes on each level ordered by their profit earned (P) and capacity used (W) values. No two tuples have the same P or W value. In FIFOBB we may have many nodes on the same level with the same P or W value. It is not easy to implement the dominance rule of Section 5.7 into FIFOBB as nodes on a level are not ordered by their P or W values. However, the bounding rules can easily be incorporated into DKnap. Toward the end of Section 5.7 we discussed some simple heuristics to determine whether a pair $(P, W) \in S^{(i)}$ should be killed. These heuristics are readily seen to be bounding functions of the type discussed here. Let the algorithm resulting from the inclusion of the bounding functions into DKnap be DKnap1. DKnap1 is expected to be superior to FIFOBB as it uses the dominance rule in addition to the bounding functions. In addition, the overhead incurred each time a node is generated is less.

To determine which of the knapsack algorithms is best, it is necessary to program them and obtain real computing times for different data sets. Since the effectiveness of the bounding functions and the dominance rule is highly data dependent, we expect a wide variation in the computing time for different problem instances having the same number of objects n. To get representative times, it is necessary to generate many problem instances for a fixed n and obtain computing times for these instances. The generation of these data sets and the problem of conducting the tests is discussed in a programming project at the end of this section. The results of some tests can be found in the references to this chapter.

Before closing our discussion of the knapsack problem, we briefly discuss a very effective heuristic to reduce a knapsack instance with large n to an equivalent one with smaller n. This heuristic, Reduce, uses some of the ideas developed for the branch-and-bound algorithm. It classifies the objects $\{1, 2, \ldots, n\}$ into one of three categories $I1, I2$, and $I3$. $I1$ is a set of objects for which x_i must be 1 in every optimal solution. $I2$ is a set for which x_i must be 0. $I3$ is $\{1, 2, \ldots, n\} - I1 - I2$. Once $I1$, $I2$, and $I3$ have been determined, we need to solve only the reduced knapsack instance

$$\text{maximize} \sum_{i \in I3} p_i x_i$$

$$\text{subject to} \sum_{i \in I3} w_i x_i \leq m - \sum_{i \in I1} w_i x_i \tag{8.2}$$

$$x_i = 0 \text{ or } 1$$

From the solution to (8.2) an optimal solution to the original knapsack instance is obtained by setting $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$.

Function Reduce (Algorithm 8.3) makes use of two functions Ubb and Lbb. The bound Ubb($I1$, $I2$) is an upper bound on the value of an optimal solution to the given knapsack instance with added constraints $x_i = 1$ if $i \in I1$ and $x_i = 0$ if $i \in I2$. The bound Lbb($I1$, $I2$) is a lower bound under the constraints of $I1$ and $I2$. Algorithm Reduce needs no further explanation. It should be clear that $I1$ and $I2$ are such that from an optimal solution to (8.2), we can easily obtain an optimal solution to the original knapsack problem.

The time complexity of Reduce is $O(n^2)$. Because the reduction procedure is very much like the heuristics used in DKnap1 and the knapsack algorithms of this chapter, the use of Reduce does not decrease the overall computing time by as much as may be expected by the reduction in the number of objects. These algorithms do dynamically what Reduce does. The exercises explore the value of Reduce further.

```
1   Algorithm Reduce(p, w, n, m, I1, I2)
2   // Variables are as described in the discussion.
3   // p[i]/w[i] ≥ p[i+1]/w[i+1], 1 ≤ i < n.
4   {
5       I1 := I2 := ∅;
6       q := Lbb(∅, ∅);
7       k := largest j such that w[1] + ··· + w[j] < m;
8       for i := 1 to k do
9       {
10          if (Ubb(∅, {i}) < q) then I1 := I1 ∪ {i};
11          else if (Lbb(∅, {i}) > q) then q := Lbb(∅, {i});
12      }
13      for i := k + 1 to n do
14      {
15          if (Ubb({i}, ∅) < q) then I2 := I2 ∪ {i};
16          else if (Lbb({i}, ∅) > q) then q := Lbb({i}, ∅);
17      }
18  }
```

Algorithm 8.3 Reduction pseudocode for knapsack problem

EXERCISES

1. Work out Example 8.2 using the variable tuple size formulation.
2. Work out Example 8.3 using the variable tuple size formulation.
3. Draw the portion of the state space tree generated by LCBB for the following knapsack instances:
   (a) $n = 5$, $(p_1, p_2, \ldots, p_5) = (10, 15, 6, 8, 4)$, $(w_1, w_2, \ldots, w_5) = (4, 6, 3, 4, 2)$, and $m = 12$.
   (b) $n = 5$, $(p_1, p_2, p_3, p_4, p_5) = (w_1, w_2, w_3, w_4, w_5) = (4, 4, 5, 8, 9)$ and $m = 15$.
4. Do Exercise 3 using LCBB on a dynamic state space tree (see Section 7.6). Use the fixed tuple size formulation.
5. Write a LCBB algorithm for the knapsack problem using the ideas given in Example 8.2.
6. Write a LCBB algorithm for the knapsack problem using the fixed tuple size formulation and the dynamic state space tree of Section 7.6.
7. [Programming project] Program the algorithms DKnap (Algorithm 5.7), DKnap1 (page 399), LCBB for knapsack, and Bknap (Algorithm 7.12). Compare these programs empirically using randomly generated data as below:
   (a) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, and $m = \sum_1^n w_i/2$.
   (b) Random w_i and p_i, $w_i \in [1, 100]$, $p_i \in [1, 100]$, and $m = 2 \max \{w_i\}$.
   (c) Random w_i, $w_i \in [1, 100]$, $p_i = w_i + 10$, and $m = \sum_1^n w_i/2$.
   (d) Same as (c) except $m = 2 \max \{w_i\}$.
   (e) Random p_i, $p_i \in [1, 100]$, $w_i = p_i + 10$, and $m = \sum_1^n w_i/2$.
   (f) Same as (e) except $m = 2 \max \{w_i\}$.

   Obtain computing times for $n = 5, 10, 20, 30, 40, \ldots$. For each n, generate (say) ten problem instances from each of the above data sets. Report average and worst-case computing times for each of the above data sets. From these times can you say anything about the expected behavior of these algorithms?

   Now, generate problem instances with $p_i = w_i$, $1 \leq i \leq n$, $m = \sum w_i/2$, and $\sum w_i x_i \neq m$ for any 0, 1 assignment to the x_i's. Obtain computing times for your four programs for $n = 10$, 20, and 30. Now study the effect of changing the range to [1, 1000] in data sets (a) through (f). In sets (c) to (f) replace $p_i = w_i + 10$ by $p_i = w_i + 100$ and $w_i = p_i + 10$ by $w_i = p_i + 100$.

8. [Programming project]

   (a) Program the reduction heuristic Reduce of Section 8.2. Generate several problem instances from the data sets of Exercise 7 and determine the size of the reduced problem instances. Use n = 100, 200, 500, and 1000.

   (b) Program DKnap and the backtracking algorithm Bknap for the knapsack problem. Compare the effectiveness of Reduce by running several problem instances (as in Exercise 7). Obtain average and worst-case computing times for DKnap and Bknap for the generated problem instances and also for the reduced instances. To the times for the reduced problem instances, add the time required by Reduce. What conclusion can you draw from your experiments?

8.3 TRAVELING SALESPERSON (*)

An $O(n^2 2^n)$ dynamic programming algorithm for the traveling salesperson problem was arrived at in Section 5.9. We now investigate branch-and-bound algorithms for this problem. Although the worst-case complexity of these algorithms will not be any better than $O(n^2 2^n)$, the use of good bounding functions will enable these branch-and-bound algorithms to solve some problem instances in much less time than required by the dynamic programming algorithm.

Let $G = (V, E)$ be a directed graph defining an instance of the traveling salesperson problem. Let c_{ij} equal the cost of edge $\langle i, j \rangle$, $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$, and let $|V| = n$. Without loss of generality, we can assume that every tour starts and ends at vertex 1. So, the solution space S is given by $S = \{1, \pi, 1 | \pi$ is a permutation of $(2, 3, \ldots, n)\}$. Then $|S| = (n-1)!$. The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots, i_{n-1}, 1) \in S$ iff $\langle i_j, i_{j+1} \rangle \in E$, $0 \leq j \leq n-1$, and $i_0 = i_n = 1$. S can be organized into a state space tree similar to that for the n-queens problem (see Figure 7.2). Figure 8.10 shows the tree organization for the case of a complete graph with $|V| = 4$. Each leaf node L is a solution node and represents the tour defined by the path from the root to L. Node 14 represents the tour $i_0 = 1, i_1 = 3, i_2 = 4, i_3 = 2$, and $i_4 = 1$.

To use LCBB to search the traveling salesperson state space tree, we need to define a cost function $c(\cdot)$ and two other functions $\hat{c}(\cdot)$ and $u(\cdot)$ such that $\hat{c}(r) \leq c(r) \leq u(r)$ for all nodes r. The cost $c(\cdot)$ is such that the solution node with least $c(\cdot)$ corresponds to a shortest tour in G. One choice for $c(\cdot)$ is

$$c(A) = \begin{cases} \text{length of tour defined by the path from the root to } A, & \text{if } A \text{ is a leaf} \\ \text{cost of a minimum-cost leaf in the subtree } A, & \text{if } A \text{ is not a leaf} \end{cases}$$

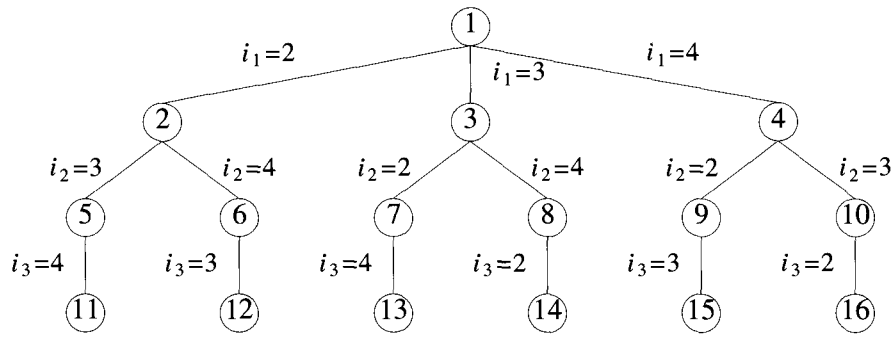


Figure 8.10 State space tree for the traveling salesperson problem with $n = 4$ and $i_0 = i_4 = 1$

A simple $\hat{c}(\cdot)$ such that $\hat{c}(A) \leq c(A)$ for all A is obtained by defining $\hat{c}(A)$ to be the length of the path defined at node A. For example, the path defined at node 6 of Figure 8.10 is $i_0, i_1, i_2 = 1, 2, 4$. It consists of the edges $\langle 1, 2\rangle$ and $\langle 2, 4\rangle$. A better $\hat{c}(\cdot)$ can be obtained by using the reduced cost matrix corresponding to G. A row (column) is said to be *reduced* iff it contains at least one zero and all remaining entries are non-negative. A matrix is *reduced* iff every row and column is reduced. As an example of how to reduce the cost matrix of a given graph G, consider the matrix of Figure 8.11(a). This corresponds to a graph with five vertices. Since every tour on this graph includes exactly one edge $\langle i, j\rangle$ with $i = k$, $1 \leq k \leq 5$, and exactly one edge $\langle i, j\rangle$ with $j = k$, $1 \leq k \leq 5$, subtracting a constant t from every entry in one column or one row of the cost matrix reduces the length of every tour by exactly t. A minimum-cost tour remains a minimum-cost tour following this subtraction operation. If t is chosen to be the minimum entry in row i (column j), then subtracting it from all entries in row i (column j) introduces a zero into row i (column j). Repeating this as often as needed, the cost matrix can be reduced. The total amount subtracted from the columns and rows is a lower bound on the length of a minimum-cost tour and can be used as the $\hat{c}$ value for the root of the state space tree. Subtracting 10, 2, 2, 3, 4, 1, and 3 from rows 1, 2, 3, 4, and 5 and columns 1 and 3 respectively of the matrix of Figure 8.11(a) yields the reduced matrix of Figure 8.11(b). The total amount subtracted is 25. Hence, all tours in the original graph have a length at least 25.

We can associate a reduced cost matrix with every node in the traveling salesperson state space tree. Let A be the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including

edge $\langle i, j \rangle$ in the tour. If S is not a leaf, then the reduced cost matrix for S may be obtained as follows: (1) Change all entries in row i and column j of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j. (2) Set $A(j, 1)$ to ∞. This prevents the use of edge $\langle j, 1 \rangle$. (3) Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞. Let the resulting matrix be B. Steps (1) and (2) are valid as no tour in the subtree s can contain edges of the type $\langle i, k \rangle$ or $\langle k, j \rangle$ or $\langle j, 1 \rangle$ (except for edge $\langle i, j \rangle$). If r is the total amount subtracted in step (3) then $\hat{c}(S) = \hat{c}(R) + A(i, j) + r$. For leaf nodes, $\hat{c}(\cdot) = c()$ is easily computed as each leaf defines a unique tour. For the upper bound function u, we can use $u(R) = \infty$ for all nodes R.

$$\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$$

(a) Cost matrix

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Reduced cost matrix
L = 25

Figure 8.11 An example

Let us now trace the progress of the LCBB algorithm on the problem instance of Figure 8.11(a). We use $\hat{c}$ and u as above. The initial reduced matrix is that of Figure 8.11(b) and *upper* $= \infty$. The portion of the state space tree that gets generated is shown in Figure 8.12. Starting with the root node as the E-node, nodes 2, 3, 4, and 5 are generated (in that order). The reduced matrices corresponding to these nodes are shown in Figure 8.13. The matrix of Figure 8.13(b) is obtained from that of 8.11(b) by (1) setting all entries in row 1 and column 3 to ∞, (2) setting the element at position (3, 1) to ∞, and (3) reducing column 1 by subtracting by 11. The $\hat{c}$ for node 3 is therefore 25 + 17 (the cost of edge $\langle 1, 3 \rangle$ in the reduced matrix) + 11 = 53. The matrices and $\hat{c}$ value for nodes 2, 4, and 5 are obtained similarly. The value of *upper* is unchanged and node 4 becomes the next E-node. Its children 6, 7, and 8 are generated. The live nodes at this time are nodes 2, 3, 5, 6, 7, and 8. Node 6 has least $\hat{c}$ value and becomes the next E-node. Nodes 9 and 10 are generated. Node 10 is the next E-node. The solution node, node 11, is generated. The tour length for this node is $\hat{c}(11) = 28$ and *upper* is updated to 28. For the next E-node, node 5, $\hat{c}(5) = 31 >$ *upper*. Hence, LCBB terminates with 1, 4, 2, 5, 3, 1 as the shortest length tour.

An exercise examines the implementation considerations for the LCBB algorithm. A different LCBB algorithm can be arrived at by considering

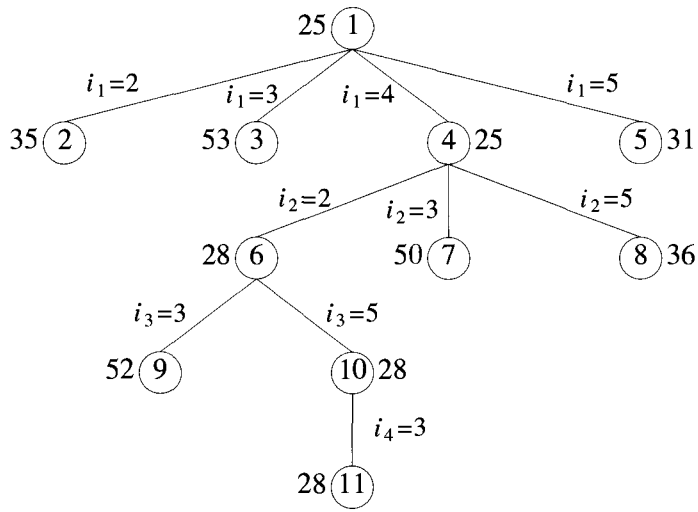


Figure 8.12 State space tree generated by procedure LCBB

a different tree organization for the solution space. This organization is reached by regarding a tour as a collection of n edges. If $G = (V, E)$ has e edges, then every tour contains exactly n of the e edges. However, for each $i, 1 \leq i \leq n$, there is exactly one edge of the form $\langle i, j \rangle$ and one of the form $\langle k, i \rangle$ in every tour. A possible organization for the state space is a binary tree in which a left branch represents the inclusion of a particular edge while the right branch represents the exclusion of that edge. Figure 8.14(b) and (c) represents the first two levels of two possible state space trees for the three vertex graph of Figure 8.14(a). As is true of all problems, many state space trees are possible for a given problem formulation. Different trees differ in the order in which decisions are made. Thus, in Figure 8.14(c) we first decide the fate of edge $\langle 1, 2 \rangle$. Rather than use a static state space tree, we now consider a dynamic state space tree (see Section 7.1). This is also a binary tree. However, the order in which edges are considered depends on the particular problem instance being solved. We compute $\hat{c}$ in the same way as we did using the earlier state space tree formulation.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

(a) Path 1,2; node 2

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$$

(b) Path 1,3; node 3

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

(c) Path 1,4; node 4

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(d) Path 1,5; node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

(e) Path 1,4,2; node 6

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$$

(f) Path 1,4,3; node 7

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

(g) Path 1,4,5; node 8

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$$

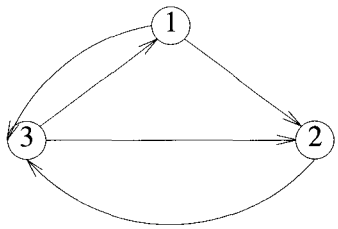
(h) Path 1,4,2,3; node 9

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

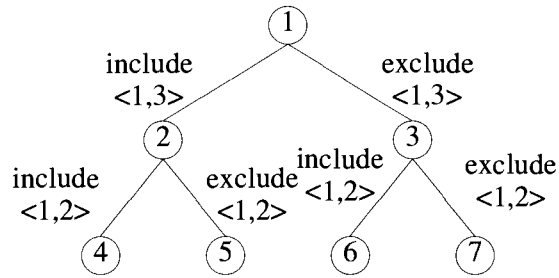
(i) Path 1,4,2,5; node 10

Figure 8.13 Reduced cost matrices corresponding to nodes in Figure 8.12

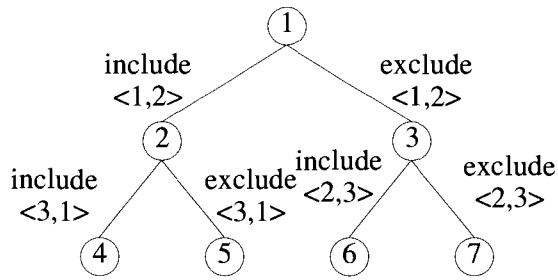
As an example of how LCBB would work on the dynamic binary tree formulation, consider the cost matrix of Figure 8.11(a). Since a total of 25



(a) Graph



(b) Part of a state space tree



(c) Part of a state space tree

Figure 8.14 An example

needs to be subtracted form the rows and columns of this matrix to obtain the reduced matrix of Figure 8.11(b), all tours have a length at least 25. This fact is represented by the root of the state space tree of Figure 8.15. Now, we must decide which edge to use to partition the solution space into two subsets. If edge $\langle i, j \rangle$ is used, then the left subtree of the root represents all tours including edge $\langle i, j \rangle$ and the right subtree represents all tours that do not include edge $\langle i, j \rangle$. If an optimal tour is included in the left subtree, then only $n - 1$ edges remain to be selected. If all optimal tours lie in the right subtree, then we have still to select n edges. Since the left subtree selects fewer edges, it should be easier to find an optimal solution in it than to find one in the right subtree. Consequently, we would like to choose as the partitioning edge an edge $\langle i, j \rangle$ that has the highest probability of being

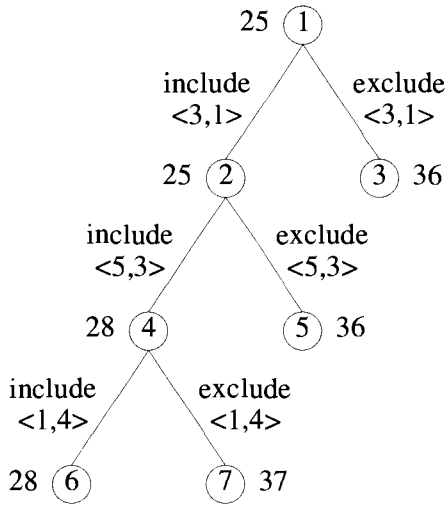


Figure 8.15 State space tree for Figure 8.11(a)

in an optimal tour. Several heuristics for determining such an edge can be formulated. A selection rule that is commonly used is select that edge which results in a right subtree that has highest $\hat{c}$ value. The logic behind this is that we soon have right subtrees (perhaps at lower levels) for which the $\hat{c}$ value is higher than the length of an optimal tour. Another possibility is to choose an edge such that the difference in the $\hat{c}$ values for the left and right subtrees is maximum. Other selection rules are also possible.

When LCBB is used with the first of the two selection rules stated above and the cost matrix of Figure 8.11(a), the tree of Figure 8.15 is generated. At the root node, we have to determine an edge $\langle i, j \rangle$ that will maximize the $\hat{c}$ value of the right subtree. If we select an edge $\langle i, j \rangle$ whose cost in the reduced matrix (Figure 8.11(b)) is positive, then the $\hat{c}$ value of the right subtree will remain 25. This is so as the reduced matrix for the right subtree will have $B(i, j) = \infty$ and all other entries will be identical to those in Figure 8.11(b). Hence B will be reduced and $\hat{c}$ cannot increase. So, we must choose an edge with reduced cost 0. If we choose $\langle 1, 4 \rangle$, then $B(1, 4) = \infty$ and we need to subtract 1 from row 1 to obtain a reduced matrix. In this case $\hat{c}$ will be 26. If $\langle 3, 1 \rangle$ is selected, then 11 needs to be subtracted from column 1 to obtain the reduced matrix for the right subtree. So, $\hat{c}$ will be 36. If A is the reduced cost matrix for node R, then the selection of edge $\langle i, j \rangle$ $(A(i, j) = 0)$ as the next partitioning edge will increase the $\hat{c}$ of the

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 11 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 12 & \infty & 0 \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

(a) Node 2

$$\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 1 & \infty & 11 & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & 12 & \infty & 0 \\ 0 & 0 & 0 & 12 & \infty \end{bmatrix}$$

(b) Node 3

$$\begin{bmatrix} \infty & 7 & \infty & 0 & \infty \\ \infty & \infty & \infty & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(c) Node 4

$$\begin{bmatrix} \infty & 10 & \infty & 0 & 1 \\ \infty & \infty & 0 & 2 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 3 & 1 & \infty & 0 \\ \infty & 0 & \infty & 12 & \infty \end{bmatrix}$$

(d) Node 5

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(e) Node 6

$$\begin{bmatrix} \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & 0 \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

(f) Node 7

Figure 8.16 Reduced cost matrices for Figure 8.15

right subtree by $\triangle = \min_{k \neq j}\{A(i,k)\} + \min_{k \neq i}\{A(k,j)\}$ as this much needs to be subtracted from row i and column j to introduce a zero into both. For edges $\langle 1,4\rangle, \langle 2,5\rangle, \langle 3,1\rangle$ $\langle 3,4\rangle, \langle 4,5\rangle, \langle 5,2\rangle$, and $\langle 5,3\rangle, \triangle = 1$, 2, 11, 0, 3, 3, and 11 respectively. So, either of the edges $\langle 3,1\rangle$ or $\langle 5,3\rangle$ can be used. Let us assume that LCBB selects edge $\langle 3,1\rangle$. The $\hat{c}(2)$ (Figure 8.15) can be computed in a manner similar to that for the state space tree of Figure 8.12. In the corresponding reduced cost matrix all entries in row 3 and column 1 will be ∞. Moreover the entry (1, 3) will also be ∞ as inclusion of this edge will result in a cycle. The reduced matrices corresponding to nodes 2 and 3 are given in Figure 8.16(a) and (b). The $\hat{c}$ values for nodes 2 and 3 (as well as for all other nodes) appear outside the respective nodes.

Node 2 is the next E-node. For edges $\langle 1,4\rangle, \langle 2,5\rangle, \langle 4,5\rangle, \langle 5,2\rangle$, and $\langle 5,3\rangle$, $\triangle = 3, 2, 3, 3$, and 11 respectively. Edge $\langle 5,3\rangle$ is selected and nodes 4 and 5 generated. The corresponding reduced matrices are given in Figure 8.16(c) and (d). Then $\hat{c}(4)$ becomes 28 as we need to subtract 3 from column 2 to reduce this column. Note that entry (1, 5) has been set to ∞ in Figure 8.16(c). This is necessary as the inclusion of edge $\langle 1,5\rangle$ to the collection $\{\langle 3,1\rangle, \langle 5,3\rangle\}$ will result in a cycle. In addition, entries in column 3 and row 5 are set to ∞. Node 4 is the next E-node. The $\triangle$ values corresponding to edges $\langle 1,4\rangle, \langle 2,5\rangle$, and $\langle 4,2\rangle$ are 9, 2, and 0 respectively. Edge $\langle 1,4\rangle$ is selected and nodes 6 and 7 generated. The edge selection at node 6 is $\{\langle 3,1\rangle, \langle 5,3\rangle, \langle 1,4\rangle\}$. This corresponds to the path 5, 3, 1, 4. So, entry (4, 5) is set to ∞ in Figure 8.16(e). In general if edge $\langle i,j\rangle$ is selected, then the entries in row i and column j are set to ∞ in the left subtree. In addition, one more entry needs to be set to ∞. This is an entry whose inclusion in

the set of edges would create a cycle (Exercise 4 examines how to determine this). The next E-node is node 6. At this time three of the five edges have already been selected. The remaining two may be selected directly. The only possibility is $\{\langle 4,2\rangle, \langle 2,5\rangle\}$. This gives the path $5,3,1,4,2,5$ with length 28. So *upper* is updated to 28. Node 3 is the next E-node. Now LCBB terminates as $\hat{c}(3) = 36 > upper$.

In the preceding example, LCBB was modified slightly to handle nodes close to a solution node differently from other nodes. Node 6 is only two levels from a solution node. Rather than evaluate $\hat{c}$ at the children of 6 and then obtain their grandchildren, we just obtained an optimal solution for that subtree by a complete search with no bounding. We could have done something similar when generating the tree of Figure 8.12. Since node 6 is only two levels from the leaf nodes, we can simply skip computing $\hat{c}$ for the children and grandchildren of 6, generate all of them, and pick the best. This works out to be quite efficient as it is easier to generate a subtree with a small number of nodes and evaluate all the solution nodes in it than it is to compute $\hat{c}$ for one of the children of 6. This latter statement is true of many applications of branch-and-bound. Branch-and-bound is used on large subtrees. Once a small subtree is reached (say one with 4 or 6 nodes in it), then that subtree is fully evaluated without using the bounding functions.

We have now seen several branch-and-bound strategies for the traveling salesperson problem. It is not possible to determine analytically which of these is the best. The exercises describe computer experiments that determine empirically the relative performance of the strategies suggested.

EXERCISES

1. Consider the traveling salesperson instance defined by the cost matrix

$$\begin{bmatrix} \infty & 7 & 3 & 12 & 8 \\ 3 & \infty & 6 & 14 & 9 \\ 5 & 8 & \infty & 6 & 18 \\ 9 & 3 & 5 & \infty & 11 \\ 18 & 14 & 9 & 8 & \infty \end{bmatrix}$$

   (a) Obtain the reduced cost matrix

   (b) Using a state space tree formulation similar to that of Figure 8.10 and $\hat{c}$ as described in Section 8.3, obtain the portion of the state space tree that will be generated by LCBB. Label each node by its $\hat{c}$ value. Write out the reduced matrices corresponding to each of these nodes.

   (c) Do part (b) using the reduced matrix method and the dynamic state space tree approach discussed in Section 8.3.

2. Do Exercise 1 using the following traveling salesperson cost matrix:

$$\begin{bmatrix} \infty & 11 & 10 & 9 & 6 \\ 8 & \infty & 7 & 3 & 4 \\ 8 & 4 & \infty & 4 & 8 \\ 11 & 10 & 5 & \infty & 5 \\ 6 & 9 & 5 & 5 & \infty \end{bmatrix}$$

3. (a) Describe an efficient implementation for a LCBB traveling salesperson algorithm using the reduced cost matrix approach and (i) a dynamic state space tree and (ii) a static tree as in Figure 8.10.
   (b) Are there any problem instances for which the LCBB will generate fewer nodes using a static tree than using a dynamic tree? Prove your answer.

4. Consider the LCBB traveling salesperson algorithm described using the dynamic state space tree formulation. Let A and B be nodes. Let B be a child of A. If the edge (A, B) represents the inclusion of edge $\langle i, j \rangle$ in the tour, then in the reduced matrix for B all entries in row i and column j are set to ∞. In addition, one more entry is set to ∞. Obtain an efficient way to determine this entry.

5. [Programming project] Write computer programs for the following traveling salesperson algorithms:
   (a) The dynamic programming algorithm of Chapter 5
   (b) A backtracking algorithm using the static tree formulation of Section 8.3
   (c) A backtracking algorithm using the dynamic tree formulation of Section 8.3
   (d) A LCBB algorithm corresponding to (b)
   (e) A LCBB algorithm corresponding to (c)

   Design data sets to be used to compare the efficiency of the above algorithms. Randomly generate problem instances from these data sets and obtain computing times for your programs. What conclusions can you draw from your computing times?

8.4 EFFICIENCY CONSIDERATIONS

One can pose several questions concerning the performance characteristics of branch-and-bound algorithms that find least-cost answer nodes. We might ask questions such as:

1. Does the use of a better starting value for *upper* always decrease the number of nodes generated?

2. Is it possible to decrease the number of nodes generated by expanding some nodes with $\hat{c}() > upper$?

3. Does the use of a better $\hat{c}$ always result in a decrease in (or at least not an increase in) the number of nodes generated? (A $\hat{c}_2$ is better than $\hat{c}_1$ iff $\hat{c}_1(x) \leq \hat{c}_2(x) \leq c(x)$ for all nodes x.)

4. Does the use of dominance relations ever result in the generation of more nodes than would otherwise be generated?

In this section we answer these questions. Although the answers to most of the questions examined agree with our intuition, the answers to others are contrary to intuition. However, even in cases in which the answer does not agree with intuition, we can expect the performance of the algorithm to generally agree with the intuitive expectations. All the following theorems assume that the branch-and-bound algorithm is to find a minimum-cost solution node. Consequently, $c(x)$ = cost of minimum-cost solution node in subtree x.

Theorem 8.2 Let t be a state space tree. The number of nodes of t generated by FIFO, LIFO, and LC branch-and-bound algorithms cannot be decreased by the expansion of any node x with $\hat{c}(x) \geq upper$, where *upper* is the current upper bound on the cost of a minimum-cost solution node in the tree t.

Proof: The theorem follows from the observation that the value of *upper* cannot be decreased by expanding x (as $\hat{c}(x) \geq upper$). Hence, such an expansion cannot affect the operation of the algorithm on the remainder of the tree. □

Theorem 8.3 Let U_1 and $U_2, U_1 < U_2$, be two initial upper bounds on the cost of a minimum-cost solution node in the state space tree t. Then FIFO, LIFO, and LC branch-and-bound algorithms beginning with U_1 will generate no more nodes than they would if they started with U_2 as the initial upper bound.

Proof: Left as an exercise. □

Theorem 8.4 The use of a better $\hat{c}$ function in conjunction with FIFO and LIFO branch-and-bound algorithms does not increase the number of nodes generated.

Proof: Left as an exercise. □

Theorem 8.5 If a better $\hat{c}$ function is used in a LC branch-and-bound algorithm, the number of nodes generated may increase.

Proof: Consider the state space tree of Figure 8.17. All leaf nodes are solution nodes. The value outside each leaf is its cost. From these values it follows that $c(1) = c(3) = 3$ and $c(2) = 4$. Outside each of nodes 1, 2, and 3 is a pair of numbers $\binom{\hat{c}_1}{\hat{c}_2}$. Clearly, $\hat{c}_2$ is a better function than $\hat{c}_1$. However, if $\hat{c}_2$ is used, node 2 can become the E-node before node 3, as $\hat{c}_2(2) = \hat{c}_2(3)$. In this case all nine nodes of the tree will get generated. When $\hat{c}_1$ is used, nodes 4, 5, and 6 are not generated. □

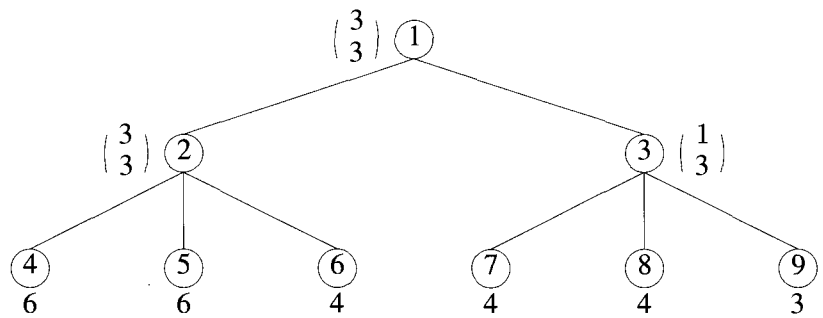


Figure 8.17 Example tree for Theorem 8.5

Now, let us look at the effect of dominance relations. Formally, a dominance relation D is given by a set of tuples, $D = \{(i_1, i_2), (i_3, i_4), (i_5, i_6), \ldots\}$. If $(i, j) \in D$, then node i is said to dominate node j. By this we mean that subtree i contains a solution node with cost no more than the cost of a minimum-cost solution node in subtree j. Dominated nodes can be killed without expansion.

Since every node dominates itself, $(i, i) \in D$ for all i and D. The relation (i, i) should not result in the killing of node i. In addition, it is quite possible for D to contain tuples (i_1, i_2), $(i_2, i_3), (i_3, i_4), \ldots, (i_n, i_1)$. In this case, the transitivity of D implies that each node i_k dominates all nodes $i_j, 1 \leq j \leq n$. Care should be taken to leave at least one of the i_j's alive. A dominance relation D_2 is said to be *stronger* than another dominance relation D_1 iff $D_1 \subset D_2$. In the following theorems I denotes the identity relation $\{(i, i) | 1 \leq i \leq n\}$.

Theorem 8.6 The number of nodes generated during a FIFO or LIFO branch-and-bound search for a least-cost solution node may increase when a stronger dominance relation is used.

Proof: Consider the state space tree of Figure 8.18. The only solution nodes are leaf nodes. Their cost is written outside the node. For the remaining nodes the number outside each node is its $\hat{c}$ value. The two dominance relations to use are $D_1 = I$ and $D_2 = I \cup \{(5,2),(5,8)\}$. Clearly, D_2 is stronger than D_1 and fewer nodes are generated using D_1 rather than D_2. □

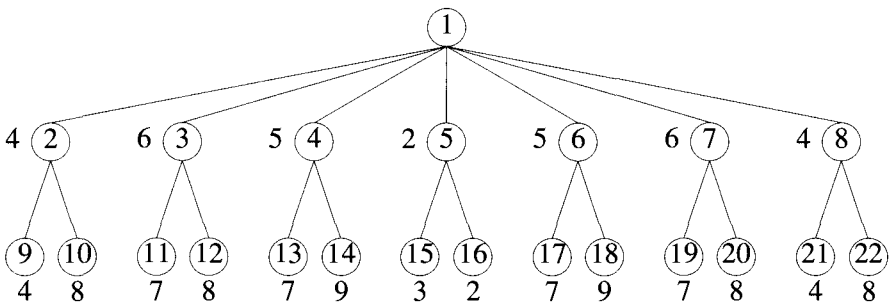


Figure 8.18 Example tree for Theorem 8.6

Theorem 8.7 Let D_1 and D_2 be two dominance relations. Let D_2 be stronger than D_1 and such that $(i,j) \in D_2, i \neq j$, implies $\hat{c}(i) < \hat{c}(j)$. An LC branch-and-bound using D_1 generates at least as many nodes as one using D_2.

Proof: Left as an exercise. □

Theorem 8.8 If the condition $\hat{c}(i) < \hat{c}(j)$ in Theorem 8.7 is removed then an LC branch-and-bound using the relation D_1 may generate fewer nodes than one using D_2.

Proof: Left as an exercise. □

EXERCISES

1. Prove Theorem 8.3.
2. Prove Theorem 8.4.
3. Prove Theorem 8.7.
4. Prove Theorem 8.8.
5. [Heuristic search] Heuristic search is a generalization of FIFO, LIFO, and LC searches. A heuristic function $h(\cdot)$ is used to evaluate all live nodes. The next E-node is the live node with least $h(\cdot)$. Discuss the advantages of using a heuristic function $h(\cdot)$ different from $\hat{c}(\cdot)$ in the search for a least-cost answer node. Consider the knapsack and traveling salesperson problems as two example problems. Also consider any other problems you wish. For these problems devise reasonable functions $h(\cdot)$ (different from $\hat{c}(\cdot)$). Obtain problem instances on which heuristic search performs better than LC-search.

8.5 REFERENCES AND READINGS

LC branch-and-bound algorithms have been extensively studied by researchers in areas such as artificial intelligence and operations research.

Branch-and-bound algorithms using dominance relations in a manner similar to that suggested by FIFOKNAP (resulting in DKnap1) were given by M. Held and R. Karp.

The reduction technique for the knapsack problem is due to G. Ingargiola and J. Korsh.

The reduced matrix technique to compute $\hat{c}$ is due to J. Little, K. Murty, D. Sweeny, and C. Karel. They employed the dynamic state space tree approach.

The results of Section 8.4 are based on the work of W. Kohler, K. Steiglitz, and T. Ibaraki.

The application of branch-and-bound and other techniques to the knapsack and related problems is discussed extensively in *Knapsack Problems: Algorithms and Computer Implementations*, by S. Martello and P. Toth, John Wiley and Sons, 1990.